Fakultät für Informatik
Technische Universität München

Axiomatic Specification and Interactive Verification of Architectural Design Patterns in FACTum

Diego Marmsoler

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:
Prof. Tobias Nipkow, Ph.D.

Prüfende der Dissertation:
1. Prof. Dr. Dr. h.c. Manfred Broy
2. Prof. Dr. Alexander Knapp,
   Universität Augsburg

Die Dissertation wurde am 09.11.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 23.01.2019 angenommen.

ARCHITECTURAL design patterns (ADPs) are an important concept in software engineering used by architects for the design and the analysis of architectures. Usually, an ADP addresses a recurring *architectural design problem* by constraining an *architectural design*. To this end, it provides a set of *guarantees* for architectures implementing the pattern, which formalize correct solutions to the pattern's addressed design problem.

With this thesis, we address the problem that ADPs, as specified in literature, are usually not *verified*, i.e., it is not verified whether the imposed design constraints indeed lead to an architecture satisfying the claimed guarantee. This entails two undesired consequences: (i) The constraints imposed by a pattern may be too weak to ensure the guarantee. Thus, an architecture satisfying the constraints may indeed fail to correctly solve the intended design problem. Therefore, since patterns are usually selected based on the design problem they address, the architecture may not satisfy its requirements. (ii) The constraints imposed by a pattern may be too restrictive for the provided guarantee. While unnecessary constraints are not as severe as missing constraints, they might unnecessarily restrict the application scope of an ADP.

Existing approaches to address this problem usually model ADPs in terms of state machines and apply model checking techniques to verify them. In this thesis, however, we argue that pattern specifications are *axiomatic*, focusing on a few, important properties an architecture must obey. Thus, their verification requires axiomatic reasoning, which is usually not supported by traditional approaches.

With this thesis, we propose an approach which is based on axiomatic specifications and interactive theorem proving. Accordingly, the major outcome of the thesis is FACTUM, a methodology for the axiomatic specification and interactive verification of ADPs. To this end, we provide the following contributions: (i) We provide *specification techniques* to support the axiomatic specification of patterns. (ii) We formalize a model for dynamic architectures in Isabelle/HOL and provide a sound *algorithm* to map an axiomatic pattern specification to a corresponding Isabelle/HOL theory. (iii) To support the axiomatic verification of patterns, we introduce a *calculus* to reason about axiomatic pattern specifications, show its soundness, and implement it in Isabelle/HOL. (iv) We evaluate the approach by means of three well-known ADPs and a larger case study from the domain of Blockchain architectures.

FACTUM is implemented in Eclipse/EMF to support the specification and interactive verification of ADPs. Our results suggest that the approach is well-suited to specify and verify patterns for (potentially dynamic) architectures. In our case studies, for example, we discovered 16 different constraints for four different ADPs. Two of them can be considered fundamental but were not mentioned in any specification of these patterns, so far.

In the long term, this research aims to establish a repository of verified ADPs, which can be filled with verification results for existing or even new patterns. When verifying an architecture, an architect can connect to the repository and verify the architecture against the assumptions provided by the ADPs. The corresponding guarantees are then automatically transferred to the architecture, where they can be used to support in its verification.

Acknowledgements

I would like to thank my adviser *Prof. Manfred Broy* for the freedom in pursuing new, sometimes also unconventional, ideas. His critical look at my work has been a source of constant improvement. I am also grateful for his financial support, which allowed me to present my work to the broader scientific community. I am thankful to Prof. Alexander Knapp for introducing me to the beauty of Formal Methods and reviewing my thesis.

I would also like to thank my fellow doctoral students, in particular the *Mensa-Broy* group, for their company over the last five years. I would like to thank *Vasileios Koutsumpas* for his support and friendship. Special thanks go also to *Mario Gleirscher*, for interesting discussions and fruitful collaborations. I am thankful also to *Maximilian Junker*, for his valuable feedback on my work and his regular reviews of my papers. I am grateful to *Wolfgang Böhm* for his support in any project-related concern. Finally, I would like to thank all the *anonymous reviewers* for their comments on papers related to my thesis.

Last but not least, I would like to thank my parents *Irmgard* and *Andreas* for their lifelong support. Thanks go also to my brother *Tobias* for being my brother. Finally, I am thankful to my wife *Veronika*, not only for her feedback on my work, but also for her support in any other concern. Thank you!

Accompanying Publications This thesis is accompanied by the following publications:

- Diego Marmsoler. Towards a theory of architectural styles. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 823–825. ACM, ACM Press, 2014
- Diego Marmsoler, Alexander Malkis, and Jonas Eckhardt. A model of layered architectures. In Bara Buhnova, Lucia Happe, and Jan Kofron, editors, *Proceedings 12th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2015, London, United Kingdom, April 12th, 2015.*, volume 178 of *EPTCS*, pages 47–61, 2015
- Diego Marmsoler and Mario Gleirscher. Specifying properties of dynamic architectures using configuration traces. In *International Colloquium on Theoretical Aspects of Computing*, pages 235–254. Springer, 2016
- D. Marmsoler and M. Gleirscher. On activation, connection, and behavior in dynamic architectures. *Scientific Annals of Computer Science*, 26(2):187–248, 2016
- Diego Marmsoler and Silvio Degenhardt. Verifying patterns of dynamic architectures using model checking. In *Proceedings International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA@ETAPS 2017, Uppsala, Sweden, 22nd April 2017.*, pages 16–30, 2017
- Diego Marmsoler. On the semantics of temporal specifications of component-behavior for dynamic architectures. In *Eleventh International Symposium on Theoretical Aspects of Software Engineering.* Springer, 2017
- Diego Marmsoler. Towards a calculus for dynamic architectures. In Dang Van Hung and Deepak Kapur, editors, *Theoretical Aspects of Computing - ICTAC 2017 - 14th International Colloquium, Hanoi, Vietnam, October 23-27, 2017, Proceedings*, volume 10580 of *Lecture Notes in Computer Science*, pages 79–99. Springer, 2017
- Diego Marmsoler. Dynamic architectures. *Archive of Formal Proofs*, July 2017. `http://isa-afp.org/entries/DynamicArchitectures.html`, Formal proof development
- Diego Marmsoler. Hierarchical specification and verification of architecture design patterns. In *Fundamental Approaches to Software Engineering - 21th International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, 2018
- Diego Marmsoler. A theory of architectural design patterns. *Archive of Formal Proofs*, March 2018. `http://isa-afp.org/entries/Architectural_Design_Patterns.html`, Formal proof development

- Diego Marmsoler and Habtom Kahsay Gidey. FACTUM Studio: A tool for the axiomatic specification and verification of architectural design patterns. In *Formal Aspects of Component Software - FACS 2018 - 15th International Conference, Proceedings*, 2018
- Diego Marmsoler. A framework for interactive verification of architectural design patterns in Isabelle/HOL. In *The 20th International Conference on Formal Engineering Methods, ICFEM 2018, Proceedings*, 2018
- Diego Marmsoler and Habtom Kahsay Gidey. Interactive verification of architectural design patterns in FACTUM. *Formal Aspects of Computing*, 2019. Under review
- Diego Marmsoler. A calculus of component behavior for dynamic architectures. *Science of Computer Programming*, 2019. Under review

Contents

Part I

Introduction

1 Introduction

The architecture of a system describes the overall organization of a system into components and connections between these components. Since software systems are becoming increasingly big and complex, the architecture of a system plays an ever more important role in their development.

There exist many different definitions of what constitutes an architecture [PW92, SG96, otSEC^{+}00, BS01, BCK07, TMD09]. For the scope of this thesis, we consider the following definition of architecture:

Definition: *Architecture.*

An architecture is a set of components and a description of how these components communicate to each other. Each component has an interface, in terms of input and output ports, and a behavior describing which output is produced for a given input. An architecture may be dynamic, in which case the number of components and connections between these components may change over time.

1.1 Architectural Design Patterns

Architectural design patterns (ADPs) are an important tool in software engineering employed for the conceptualization and analysis of architectures. They capture design experience and are regarded as the "Grand Tool" for designing a software system's architecture [TMD09]. Similar as for architectures, there exist many different definitions of ADPs. In the following, we list some of them:

"An *architectural pattern* is a named collection of *architectural design decisions* that are applicable to a recurring *design problem*, parametrized to account for different software development contexts in which that problem appears." [TMD09]

"An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes *rules and guidelines* for organizing the relationships between them." [BMR^{+}96]

"An *architectural pattern* is a description of element and relation types together with a set of constraints on how they may be used. A pattern can be thought of as a *set of constraints* on an architecture — on the element types and their patterns of interaction — and these constraints define a set or family of architectures that satisfy them." [BCK07]

"An *architectural style* [...] defines a family of [...] systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types, and a *set of constraints* on how they can be combined.” [SG96]

Although these definitions vary in some aspects, two characteristic properties about ADPs can be identified:

1. An ADP solves a recurring *architectural design problem.*
2. An ADP consists of a collection of *architectural design constraints* which restrict the design of architectures.

In the following, we demonstrate this observation by means of three prominent examples: the singleton pattern, the Publisher-Subscriber pattern, and the Blackboard pattern.

Example 1 (The Singleton Pattern)**.** *A very basic example of an ADP, used often in object-oriented systems, is the so-called Singleton pattern [GHJV94]. It aims to address the* problem *that a system must have at most one component of a certain type, activated at each point in time.*

If we look at a Singleton's specification, we usually find a diagram similar to the one depicted in Fig. 1.1. The diagram is accompanied with a description explaining that "instance" contains an instance of the singleton which can be accessed through the interface "getInstance()". Moreover, the description poses a constraint *on an architecture, requiring that a new instance of type singleton is only created if no instance exists yet.*

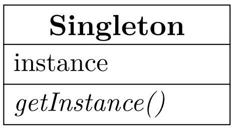


Figure 1.1: Specification of the Singleton pattern as it is usually found in literature.

Example 2 (The Publisher-Subscriber Pattern)**.** *Another ADP often employed to design architectures is the so-called Publisher-Subscriber pattern. It aims to address the* problem *of obtaining a "flexible way of communication" between certain components of an architecture. Thereby, flexibility means that a component can register for certain events at other components and they are notified about the occurrence of such events.*

The pattern is usually described with a diagram similar to the one depicted in Fig.1.2. The description usually requires the existence of two types of components: publishers and subscribers. Thereby, subscribers need to provide a mechanism to subscribe to certain events and publishers are able to publish messages associated to an event. Moreover, the description usually poses a constraint *on the connection between publisher and subscriber components which requires that, whenever a publisher component publishes a message associated to an event for which a subscriber component was registered, a connection between the corresponding publisher and subscriber component needs to be established.*

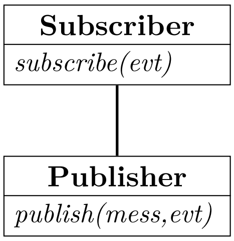


Figure 1.2: Specification of the Publisher-Subscriber pattern as it is usually found in literature.

Example 3 (The Blackboard Pattern)**.** *Another, more complex, pattern found in literature, is the Blackboard pattern [TMD09, BMR+96, SG96] which is often employed for the design of systems solving logical equations.*

The Blackboard pattern aims to address the design problem *known as "collaborative problem*[1] *solving". Thereby, it is desired to design an architecture for a system which can solve a complex problem by breaking it down into simpler subproblems, which can be solved and assembled to a solution for the original problem. For example, solving a complex, logical equation (involving multiple operators), can be split into the problem of solving simpler sub-formulas and combining their solutions according to the involved logical operators.*

Figure 1.3 shows the diagram for the Blackboard pattern as it is usually found in literature. The pattern requires from an architecture existence of the following types of components: blackboards, knowledge sources, and an optional controller component. Thereby, a blackboard keeps the overall state towards solving the original problem and knowledge sources are able to solve specific subproblems. Amongst others, the pattern requires that knowledge sources communicate exclusively through the blackboard component: they either provide solutions to currently open subproblems (given that solutions for other subproblems are available), or they communicate their ability to solve open subproblems and require a set of other subproblems to be solved first. The controller component is optional and can be employed to improve the communication between blackboard and knowledge sources.

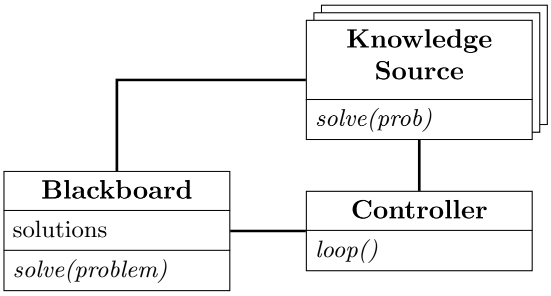


Figure 1.3: Specification of the Blackboard pattern as it is usually found in literature.

---

[1] Problem in this context is different from architectural design problem.

For the scope of this thesis, we define an ADP as follows:

Definition: *Architectural design pattern.*

An architectural design pattern consists of a set of *architectural constraints*, i.e., constraints about different aspects of an architecture, such as:

- The types of *data* exchanged by the components.
- The types of *components* involved in an architecture (including assumptions about its syntactic and semantic interface) as well as the existence of components of a certain type.
- *Activation* and *deactivation* of components of certain types.
- *Connections* between components of certain types.

An architectural design pattern usually comes with a set of invariants in terms of safety/liveness properties for an architecture implementing the pattern. In the following, we call such invariants *architectural guarantees* and usually they characterize *correct* solutions for the architectural design problem addressed by an ADP. Figure 1.4 summarizes the situation: An architecture which follows the constraints imposed by a pattern is assumed to satisfy the guarantees provided by the pattern.

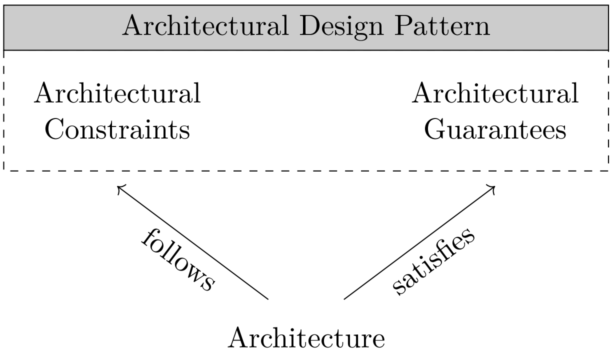


Figure 1.4: Architectures and ADPs.

In the following, we demonstrate the definition by means of the three example patterns introduced above[2].

Example 4 (The Singleton Pattern)**.** *Let us first consider the Singleton pattern introduced in Ex. 1 and reformulate it in terms of our new definition. The constraints imposed by a Singleton pattern usually concern two aspects of an architecture: the types of components as well as the activation and deactivation of components. Thus, we may formulate two corresponding types of architectural constraints:*

---

[2] We provide only informal specifications here. Corresponding formalizations are provided in Chap. 3 and Chap. 4.

- *A Singleton pattern usually requires the existence of one type of component: the singleton. However, it does not pose any constraints on the interface of a singleton component and as a consequence it does also not constrain its behavior.*

- *In addition, our version of the Singleton pattern requires that a component of type singleton is always active and only one component of type singleton is active at each point in time. Note that other versions of the Singleton pattern may require that* at most one *component of type singleton is active at each point in time. Moreover, in our version of the Singleton pattern, we also require that the active component of type singleton is unique over time, i.e., the component does not change over time. Also here, we could think of different versions of the pattern, in which, for example, the singleton component is allowed to change over time.*

If we reformulate the addressed design problem, we get the following architectural guarantee (in terms of a safety property) for an architecture implementing the singleton pattern:

> *A system implementing the Singleton pattern is guaranteed to have a* unique *component of type singleton which is active at each point in time.*

Example 5 (The Publisher-Subscriber Pattern)**.** *Let us now turn to the Publisher-Subscriber pattern and derive architectural constraints and architectural guarantees from the pattern's description provided in Ex. 2. In contrast to a Singleton pattern, a Publisher-Subscriber pattern usually constrains three aspects of an architecture: data types, component types, and connections between the ports of certain components. In the following, we provide corresponding architectural constraints:*

- *A Publisher-Subscriber pattern usually requires the existence of an abstract data type to represent subscriptions and un-subscriptions for certain events.*

- *In addition, a Publisher-Subscriber pattern requires two types of components: publisher and subscriber components. However, we do not require any assumptions about the behavior of these components.*

- *Finally, a Publisher-Subscriber pattern requires that, whenever a publisher component sends a message associated to an event for which a subscriber component is registered, the subscriber must be connected to the publisher, i.e. a channel between the corresponding ports of the publisher and subscriber component must be active in such a situation.*

The following guarantee may be derived from the pattern's addressed design problem:

> *A subscriber receives all the messages associated to an event for which it is subscribed.*

This time, the guarantee is given in terms of a liveness property for architectures implementing the pattern.

Example 6 (The Blackboard Pattern)**.** *Finally, let us derive some architectural constraints from the description of the Blackboard pattern presented in Ex. 3. Compared to the other examples, the Blackboard pattern constrains every aspect of an architecture: data types, component types, component activation, and connections between components.*

- *First of all, a Blackboard pattern requires the existence of data types for the problems to be solved and the corresponding solutions. It even requires the existence of a* well-founded *relation between problems and corresponding subproblems.*
- *Moreover, a Blackboard pattern requires the existence of two types of components: blackboards and knowledge sources. Thereby, it requires that blackboard components forward the current state towards solving the original problem, i.e., they are required to forward currently open subproblems as well as solutions for already solved subproblems. Knowledge source components, on the other hand, are required to solve a problem, whenever solutions for all the required subproblems are available. Moreover, they are also required to communicate subproblems for which they require solutions in order to solve a currently open subproblem.*
- *In order to guarantee that a Blackboard architecture indeed solves a given problem, the pattern requires that a blackboard component is unique and always activated. Moreover, the pattern requires that for every open subproblem, a knowledge source able to solve this problem is eventually activated.*
- *Finally, a Blackboard constrains also possible connections between blackboard and knowledge source components: whenever a knowledge source publishes a solution to a problem, or subproblems it requires to solve an open problem, the pattern requires the knowledge source to be connected to the blackboard component.*

A guarantee provided by the Blackboard pattern may be stated as follows:

> *An architecture is guaranteed to collaboratively solve a given problem, even if no knowledge source can solve the problem on its own.*

Again, it is a liveness property formalizing a correct solution to the pattern's addressed design problem.

1.2 Problem: Unverified Patterns

The main problem addressed with this thesis is that patterns found in current literature are usually not verified. Figure 1.5 depicts this situation: usually it is not clear whether the architectural constraints imposed by a pattern indeed lead to the corresponding guarantee. There are two possible consequences of this problem:

- The constraints may be too weak for the guarantee.
- The constraints may be unnecessarily strong for the provided guarantee.

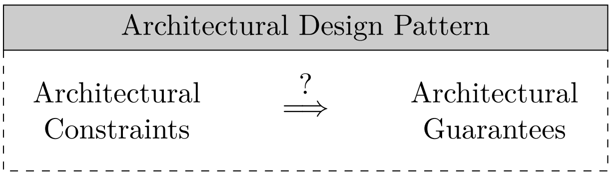


Figure 1.5: Problem: unverified patterns.

1.2.1 Missing Constraints

If the architectural constraints required by an ADP are too weak to ensure the claimed guarantees, the pattern may not *correctly* solve the addressed design problem. Important constraints might be missing and architectures implementing the pattern may not satisfy the pattern's guarantees. In order to understand why missing design constraints indeed constitute a problem, let us first look at how ADPs are usually used for the design of an architecture. The situation is shown in Fig. 1.6: When designing an architecture based on some requirements, ADPs are usually selected based on the problem they solve. The architecture is then designed according to the constraints imposed by the pattern. If the constraints, however, do not solve the problem, the corresponding architecture does not correctly solve the problem either, which might lead to a system which does not fulfill its requirements.

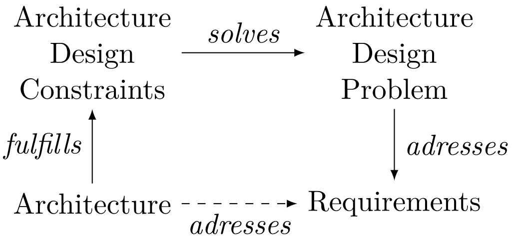


Figure 1.6: The use of ADPs for the design of architectures.

1.2.2 Unnecessary Constraints

Even if the constraints required by an ADP are strong enough to ensure a pattern's guarantee, not all constraints may be needed in order to ensure the pattern's guarantee. Thus, a pattern's specification may contain unnecessary constraints. While this problem is actually not as severe as the problem of missing constraints mentioned above, it does also have some undesired consequences: Since architectural design constraints restrict the design space for an architecture, unnecessary constraints exclude possible designs for an architecture. This becomes a problem if such an unnecessary constraint excludes an optimal design and requires an architect to select only a suboptimal architecture for the given requirements.

1.3 Approach

Over the last decades, several so-called architecture description languages (ADLs) emerged to support the formal specification and analysis of software architectures [GR91, LKA^{+}95, All97, GMW00, DVdHT01, FLV06, HF10]. Some of those even support the specification of dynamic aspects [MK96, ADG98, vOvdLKM00]. These techniques usually specify an architecture using some type of stat machine and the specification is then analyzed using model-checking techniques.Traditional approaches to address the problems identified above tried to apply these techniques, developed for the specification and verification of architectures, to ADPs. Kim and Garlan [KG06], for example, apply the Alloy [Jac02] model checker to automatically verify architectural styles specified in ACME [Gar03]. First approaches in this area come from some early attempts to formalize design patterns using UML [MCL04, SH04, ZA05]. A similar approach comes from Wong et al. [WSWS08] which applies Alloy for the verification of architecture patterns. Zhang et al. [ZLS^{+}12] applied model-checking techniques to verify architectural styles formulated in Wright#, an extension of Wright [All97]. More recently, Marmsoler and Degenhardt [MD17] apply the NuSMV model checker [CCGR00] to verify properties of design patterns and Goethel et al. [GJS17] model patterns for self-adaptive systems using CSP [Hoa78] and use the FDR3 model checker [GRABR14] to analyze them.

Specifications of ADPs, however, have some peculiarities, which limit the application of the above techniques for their specification and verification: (i) Pattern specifications are usually axiomatic, focusing on a minimal set of constraints (about component behavior or architecture configurations), in order to ensure its guarantee. For example, in a Singleton pattern, we do not care about the concrete implementation of component activation, as long as it is guaranteed that a component of type singleton is only activated if no other component of that type is already active. For a Publisher-Subscriber pattern, on the other hand, we are not interested in the concrete mechanism which implements communication between components, as long as it is guaranteed that a subscriber component is connected to a publisher, whenever latter sends out some message for which the former is currently subscribed. Or in a Blackboard pattern, we are not concerned with how a knowledge source solves a certain problem, as long as it is guaranteed that it solves it somehow. (ii) Moreover, the specification of patterns does not necessarily contain a fixed number of components. Rather it provides upper or lower bounds and sometimes the number might be even unbounded. For example, in a Publisher-Subscriber pattern, we do not know the exact number of subscriber components. Or in a Blackboard pattern, we do not know the exact number of knowledge source components.

In this thesis, we propose an approach based on *axiomatic* specification techniques and *interactive theorem proving*, to address the problems identified above.

Thereby, to the best of our knowledge, this is the first approach applying *interactive theorem proving* for the *verification of architectural design patterns.*

Interactive theorem proving supports verification at an axiomatic level. This allows for the verification of the axiomatic specifications inherent in ADPs. The additional effort which comes with interactive theorem proving (compared to automatic verification techniques, employed in traditional approaches), is justified by the impact of verification results at pattern level: *Each result obtained at the level of an ADP applies to every architecture which implements that pattern.* Thus, if we think about how many architectures implement a Singleton or a Publisher-Subscriber pattern, this should justify the additional effort induced by manual verification approaches.

1.4 Contributions

Perhaps the major outcome of this thesis is a methodology for the specification and verification of ADPs. In the following, we briefly introduce the proposed methodology and summarize the major contributions of this thesis.

1.4.1 FACTum: Focus on Architectural Design Constraints

Figure 1.7 depicts a general overview of the FACTUM methodology. Thereby, verifying a pattern proceeds in three main phases:

- First, the pattern is formally specified. Therefore, one needs to specify the architectural constraints imposed by the pattern as well as the architectural guarantees derived from the pattern's addressed architectural design problem. The stick figure indicates that these activities are executed manually, by the person analyzing the ADP. The outcome of this activity is a formal specification of the constraints imposed by the pattern and the corresponding guarantees.
- In the next phase, an Isabelle/HOL theory is created from the specification of the pattern and theorems are created from the corresponding guarantees. As indicated by the gear-wheel, creating the theory as well as the corresponding theorems are fully automated activities.
- In the last phase, the pattern is finally verified by proving the theorem from the specification. As indicated by the symbols, this is a semi-automatic activity: a user writes the proof using Isabelle/HOLs structured proof language Isar. The soundness of the different steps is then automatically checked by Isabelle.

Note that the activities depicted in Fig. 1.7 are annotated with labeled stars. They indicate where the particular contributions of this thesis are located in the overall methodology. In total, the thesis provides 4 major contributions (3 of which contribute to the activities of the methodology and one additional contribution comes from the evaluation of the methodology):

C1 We provide an axiomatic *specification framework* which can be used to formally specify ADPs as well as its guarantees and implement it in Eclipse/EMF.

C2 We provide an *algorithm* to map a FACTum specification to a corresponding Isabelle/HOL theory, show its soundness, and implement it in Eclipse/EMF.

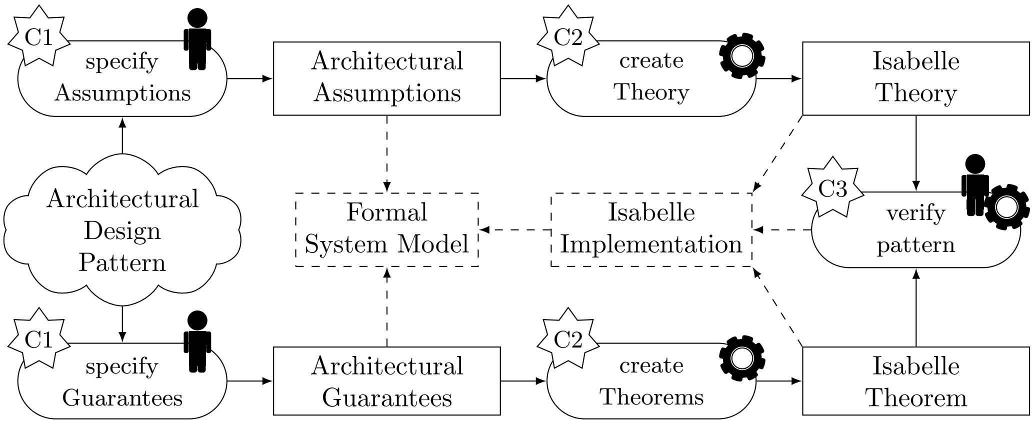


Figure 1.7: The FACTUM methodology for interactive pattern verification.

C3 We provide a *calculus* to reason about pattern specifications, show its soundness and relative completeness, and implement it in Isabelle/HOL.

C4 We demonstrate our approach in terms of three running examples and evaluate it in terms of a larger case study.

In the following, we discuss each of these contributions in more detail.

1.4.2 C1: Axiomatic Pattern Specification Framework

To support the specification activity, we developed a framework for the axiomatic specification of ADPs. The framework consists of several languages to specify the different types of constraints imposed by a pattern as well as the architectural guarantees given by the pattern. The framework also comes with a denotational semantics for every language, which allows for a *formal* specification of an ADP. To support the specification, the basic, textual specification languages are complemented with a graphical extension which allows to easily express common activation and connection constraints. To support the hierarchical nature inherent in ADPs, the framework also supports hierarchical specifications: a pattern specification can instantiate another pattern specification by interpreting the corresponding component types.

The framework was also implemented as a tool in Eclipse/EMF, which supports the specification of ADPs by combining graphical and textual elements. Thereby, comprehensive type checking supports the user in the specification of architectural constraints as well as architectural guarantees.

1.4.3 C2: Theory Generation Algorithm

We also provide an algorithm to map a FACTum specification to a corresponding Isabelle/HOL theory. Therefore, we first implemented our formal model of architectures as an Isabelle/HOL theory. Then, we developed an algorithm to map a (hierarchical)

FACTum specification to a corresponding Isabelle/HOL theory. Thereby, the generated theory extends the theory given by the model implementation. The algorithm is shown to be sound and implemented in Eclipse/EMF. Thus, a user of the specification tool (presented as an outcome of C2), can automatically generate Isabelle theories and corresponding theorems from its specification of an ADP. Thereby, the original meaning of the specification is guaranteed to be preserved by the generated Isabelle/HOL theory.

1.4.4 C3: Verification Framework

To support the verification of a given pattern specification, we also provide a calculus which formalizes reasoning about behavior specifications of component types. The calculus comes in a natural deduction style and provides introduction and elimination rules for all the operators involved in a FACTum specification. The calculus is shown to be sound and it is implemented in Isabelle/HOL to support the verification of pattern specifications.

1.4.5 C4: Running Examples and Case Study

Throughout the thesis we shall use three running examples to demonstrate our concepts and ideas: the Singleton, the Publisher-Subscriber, and the Blackboard pattern. Thereby, we also provide verification results for these patterns. To evaluate our approach in more depth, however, we provide a larger case study in the area of blockchain architectures. Thereby, we specify a pattern for blockchain architectures based on the proof of work consensus algorithm and verify an important property for blockchain architectures: that entries of a blockchain are indeed resistant to modifications from untrusted nodes.

1.5 Related Work

As mentioned in the introduction, architecture description languages (ADLs) have been an active area of research and many approaches emerged to support the formal specification of architectures. Famous examples are Weaves [GR91], Rapide [LKA$^+$95], Wright [All97], AADL [FLV06], ACME [GMW00], xADL [DVdHT01], and Autofocus [HF10]. Over the last years, specification and verification of dynamic aspects were of particular interest. Table 1.1 provides an overview of some representative examples in this area. For each of them, we list the underlying formalism as well as its support for dynamic aspects. To this end, we distinguish between *Separate* and *Combined* approaches: While the former separate the specification of behavior from the specification of architectural aspects, the latter combine the two.

Similar to most of the approaches shown in Tab. 1.1, we also separate the specification of behavioral aspects from that of structural aspects. The difference comes, however, in the verification: While most of these approaches focus on operational specifications and automatic verification techniques, with our work we aim towards axiomatic specifications and interactive theorem proving.

approach	dynamics	specification
Darwin [MK96]	S & C	Π-Calculus [Mil99]
Wright [All97, ADG98]	S	CSP[Hoa78]
COMMUNITY [WLF01, WF02]	S	Unity [Cha89]/SM
Aguirre and Maibaum [AM02b, AM02a]	S	TL [MP92]
Π-ADL [Oqu04]	S	Π-Calculus [Mil99]
Reo [Arb04, BSAR06, KMLA11]	S	circuits
Castro et. al [CAPM10]	C	Category Theory
Canal et al. [CCS12]	S	LTS
Archery [SBR12, SMB15]	S	ACP [BK86]

Table 1.1: Overview of dynamic ADLs and Coordination Languages.

1.5.1 Axiomatic approaches

Even though they were not invented for the purpose of pattern verification, there exist some approaches which focus on the axiomatic specification of architectures, in general. One of the first attempts in this direction is done by Bergner [Ber96]. The author proposes an approach to specify component networks and verify whether a given (runtime) component network satisfies its specification. The approach is implemented in Spectrum [BFGea93], a functional programming language which allows for axiomatic specifications of functions. Another approach comes from Fensel and Schnogge [FS97], which apply the KIV interactive theorem prover [Rei95] to verify concrete architectures in the area of knowledge-based systems. Another example is Spichkova [Spi07], which provides a mapping from a FOCUS [BS01] specification to a corresponding Isabelle/HOL [NPW02] theory. More recently, some attempts were made to apply interactive theorem proving to the verification of architectural connectors. Li and Sun [LS13], for example, apply the Coq proof assistant [BC13] to verify connectors specified in Reo [Arb04]. These approaches, both, apply interactive theorem proving to verify architectures.

While also these approaches indeed support axiomatic specifications and verification of architectures, there are two major differences to our work.

1.5.1.1 Scope of Application

The first difference lies in the scope of the application: The approaches discussed so far apply axiomatic verification at the level of concrete architectures which might be too expensive, in general. Thus, we argue, that application of axiomatic verification should be restricted to architecture patterns, rather than concrete architectures. Thus, the expenses would pay off since each result at the level of pattern applies for each concrete architecture implementing the pattern. Just think about how many patterns employ a Singleton or Publisher-Subscriber pattern.

1.5.1.2 Dynamic Aspects

Another difference lies in the expressiveness of the specification languages: The above approaches mainly focus on the specification of static architectures. However, as shown at the beginning, some commonly used patterns require also the specification of dynamic aspects, such as:

Component Activation Some patterns, such as the Singleton pattern or the Blackboard pattern, require to specify activation and deactivation of components.

Reconfiguration Other patterns, such as the Publisher-Subscriber pattern or the Blackboard pattern, require means to specify architecture reconfiguration, i.e., means to specify activation and deactivation of connections between component ports.

There are two exceptions to this which support axiomatic specifications of even dynamic architectures. They are closely related to our approach and thus deserve a detailed analysis.

1.5.2 Componentware

One example which uses a model similar to ours to formalize UML models is Componentware [Rau01]. Here, the author provides means to specify architectural constraints in an UML-like notation [RJB04]. There are, however, some differences to our specification approach which makes the specification of patterns difficult:

- The main restriction is probably the use of OCL for the specification of the behavior of components. As our examples later on show, specifying component types involves the specification of temporal aspects which is not supported by OCL and consequently not possible in their approach.
- Another restriction is the limited possibility for analysis of specifications. The approach does not provide any calculus to analyze an axiomatic specification.

Nevertheless, the approach provides many interesting insights into axiomatic specification of dynamic architectures and indeed the underlying model of dynamic architectures is similar to the model used in the approach presented with this paper.

1.5.3 CommUnity

Another, closely related approach is the one of Aguirre and Maibaum [AM02b, AM02a]. The approach builds on top of CommUnity [FM97] and provides many interesting ideas found in our approach as well:

- It allows for the specification of abstract data types used by the components.
- It allows for the specification of classes which are similar to our notion of component types.

- Instance of classes as well as reconfigurations can then be specified using so-called subsystems which are similar to our notion of architecture constraint specification.

There are, however, some subtle differences to our approach which limits its application for the specification of patterns:

- Instantiation of components as well as architecture reconfiguration must be explicitly triggered from outside. However, as shown later on, for some patterns there is no such well-defined trigger, i.e., the trigger may change in different implementations of the pattern.
- Their approach does not support the notion of parametric interfaces which turn out to be useful when it comes to the specification of patterns.
- The approach does not support hierarchical specifications which are very important when it comes to the specification of patterns since they are usually specified on top of each other.
- The approach is based on an action-synchronous model of systems. Some patterns are, however, better described using a time-synchronous model of communication.

Another key constraint of this approach is the lack of analysis methods to reason about such specifications.

1.6 Outline

This thesis is structured into five main parts: An introductory part, containing this introduction and our formal model of architectures; a specification part, describing techniques for the specification of ADPs over the model and demonstrating them by means of our three running examples; a verification part describing our verification framework and its formalization in Isabelle/HOL and demonstrating it in terms of our running examples; an evaluation part in which we present the outcome of evaluating the approach by means of our running examples and a larger case study from the domain of blockchain architectures; a concluding part containing an outlook and suggestions for future work as well as several appendices.

2 A Model of Dynamic Architectures

Since ADPs often involve the specification of dynamic aspects, our approach relies on a model for dynamic architectures. The model is based on Broy's FOCUS theory [Bro10] and its dynamic extension [Bro14]. It assumes a set of ports and messages to be given, together with a corresponding type function which assigns a set of messages to each port. Then, it defines the notion of an *interface*, which consists of a set of input and output ports. It then introduces the central notion of *component type*, which extends an interface with a set of so-called *component parameters* (formally represented as a set of ports and associated messages) and behavior. Behavior of component types is modeled in terms of sets of so-called *behavior traces*, i.e., causal [Bro10] streams of valuations of the ports of the component's interface. Besides component types, the model introduces the concept of an *architecture trace*: an infinite sequence of so-called *architecture snapshots*, which consists of a set of active components (belonging to some type), connections between their ports, and a valuation of the ports of active components. An *architecture specification* is then defined as a set of architecture traces which does not restrict the behavior of components. Finally, the notion of *behavior projection* is introduced to extract the behavior of a certain component out of a given architecture trace. Behavior projection is then used to define composition of component types under a given architecture specification: the result of composing a set of component types with an architecture specification is defined to consist of all the architecture traces from the architecture specification for which the projection to any component leads to a behavior trace allowed by the component's type. The model is formalized in Isabelle/HOL and available as the entry `DynamicArchitectures` [Mar17a] in the Archive of Formal Proofs. In order to deal with infinite streams, the formalization is based on Lochbihler's theory of coinductive (lazy) lists [Loc10]. Thereby, architecture traces are formalized in terms of lazy lists and behavior projection is formalized using the lazy filter operation.

In the following, we first introduce the basic concepts of messages, ports, and interfaces. Then, we describe two key concepts of our model: component types and architecture specifications. Thereby, we describe also the notion of behavior projection and composition. We conclude with a brief summary of the introduced concepts and their interrelationships.

2.1 Messages and Ports

In our model, components communicate to each other by exchanging messages over ports. Thus, we assume the existence of set $\mathcal{M}$, containing all *messages*, and set $\mathcal{P}$, containing all *ports*, respectively. Moreover, we postulate the existence of a type function

$$\mathcal{T}\colon \mathcal{P} \to \wp(\mathcal{M}) \tag{2.1}$$

which assigns a set of messages to each port.

2.2 Port Valuations

Ports are means to exchange messages between a component and its environment. This is achieved through the notion of port valuation. Roughly speaking, a valuation for a set of ports is an assignment of messages to each port.

Definition 1 (Port valuation)**.** *For a set of ports $P \subseteq \mathcal{P}$, we denote with $\overline{P}$ the set of all possible, type-compatible* port valuations*, formally:*

$$\overline{P} \quad \overset{\mathit{def}}{=} \quad \Big\{\mu \in (P \to \wp(\mathcal{M})) \mid \forall p \in P\colon \mu(p) \subseteq \mathcal{T}(p)\Big\}$$

Moreover, we denote by $[p_1, p_2, \ldots \mapsto M_1, M_2, \ldots]$ the valuation of ports $p_1, p_2, \ldots$ with sets M_1, M_2, …, respectively. For singleton sets we shall sometimes omit the set parentheses and simply write $[p_1, p_2, \ldots \mapsto m_1, m_2, \ldots]$.

In our model, ports may be valuated by *sets* of messages, meaning that a component can send/receive a set of messages via each of its ports at each point in time. A component may also send no message at all, in which case the corresponding port is valuated by the empty set.

2.3 Interfaces

The ports which a component may use to send and receive messages are grouped into so-called interfaces.

Definition 2 (Interface)**.** *An* interface *is a pair (CI, CO), consisting of* disjoint *sets of* input ports *$CI \subseteq \mathcal{P}$ and* output ports *$CO \subseteq \mathcal{P}$. The set of all interfaces is denoted by $IF_{\mathcal{P}}$. For an interface $if = (CI, CO)$, we denote by*

- $\mathsf{in}(if) \overset{\mathit{def}}{=} CI$ *the set of input ports,*
- $\mathsf{out}(if) \overset{\mathit{def}}{=} CO$ *the set of output ports, and*
- $\mathsf{port}(if) \overset{\mathit{def}}{=} CI \cup CO$ *the set of all interface ports.*

2.4 Component Types

An important concept of our model are component types, i.e., interfaces with associated component behavior.

2.4.1 Streams

In the following, we shall make use of finite as well as infinite *streams* [BS01]. Thereby, we denote with $(E)^*$ the set of all finite streams over elements of a given set E, by $(E)^\infty$ the set of all infinite streams over E, and by $(E)^\omega$ the set of all finite and infinite streams over E. The n-th element of a stream s is denoted with $s(n)$ and the first element is $s(0)$. Moreover, we shall use the following conventions for streams:

- With $\langle\rangle$ we denote the empty stream.
- With $e\&s$ we denote the stream resulting from appending stream s to element e.
- With $s \frown s'$ we denote the concatenation of stream s with stream s'.
- With $rg(s)$ we denote the set of all elements of a given stream s.
- With $\#s \in \mathbb{N}_\infty$ we denote the length of s.
- We use $s{\downarrow_n}$ to extract the first n (excluding the n-th) elements of a stream. Thereby $s{\downarrow_0} \overset{\text{def}}{=} \langle\rangle$.
- With $s' \sqsubseteq s$, we denote that s' is a prefix of s.
- We may also lift the restriction operator from functions to streams of functions and use $s|_D$ to denote a stream of length $\#s$, with $s|_D(n) \overset{\text{def}}{=} s(n)|_D$ for every time point $n < \#(s|_D)$.

2.4.2 Component Type

Essentially, a component type is an interface with associated behavior. The behavior is given in terms of so-called behavior traces, streams of valuations of the corresponding interface ports.

Definition 3 (Component type)**.** *A* component type *is a pair* $(\mathit{if}, \mathit{bhv})$*, consisting of*

- *an interface* $\mathit{if} \in \mathit{IF}_{\mathcal{P}}$*,*
- *and a* non-empty *set of so-called* behavior traces $\mathit{bhv} \subseteq (\overline{\mathsf{port}(\mathit{if})})^\infty$*, such that:*
  - *the behavior of a component is* input-complete*, i.e., for all* $t \in \mathit{bhv}$ *and all time points* $n \in \mathbb{N}$*:*

$$\forall \mu \in \overline{\mathsf{in}(\mathit{if})}\ \exists t' \in \mathit{bhv}\colon t'{\downarrow_n} = t{\downarrow_n} \ \land\ t'(n)|_{\mathsf{in}(\mathit{if})} = \mu \tag{2.2}$$

  - *the behavior of a component is* causal*, i.e., for all* $t, t' \in \mathit{bhv}$ *and all time points* $n \in \mathbb{N}$*, we have:*

$$(t{\downarrow_{n-1}})|_{\mathsf{in}(\mathit{if})} = (t'{\downarrow_{n-1}})|_{\mathsf{in}(\mathit{if})} \tag{2.3}$$

$$\implies \exists t'' \in \mathit{bhv}\colon (t''{\downarrow_n})\ |_{\mathsf{in}(\mathit{if})} = (t'{\downarrow_n})|_{\mathsf{in}(\mathit{if})} \land t''{\downarrow_n} = t{\downarrow_n} \tag{2.4}$$

> *Actually, we could relax the second condition to require only equality of valuations for output ports. However, due to the first condition and Eq.* (2.3)*, this is equal to requiring equality for the valuations of all the ports and thus the complete valuation.*

We shall use the same notation as introduced in Def. 2 to denote input, output, and all interface ports for component types. Moreover, for a component type $ct = (\mathit{if}, \mathit{bhv})$*, we denote by*

$$\mathsf{bhv}(ct) \overset{def}{=} \mathit{bhv} \tag{2.5}$$

the behavior of that type.

Example 7 (Component type)**.** *Assuming* $\mathcal{P}$ *contains ports* i_0, i_1, o_0, o_1*, Fig. 2.1 shows a conceptual representation of a component type* $(\mathit{if}, \mathit{bhv})$*, consisting of:*

- *Interface* $\mathit{if} = (\mathit{CI}, \mathit{CO})$*, with*
  - *input ports* $\mathit{CI} = \{i_0, i_1\}$*, and*
  - *output ports* $\mathit{CO} = \{o_0, o_1\}$*.*
- *Behavior* $\mathit{bhv} = \{\epsilon, \sigma, \nu, \mu, \omega, \delta, \eta\}$ *which is assumed to be input complete and causal.*

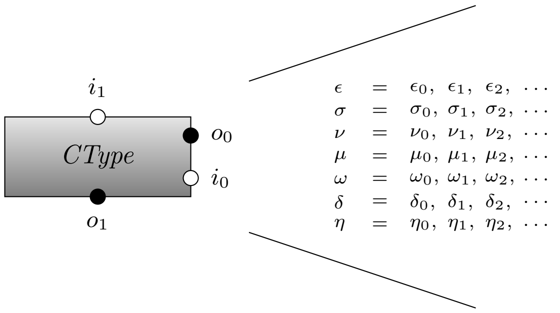


Figure 2.1: Conceptual representation of a component type with behavior $\mathit{bhv} = \{\epsilon, \sigma, \nu, \mu, \omega, \delta, \eta\}$.

2.4.3 Parametrized Component Types

Sometimes, it is convenient to specify and reason about groups of related components of a certain type. Consider, for example, the Blackboard pattern in which a set of knowledge source components work together to collaboratively solve an overall problem. Thereby, certain knowledge sources are only able to solve certain problems, which is why they can be classified into different groups, depending on the problem they can solve. In such cases, it is useful to extend the notion of component type by adding a set of parameters, used to group related components based on the value of the parameter.

Definition 4 (Parametrized component type)**.** *A* parametrized component type *is a triple* (ct, CP, ν)*, consisting of*

- *a component type ct,*
- *so-called* component parameters *$CP \subseteq \mathcal{P}$ which are required to be disjoint from the component type's input and output ports,*
- *a valuation of the component parameters $\nu \in \overline{CP}$,*

The set of all possible parametrized component types over a set of interfaces $\mathcal{I}$ is denoted $CT_{\mathcal{I}}$. We shall use the same notation as introduced for component types in Def. 3 to denote the ports and behavior for parametrized component types. Moreover, for a parametrized component type (ct, CP, ν), we denote by

- $\mathsf{par}(ct) \overset{def}{=} CP$ *its component parameters and*
- $\mathsf{val}(ct) \overset{def}{=} \nu$ *the valuation of component parameters.*

Example 8 (Parametrized component type)**.** *Figure 2.2 shows a conceptual representation of a parametrized component type (ct, CP, ν), extending the component type described in Ex. 7 by a component parameter $CP = \{p\}$ valuated with a set of messages $\nu(p) = M$.*

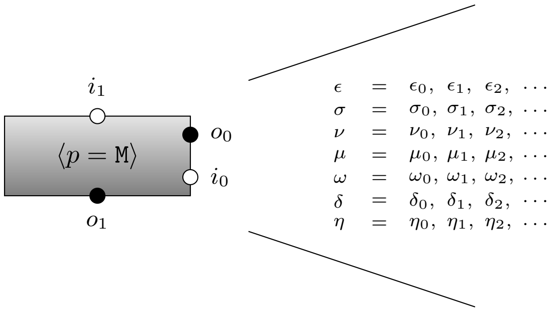


Figure 2.2: Conceptual representation of a parametrized component type with component parameter p valuated with a set of messages M.

Formally, component parameters are just normal ports, valuated with some messages. However, they have a special meaning in a specification, which distinguishes them from input and output ports. First, the valuation of a component parameter is *bound to a component* and does not change its value over time (compared to input and output ports which may change their valuation at each point in time). Second, for each parametrized component type, we require the existence of at least one component, for each possible valuation of the parameter (respecting its type). Note that existence does not require activation of that component, however it is required to ensure soundness of specifications involving parameterized component variables. Such variables are interpreted only by components with a corresponding parameter valuation. However, if for a certain parameter valuation no such component exists, the semantics of the specification is not well-defined. More on details on parametrized component variables can be found in Chap. 3.

2.5 Architecture Specifications

Component types specify the interface and the allowed behavior for components. However, they do not say anything about the activation and deactivation of components or their interconnections. Thus, in the following, we introduce the concept of an architecture specification to address these aspects. We conclude the section with the definition of a composition operator which allows to combine component types with an architecture specification.

2.5.1 Components

Component types can be instantiated to obtain components of that type. We shall use the same notation as introduced for parametrized component types in Def. 4, to access ports, valuation of component parameters, and behavior assigned to a component. Note, however, that instantiating a component leads to the notion of *component port*, which is a port combined with the corresponding component identifier. Thus, for a family of components $(\mathcal{C}_{ct})_{ct \in \mathcal{CT}}$ over a set of parametrized component types $\mathcal{CT} \subseteq CT_{\mathcal{I}}$, we denote by:

- $\mathsf{in}(\mathcal{C}) \overset{\text{def}}{=} \bigcup_{c \in \mathcal{C}} (\{c\} \times \mathsf{in}(c))$, the set of *component input ports*,
- $\mathsf{out}(\mathcal{C}) \overset{\text{def}}{=} \bigcup_{c \in \mathcal{C}} (\{c\} \times \mathsf{out}(c))$, the set of *component output ports*,
- $\mathsf{port}(\mathcal{C}) \overset{\text{def}}{=} \mathsf{in}(\mathcal{C}) \cup \mathsf{out}(\mathcal{C})$, the set of all *component ports*.

Moreover, we may lift the typing function (introduced for ports at the beginning of the chapter), to corresponding component ports:

$$\mathcal{T}((c,p)) \overset{\text{def}}{=} \mathcal{T}(p) \ .$$

Finally, we can generalize our notion of port valuation (Def. 1) for *component ports* $CP \subseteq \mathcal{C} \times \mathcal{P}$ to so-called *component port valuations*:

$$\overline{CP} \overset{\text{def}}{=} \Big\{\mu \in (CP \to \wp(\mathcal{M})) \mid \forall cp \in CP \colon \mu(cp) \subseteq \mathcal{T}(cp)\Big\}$$

To better distinguish between ports and component ports, in the following, we shall use p, q, pi, po, … to denote ports and cp, cq, ci, co, … to denote component ports.

2.5.2 Architecture Snapshots

An architecture is modeled as a sequence of snapshots of its state during execution. To this end, in the following, we introduce the notion of architecture snapshot. Such a snapshot consists of snapshots of currently active components, i.e., interfaces with its ports valuated with messages, and connections between the ports of these components. Message exchange between components requires the valuation of connected ports to be equal.

Definition 5 (Architecture snapshot). *An* architecture snapshot *is a triple* (C', N, μ)*, consisting of:*

- *a set of components* $C' \subseteq \mathcal{C}$*,*
- *a connection* $N \colon \mathsf{in}(C') \to \wp(\mathsf{out}(C'))$*, such that*

$$\forall ci \in \mathsf{in}(C') \colon \bigcup_{co \in N(ci)} \mathcal{T}(co) \subseteq \mathcal{T}(ci) \tag{2.6}$$

- *a component port valuation* $\mu \in \overline{\mathsf{port}(C')}$ *.*

We require connected ports to be consistent in their valuation, i.e., if a component provides messages at its output port, these messages are transferred to the corresponding, connected input ports:

$$\forall ci \in \mathsf{in}(C') \colon N(ci) \neq \emptyset \implies \mu(ci) = \bigcup_{co \in N(ci)} \mu(co) \tag{2.7}$$

Note that Eq. (2.6) *guarantees that Eq.* (2.7) *does not violate type restrictions. The set of all possible architecture snapshots is denoted by* $AS_{\mathcal{T}}^{\mathcal{C}}$*.*

For an architecture snapshot $as = (C', N, \mu) \in AS_{\mathcal{T}}^{\mathcal{C}}$*, we denote by*

- $CMP_{as} \overset{def}{=} C'$ *the set of active components and with* $\wr c \wr_{as} \overset{def}{\iff} c \in C'$*, that a component* $c \in \mathcal{C}$ *is* active *in as,*
- $CN_{as} \overset{def}{=} N$*, its connection, and*
- $val_{as} \overset{def}{=} \mu$*, the port valuation.*

Moreover, given a component $c \in C'$*, we denote by*

$$\mathsf{cmp}_{as}^{c} \in \overline{\mathsf{port}(\{c\})} \overset{def}{=} \Big(\lambda cp \in \mathsf{port}(\{c\}) \colon \mu(cp)\Big) \tag{2.8}$$

the valuation of the component's ports.

Note that cmp_{as}^{c} is well-defined iff $\wr c \wr_{as}$.

Moreover, note that connection N is modeled as a set-valued function from component input ports to component output ports, meaning that:

1. input/output ports can be connected to several output/input ports, respectively[1], and
2. not every input/output port needs to be connected to an output/input port (in which case the connection returns the empty set).

---

[1] As indicated by Eq. 2.7, if multiple output ports are connected to one input port, the corresponding input port is valuated with the union of messages from all connected output ports.

Thus, ports of an architecture snapshot can be classified as either open or connected, depending on whether they are connected to any other ports or not. Ports which are not connected to any other port are called open architecture ports.

Definition 6 (Open architecture port)**.** *For an architecture snapshot* $as = (C', N, \mu) \in AS_{\mathcal{T}}^{\mathcal{C}}$*, we denote by:*

- $\mathsf{oin}(as) \overset{def}{=} \{ci \in \mathsf{in}(C') \mid N(ci) = \emptyset\}$*, the set of* open input ports*,*
- $\mathsf{oout}(as) \overset{def}{=} \{co \in \mathsf{out}(C') \mid \nexists ci \in \mathsf{in}(C')\colon co \in N(ci)\}$*, the set of* open output ports*,*
- $\mathsf{oport}(as) \overset{def}{=} \mathsf{oin}(as) \cup \mathsf{oout}(as)$*, the set of all* open architecture ports*.*

On the other hand, ports which are connected to other ports are called connected architecture ports.

Definition 7 (Connected architecture port)**.** *For an architecture snapshot* $as = (C', N, \mu) \in AS_{\mathcal{T}}^{\mathcal{C}}$*, we denote by:*

- $\mathsf{cin}(as) \overset{def}{=} \{ci \in \mathsf{in}(C') \mid N(ci) \neq \emptyset\}$*, the set of* connected input ports*,*
- $\mathsf{cout}(as) \overset{def}{=} \{co \in \mathsf{out}(C') \mid \exists ci \in \mathsf{in}(C')\colon co \in N(ci)\}$*, the set of* connected output ports*,*
- $\mathsf{cport}(as) \overset{def}{=} \mathsf{cin}(as) \cup \mathsf{cout}(as)$*, the set of all* connected architecture ports*.*

Note that for an architecture snapshot $as = (C', N, \mu)$,

$$\mathsf{oin}(as) \cup \mathsf{cin}(as) = \mathsf{in}(C') \quad \text{and} \quad \mathsf{oout}(as) \cup \mathsf{cout}(as) = \mathsf{out}(C') \ .$$

Moreover, note that by Eq. (2.7), the valuation of an input port connected to many output ports is defined to be the *union* of all the valuations of the corresponding, connected output ports.

Example 9 (Architecture snapshot)**.** *Figure 2.3 shows a conceptual representation of an architecture snapshot* (C', N, μ)*, consisting of:*

- *active components* $C' = \{c_1, c_2, c_3\}$ *with corresponding component types (*c_3*, for example, is of a type as described in Ex. 7);*
- *connection* N*, defined as follows:*
  - $N((c_2, i_1)) = \{(c_1, o_1)\}$*,*
  - $N((c_3, i_1)) = \{(c_1, o_2)\}$*,*
  - $N((c_2, i_2)) = \{(c_3, o_1)\}$*, and*
  - $N((c_1, i_0)) = N((c_2, i_0)) = N((c_3, i_0)) = \emptyset$*; and*
- *component port valuation* $[(c_1, o_0), (c_2, i_1), (c_3, o_1), \dots \mapsto \mathtt{M}_3, \mathtt{M}_5, \mathtt{M}_3, \dots]$.

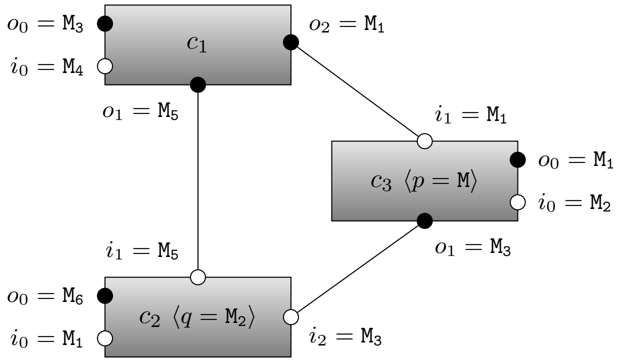


Figure 2.3: Architecture snapshot consisting of three components c_1, c_2, and c_3; a connection between ports (c_2, i_1) and (c_1, o_1), (c_2, i_2) and (c_3, o_1), and (c_3, i_1) and (c_1, o_2); and valuations of the component parameters and ports.

2.5.3 Architecture Traces

An *architecture trace* consists of a series of snapshots of an architecture during system execution. Thus, an architecture trace is modeled as a stream of architecture snapshots at certain points in time.

Definition 8 (Architecture trace)**.** *An* architecture trace *is an infinite stream* $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^\infty$*. For an architecture trace* $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^\infty$ *and a component* $c \in \mathcal{C}$*, we denote with*[2]

- *$last(c, t)$, the* greatest $i \in \mathbb{N}$*, such that* $\wr c \wr_{t(i)}$*,*
- $c \overset{n}{\Lleftarrow} t$*, the last time point less or equal to* n *at which* c *was* not *active in* t*, i.e., the* least $n' \in \mathbb{N}$*, such that* $n' = n \vee \big(n' < n \wedge \nexists n' \leq \underset{\cdot}{k} < n \colon \wr c \wr_{t(k)}\big)$*,*
- $c \overset{n}{\leftarrow} t$*, the latest activation of component* c *(strictly) before* n*, and*
- $c \overset{n}{\rightarrow} t$*, the next point in time (after* n*) at which* c *is active in* t*.*

Note that $c \overset{n}{\Lleftarrow} t$ is always well-defined, while $c \overset{n}{\leftarrow} t$ and $c \overset{n}{\rightarrow} t$ are only well-defined iff there exists at least one activation of component c in the past (a point in time strictly less than n) or in the future (a point in time greater or equal to n), respectively. *last*(c, t), on the other hand, is well-defined iff i) component c is activated at least once in t: $\exists i \in \mathbb{N} \colon \wr c \wr_{t(i)}$ and ii) component c is not activated infinitely often, i.e., $\exists n \in \mathbb{N} \colon \forall n' \geq n \colon \neg \wr c \wr_{t(n')}$.

Example 10 (Architecture trace)**.** *Figure 2.4 shows an architecture trace* $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^\infty$ *with corresponding architecture snapshots* $t(0) = k_0$*,* $t(1) = k_1$*, and* $t(2) = k_2$*. Architecture snapshot* k_0*, for example, is described in Ex. 9.*

---

[2] From now on, we shall sometimes use a dot for variables after a quantifier to highlight the variable bound by the corresponding quantifier.

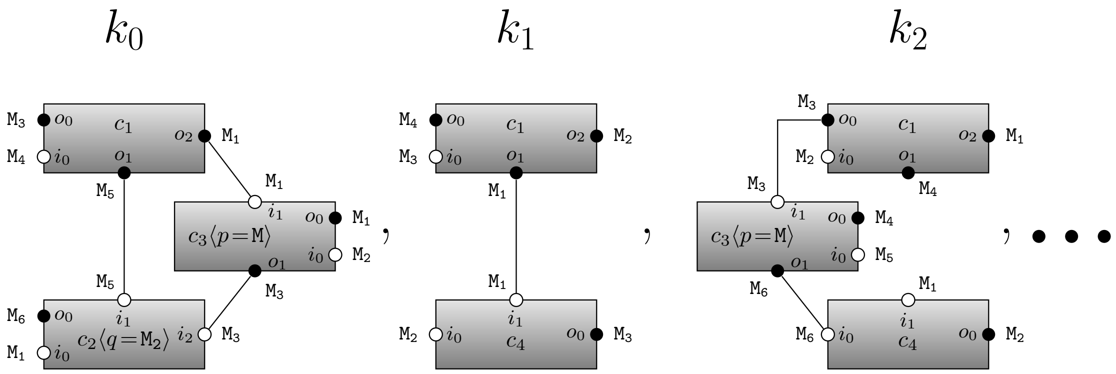


Figure 2.4: The first three architecture snapshots of an architecture trace.

Figure 2.5 lists some properties derived for the operators introduced for architecture traces in Def. 8. As indicated by the small Isabelle logo on the top right, these properties are all mechanically verified in our formalization of the model (App. D.2).

Properties of component activation

$$\begin{aligned}
\wr c\wr_{t(last(c,t))} &\ [\text{if } \exists i\colon \wr c\wr_{t(i)} \text{ and } \exists n\colon \forall n' > n\colon \neg\wr c\wr_{t(n')}]\\
\nexists n > last(c,t)\colon \wr c\wr_{t(n)} &\ [\text{if } \exists n'\colon \forall n'' > n'\colon \neg\wr c\wr_{t(n'')}]\\
c \overset{0}{\Leftarrow} t = 0 &\\
\wr c\wr_{t(n-1)} \implies c \overset{n}{\Leftarrow} t = n &\ [\text{if } n \geq 1]\\
\forall c \overset{n}{\Leftarrow} t \leq \underset{\cdot}{n}' < n\colon \neg\wr c\wr_{t(n')} &\\
c \overset{n}{\Leftarrow} t \leq n &\\
c \overset{c \overset{n}{\leftarrow} t}{\rightarrow} t = c \overset{n}{\leftarrow} t &\ [\text{if } \exists i < n\colon \wr c\wr_{t(i)}]\\
c \overset{n}{\rightarrow} t > c \overset{n}{\leftarrow} t &\ [\text{if } \exists i \geq n\colon \wr c\wr_{t(i)} \text{ and } \exists i < n\colon \wr c\wr_{t(i)}]\\
c \overset{n}{\rightarrow} t \geq n &\ [\text{if } \exists i \geq n\colon \wr c\wr_{t(i)}]\\
\wr c\wr_{t(c \overset{n}{\rightarrow} t)} &\ [\text{if } \exists i \geq n\colon \wr c\wr_{t(i)}]\\
\nexists n \leq \underset{\cdot}{k} < c \overset{n}{\rightarrow} t\colon \wr c\wr_{t(k)} &\ [\text{if } \exists i \geq n\colon \wr c\wr_{t(i)}]\\
\wr c\wr_{t(n)} \implies c \overset{n}{\rightarrow} t = n &\\
c \overset{n}{\rightarrow} t \geq c \overset{n}{\Leftarrow} t &\ [\text{if } \exists i \geq n\colon \wr c\wr_{t(i)}]
\end{aligned}$$

Figure 2.5: Properties of component activation.

Behavior projection An important concept for our model is the notion of behavior projection. It is used to extract the behavior of a certain component out of a given architecture trace (Figure 2.6).

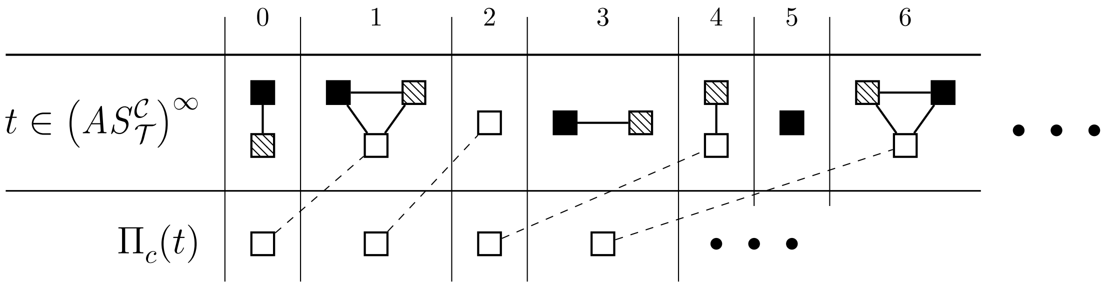


Figure 2.6: Conceptual representation of behavior projection.

In the following, we provide a *co-recursive* definition for behavior projection. This allows us to easily specify the operator also for infinite input traces by following a certain pattern in its specification. Then, we can use co-induction to reason about behavior projection[3].

Definition 9 (Behavior projection)**.** *Given an architecture trace* $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\omega}$*. The* behavior projection *to component* $c \in \mathcal{C}_{ct}$ *of type* $ct \in \mathcal{CT}$ *is denoted by* $\Pi_c(t) \in (\overline{\mathsf{port}(c)})^{\omega}$ *and defined by the following equations:*

$$\begin{aligned} & & \Pi_c(\langle\rangle) &= \langle\rangle & (2.9)\\ \wr c \wr_{as} &\Longrightarrow & \Pi_c(as \mathbin{\&} t) &= \mathsf{cmp}_{as}^{c} \mathbin{\&} \Pi_c(t) & (2.10)\\ \neg \wr c \wr_{as} &\Longrightarrow & \Pi_c(as \mathbin{\&} t) &= \Pi_c(t) & (2.11)\\ (\forall as \in rg(t) \colon \neg \wr c \wr_{as}) &\Longrightarrow & \Pi_c(t) &= \langle\rangle & (2.12) \end{aligned}$$

Note that the structure of the equations provided in Def. 9 ensures productivity [JR97] and hence they form a valid *co-recursive* definition. Thus, projection is indeed well-defined by those equations.

Example 11 (Behavior projection)**.** *Applying behavior projection of component* c_3 *to the architecture trace described in Ex. 10 results in a behavior trace starting as follows:*

$$[i_0, i_1, o_0, o_1 \mapsto \mathtt{M}_2, \mathtt{M}_1, \mathtt{M}_1, \mathtt{M}_3],\ [i_0, i_1, o_0, o_1 \mapsto \mathtt{M}_5, \mathtt{M}_3, \mathtt{M}_4, \mathtt{M}_6],\ \cdots$$

Figure 2.7 lists some characteristic properties of behavior projection.

2.5.4 Architecture Specifications

Finally, we can define our notion of architecture specification as a set of architecture traces with certain properties.

Definition 10 (Architecture specification)**.** *An* architecture specification *is a set* $\mathcal{A} \subseteq (AS_{\mathcal{T}}^{\mathcal{C}})^{\infty}$ *of architecture traces, such that:*

---

[3] Alternatively we could have used traditional recursion, show that behavior projection is continuous, and use fixpoint induction [GH05] to proof properties about it. The reason to choose co-recursion here is that it simplifies subsequent formalization in Isabelle/HOL.

Properties of behavior projection

$$
\begin{aligned}
\#\Pi_c(t) \quad &\leq \quad \#t \\
\Pi_c(t) \quad &= \quad \Pi_c(t\downarrow_n) \text{ [if } \forall n \leq n' \leq \#t \colon \neg \wr c \wr_{t(n')}\text{]} \\
\mathit{finite}(\Pi_c(t)) \quad &\iff \quad \exists n\ \forall n' > n \colon \neg \wr c \wr_{t(n')} \\
t \sqsubseteq t' \quad &\implies \quad \Pi_c(t) \sqsubseteq \Pi_c(t') \\
\Pi_c(t \frown t') \quad &= \quad \Pi_c(t) \frown \Pi_c(t') \text{ [if } \mathit{finite}(t)\text{]} \\
\Pi_c(t\downarrow_{n+1}) \quad &= \quad \Pi_c(t\downarrow_n) \text{ [if } n < \#t \text{ and } \neg \wr c \wr_{t(n)}\text{]} \\
\Pi_c(t\downarrow_{i+1}) \quad &= \quad \Pi_c(t\downarrow_i) \frown \mathsf{cmp}^c_{t(i)} \mathbin{\&} \langle\rangle \text{ [if } i < \#t \text{ and } \wr c \wr_{t(i)}\text{]}
\end{aligned}
$$

Figure 2.7: Properties of behavior projection.

- *it is input-complete, i.e., that for all* $t \in \mathcal{A}$ *and all time points* $n \in \mathbb{N}$*:*

$$
\begin{aligned}
\forall \mu \in \overline{\mathsf{oin}(t(n))}\ \exists t' \in \mathcal{A} \colon t'\downarrow_n = t\downarrow_n \ \wedge \\
\mathit{CMP}_{t(n)} = \mathit{CMP}_{t'(n)} \ \wedge \\
\mathit{CN}_{t(n)} = \mathit{CN}_{t'(n)} \ \wedge \\
\mathit{val}_{t'(n)}|_{\mathsf{oin}(t(n))} = \mu
\end{aligned}
\tag{2.13}
$$

- *it does not restrict the behavior of components, i.e., that for all* $t \in \mathcal{A}$ *and all time points* $n \in \mathbb{N}$*:*

$$
\begin{aligned}
\forall \mu \in \overline{\mathsf{out}(\mathit{CMP}_{t(n)})}\ \exists t' \in \mathcal{A} \colon t'\downarrow_n = t\downarrow_n \ \wedge \\
\mathit{CMP}_{t(n)} = \mathit{CMP}_{t'(n)} \ \wedge \\
\mathit{CN}_{t(n)} = \mathit{CN}_{t'(n)} \ \wedge \\
\mathit{val}_{t'(n)}|_{\mathsf{out}(\mathit{CMP}_{t(n)})} = \mu
\end{aligned}
\tag{2.14}
$$

Note that an architecture specification does not restrict the behavior of components. A component's behavior, on the other hand is restricted in the specification of component types. We conclude the section by introducing the notion of composition as a means to combine a specification of component types with a corresponding architecture specification.

Definition 11 (Composition)**.** Composition *of a family of components* $(\mathcal{C}_{ct})_{ct \in \mathcal{CT}}$ *with an architecture specification* $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^\infty$*, is defined as follows:*

$$
\bigotimes_{\mathcal{A}}(\mathcal{C}) \overset{\mathit{def}}{=} \Big\{ t \in \mathcal{A} \mid \forall ct \in CT,\ c \in C_{ct}\ \exists t' \subseteq (\overline{\mathsf{port}(ct)})^\infty \colon \Pi_c(t) \frown t' \in \mathsf{bhv}(ct) \Big\}
\tag{2.15}
$$

Note that the projection to an *unfair* architecture trace t, i.e., a trace in which a component is activated only finitely many times, the projection to this component results in only a finite behavior trace. Thus, we actually search for a valid *continuation* t', such that the concatenation for the projection $\Pi_c(t)$ with t' is a valid behavior of c. The situation is depicted in Fig. 2.8: The projection to component c (represented by the empty rectangle) in architecture trace t, is combined with a possible continuation t' to obtain a behavior trace $\Pi_c(t) \frown t'$ (shown at the bottom of Fig. 2.8).

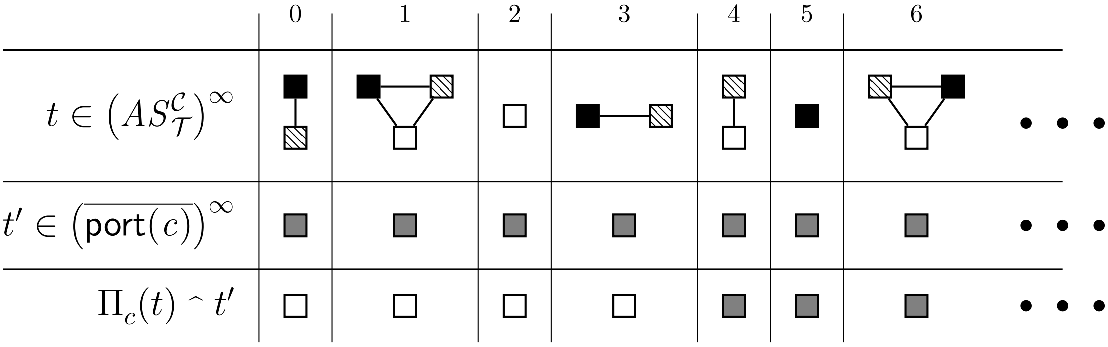


Figure 2.8: Continuations for unfair architecture traces.

2.5.5 A Note on Compositionality

We conclude the section with a brief discussion about compositionality, since it is an important property, which allows to combine specifications of components (usually by means of logical conjunction) to reason about its composition. In the following, we use Γ to denote a specification of component activation and port connection, as introduced by Def. 10. Moreover, we denote with γ_{ct} a specification of component type ct, as described in Def. 4.

In the presented approach, the behavior of an architecture is fully determined by the behavior of the single components *and* an additional specification of architectural aspects, such as activation of components and connections (which is in line with our definition of architecture, presented in Sect. 1).

Theorem 1. *Γ holds for an architecture specification $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$ and for each component type $ct \in CT_{\mathcal{I}}$ a specification γ_{ct} holds, iff Γ holds for $\otimes_{\mathcal{A}}(\mathcal{C})$ and γ_{ct} holds for the projection to every component $c \in \mathcal{C}$ of type ct in $\otimes_{\mathcal{A}}(\mathcal{C})$.*

Figure 2.9 summarizes the situation (an informal proof is provided in App. B): Whenever we have a specification γ_{ct} for component types $ct \in \mathcal{CT}$ and a corresponding specification Γ for architecture specification $\mathcal{A}$, we can simply combine them using logical conjunction to have a specification for $\otimes_{\mathcal{A}}(\mathcal{C})$.

Note that this corresponds to a situation in which we have a designated *controller component* which, at every point in time, knows the state of an architecture and based on that determines activation and deactivation of components and connections.

$$\begin{tikzcd}
\gamma_{ct} \arrow[d] \arrow[dr, dashed, "\Pi"] & \wedge & \Gamma \arrow[d] \arrow[dl, dashed] \\
(\mathcal{C})_{ct} & \otimes & \mathcal{A}
\end{tikzcd}$$

Figure 2.9: Compositionality.

For a more decentralized approach, however, one could just require the existence of a separate specification of architecture reconfiguration, for each single component (or component type). Then, the behavior of an architecture is fully determined by the specification of components only (without any designated controller component). Again, there are two possible options for such a design:

- In one version, every component knows the state of the whole architecture, at each point in time.
- In another version, every component only knows the state of itself.

While the first option is actually equal to the centralized approach, the second option is more restrictive, i.e., not every set of architecture traces which can be specified with the first approach, can also be specified with the second approach.

2.6 Summary

Figure 2.10 summarizes the main concepts of our model and their interrelationships: Messages and ports (typed by sets of messages) form the basic concepts of the model. A key concept of the model is the notion of *component type* which consists of an interface and a behavior in terms of behavior traces (streams of port valuations, i.e., valuations of ports with messages). In order to deal with related groups of components, we extended the notion of component type to *parametrized component type*. Another important concept is the concept of *architecture specification*: a special set of architecture traces (streams of architecture snapshots, i.e., states of an architecture during execution). Finally, the model provides an operator to combine a given architecture specification with a set of component types and corresponding components. Therefore, the operator uses the concept of *behavior projection* which extracts the behavior of a certain component out of a given architecture trace.

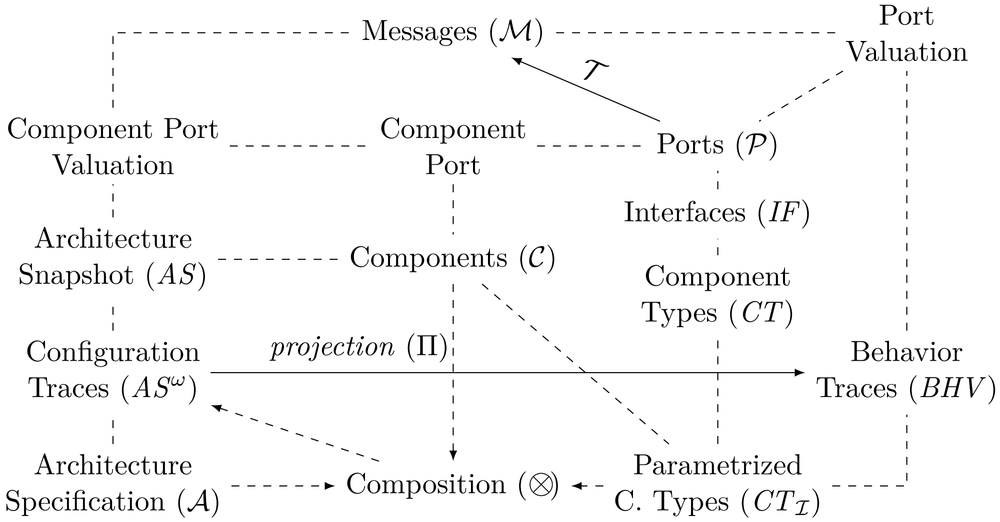


Figure 2.10: Concept map summarizing major concepts and their interrelationships.

Part II

Specification

3 Specifying Architectural Design Patterns

In the last section, we described a model for dynamic architectures based on the concept of component types and architecture specifications. However, we did not yet provide any techniques to specify ADPs over the model. Figure 3.1 provides an overview of techniques which can be used to specify ADPs over the model introduced in Chap. 2. First, data types are specified for the messages exchanged by the components of an architecture using traditional, *algebraic specification techniques* [Bro96, Wir90]. Then, component types are specified on top of these datatypes: Therefore, corresponding interfaces are specified for each type of component using a graphical notation called *architecture diagrams*. Then, component behavior is specified over these interfaces using so-called *behavior trace assertions*, i.e., linear temporal logic formulæ with ports as free variables. Finally, an architectural specification is given by means of so-called *architecture trace assertions*: linear temporal logic formulæ with component variables and architectural predicates. The techniques come with a formal semantics in terms of the model introduced in Chap. 2 and they are implemented in terms of an Eclipse/EMF application [GM18] which supports the specification of ADPs by rigorous type checking mechanisms. In the following section, we detail on each of the techniques and demonstrate them by means of our three running example: the Singleton, the Publisher-Subscriber, and the Blackboard pattern.

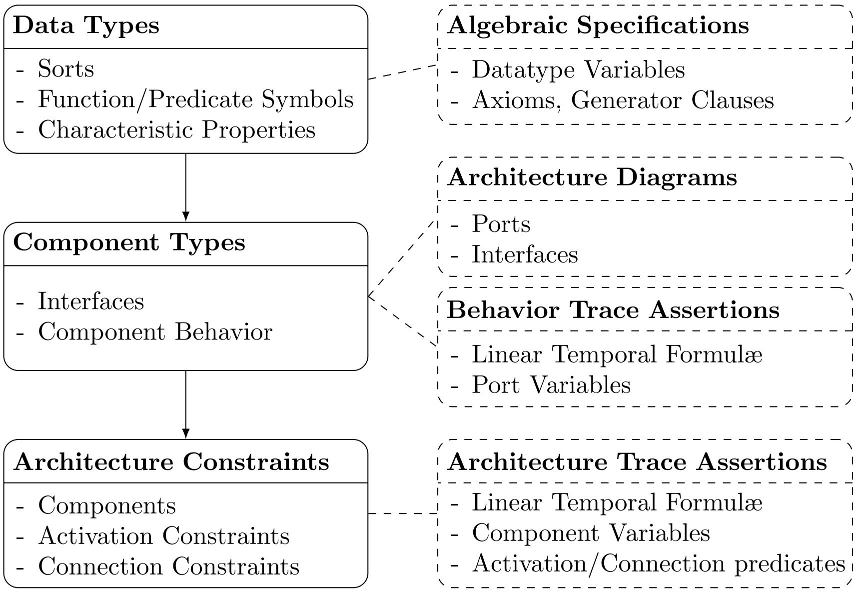


Figure 3.1: Specifying architectural design patterns.

3.1 Specifying Data Types

As a first step, a set of data types is specified for a pattern. Data types are specified in terms of axioms over a signature and corresponding variables. They can be specified using traditional, algebraic specification techniques [Bro96, Wir90]. Figure 3.2 depicts a schematic example of an algebraic specification. Each specification has a name and may be parametrized by several sorts. Moreover, other data type specifications can by imported by means of their name. Function/predicate symbols are introduced with the corresponding sorts at the beginning of the specification. Some of the symbols might be declared as generator clauses, requiring that every element of the corresponding datatype can be “reached” by a term formulated with these symbols. Finally, a list of variables for the different sorts is defined and a set of axioms is specified to describe the characteristic properties of a data type.

DTSpec Name(`param`)	**imports** OtherDatatype
symbol1 :	`Sort1`
symbol2 :	`Sort1` → `Sort2`
⋮	
generated by *symbol1*, *symbol2*	
⋮	
flex *var1*, *var2* :	`Sort1`
var3 :	`Sort2`
⋮	
assertion1(*symbol1*, *var1*, *var2*, *var4*)	
assertion2(*symbol1*, *symbol2*, *var1*, *var4*)	
⋮	

Figure 3.2: Schematic algebraic specification to for data types.

In the following, we demonstrate datatype specifications using our three running examples. The specification of the Singleton pattern does not require any special data types. Data types are required, however, for the specification of the Publisher-Subscriber pattern as well as the Blackboard pattern.

Example 12 (Datatype specification for the Publisher-Subscriber pattern)**.** *In a Publisher-Subscriber pattern, we usually have two types of messages:* subscriptions *for, and* unsubscriptions *from events. Figure 3.3 depicts the corresponding data type specification. Subscriptions are modeled as* parametric *data types over two type parameters: a type* `id` *for component identifiers and some type* `evt` *denoting events to subscribe for. The data type is freely generated by the constructor terms “sub* `id` `evt`*” and “unsub* `id` `evt`*”, meaning that every element of the type has the form “sub* `id` `evt`*” or “unsub* `id` `evt`*”.*

DTSpec subscription($\texttt{id}, \texttt{evt}$)
generated by sub $\texttt{id}$ $\wp(\texttt{evt})$, unsub $\texttt{id}$ $\wp(\texttt{evt})$

Figure 3.3: Data type specification for the Publisher-Subscriber pattern.

Example 13 (Datatype specification for the Blackboard pattern)**.** *Blackboard architectures usually work with* problems *and* solutions *for them. Figure 3.4 provides a specification of the corresponding data types. We denote by* $\texttt{PROB}$ *the set of all problems and by* $\texttt{SOL}$ *the set of all solutions. Complex problems consist of* subproblems *which can be complex themselves. To solve a problem, its subproblems have to be solved first. Therefore, we assume the existence of a* subproblem relation $\prec \subseteq \texttt{PROB} \times \texttt{PROB}$ *which relates problems with corresponding subproblems. For complex problems, the* details *of the relation may not be known in advance. Indeed, one of the benefits of a Blackboard architecture is that a problem can be solved even without knowing the exact nature of this relation in advance. However, the subproblem relation has to be well-founded*[1] *(Eq.* (3.1)*) for a problem to be solvable. In particular, we do not allow for cycles in the transitive closure of* $\prec$*. While there may be different approaches to solve a problem (i.e., several ways to split a problem into subproblems), we assume that the final solution for a problem is always unique. Thus, we assume the existence of a function* $solve \colon \texttt{PROB} \to \texttt{SOL}$ *which assigns the* correct *solution to each problem. Note, however, that it is not known in advance* how *to compute this function and it is indeed one of the reasons for using this pattern to calculate this function.*

DTSpec ProbSol	**imports** SET
$\prec$:	$\texttt{PROB} \times \texttt{PROB}$
solve:	$\texttt{PROB} \to \texttt{SOL}$
well-founded($\prec$)	(3.1)

Figure 3.4: Data type specification for the Blackboard pattern.

3.2 Specifying Component Types

On top of the specified data types, a set of parametrized component types (as described in Def. 4) is specified for the pattern. Component types are specified in two steps: First, an interface is specified for them using a graphical notation called architecture diagrams. Then, behavior is specified in terms of behavior trace assertions.

3.2.1 Specifying Interfaces

On top of the specified data types, a set of interfaces for the component types (as introduced in Def. 2) are specified. The specification of interfaces proceeds in two steps:

---

[1] A *well-founded* relation is a partial order which has no infinite decreasing chains.

First, a set of ports is specified as means to exchange messages of a certain type. Then, interfaces are specified over the ports.

3.2.1.1 Port Specifications

Ports are specified in terms of templates which declare a set of port identifiers and a corresponding typing. Figure 3.5 shows such a template which specifies two ports *port1* of type `Sort1` and *port2* of type `Sort2`, respectively.

PSpec Port Specification	**imports** Datatype
port1 :	`Sort1`
port2 :	`Sort2`
⋮	

Figure 3.5: Exemplary port specification.

Again, we demonstrate port specifications by means of our running examples and again, the specification of the Singleton pattern does not require any ports, at all. However, Publisher-Subscriber architectures and also Blackboard architectures require ports to be specified.

Example 14 (Port specification for the Publisher-Subscriber pattern)**.** *Two port types are specified for the Publisher-Subscriber pattern by the specification given in Fig. 3.6: a type sb which allows to exchange subscriptions for a specific event and a type nt which allows to exchange messages associated with a certain event. To this end, it uses a type parameter* `msg` *and imports the data type specification for subscriptions described in Ex. 12.*

PSpec PSPorts(`msg`)	**imports** subscription(`id`, `evt`)
sb:	subscription(`id`, `evt`)
nt:	`evt` × `msg`

Figure 3.6: Port specification for the Publisher-Subscriber pattern.

Example 15 (Port specification for the Blackboard pattern)**.** *For the specification of the Blackboard pattern we require* 4 *different ports as specified in Fig. 3.7:*

- *rp is used to exchange a problem which a knowledge source is able to solve, together with a set of subproblems the knowledge source requires to be solved first.*
- *ns is used to exchange a problem solved by a knowledge source, together with the corresponding solution.*
- *op is used to exchange problems which still need to be solved.*

- *cs is used to exchange solutions for problems.*

Moreover, a component parameter prob is specified to parametrize knowledge sources according to the problems they can solve.

PSpec BBPorts	**imports** ProbSol
rp:	$\mathtt{PROB} \times \wp(\mathtt{PROB})$
ns, cs:	$\mathtt{PROB} \times \mathtt{SOL}$
op, $prob$:	$\wp(\mathtt{PROB})$

Figure 3.7: Port specification for the Blackboard pattern.

3.2.1.2 Interface Specification

Interfaces consist of a set of input and output ports. Moreover, they consist of a set of so-called *component parameters* with a corresponding *strictness condition* to specify groups of related components. Formally, component parameters are represented as ports, however, they have a special meaning in the following sense:

- The valuation of component parameters is bound to a concrete component (as required by Def. 3), i.e., the valuation does not change over time, compared to valuations of input and output ports.
- For each possible valuation of the component parameters, at least one component is guaranteed to exist (as required in Sect. 2.5.1). This is not the case for input and output port valuations.
- If the interface is declared to be strict, then exactly one component exists for each valuation of the parameter ports.

Interfaces are specified over a given port specification and they are best expressed graphically using so-called *architecture diagrams*. Thereby, an interface is represented by a rectangle and consists of two parts: i) A name followed by a list of component parameters (enclosed between '⟨' and '⟩' for non-strict interfaces and '⟪' and '⟫' for strict ones). ii) A set of input and output ports which are represented by empty and filled circles, respectively. Figure 3.8 shows a conceptual representation of an architecture diagram *Name*, which is based on a port specification "PortSpecification" and which specifies two interfaces:

- Interface *If1* which consists of one input port i, one output port o, and a non-strict component parameter par.
- Interface *If2* which consists of a single output port o, and a strict component parameter par.

In the following, we provide interface specifications for all of our running examples.

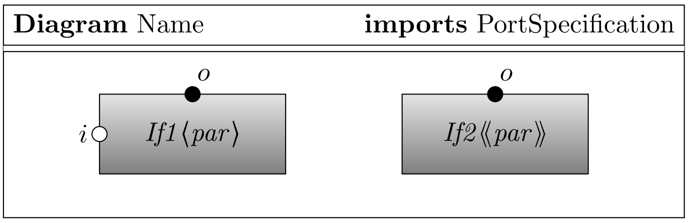


Figure 3.8: Exemplary architecture diagram specifying two interfaces.

Example 16 (Interface specification for the Singleton pattern)**.** *The interface for the Singleton pattern is specified by the architecture diagram depicted in Fig. 3.9: It consists of a single interface Singleton and does not require any special ports.*

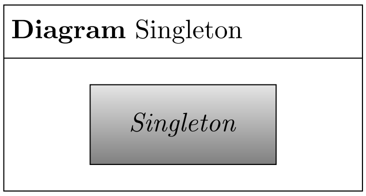


Figure 3.9: Architecture diagram for the Singleton pattern.

Example 17 (Interface specification for the Publisher-Subscriber pattern)**.** *The architecture diagram depicted in Fig. 3.10 shows the specification of the interfaces of the two types of components involved in a Publisher-Subscriber pattern: An interface "Publisher" is defined with an input port sb to receive subscriptions and an output port nt to send out notifications. Moreover, an interface "Subscriber" is defined with an input port nt receiving notifications and an output port sb to send out subscriptions. Note also that the diagram imports the specification of ports discussed in Ex. 14.*

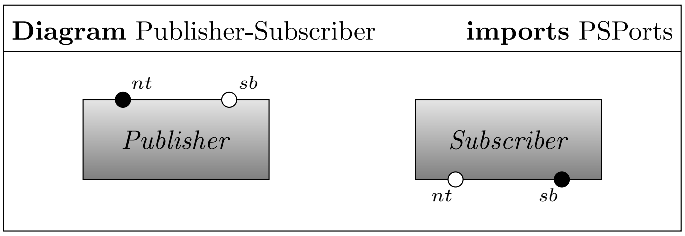


Figure 3.10: Architecture diagram for the Publisher-Subscriber pattern.

Example 18 (Interface specification for the Blackboard pattern)**.** *A Blackboard pattern usually involves two types of components: blackboards and knowledge sources. The corresponding interfaces are specified by the architecture diagram depicted in Fig. 3.11:*

The blackboard interface is denoted "BB" and consists of two input ports rp and ns to receive subproblems for which solutions are required and new solutions to currently open *problems. Moreover, it specifies two output ports op and cs to communicate currently open problems and solutions for all currently solved problems.*

The interface for knowledge sources is denoted "KS" and its specification actually mirrors the specification of the blackboard interface: A knowledge source is required to have two input ports op and cs to receive currently open problems and solutions for all currently solved problems, and two output ports rp and ns to communicate required subproblems and new solutions. Note that each knowledge source can only solve certain problems, which is why a knowledge source is parameterized by a set of problems "prob" it is able to solve. Since there may be different knowledge sources which are able to solve the same set of problems, the parameter is not *declared to be strict. Again, the diagram imports the corresponding port specification from Ex. 15.*

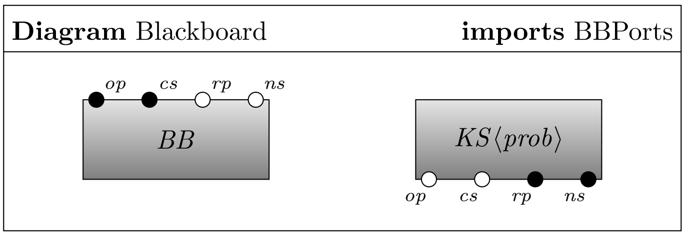


Figure 3.11: Architecture diagram for the Blackboard pattern.

3.2.2 Specifying Behavioral Constraints

We conclude the specification of component types by assigning constraints about component behavior to each interface. These constraints are expressed in terms of so-called *behavior trace assertions*. In the following, we introduce behavior trace assertions informally and demonstrate them by means of our Blackboard example. However, in App. C, we provide also a formal definition of the syntax and semantics of behavior trace assertions.

3.2.2.1 Behavior Trace Assertions

Behavior trace assertions are a means to specify a component's behavior in terms of behavior traces introduced in Def. 3. They are formulated by means of first-order linear temporal logic formulæ [MP92] over *datatype variables* and *behavior assertions.*

Datatype variables Behavior trace assertions may be specified over variables for messages. Thereby, variables are typed by the sorts of the pattern's datatype specification and we distinguish between two types of variables: *rigid* and *flexible* datatype variables. While rigid variables are only interpreted once, flexible variables are newly interpreted at each point in time.

Behavior assertions Roughly speaking, behavior assertions are predicate logic formulæ specified over a set of *datatype variables* and a set of ports, with terms consisting of:

- Datatype variables as well as the ports and parameters of a component type's interface.
- Function and predicate symbols of the corresponding data type specification.

They specify the state of a component (in terms of valuations of input and output ports) during execution.

3.2.2.2 Component Type Specification

Component types are specified using templates as shown in Fig. 3.12. The specification has a name and is associated with an interface of a corresponding interface specification. Then, a set of flexible and rigid variables for different sorts are declared. A component type specification concludes with a list of behavior trace assertions, formulated over the ports of the corresponding interface and the introduced data type variables.

BSpec Name	**for** iface **of** ispec
flex aDt1:	`Sort1`
rig aDt2:	`Sort1`
⋮	
assertion1 (iface, aDt1, aDt2)	
assertion2 (iface, aDt1, aDt2)	
⋮	

Figure 3.12: Schematic component type specification.

In the following, we demonstrate component type specifications in terms of our running examples. However, since the Singleton pattern and the Publisher-Subscriber pattern do not pose any constraints on the behavior of components, we only provide behavior specifications for the two types of components involved in a Blackboard pattern.

Example 19 (Behavioral constraints for blackboard components)**.** *A blackboard provides the* current state *towards solving the original problem and forwards problems and solutions from knowledge sources. Figure 3.13 provides a specification of the blackboard's behavior in terms of three behavior trace assertions:*

- *If a solution s' to a subproblem p' is received on its input port ns, then it is eventually provided at its output port cs (Eq. 3.2).*
- *If it gets notified that solutions for subproblems P are required in order to solve a certain problem p on its input port rp, these problems are eventually provided at its output port op (Eq.* (3.3)*).*
- *A problem p' is provided at its output port op, as long as it is not solved (Eq.* (3.4)*).*

Note that the last assertion (Eq. (3.4)*) is formulated using a* weak *until operator which is defined as follows:* $\gamma' \; \mathcal{W} \; \gamma \quad \overset{def}{=} \quad \Box(\gamma') \lor (\gamma' \; \mathcal{U} \; \gamma)$*.*

BSpec Blackboard **for** *BB* **of** Blackboard

flex p: `PROB`
P: `PROB SET`
rig p': `PROB`
s': `SOL`

$$\Box\Big((p', s') \in ns \longrightarrow \Diamond\big((p', s') \in cs\big)\Big) \tag{3.2}$$

$$\Box\Big((p, P) \in rp \longrightarrow \big(\forall p' \in P\colon \; (\Diamond(p' \in op))\big)\Big) \tag{3.3}$$

$$\Box\Big(p' \in op \longrightarrow (p' \in op \; \mathcal{W} \; (p', solve(p')) \in ns)\Big) \tag{3.4}$$

Figure 3.13: Specification of behavior for blackboard components.

Example 20 (Behavioral constraints for knowledge source components)**.** *A knowledge source receives open problems and solutions for already solved problems. It might contribute to the solution of the original problem by solving currently open subproblems. Figure 3.14 provides a specification of knowledge source behavior in terms of three behavior trace assertions:*

- *If a knowledge source requires some subproblems* P *to be solved in order to solve a problem* p' *and it gets solutions for all these subproblems* q' *on its input port cs, then it eventually solves the original problem* p' *and provides the solution through its output port ns (Eq.* (3.5)*).*
- *To solve a problem* p*, a knowledge source requires solutions only for smaller problems* q *(Eq.* (3.6)*).*
- *A knowledge source will eventually communicate its ability to solve an open problem via its output port rp (Eq.* (3.7)*).*

3.3 Specifying Architectural Constraints

As a last step, an architecture specification (as described in Def. 10) is specified by means of constraints about the activation and deactivation of components as well as constraints about connections between component ports. Both types of constraints may be expressed in terms of *architecture trace assertions*, i.e, linear temporal logic formulæ [MP92] over datatype variables (introduced in the description of behavior trace assertions above), component variables, and architecture assertions. Their semantics is given in terms of architecture traces (as described in Def. 8). Again, in the following, we introduce architecture trace assertions informally, by means of our running examples and we provide a formal definition of the syntax and semantics in App. C.4.

BSpec Knowledge Source **for** $KS\langle prob\rangle$ **of** Blackboard

flex $p,\ q$: `PROB`
P: $\wp$(`PROB`)
rig $p',\ q'$: `PROB`

$$\Box\Big(\forall (p',P)\in rp\colon\ \big((\forall q'\in P\colon\ \Diamond(q',solve(q'))\in cs)\longrightarrow \Diamond(p',solve(p'))\in ns\big)\Big) \tag{3.5}$$

$$\Box\Big(\forall (p,P)\in rp\colon\ \forall q\in P\colon\ q\prec p\Big) \tag{3.6}$$

$$\Box\Big(prob\in op\longrightarrow \Diamond(\exists P\colon\ (prob,P)\in rp)\Big) \tag{3.7}$$

Figure 3.14: Specification of behavior for knowledge source components.

3.3.1 Component Variables

Component variables are typed by component types and may be interpreted by corresponding components. Similar to datatype variables, component variables can be classified into “flexible” and “rigid”, depending on whether they are newly interpreted at each point in time or whether they keep their value. Since component types may be parametrized, variables are assumed to be available for each possible valuation of the parameters. For example, a component variable declaration x for component type $X\langle bool\rangle$ would actually induce two component variables $x_{\langle true\rangle}$ and $x_{\langle false\rangle}$ which can be interpreted by components where parameter *bool* is valuated with the interpretations of *true* and *false*, respectively. Note that such parametrized component variables are only feasible since the semantics of a FACTUM specification requires the existence of at least one component for each different valuation of a component type’s component parameters (as discussed in Sect. 3.2.1.2).

3.3.2 Architecture Assertions

Architecture assertions are predicates to specify snapshots of an architecture during execution (as described in Def. 5). Roughly speaking, they are predicate-logic formulæ specified over datatype and component variables, with terms consisting of:

- Datatype variables as well as component ports and component parameters, i.e., ports or parameters combined with corresponding component variables.
- Function and predicate symbols of the corresponding data type specification.

Moreover, several pre-defined, *architectural predicates* may be used for the formulation of architecture assertions:

- $\widehat{c.p}$ denotes that a component c is currently sending/receiving a message over port p,
- $\wr c \wr$ denotes that a component c is currently active, and
- $c.p \rightsquigarrow c'.p'$ denotes that output port p' of component c' is connected to input port p of component c.

3.3.3 Architecture Constraint Specification

Architectural constraints can be specified by means of specification templates (Fig. 3.15). Each template has a name and is based on a corresponding interface specification. Then, a list of flexible and rigid variables for the data types and components are defined. Finally, a list of architecture trace assertions is formulated over the variables. Note that the semantics of architecture constraint specifications is given in terms of architecture specifications as described in Chap. 2. Thus, they can only be used to specify component activation and reconfiguration and *not* to restrict the behavior of components.

ASpec Name		**for** ifSpec
flex	aDt1:	`Sort1`
	aCmp1:	*If1*
rig	aDt2:	`Sort1`
	aCmp2:	*If1*
⋮		

assertion1 (aDt1, aDt2, aCmp1, aCmp2)

assertion2 (aDt1, aDt2, aCmp1, aCmp2)

⋮

Figure 3.15: Exemplary architecture constraint specification.

In the following, we provide architecture constraint specifications for all three patterns.

Example 21 (Architectural constraints for the Singleton pattern)**.** *Architectural constraints for the Singleton pattern are formalized by the specification depicted in Fig. 3.16. The specification requires two constraints for the activation of components: Equation 3.8 requires that at each point in time there exists a singleton component* c *which is activated. Equation 3.9 further asserts that there exists a unique component* c'*, such that every active component* c *of type singleton is equal to* c' *at every point in time. In our version of the singleton, we require that the singleton component is not allowed to change over time. This is why variable* c' *is declared to be rigid in Fig. 3.16. Indeed, other versions of the singleton are possible in which the singleton may change over time.*

ASpec Singleton		**for** Singleton
flex	c:	*Singleton*
rig	c':	*Singleton*

$$\Box(\exists c \colon \wr c \wr) \tag{3.8}$$

$$\exists c' \colon \Big(\Box(\forall c \colon (\wr c \wr \longrightarrow c = c'))\Big) \tag{3.9}$$

Figure 3.16: Activation constraints for a Singleton pattern.

Example 22 (Architectural constraints for the Publisher-Subscriber pattern)**.** *Activation constraints for the publisher component of a Publisher-Subscriber pattern are similar* *to the ones specified for the Singleton pattern in Fig. 3.16. Moreover, a Publisher-Subscriber pattern requires two additional constraints, regarding the connections between publisher and subscriber components, which are specified in Fig. 3.17:*

- *With Eq.* (3.10)*, we require that a publisher's sb port is always connected to a subscriber's sb port, whenever both of them are active.*
- *With Eq.* (3.11)*, we require that port nt of a subscriber s' is always connected to a publisher's nt port, whenever the publisher sends out a message associated to an event e for which s' was subscribed for.*

ASpec Publisher-Subscriber **for** Publisher-Subscriber

flex s: *Subscriber*
p: *Publisher*
m: `msg`
E: $\wp($`evt`$)$
rig s': *Subscriber*
e: `evt`

$$\Box\Big(\wr p\wr \land \wr s\wr \land \widehat{s.sb} \longrightarrow p.sb \rightsquigarrow s.sb\Big) \tag{3.10}$$

$$\begin{aligned}\Box\Bigg(&\wr s'\wr \land (\exists E\colon \text{sub } s'\ E \in s'.sb \land e \in E)\\ &\longrightarrow \Big((\wr p\wr \land \wr s'\wr \land (e,m) \in p.nt \longrightarrow s'.nt \rightsquigarrow p.nt)\\ &\qquad \mathcal{W}\ (\wr s'\wr \land (\exists E\colon \text{unsub } s'\ E \in s'.sb \land e \in E))\Big)\Bigg)\end{aligned} \tag{3.11}$$

Figure 3.17: Architectural constraints for the Publisher-Subscriber pattern (in addition to the ones specified for the Singleton pattern).

Example 23 (Architectural constraints for the Blackboard pattern)**.** *Also for the Blackboard pattern we get similar activation constraints for blackboard components as the ones specified for the Singleton pattern in Fig. 3.16. Moreover, the Blackboard pattern requires similar connection constraints as required for the Publisher-Subscriber pattern in Fig. 3.17. Thereby, port rp of the Blackboard pattern corresponds to port sb and port cs of the Blackboard pattern to port nt.*

In addition, Fig. 3.18 provides two connection constraints and three activation constraints for Blackboard architectures:

- *By Eq.* (3.12)*, we require that a blackboard's op port is always connected to a knowledge source's op port, whenever both of them are active.*
- *By Eq.* (3.13)*, we require that a blackboard's ns port is always connected to a knowledge source's ns port, whenever both of them are active.*

ASpec Blackboard **for** Blackboard

flex	ks:	$KS\langle prob\rangle$
	bb:	BB
	p:	$\texttt{PROB}$
	s:	$\texttt{SOL}$
	P:	$\wp(\texttt{PROB})$
rig	ks':	$KS\langle prob\rangle$
	p':	$\texttt{PROB}$

$$\Box\Big(\wr ks\wr \land \wr bb\wr \land \widehat{bb.op} \longrightarrow ks.op \rightsquigarrow bb.op\Big) \tag{3.12}$$

$$\Box\Big(\wr bb\wr \land \wr ks\wr \land \widehat{ks.ns} \longrightarrow bb.ns \rightsquigarrow ks.ns\Big) \tag{3.13}$$

$$\Box\Big(\wr ks'\wr \longrightarrow \Box\big(\Diamond \wr ks'\wr\big)\Big) \tag{3.14}$$

$$\begin{aligned}&\Box\Big(\wr ks'\wr \land (p, P) \in ks'.rp \land p' \in P\\ &\quad \longrightarrow \Box\big((\exists bb\colon \wr bb\wr \land (p', s) \in bb.cs) \longrightarrow \wr ks'\wr\big)\Big)\end{aligned} \tag{3.15}$$

$$\Box\Big(\forall p' \in bb.op\colon \Diamond\big(\exists ks_{\langle prob=p'\rangle}\colon \wr ks\wr\big)\Big) \tag{3.16}$$

Figure 3.18: Architectural constraints for Blackboard architectures (in addition to the ones specified for the Singleton and the Publisher-Subscriber pattern).

- *By Eq.* (3.14)*, we require a fairness condition for the activation of already activated knowledge sources.*
- *By Eq.* (3.15)*, we require that whenever a knowledge source offers to solve some problem p, given that it receives solutions for corresponding subproblems P, then the knowledge source is activated, whenever a solution for any of the problems of P is provided.*
- *By Eq.* (3.16)*, we require that for each open problem p', a knowledge source ks which is able to solve p' is eventually activated.*

Note expression $\exists ks_{\langle prob=p'\rangle}\colon \wr ks\wr$ *of Eq.* (3.16) *which demonstrates the use of parametrized component variables. Indeed, the variable* $ks_{\langle prob=p'\rangle}$ *actually represents a knowledge source component which has its parameter "prob" valuated with the problem represented by datatype variable p'. Such parametrized variables provide a convenient way to specify constraints about components of parametrized component types.*

3.4 Summary

Table 3.1 summarizes techniques for the specification of ADPs described in this chapter. For each technique it lists the specified model concept (with reference to the definition of the concept), the type of technique, and important specification elements.

	concept	*type*	*elements*
Algebraic Specifications	data types for messages (Sect. 2.1)	template	function symbols, datatype variables, characteristic properties
Architecture Diagrams	interfaces (Def. 2), architecture specification (Def. 10)	graphical, annotations*	interfaces, connection annotations*, activation annotations*
Behavior Trace Assertions	parameterized component types (Def. 4)	template	datatype variables, temporal operators, ports
Architecture Trace Assertions	architecture specification (Def. 10)	template	datatype variables, component variables, temporal operators, component ports, architectural predicates

* Introduced in the next chapter.

Table 3.1: Techniques used for the specification of architectural design patterns.

4 Advanced Specifications

In the previous chapter, we introduced basic techniques to specify ADPs. We also applied the techniques to specify versions of three, well-known, ADPs: the Singleton, the Publisher-Subscriber, and the Blackboard pattern.

Specifying these patterns, however, led to two further observations:

1. Some architectural constraints are common to different ADPs. One example is the constraint that components of a certain type are required to be always activated. Another example is that components of a certain type are connected via certain ports, whenever they are activated.

2. Another observation is that, sometimes, pattern specifications reuse specifications from other patterns. For example, the specification of the Publisher-Subscriber pattern reused the activation specification from the Singleton pattern for the publisher component. Or the Blackboard pattern reused the whole specification of the Publisher-Subscriber pattern.

Building on these observations, in the following chapter, we extend our specification approach with two features which turn out to be useful for the specification of patterns:

1. In order to facilitate the specification of common activation and connection constraints, we extend our notion of architecture diagram with so-called *activation/connection annotations*. These annotations provide a convenient way to express certain architectural constraints graphically by annotating the given architecture diagram.

2. In order to support hierarchical pattern specifications, we introduce the notion of *pattern instantiations* which allow to import a pattern specification within another pattern specification and instantiate the corresponding component types. Therefore, we provide additional annotations for architecture diagrams, which allow to easily express such instantiations in a graphical manner.

Again, the different techniques are demonstrated in terms of the three running examples introduced in Chap. 1.

4.1 Activation Annotations

Activation annotations enhance an architecture diagram with constraints regarding the activation and deactivation of components. They are expressed by annotating component types with corresponding architecture assertions, determining situations in which components of the annotated type are required to be active or inactive. Figure 4.1, for example, depicts an activation annotation for a component type CT, parametrized by a parameter P. The annotation is enclosed between square brackets and takes a variable c of a component of type CT, with parameter valuation ω, as input. It then specifies two architecture assertions using variable c: $\gamma(c)$ determines situations in which the component is required to be activated, while $\gamma'(c)$ determines a situation in which the component is required to be deactivated. For the case neither $\gamma(c)$ nor $\gamma'(c)$ holds, c may be either active or not. If omitted, we assume default values *true* and *false*, respectively. In order to specify only the first parameter and leave the default value for the other one, we write $\lfloor c_{\langle\omega\rangle}\colon \gamma(c)\rfloor$. Similarly, we write $\lceil c_{\langle\omega\rangle}\colon \gamma(c)\rceil$ to only specify the second parameter and leave the default value for the other one.

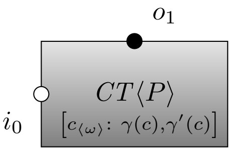


Figure 4.1: Activation annotation for a component type $CT\langle P\rangle$ with minimal activation condition $\gamma(c)$ and deactivation condition $\gamma'(c)$.

Activation annotations as described so far specify the activation/deactivation of components of a certain type. However, they do not say anything about the identity of these components. The annotation in Fig. 4.1, for example, allows c to be a different component at different points in time. In order to require that the identity of the components does not change over time, we need a stronger notion of activation annotation. We call it *rigid activation annotation* and it is expressed using double square brackets, instead of single square brackets. Figure 4.2, for example, depicts an activation annotation, similar to the one presented in Fig. 4.1. However, since we use double square brackets, it is to be interpreted as a rigid activation annotation and variable c is not allowed to change over time. Similar as for activation annotations we take *true* and *false* as default values and write $\|\lfloor c\colon \gamma(c)\rfloor\|$ and $\|\lceil c\colon \gamma(c)\rceil\|$ to take the default values for the second and first condition only.

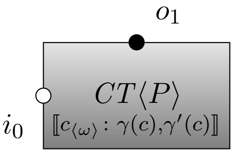


Figure 4.2: Rigid activation annotation for a component type $CT\langle P\rangle$, with activation condition $\gamma(c)$ and deactivation condition $\gamma'(c)$.

Using activation annotations, we can now specify a Singleton by adapting the architecture diagram presented in Fig. 3.9.

Example 24 (Annotations for the Singleton pattern)**.** *Figure 4.3 depicts the adapted architecture diagram for Singletons. The first condition requires that a singleton is always active. The second condition, on the other hand, requires that, whenever a singleton is active, it is the only component of that type which is active. Since we do not want singletons to change over time, we enclose the conditions into double squared brackets,* *making it a rigid activation annotation. Note that the new architecture diagram now makes the activation specification presented in Fig. 3.16 superfluous.*

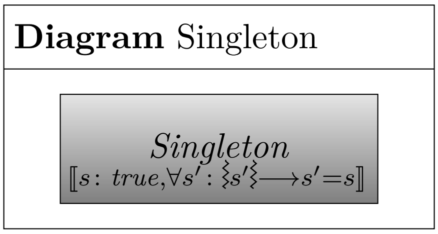


Figure 4.3: Annotated architecture diagram for the Singleton pattern.

4.2 Connection Annotations

Connection annotations enhance an architecture diagram by constraints regarding the connection of certain components. Such annotations are added to each type-consistent pair of input and output ports of component types and specify conditions under which the corresponding ports of components of these types are required to be connected or disconnected. Figure 4.4, for example, depicts a connection annotation for ports i and o of component types *CT1* and *CT2*, respectively. The annotation takes two component variables c and c' as input and specifies two architecture assertions $\gamma(c, c')$ and $\gamma'(c, c')$ over these variables: $\gamma(c, c')$ determines situations in which the connection is required to be established, while $\gamma'(c, c')$ determines a situation in which a connection is not allowed. For the case neither $\gamma(c, c')$ nor $\gamma'(c, c')$ holds, the connection may be established or not. Again, we assume default values of *true* and *false* and may omit one of the conditions to take its default value. Also for connection annotations we may require components not to change over time, which is why we also introduce the notion of *rigid connection annotation.* Similar as for rigid activation annotations, we mark connection annotations as rigid by enclosing them into double square brackets, instead of single square brackets.

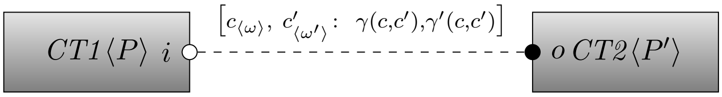


Figure 4.4: Connection annotation for connections between port i of component type $CT1\langle P\rangle$ and port o of component type $CT2\langle P'\rangle$ with connection condition $\gamma(c, c')$ and disconnection condition $\gamma'(c, c')$.

In the following, we demonstrate the use of connection annotations in terms of the examples introduced above.

Example 25 (Annotations for the Publisher-Subscriber pattern)**.** *We first adapt the architecture diagram for the Publisher-Subscriber pattern introduced in Fig. 3.10. The* *resulting architecture diagram is depicted in Fig. 4.5. We require a similar activation annotation for a publisher component as the one introduced for singletons. To increase readability, however, the annotation uses abbreviations γ and γ', which are expanded at the bottom of the diagram. Moreover, we add a connection annotation which requires port sb of a publisher component to be connected to port sb of a subscriber component, whenever the subscriber sends out some message. The dashed line without any annotation, denotes a connection constraint using the default values for connections and disconnections. Indeed the line could have been omitted and the semantics would not change. However, it is put there to highlight the fact that there is an additional connection constraint specified as architecture trace assertion in Eq.* (3.11)*. The new architecture diagram allows us now to get rid of some of the architectural assertions introduced in Ex. 22. Indeed, the only remaining assertion is Eq.* (3.11)*, which cannot be replaced with any annotation so far. Note that the connection annotation used in this example is used frequently which is why, from now one, we shall use a solid connection between ports to denote that the corresponding ports are required to be connected, whenever the output port is valuated with some message.*

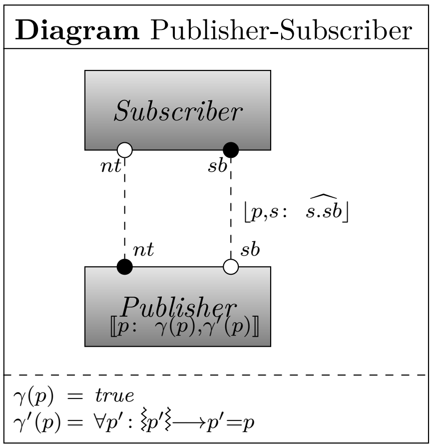


Figure 4.5: Annotated architecture diagram for the Publisher-Subscriber pattern.

Example 26 (Annotations for the Blackboard pattern)**.** *Next we adapt the architecture diagram for the Blackboard pattern as introduced in Fig. 3.11. The resulting architecture diagram is depicted in Fig. 4.6. Again, we add a similar annotation as the one required for singletons to the blackboard component type. Moreover, we add three connection annotations: the three solid lines between the ports of blackboard and knowledge source components use the new notation introduced in the last example to denote a required connection between the corresponding ports, whenever the output port sends out a message. The architecture diagram now captures all the architectural assertions imposed to a Blackboard architecture, except the activation assertions Eq.* (3.14)*, Eq.* (3.15)*, and Eq.* (3.16)*, as well as the connection assertion formulated by Eq.* (3.11)*.*

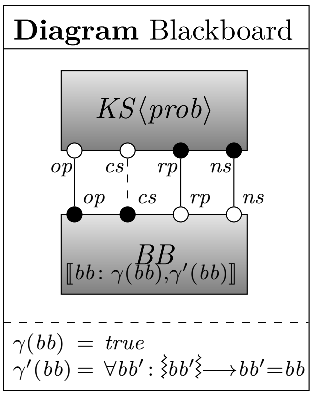


Figure 4.6: Annotated architecture diagram for the Blackboard pattern.

4.3 Dependencies

The annotations introduced so far affect all components of a certain type and they do not consider a component's context. Sometimes, however, activation as well as connection of certain components needs to be expressed *relative* to other components. Suppose, for example, we want to specify a Publisher-Subscriber pattern in which we can have multiple, different publisher components. Thus, we would first remove the activation constraint for the publisher component type which allows for multiple components of type publisher. Then, however, the activation and connection constraints for subscriber components need to be interpreted relatively to one of the publisher components.

To express this kind of constraints, in the following, we introduce so-called component relationships. They allow one to specify relationships between components of certain types and modify the semantics of activation and connection annotations accordingly.

4.3.1 Specifying Dependencies

Basically, we distinguish between two types of dependencies: weak and strong dependencies.

Weak dependencies between two components allow them to be shared amongst other components. A weak dependency specifies an upper and a lower bound of how many components of one type are dependent on a component of another type. They are specified graphically by connecting the dependent components with an edge which starts with an empty diamond and ends with the corresponding cardinality. Specifying a weak dependency between two component types actually determines a relation between components of the corresponding types. The dependency specified in Fig. 4.7, for example, determines a relation between components of type *CT1* and *CT2* where each component of type *CT1* is related with 5 to 10 components of type *CT2*. Since it is a weak dependency, components may be shared, i.e., the relation may contain an entry (c_1, c') and also an entry (c_2, c') in which case c' is dependent on both c_1 and c_2.

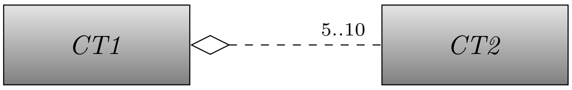


Figure 4.7: Weak dependency between components of type *CT1* and components of type *CT2*.

In order to avoid shared components, we also introduce the notion of *strong dependency*. Similarly to weak dependent annotations, they are specified graphically by connecting dependent component types with an edge which starts with a diamond and ends with corresponding cardinalities. However, in order to highlight that this is a strong dependency, we use filled diamonds, instead of empty ones. Similar as the specification of weak dependencies, also the specification of strong dependencies induces a relation between components of the corresponding type. Compared to weak dependencies, however, strong dependencies require the relation to be a function, i.e., that dependent components are not shared amongst components. The dependency specified in Fig. 4.8, for example, determines a relation between components of type *CT1* and *CT2* where each component of type *CT1* is related with 5 to 10 components of type *CT2*. Since it is a strong dependency, however, components must not be shared, i.e., the relation is not allowed to contain entries (c_1, c') and (c_2, c') whenever $c_1 \neq c_2$.

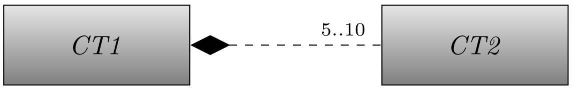


Figure 4.8: Strong dependency between components of type *CT1* and components of type *CT2*.

4.3.2 Dependent Connections

As hinted in the introduction of this chapter, adding dependencies refines the semantics of the corresponding activation and connection annotations. Connection annotations are now interpreted w.r.t. the relation induced by the dependency specification. Thus, conditions for required and prohibited connections are combined with a requirement that the components are indeed related according to the dependency specification. Figure 4.9, for example, establishes a relation between components of type *CT1* and components of type *CT2* and requires that a connection between the corresponding ports of related components is established, whenever $\gamma(c, c')$ holds. The ports must not be connected in situations described by $\gamma'(c, c')$.

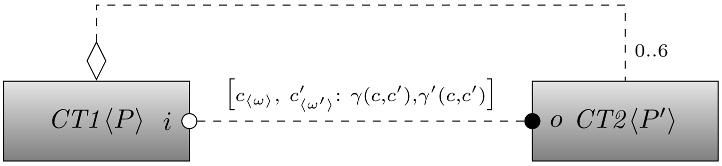


Figure 4.9: Dependent connection between components of type *CT1* and *CT2*.

4.3.3 Dependent Activations

For activation annotations, dependencies change the semantics of activation annotations in such a way that the activation of one of the components becomes a precondition to the activation of the other one. Thereby, we distinguish four types of dependent activations.

A *required activation annotation* specifies that the activation of dependent components requires the activation of the components they depend on. Such annotations can be easily expressed with a solid line for dependencies. The specification in Fig. 4.10, for example, introduces a dependency between components of type *CT1* and components of type *CT2*. Activations of components of type *CT2* then depend on the activation of a component of type *CT1*, on which the component of type *CT2* depends on.

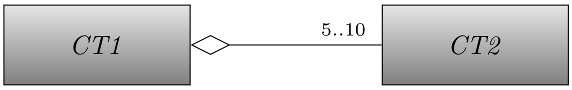


Figure 4.10: Required activation annotation for components of type *CT1* and *CT2*.

Sometimes, however, we want deactivation of a component to depend on the deactivation of dependent components. Such *required deactivations* can be expressed by changing the dependency line to a double dashed one. Figure 4.11, for example, requires a component of type *CT1* to be deactivated in order for dependent components of type *CT2* to be deactivated.

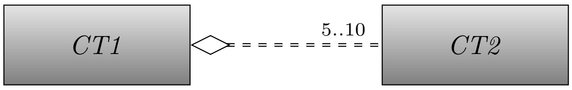


Figure 4.11: Required deactivation annotation for components of type *if1* and *if2*.

Finally, we may require that the activation of a component is completely determined by the activation of a depending component. This can be expressed with a double line in the dependency specification. Figure 4.12, for example, depicts how to specify that components of type *CT2* are activated whenever depending components of type *CT1* are.

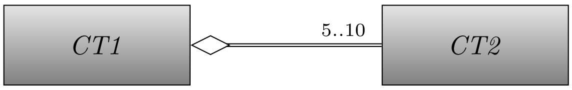
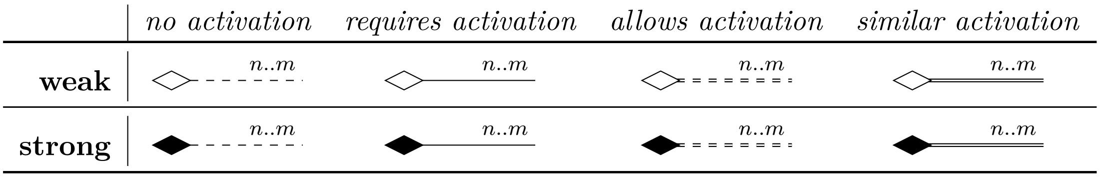


Figure 4.12: Dependent activation annotation for components of type *if1* and *if2*.

Table 4.1 summarizes the different types of dependency annotations which can be used for architecture diagrams.

4.3.4 Publisher-Subscriber with multiple Publishers

We can now use dependencies to specify an alternative version of the Publisher-Subscriber pattern.



	no activation	*requires activation*	*allows activation*	*similar activation*
weak	◇- - - - - - $n..m$	◇—— $n..m$	◇= = = = = = $n..m$	◇══ $n..m$
strong	◆- - - - - - $n..m$	◆—— $n..m$	◆= = = = = = $n..m$	◆══ $n..m$

Table 4.1: Overview of dependency annotations for architecture diagrams.

Example 27 (Publisher-Subscriber with multiple Publishers)**.** *Figure 4.13 depicts the specification of a Publisher-Subscriber pattern which allows for multiple publisher components with subscriber components which can subscribe at different publishers. The architecture diagram has two major changes compared to the original one: First, we relaxed the activation annotation by removing the postcondition. The new diagram only requires each publisher to be always activated, however it does not require anymore that only one component of type publisher exists. The second difference is that we added a weak dependency relationship between publishers and subscribers which allows subscribers to be subscribed at different publishers. Note that without adding the dependency relation, each subscriber would be required to be subscribed at each available publisher.*

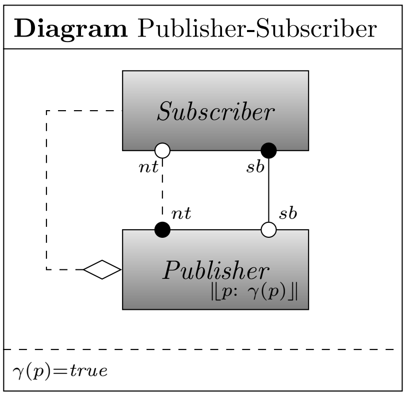


Figure 4.13: Alternative version of a Publisher-Subscriber pattern with multiple publishers.

In another version of the Publisher-Subscriber pattern we could require that subscribers are only allowed to subscribe at one single publisher. To specify this version we would only change the dependency to be a strong one (with a filled diamond).

4.4 Specifying Pattern Instantiations

As described above, pattern specifications may be built on top of other pattern specifications by instantiating their component types. Instantiating a pattern requires to provide a mapping which relates component types and port types. Such instantiations can be directly specified in a pattern's architecture diagram by annotating the corresponding interfaces.

Figure 4.14 depicts a schematic pattern instantiation. The diagram specifies a pattern *PatternB* and thereby instantiates another, existing specification *PatternA*, which is assumed to specify a component type *CT1* with one single input port i and one single output port o. *PatternB* specifies one component type *CT2* which is declared to be an instance of component type *CT1* of *PatternA*. Since *CT1* has two ports, the instantiation must provide mappings for these two ports which is done in square brackets right after the name of the instantiated component type. In our case, port i of component type *CT1* is mapped to input port i_1 and o to output port o_1. Note that the interface of *CT2* has two additional ports i_2 and o_2, which do not instantiate any port of interface *CT1*. Indeed, a port mapping is *not* required to be bijective, which means that a component type may add more ports to the interface of the component type it instantiates. However, we do require that the types of the ports refine the types of the port of the instantiated component type. For our example that means that the type of port i_1 must be a refined version of the type of i and the type of o_1 must refine the type of o.

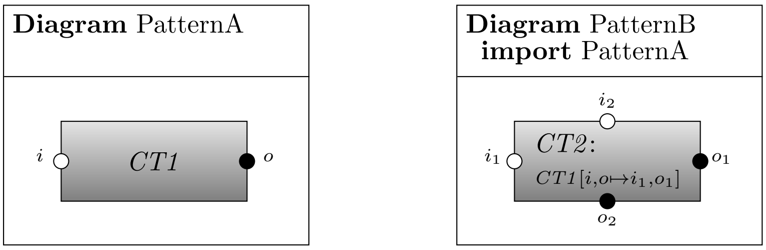


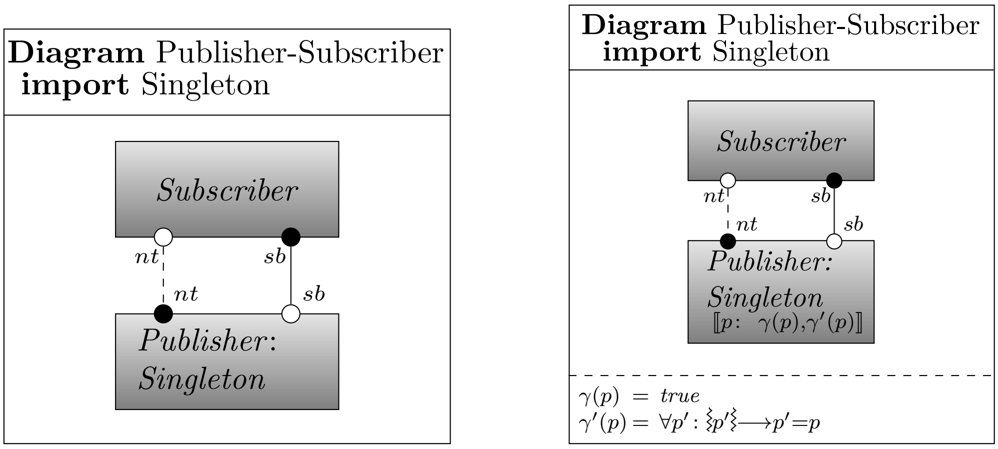
Figure 4.14: Architecture diagram in which component type *CT1* of *PatternB* instantiates component type *CT1* of a pattern *PatternA*.

In the following, we demonstrate hierarchical specifications in terms of our running examples.

Example 28 (Publisher-Subscriber instantiating the Singleton pattern)**.** *First, we adapt the specification of the Publisher-Subscriber pattern introduced above such that a publisher component is considered to be an instance of the singleton type. Figure 4.15 depicts the adapted architecture diagrams, excluding (Fig. 4.15a) and including (Fig. 4.15b) annotations inherited from the imported Singleton pattern. The diagram first imports the specification of the Singleton. Then it declares the publisher component type to be an instance of a singleton component type from the Singleton pattern.*

By instantiating the singleton, the publisher will inherit all its specified properties, i.e., an adapted version of the activation annotation of the Singleton will be available for publisher components.

Example 29 (Blackboard pattern instantiating the Publisher-Subscriber pattern)**.** *Finally we model a Blackboard pattern as an instance of the Publisher-Subscriber pattern. Thereby, the blackboard is specified to be an instance of the publisher type and knowledge sources instances of subscriber components, respectively. Figure 4.16 depicts the adapted architecture diagrams, excluding (Fig. 4.16a) and including (Fig. 4.16b) annotations inherited from the imported Publisher-Subscriber pattern. Again, the diagram imports the*



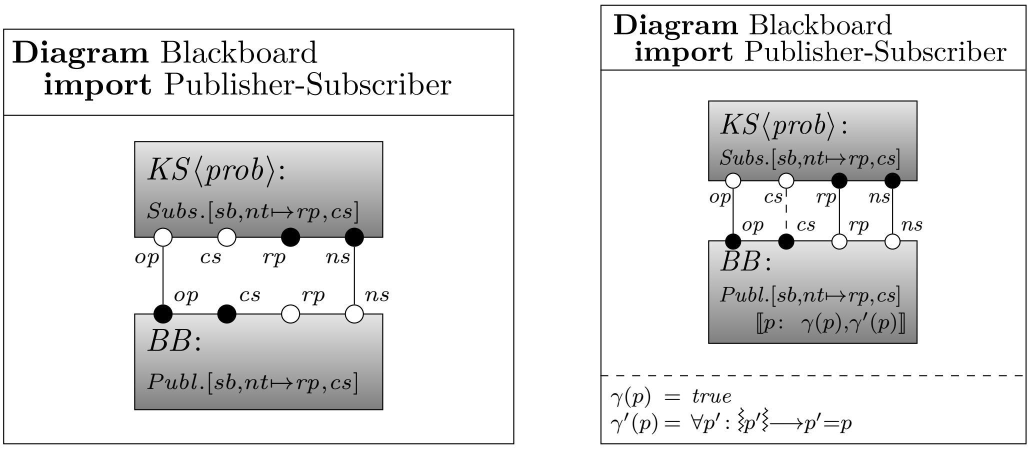
(a) Without inherited annotations. **(b)** With inherited annotations.

Figure 4.15: Architecture diagrams for the Publisher-Subscriber pattern instantiating the Singleton pattern.

specification of the Publisher-Subscriber pattern and declares a blackboard to be an instance of a publisher and a knowledge source to be an instance of a subscriber. However, since publisher as well as subscriber interfaces have ports, we need to provide an additional port-mapping which maps every port of a publisher / subscriber to a corresponding port of a blackboard/knowledge source. Note also that a blackboard / knowledge source adds two additional ports which do not map to any of the ports of the Publisher-Subscriber pattern. Again, by instantiating the Publisher-Subscriber pattern, the Blackboard will inherit the specification of the Publisher-Subscriber pattern. Thereby, the properties are adapted as specified by the instantiation, i.e., publisher components will become blackboard components and subscriber components will become knowledge sources in the specified properties. Note that inherited properties will be propagated as well which means that the Blackboard will even inherit the properties of the Singleton since the publisher component instantiates the Singleton.

4.5 Summary

Table 4.2 provides an overview of the advanced specification techniques for architecture diagrams introduced in this chapter. For each technique it lists the specified model concept, the type of technique, and important specification elements.



(a) Without inherited annotations. **(b)** With inherited annotations.

Figure 4.16: Architecture diagram for the Blackboard pattern instantiating the Publisher-Subscriber pattern.

	concept	*type*	*elements*
Activation Annotations	component activation and deactivation	textual for component types	activation conditions deactivation conditions
Connection Annotations	textual for connections	architecture assertions for connection	connection conditions deconnection conditions
Dependencies	relations between components	graphical	weak dependency strong dependency cardinalities
Instantiations	pattern instantiations	textual for component types	port mappings

Table 4.2: Summary of advanced architecture diagrams.

Part III

Verification

5 A Calculus for Architectural Design Patterns

In the last chapter, we introduced the notion of behavior trace assertion, as a means to specify component types (introduced in Sect. 2.4). Moreover, we also introduced the notion of architecture trace assertion, as a means to formulate architecture specifications (introduced in Sect. 2.5). Verifying an ADP now requires to show that the composition of component types with the architectural specification satisfies a certain property. The process is summarized in Fig. 5.1: First, behavior for component types is specified in terms of a set of behavior traces. Then, an architectural specification is interpreted over a set of architecture traces. Finally, the architecture specification is combined with the component type specification using behavior projection for each involved component c. The desired property is then verified over the resulting set of architecture traces.

Verifying ADPs using this approach led to the observation that certain proof steps are common for the verification of different ADPs. In an effort to shorten the verification process, we developed a calculus to reason about component behavior in a dynamic context by combining behavior and architecture specifications. Therefore, we first introduced a means to interpret a behavior specification directly over a set of architecture traces (dashed arrow in Fig. 5.1). Then, we introduced introduction and elimination rules for all the temporal operators involved in behavior specifications, to combine them with corresponding activation specifications and reason at a more abstract level ($\vdash_c$ in Fig. 5.1). Finally, we showed soundness of each of the rules w.r.t. the interpretation function introduced at the beginning.

In the following chapter, we first introduce an operator to interpret behavior specifications over architecture traces. Thereby, we extend our model introduced in Chap. 2 with some operators to map time points between architecture traces and corresponding projections. Then, we present our calculus in terms of 35 different rules and conclude the chapter with a brief summary.

5.1 Evaluating Behavior Trace Assertions over Architecture Traces

Evaluating behavior specifications over architecture traces requires to first extract the behavior of a certain component out of the architecture trace and evaluate it against the behavior specification using the traditional semantics of liner temporal logics [MP92]. In order to define our new evaluation operator, we first need to extend our model intro-

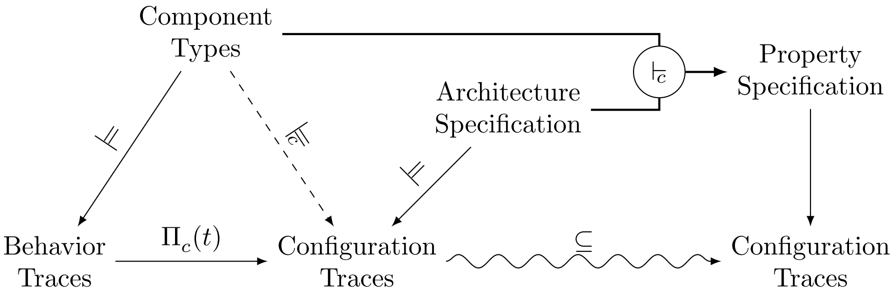


Figure 5.1: Interactive verification of ADPs.

duced in Chap. 2 with three more operators to relate the states of a component in an architecture trace with the corresponding state of the extracted behavior trace.

5.1.1 Component Activations

For the states of a component obtained through behavior projection, we can simply use the number of activations of a component to obtain the corresponding time point. If we look, for example, at Fig. 2.8, the state of the component in the resulting trace $\Pi_c(t) ^\frown t'$ at time point 2 corresponds to the state of the empty component at time point 4 in t, since it is the third activation of that component in t. This, however, corresponds to the number of activations of the empty component in t, up to time point 6 (exclusive).

In the following, we define component activation using the projection operator introduced in Chap. 2.

Definition 12 (Component activation)**.** *With* $\#_c^n(t) \in \mathbb{N}_\infty$*, we denote the number of* component activations *of a component* c *in an (possibly finite) architecture trace* t *up to (including) point in time* n*:*

$$\#_c^n(t) \overset{def}{=} \#\Pi_c(t\downarrow_n) \ . \tag{5.1}$$

Note that parameter n as well as the return value of $\#_c^n(t)$ is an element of the *extended* natural numbers $\mathbb{N}_\infty$, including ∞. Figure 5.2 lists some characteristic properties of activations and Fig. 5.3 lists some properties about the relationship of behavior projection and activations. As indicated by the small Isabelle logo on the top right, these properties are all mechanically verified in our formalization of the calculus (App. D.2).

5.1.2 Continuations

In chapter 2 we mentioned that for the case in which a component is not activated infinitely often in an architecture trace, the corresponding projection yields only a *finite* architecture trace. In order to evaluate a temporal specification over such a finite behavior trace, we search for a valid *continuation*, i.e., an arbitrary behavior trace for the component which is appended to the projection. In order to calculate the time points

Properties of activations

$$\begin{array}{rcl}
\#_c^0(t) & = & 0 \\
\#_c^n(\langle\rangle) & = & 0 \\
\#_c^n(t) & \neq & \infty \text{ [if } n \neq \infty] \\
c \overset{n}{\Leftarrow} t \leq n' & \Longrightarrow & \#_c^n(t) \leq \#_c^{n'}(t) \\
\#_c^n(t) < \#_c^{n'}(t)] & \Longrightarrow & n < n' \\
\#_c^n(t) \leq \#_c^{n'}(t) & \Longrightarrow & c \overset{n}{\Leftarrow} t \leq n' \\
\#_c^{i+1}(t) & = & \#_c^i(t) \text{ [if } i < \#t \text{ and } \neg \wr c \wr_{t(i)}] \\
\#_c^{i+1}(t) & = & \#_c^i(t) + 1 \text{ [if } i < \#t \text{ and } \wr c \wr_{t(i)}] \\
\#_c^n(t) < \#_c^{n'}(t) & \Longrightarrow & \exists n \leq \underset{.}{i} < n' \colon \wr c \wr_{t(i)} \text{ [if } n' - 1 < \#t] \\
x < \#\Pi_c(t) & \Longrightarrow & \exists n' \colon x = \#_c^{n'}(t)
\end{array}$$

Figure 5.2: Properties of activations.

Relating activation and projection

$$\begin{array}{rcl}
\#_c^n(t) & \leq & \#\Pi_c(t) \\
\#_c^n(t) & = & \#\Pi_c(t) \text{ [if } \nexists i \geq n \colon \wr c \wr_{t(i)}] \\
\Pi_c(t\downarrow_n) & = & \Pi_c(t)\downarrow_{\#_c^n(t)} \text{ [if } n < \#t] \\
\Pi_c(t)(\#_c^i(t)) & = & \mathsf{cmp}_{t(i)}^c \text{ [if } i + 1 < \#t \text{ and } \wr c \wr_{t(i)}]
\end{array}$$

Figure 5.3: Properties of projection and activations.

of a corresponding continuation, we are going to introduce two operators to map time points from an architecture trace to a corresponding behavior trace and vice versa.

Definition 13 (Architecture to behavior trace)**.** *Given an architecture trace* $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\infty}$*, a component* $c \in \mathcal{C}$*, and a time point* $n \in \mathbb{N}$ *(for architecture trace* t*). With*

$$
{}_{c}{\Downarrow}_{t}(n) \quad \overset{def}{=} \quad \#\Pi_{c}(t) - 1 + (n - \mathit{last}(c,t)) \tag{5.2}
$$

we denote the corresponding point in time for a corresponding behavior projection.

Figure 5.4 lists some properties of this mapping.

Properties of architecture to behavior trace mapping

$$
\begin{array}{rcl}
n' \geq n & \Longrightarrow & {}_{c}{\Downarrow}_{t}(n') \geq {}_{c}{\Downarrow}_{t}(n) \\
n \geq \mathit{last}(c,t) & \Longrightarrow & {}_{c}{\Downarrow}_{t}(n') > {}_{c}{\Downarrow}_{t}(n) \text{ [if and } n' > n] \\
{}_{c}{\Downarrow}_{t}(n+1) & = & {}_{c}{\Downarrow}_{t}(n) + 1 \text{ [if } n \geq \mathit{last}(c,t)] \\
{}_{c}{\Downarrow}_{t}(\mathit{last}(c,t)) & = & \#\Pi_{c}(t) - 1 \\
\Pi_{c}(t) \& t'({}_{c}{\Downarrow}_{t}(n)) & = & t'(n - \mathit{last}(c,t) - 1) \text{ [if } \exists i\colon \wr c \wr_{t(i)} \text{ and } \nexists i \geq n\colon \wr c \wr_{t(i)}]
\end{array}
$$

Figure 5.4: Properties of architecture to behavior trace mapping.

As mentioned before, the mapping has a corresponding dual which is defined in the following.

Definition 14 (Behavior to architecture trace)**.** *Given an architecture trace* $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\infty}$*, a component* $c \in \mathcal{C}$*, and a time point* $n \in \mathbb{N}$ *(for the corresponding behavior projection of* c *to* t*). With*

$$
{}_{c}{\Uparrow}_{t}(n) \quad \overset{def}{=} \quad \mathit{last}(c,t) + (n - (\#\Pi_{c}(t) - 1) \enspace , \tag{5.3}
$$

we denote the corresponding point in time for architecture trace t*.*

Figure 5.5 lists some properties of this mapping and Fig. 5.6 lists some properties of the relationship between the two mappings.

In the following, we describe how a behavior trace assertion can be interpreted over architecture traces. We can define the interpretation using component projection (Def. 9), component activation (Def. 12), and mappings between time points (Def. 13 and Def. 14).

Definition 15 (Evaluating behavior trace assertions over architecture traces)**.** *With*

$$
(t, t', n) \underset{c}{\models} \gamma \overset{def}{\Longleftrightarrow}
$$

$$
\left(\exists i \geq n\colon \wr c \wr_{t(i)} \wedge (\Pi_{c}(t) {}^\frown t', \#_{c}^{n}(t)) \models \gamma\right) \quad \vee \tag{5.4}
$$

$$
\left(\exists i\colon \wr c \wr_{t(i)} \wedge (\nexists i \geq n\colon \wr c \wr_{t(i)}) \wedge (\Pi_{c}(t) {}^\frown t', {}_{c}{\Downarrow}_{t}(n)) \models \gamma\right) \quad \vee \tag{5.5}
$$

$$
\left(\nexists i\colon \wr c \wr_{t(i)} \wedge (t', n) \models \gamma\right) \enspace , \tag{5.6}
$$

Properties of behavior to architecture trace mapping

$$\begin{aligned} n' \geq n &\implies {}_{c}{\Uparrow}_{t}(n') \geq {}_{c}{\Uparrow}_{t}(n) \\ n' > n &\implies {}_{c}{\Uparrow}_{t}(n') > {}_{c}{\Uparrow}_{t}(n) \text{ [if } n \geq \#\Pi_c(t) - 1] \\ {}_{c}{\Uparrow}_{t}(\#\Pi_c(t)) &= \mathit{last}(c,t) + 1 \text{ [if } \exists i \colon \wr c \wr_{t(i)} \text{ and } \mathit{finite}(\Pi_c(t))] \end{aligned}$$

Figure 5.5: Properties of behavior to architecture trace mapping.

Relationship between mappings

$$\begin{aligned} {}_{c}{\Uparrow}_{t}({}_{c}{\Downarrow}_{t}(n)) &= n \text{ [if } n \geq \mathit{last}(c,t)] \\ {}_{c}{\Downarrow}_{t}({}_{c}{\Uparrow}_{t}(n)) &= n \text{ [if } n \geq \#\Pi_c(t) - 1] \\ n' \geq {}_{c}{\Downarrow}_{t}(n) &\implies {}_{c}{\Uparrow}_{t}(n') \geq n \text{ [if } n \geq \mathit{last}(c,t)] \\ n' \geq {}_{c}{\Uparrow}_{t}(n) &\implies {}_{c}{\Downarrow}_{t}(n') \geq n \text{ [if } n \geq \#\Pi_c(t) - 1] \\ n < {}_{c}{\Downarrow}_{t}(n') &\implies {}_{c}{\Uparrow}_{t}(n) < n' \text{ [if } n \geq \#\Pi_c(t) - 1] \end{aligned}$$

Figure 5.6: Relationship between mappings.

we denote that architecture trace $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\infty}$ *satisfies behavior trace assertion* γ *at time point* $n \in \mathbb{N}$ *for continuation* $t' \in (\overline{\mathsf{port}(c)})^{\infty}$*. We denote with* $(t,t') \models_{c} \gamma \overset{def}{\iff} (t,t',0) \models_{c} \gamma$ *that architecture trace* t *satisfies behavior assertion* γ *for continuation* t' *and with* $t \models_{c} \gamma \overset{def}{\iff} \exists t' \in (\overline{\mathsf{port}(c)})^{\infty} \colon (t,t') \models_{c} \gamma$ *that architecture trace* t *satisfies behavior trace assertion* γ*.*

To satisfy a behavior trace assertion γ for a component c at a certain point in time n under a given continuation t', an architecture trace t is required to fulfill one of the following conditions:

- By Eq. (5.4): Component c is activated again (after time point n) and the projection to c for t fulfills γ at the point in time given by the current number of activations of c in t.
- By Eq. (5.5): Component c is activated at least once, but not again in the future and the continuation fulfills γ at the point in time resulting from the difference of the current point in time and the last activation of c.
- By Eq. (5.6): Component c is never activated and the continuation fulfills γ at point in time n.

The following proposition relates Def. 15 with our notion of composition (Def. 11).

Proposition 1. *Given a behavior specification* $(\gamma_{ct})_{ct \in \mathcal{CT}}$ *for a set of component types* $\mathcal{CT}$*, such that* $\forall ct \in \mathcal{CT}, \mathsf{bhv}(ct) \models \gamma_{ct}$*; a specification* Γ*, such that* $\mathcal{A} \models \Gamma$*; and a set of components* $(\mathcal{C}_{ct})_{ct \in \mathcal{CT}}$ *for each component type. Then, the composition of components* $\mathcal{C}$

under architecture specification $\mathcal{A}$ (as defined by Def. 11) can be derived using the newly introduced evaluation operator:

$$\bigotimes_{\mathcal{A}}(\mathcal{C}) = \left\{t \in \mathcal{A} \mid \forall ct \in \mathcal{CT},\ c \in \mathcal{C}_{ct},\ \exists t' \in (\overline{\mathsf{port}(ct)})^{\infty} \colon (t, t') \models_{c} \gamma_{ct}\right\} \ .$$

□

5.2 Rules of the Calculus

In the following, we present introduction and elimination rules for all the temporal operators introduced for the specification of component behavior.

5.2.1 Basic Logical Operators

In the following, we provide rules for the common logical operators. Essentially, for each operator we provide one introduction and one elimination rule. However, since elimination rules are symmetric to the corresponding introduction rules, in the following we list only the former type of rules. The elimination rules can be found in D.1.

NegI

$$\frac{\neg(t, t', n) \models_{c} \text{“}\gamma\text{”}}{(t, t', n) \models_{c} \text{“}\neg\gamma\text{”}}$$

ConjI

$$\frac{(t, t', n) \models_{c} \text{“}\gamma\text{”} \wedge (t, t', n) \models_{c} \text{“}\gamma'\text{”}}{(t, t', n) \models_{c} \text{“}\gamma \wedge \gamma'\text{”}}$$

DisjI

$$\frac{(t, t', n) \models_{c} \text{“}\gamma\text{”} \vee (t, t', n) \models_{c} \text{“}\gamma'\text{”}}{(t, t', n) \models_{c} \text{“}\gamma \vee \gamma'\text{”}}$$

ImpI

$$\frac{(t, t', n) \models_{c} \text{“}\gamma\text{”} \longrightarrow (t, t', n) \models_{c} \text{“}\gamma'\text{”}}{(t, t', n) \models_{c} \text{“}\gamma \longrightarrow \gamma'\text{”}}$$

AllI

$$\frac{\forall x\colon\ (t, t', n) \models_{c} \text{“}\gamma\text{”}}{(t, t', n) \models_{c} \text{“}\forall x\colon\ \gamma\text{”}}$$

ExI

$$\frac{\exists x\colon\ (t, t', n) \models_{c} \text{“}\gamma\text{”}}{(t, t', n) \models_{c} \text{“}\exists x\colon\ \gamma\text{”}}$$

The rules essentially resemble their logical counterparts. Note, however, that basic logic operators are evaluated at the current point in time, no matter whether or not the component under consideration is currently active.

5.2.2 Behavior Assertions

Next, we provide rules for the introduction and elimination of basic behavior assertions.

Introduction The first rules characterize introduction for *basic* behavior assertions. Therefore, we distinguish between three cases. First, the case in which a component is guaranteed to be eventually activated in the future:

$\mathrm{BaI}_{\mathrm{a}}$

$$\frac{\mathsf{val}(c) \cup \left(\lambda p \in \mathsf{port}(c) \colon \mathit{val}_{t(c \xrightarrow{n} t)}(c, p)\right) \ \models \ \phi}{(t, t', n) \models_{c} \phi} \ \exists i \geq n \colon \wr c \wr_{t(i)}$$

For this case, in order to show that a BA ϕ holds at time point n, we have to show that ϕ holds at the very next point in time at which component c is active.

For the case in which a component was sometimes active but is not activated again in the future, we get the following rule:

BaI$_{\mathrm{n1}}$

$$\frac{t'(n - \mathit{last}(c,t) - 1) \;\models\; \phi}{(t,t',n) \models_{\overline{c}} \phi} \; \exists i \colon \wr c \wr_{t(i)} \wedge \nexists \underset{.}{i} \geq n \colon \wr c \wr_{t(i)}$$

In order to show that BA ϕ holds at a certain point in time n, we have to show that ϕ holds for the continuation t'. Note that the corresponding time point is calculated as the difference from n to the last point in time at which component c was active in t.

Finally, we provide another rule for the case in which a component is never activated, at all:

BaI$_{\mathrm{n2}}$

$$\frac{t'(n) \;\models\; \phi}{(t,t',n) \models_{\overline{c}} \phi} \; \nexists i \colon \wr c \wr_{t(i)}$$

For such cases, BA ϕ holds at a certain point in time n when ϕ holds for t' at time point n.

Elimination Elimination for behavior assertions is actually symmetric to the corresponding introduction rules. For the sake of completeness they are provided in D.2.

5.2.3 Next

The next rules characterize introduction and elimination for the *next* operator.

Introduction We provide two different rules for introducing a next operator. The first rule describes introduction for the case in which a component is guaranteed to be eventually activated in the future:

NxtI$_{\mathrm{a}}$

$$\frac{\begin{array}{cc} \left[\exists \underset{.}{i} > c \xrightarrow{n} t \colon \wr c \wr_{t(i)}\right] & \left[\nexists \underset{.}{i} > c \xrightarrow{n} t \colon \wr c \wr_{t(i)}\right] \\ \vdots & \vdots \\ \exists \underset{.}{n}' \geq n \colon (\exists! n \leq \underset{.}{i} < n' \colon \wr c \wr_{t(i)}) \wedge (t,t',n') \models_{\overline{c}} \text{“}\gamma\text{”} & (t,t',c \xrightarrow{n} t+1) \models_{\overline{c}} \text{“}\gamma\text{”} \end{array}}{(t,t',n) \models_{\overline{c}} \text{“}\bigcirc\gamma\text{”}} \; \exists \underset{.}{i} \geq n \colon \wr c \wr_{t(i)}$$

The rule distinguishes between two cases: For the case in which the component is activated again *after* its next activation in t, we have to show that BTA $\bigcirc\gamma$ holds at some time point n' with one single activation in between n and n'. For the case in which the component is activated only once in the future, we have to show that BTA $\bigcirc\gamma$ holds at the next point in time *after* its next activation.

A second rule describes introduction of the next operator for the case in which a component is not activated again in the future:

NxtI$_\mathrm{n}$

$$\frac{(t,t',n+1)\underset{c}{\models}\text{“}\gamma\text{”}}{(t,t',n)\underset{c}{\models}\text{“}\bigcirc\gamma\text{”}}\ \nexists i \geq n\colon \wr c \wr_{t(i)}$$

In this case, the dynamic interpretation of the operator resembles its traditional one. Thus, it suffices to show that BTA γ holds for the *next* point in time $n+1$, in order to conclude that $\bigcirc\gamma$ holds at n.

Elimination In contrary to introduction, we provide three rules to eliminate a *next* operator: The first rule deals with the case in which a component is guaranteed to be activated at least twice in the future:

NxtE$_\mathrm{a1}$

$$\frac{(t,t',n)\underset{c}{\models}\text{“}\bigcirc\gamma\text{”}\quad n\leq n'\quad \exists! n\leq i<n'\colon \wr c \wr_{t(i)}}{(t,t',n')\underset{c}{\models}\text{“}\gamma\text{”}}\ \exists i>c\overset{n}{\to}t\colon \wr c \wr_{t(i)}$$

Similar to the corresponding introduction rule, this rule allows us to conclude BTA γ for each point in time n' where there is one single activation of component c in between n and n'.

For the case in which a component is activated *exactly* once in the future, we get the following rule:

NxtE$_\mathrm{a2}$

$$\frac{(t,t',n)\underset{c}{\models}\text{“}\bigcirc\gamma\text{”}}{(t,t',c\overset{n}{\to}t+1)\underset{c}{\models}\text{“}\gamma\text{”}}\ (\exists i \geq n\colon \wr c \wr_{t(i)}) \wedge \nexists i > c\overset{n}{\to}t\colon \wr c \wr_{t(i)}$$

The rule allows us to conclude γ right *after* the next activation of c in t.

If a component is not activated in the future at all, we get the following rule for eliminating a *next* operator:

NxtE$_\mathrm{n}$

$$\frac{(t,t',n)\underset{c}{\models}\text{“}\bigcirc\gamma\text{”}}{(t,t',n+1)\underset{c}{\models}\text{“}\gamma\text{”}}\ \nexists i \geq n\colon \wr c \wr_{t(i)}$$

Again, the rule resembles the traditional interpretation of next which allows us to conclude that BTA γ holds for a certain point in time $n+1$, whenever $\bigcirc\gamma$ holds at n.

5.2.4 Eventually

In the following, we provide introduction and elimination rules for the eventually operator.

Introduction Two rules characterize introduction for the eventually operator. Again, the first rule applies for the case in which the corresponding component is guaranteed to be active in the future.

EvtI$_{\text{a}}$

$$\frac{c \overset{n}{\Leftarrow} t \leq n' \quad \overset{\begin{array}{c}\left[\exists i \geq n' \colon \wr c \wr_{t(i)}\right]\\ \vdots\end{array}}{\exists c \overset{n'}{\Leftarrow} t \leq n'' \leq c \overset{n'}{\rightarrow} t \colon (t,t',n'') \underset{c}{\models} ``\gamma"} \quad \overset{\begin{array}{c}\left[\nexists i \geq n' \colon \wr c \wr_{t(i)}\right]\\ \vdots\end{array}}{(t,t',n') \underset{c}{\models} ``\gamma"}}{(t,t',n) \underset{c}{\models} ``\Diamond\gamma"} \; \exists i \geq n \colon \wr c \wr_{t(i)}$$

Similar to its traditional interpretation, the rule requires the existence of a future point in time n' for which γ holds. There are, however, some peculiarities. First, n' does not necessarily have to be in the future. Rather every point in time greater than the last activation of component c in t is allowed. Moreover, for the case in which the component is again activated after n', it suffices to show the existence of a single point in time n'' in between the last activation before n' and the next activation after n' for which γ holds. For the case in which there is no activation of the component after n', we must indeed show that γ holds for time point n'.

In the case for which there is no future activation of the component, introduction of elimination again resembles its traditional interpretation:

EvtI$_{\text{n}}$

$$\frac{n \leq n' \quad (t,t',n') \underset{c}{\models} ``\gamma"}{(t,t',n) \underset{c}{\models} ``\Diamond\gamma"} \; \nexists i \geq n \colon \wr c \wr_{t(i)}$$

Elimination Similar as for introduction, we provide two rules to eliminate an *eventually* operator. Again, the first rule applies for the case in which there is a future activation of the component:

EvtE$_{\text{a}}$

$$\frac{(t,t',n) \underset{c}{\models} ``\Diamond\gamma"}{\begin{array}{l}\exists n' \geq c \overset{n}{\rightarrow} t \colon\\ \quad (\exists i \geq n' \colon \wr c \wr_{t(i)}) \wedge (\forall c \overset{n'}{\Leftarrow} t \leq n'' \leq c \overset{n'}{\rightarrow} t \colon (t,t',n'') \underset{c}{\models} ``\gamma" \vee\\ \quad (\nexists i \geq n' \colon \wr c \wr_{t(i)}) \wedge (t,t',n') \underset{c}{\models} ``\gamma"\end{array}} \; \exists i \geq n \colon \wr c \wr_{t(i)}$$

Similar as for its traditional interpretation, the rule allows us to conclude that γ holds at some time point n' in the future. However, there are two subtleties with the dynamic interpretation: First, we can conclude that the corresponding time point n' is greater or equal to the component's next activation. Moreover, if there is an activation of component c after n', then, we can conclude that γ holds for all time points in between the components last activation and next activation.

Again, the case in which there is no future activation of the component just resembles the operator's traditional interpretation:

EvtE$_{\text{n}}$

$$\frac{(t,t',n) \underset{c}{\models} ``\Diamond\gamma"}{\exists n' \geq n \colon (t,t',n') \underset{c}{\models} ``\gamma"} \; \nexists i \geq n \colon \wr c \wr_{t(i)}$$

5.2.5 Globally

Next, we discuss introduction and elimination for the globally operator.

Introduction Similar as for the eventually operator, we provide two introduction rules for the globally operator. As usual, the first rule applies for the case in which c is activated again in the future:

GlobI$_{\mathrm{a}}$

$$\dfrac{\begin{array}{c}\begin{bmatrix}\exists i \ge n' \colon \wr c \wr_{t(i)}\\ \land\; c \xrightarrow{n} t \le n'\end{bmatrix}\\ \vdots \\ \exists c \xLeftarrow{n'} t \le n'' \le c \xrightarrow{n'} t \colon (t, t', n'') \models_{c} \text{“}\gamma\text{”}\end{array} \qquad \begin{array}{c}\begin{bmatrix}\nexists i \ge n' \colon \wr c \wr_{t(i)}\\ \land\; c \xrightarrow{n} t \le n'\end{bmatrix}\\ \vdots \\ (t, t', n') \models_{c} \text{“}\gamma\text{”}\end{array}}{(t, t', n) \models_{c} \text{“}\Box\gamma\text{”}}\; \exists i \ge n \colon \wr c \wr_{t(i)}$$

While the traditional interpretation requires that γ holds for all time points in the future, its dynamic interpretation allows us to weaken the corresponding introduction rule in two ways: First, we only need to consider time points n' after the next activation of component c. Moreover, for the case in which there exists an activation of component c after n', it suffices to show that γ holds at an arbitrary point in time between the component's last activation and its next activation.

Introducing a globally operator for the case in which a component is not activated again in the future again resembles its traditional interpretation:

GlobI$_{\mathrm{n}}$

$$\dfrac{\begin{array}{c}[n \le n']\\ \vdots \\ (t, t', n') \models_{c} \text{“}\gamma\text{”}\end{array}}{(t, t', n) \models_{c} \text{“}\Box\gamma\text{”}}\; \nexists i \ge n \colon \wr c \wr_{t(i)}$$

Elimination Rules for the elimination of a globally operator are indeed very similar to its traditional interpretation.

There is, however, a small difference for the case in which a component is active again:

GlobE$_{\mathrm{a}}$

$$\dfrac{(t, t', n) \models_{c} \text{“}\Box\gamma\text{”} \quad n' \ge c \xLeftarrow{n} t}{(t, t', n') \models_{c} \text{“}\gamma\text{”}}\; \exists i \ge n \colon \wr c \wr_{t(i)}$$

For that case, we can conclude γ for every time point after the component's last activation compared to its traditional interpretation which requires n' to be in the future.

If a component is not active in the future, the corresponding elimination rule is as expected:

GlobE$_{\mathrm{n}}$

$$\dfrac{(t, t', n) \models_{c} \text{“}\Box\gamma\text{”} \quad n' \ge n}{(t, t', n') \models_{c} \text{“}\gamma\text{”}}\; \nexists i \ge n \colon \wr c \wr_{t(i)}$$

5.2.6 Until

We conclude the presentation with introduction and elimination rules for the until operator.

Introduction We provide two rules to introduce until operators. The first rule applies for the case in which component c is activated in the future.

UntilI$_\text{a}$

$$\cfrac{c \overset{n}{\Leftarrow} t \leq n' \qquad \begin{matrix}\left[\exists \underset{\cdot}{i} \geq n' \colon \wr c \wr_{t(i)}\right] \\ \vdots \\ \textcircled{1}\end{matrix} \qquad \begin{matrix}\left[\nexists i \geq n' \colon \wr c \wr_{t(i)}\right] \\ \vdots \\ \textcircled{2}\end{matrix}}{(t, t', n) \models_{c} \text{“}\gamma' \mathcal{U}\, \gamma\text{”}} \; \exists i \geq n \colon \wr c \wr_{t(i)}$$

In order to introduce an until operator, we have to show the existence of a time point n' greater than the component's last activation, such that one of the following conditions hold.

For the case component c is activated after n', condition ① needs to be satisfied:

$$\begin{aligned}
&\exists c \overset{n'}{\Leftarrow} t \leq \underset{\cdot}{n}'' \leq c \overset{n'}{\rightarrow} t \colon (t, t', n'') \models_{c} \text{“}\gamma\text{”} \wedge \\
&\forall c \overset{n}{\rightarrow} t \leq \underset{\cdot}{n}''' < c \overset{n''}{\Leftarrow} t \colon \\
&\quad \exists c \overset{n'''}{\Leftarrow} t \leq \underset{\cdot}{n}'''' \leq c \overset{n'''}{\rightarrow} t \colon (t, t', n'''') \models_{c} \text{“}\gamma'\text{”}
\end{aligned}$$

It requires the existence of some time point n'' in between the component's last activation (before n') and next activation (after n'), such that γ holds at n''. Moreover it requires that for all time points n''' after the component's next activation (after n) and before its last activation (before n''), there exists another time point n'''' in between the last and next activation (of n'''), such that γ' holds at n'''.

For the case that there is no activation of component c after n', condition ② needs to be satisfied:

$$\begin{aligned}
&(t, t', n') \models_{c} \text{“}\gamma\text{”} \wedge \\
&\forall c \overset{n}{\rightarrow} t \leq \underset{\cdot}{n}'' < n' \colon \\
&\quad (\exists i \geq n'' \colon \wr c \wr_{t(i)}) \wedge (\exists c \overset{n''}{\Leftarrow} t \leq \underset{\cdot}{n}''' \leq c \overset{n''}{\rightarrow} t \colon (t, t', n''') \models_{c} \text{“}\gamma'\text{”}) \\
&\quad \vee (\nexists i \geq n'' \colon \wr c \wr_{t(i)}) \wedge (t, t', n'') \models_{c} \text{“}\gamma'\text{”}
\end{aligned}$$

For this case, we need to show that γ holds for n'. In addition, we need to show for n'' after the next activation of component c (after n) and before n' that one of the following two conditions hold: Either component c is activated after n'' and there exists a time point n''' for which γ' holds and which is in between the component's last activation (before n'') and its next activation (after n''). If component c is not activated after n'', it must be shown that γ' holds for n'' itself.

Introduction for until for the case in which there is no future activation of component c is similar to its traditional interpretation:

UntilI$_{\mathrm{n}}$

$$\frac{n \leq n' \quad (t,t',n')\models_{c}\text{“}\gamma\text{”} \quad \begin{array}{c}[n \leq n'' \wedge n'' < n'] \\ \vdots \\ (t,t',n'')\models_{c}\text{“}\gamma'\text{”}\end{array}}{(t,t',n)\models_{c}\text{“}\gamma' \mathcal{U}\ \gamma\text{”}}\ \nexists i \geq n\colon \wr c \wr_{t(i)}$$

Elimination Finally, we provide two rules to eliminate *until* operators. The first one is, again, the one characterizing elimination for the case in which component c is activated in the future:

UntilE$_{\mathrm{a}}$

$$\frac{(t,t',n)\models_{c}\text{“}\gamma' \mathcal{U}\ \gamma\text{”}}{\begin{array}{l}\exists \underset{\cdot}{n}' {\geq} c\overset{n}{\rightarrow}t\colon \\ (\exists i{\geq}n'\colon \wr c \wr_{t(i)}) \wedge (\forall c\overset{n'}{\Leftarrow}t {\leq} \underset{\cdot}{n}'' {\leq} c\overset{n'}{\rightarrow}t\colon (t,t',n'')\models_{c}\text{“}\gamma\text{”}) \wedge \\ \quad (\forall c\overset{n}{\Leftarrow}t {\leq} \underset{\cdot}{n}'' {<} c\overset{n'}{\Leftarrow}t\colon (t,t',n'')\models_{c}\text{“}\gamma'\text{”}) \vee \\ (\nexists i{\geq}n'\colon \wr c \wr_{t(i)}) \wedge (t,t',n')\models_{c}\text{“}\gamma\text{”} \wedge (\forall c\overset{n}{\Leftarrow}t {\leq} \underset{\cdot}{n}'' {<} n'\colon (t,t',n'')\models_{c}\text{“}\gamma'\text{”})\end{array}}\ \exists i{\geq}n\colon \wr c \wr_{t(i)}$$

Assuming that $\gamma'\ \mathcal{U}\ \gamma$ holds at some time point n, the rule allows us to conclude that there exists an n' later than the component's next activation after n for which the following conditions are satisfied: Either component c is activated after n' and γ holds for all n'' in between the component's last activation (before n') and its next activation (after n'). In addition, γ' holds for all n'' after the component's last activation (before n) and *strictly* before the component's last activation (before n'). If component c is not activated after n', we can conclude γ for time point n' and γ' for all time points n'' after the last activation (before n) and before n'.

The rule for eliminating until for the case in which component c is not activated anymore is as expected:

UntilE$_{\mathrm{n}}$

$$\frac{(t,t',n)\models_{c}\text{“}\gamma' \mathcal{U}\ \gamma\text{”}}{\begin{array}{l}\exists \underset{\cdot}{n}' \geq n\colon (t,t',n')\models_{c}\text{“}\gamma\text{”}\ \wedge \\ \forall n \leq \underset{\cdot}{n}'' < n'\colon (t,t',n'')\models_{c}\text{“}\gamma'\text{”}\end{array}}\ \nexists i \geq n\colon \wr c \wr_{t(i)}$$

5.2.7 Soundness and Completeness

In the following, we show *soundness* and *completeness* of the calculus. Thereby, we denote with $(t,t',n)\vdash_{c}\gamma$ that it is possible to derive $(t,t',n)\models_{c}\gamma$ with the rules introduced above.

Theorem 2 (Soundness)**.** *The calculus presented in this subsection is sound:*

$$(t,t',n) \vdash_{c} \gamma \qquad \Longrightarrow \qquad (t,t',n) \models_{c} \gamma\ .$$

The proof consists of soundness proofs for each of the rules and it is fully mechanized in Isabelle/HOL's structured proof language Isabelle/Isar [Wen07]. It is available in theory `Dynamic_Architecture_Calculus` at the AFP-Entry `Dynamic_Architectures` [Mar17a] and further discussed in the next section.

Theorem 3 (Relative Completeness)**.** *The calculus presented in this subsection is complete w.r.t. Def. 15:*

$$(t, t', n) \models_{c} \gamma \implies (t, t', n) \vdash_{c} \gamma \qquad .$$

The proof is done by structural induction over γ: Thus, for each operator, we assume that $\gamma = OP\gamma'$. Then, we apply Def. 15 to obtain facts about a model which satisfies γ. Finally, we use these facts to apply one of the introduction rules for OP. The detailed proof is provided in App. D.21.

5.3 Summary

Table 5.1 depicts an overview of the rules of the calculus grouped by the corresponding logical operator. For each rule it lists its type (introduction vs. elimination) and the required condition on a component's activation state to apply the rule.

	rule	*type*	*condition*
Negation	NegI	intro.	–
	NegE	elim.	–
Conjunction	ConjI	intro.	–
	ConjE	elim.	–
Disjunction	DisjI	intro.	–
	DisjE	elim.	–
Implication	ImpI	intro.	–
	ImpE	elim.	–
Existential quantification	ExI	intro.	–
	ExE	elim.	–
All quantification	AllI	intro.	–
	AllE	elim.	–
Behavior assertion	BaI$_{\mathrm{a}}$	intro.	component activated in the future
	BaI$_{\mathrm{n1}}$	intro.	component activated in the past
	BaI$_{\mathrm{n2}}$	intro.	component never activated
	BaE$_{\mathrm{a}}$	elim.	component activated in the future
	BaE$_{\mathrm{n1}}$	elim.	component activated in the past
	BaE$_{\mathrm{n2}}$	elim.	component never activated
Next	NxtI$_{\mathrm{a}}$	intro.	component activated in the future
	NxtI$_{\mathrm{n}}$	intro.	component *not* activated in the future
	NxtE$_{\mathrm{a1}}$	elim.	component activated at least twice in the future
	NxtE$_{\mathrm{a2}}$	elim.	component activated once in the future
	NxtE$_{\mathrm{n}}$	elim.	component *not* activated in the future
Eventually	EvtI$_{\mathrm{a}}$	intro.	component activated in the future
	EvtI$_{\mathrm{n}}$	intro.	component *not* activated in the future
	EvtE$_{\mathrm{a}}$	elim.	component activated in the future
	EvtE$_{\mathrm{n}}$	elim.	component *not* activated in the future
Globally	GlobI$_{\mathrm{a}}$	intro.	component activated in the future
	GlobI$_{\mathrm{n}}$	intro.	component *not* activated in the future
	GlobE$_{\mathrm{a}}$	elim.	component activated in the future
	GlobE$_{\mathrm{n}}$	elim.	component *not* activated in the future
Until	UntilI$_{\mathrm{a}}$	intro.	component activated in the future
	UntilI$_{\mathrm{n}}$	intro.	component *not* activated in the future
	UntilE$_{\mathrm{a}}$	elim.	component activated in the future
	UntilE$_{\mathrm{n}}$	elim.	component *not* activated in the future

Table 5.1: Rules to reason about component types.

6 Interactive Pattern Verification in Isabelle/HOL

So far, we presented a model for dynamic architectures and techniques to specify ADPs over this model. We even implemented these techniques as an Eclipse/EMF modeling application to support a user in the development of specifications. In the last chapter, we then presented a calculus to reason about ADPs and thus support the verification of such specifications. Until now, however, verification needs to be done using plain "pen and paper", and the correctness of it is not mechanically verified. To address this problem, we implemented our calculus in Isabelle/HOL and developed an algorithm to map a pattern specification to a corresponding Isabelle theory. The algorithm was implemented in Eclipse/EMF and can be used to automatically generate Isabelle/HOL theories from a pattern specification. A generated pattern theory is based on the formalization of the calculus to allow the rules of the calculus to be used in the development of verification proofs. Moreover, pattern theories may instantiate other pattern theories and all the verification results of an instantiated pattern are automatically available to support the verification of the instantiating pattern.

Figure 6.1 provides an overview of our formalization. It is based on Lochbihler's formalization of co-inductive lists [Loc10] and consists of two Isabelle/HOL theories which are available as entry *DynamicArchitectures* [Mar17a] in the archive of formal proofs: `Configuration_Traces` and `Dynamic_Architecture_Calculus`. To this end `Configuration_Traces` formalizes the model presented in Chap. 2. Therefore, it introduces an Isabelle locale `dynamic_component` which requires two parameters: a function *tCMP* to obtain a snapshot of a component from an architecture snapshot, and a function *active* to assert whether a certain component is active in an architecture snapshot. Then, it introduces several definitions for the locale, reflecting the definitions presented in Chap. 2. Moreover, it provides formalizations for several, characteristic properties of the defined concepts and provides proofs for them in terms of Isabelle's structured proof language Isabelle/Isar [Wen07]. Theory `Dynamic_Architecture_Calculus`, on the other hand, formalizes the calculus presented in Chap. 5. To this end, it extends locale `dynamic_component` with definitions for each operator used in the specification of behavior trace assertions, as introduced in Chap. 3. Moreover, it formalizes the evaluation operator introduced by Def. 15 in the last chapter as well as all the rules of the calculus and provides Isabelle/Isar proofs for all of them.

In the following, we first summarize Lochbihler's formalization of co-inductive lists. Then, we present the formalization of the model presented in Chap. 2 on top of co-inductive lists and summarize our formalization of the calculus presented in Chap. 5.

Finally, we present an algorithm to map a pattern specification to a corresponding Isabelle/HOL theory.

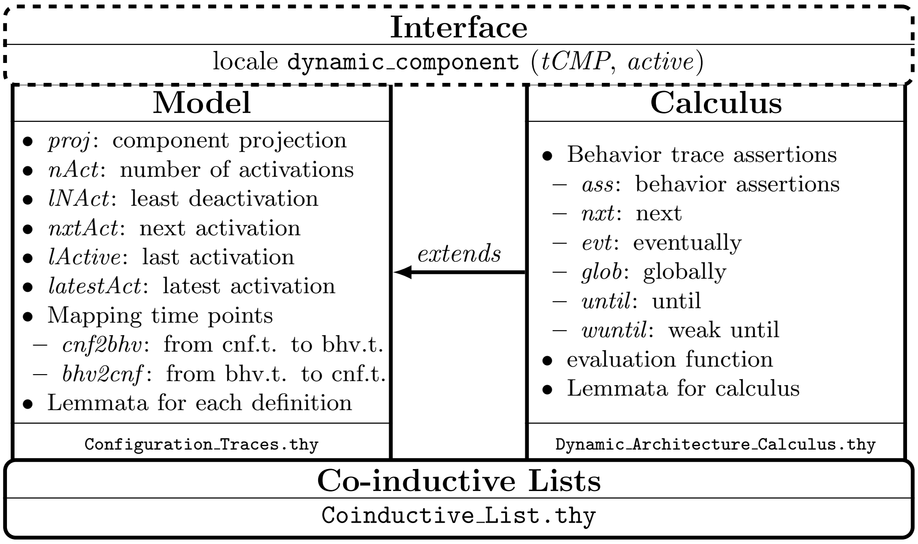


Figure 6.1: Overview of formalization in Isabelle/HOL.

6.1 Coinductive Lists

In order to deal with possibly infinite architecture traces, our formalization is based on Lochbihler's theory of coinductive (lazy) lists [Loc10]. Lazy lists are formalized using Isabelle/HOL's notion of coinductive datatypes [BHL$^+$14]. Figure 6.2 depicts the corresponding Isabelle/HOL fragment: Besides introducing the codatatype itself, the declaration also introduces some auxiliary constants:

- Destructors *LNil* and *LCons*.
- Discriminator *lnull*, to test whether a list is empty.
- Selectors *lhd* and *ltl*, to select the first element of a given list and the remaining tail, respectively.
- Set function *lset*, which returns a (possibly infinite) set containing all the elements of a given list.
- Map function *map*, to apply a given function to a certain list.
- Relator *rel*, to compare two lists based on their elements.

The where clause at the end of the command specifies a default value for selectors *lhd* and *ltl* applied to *LNil* on which they are not a priori specified.

```
codatatype (lset: 'a) llist =
    lnull: LNil
  | LCons (lhd: 'a) (ltl: 'a llist)
for
  map: lmap
  rel: llist-all2
where
  lhd LNil = undefined
| ltl LNil = LNil
```

Figure 6.2: Formalization of lazy lists in Isabelle/HOL (excerpt from [Loc10]).

In addition, Lochbihler's theory introduces formalizations of different concepts for lazy lists, of which the following are most relevant for our theory:

inf-llist converts a function with domain of natural numbers to a corresponding infinite list.

llength returns the (possible infinite) length of a list.

lnth returns the n-th element of a list.

lappend concatenates two lists.

lfilter extracts a sublist which contains only elements characterized by a given predicate.

ltake returns a prefix of a certain length of a given list.

Since *lfilter* and *ltake* are of particular importance for our theory, we discuss them in more detail.

6.1.1 Lazy Filter Function

The *lfilter* function is important, since it forms the foundation for our formalization of the behavior projection operator. The function takes a predicate P and a lazy list xs and returns a sublist, containing only those elements of xs for which P holds. Its definition is provided in Fig. 6.3: It is formalized as a recursive function based on fixpoints in complete partial orders. Note that the definition does not require any termination proof. Rather, in order to guarantee the existence of a fixpoint, the definition must ensure that the induced functional is monotonic w.r.t. the prefix order for lazy lists.

```
partial-function (llist) lfilter :: 'a llist ⇒ 'a llist
where lfilter xs = (case xs of LNil ⇒ LNil
                 | LCons x xs' ⇒ if P x then LCons x (lfilter xs') else lfilter xs')
```

Figure 6.3: Formalization of lazy filter function in Isabelle/HOL (excerpt from [Loc10]).

6.1.2 Lazy Take Function

Another important function is the *ltake* function, since it is used to formalize our notion of number of activations of a component in an architecture trace. The function takes an extended natural number n (including ∞) and a lazy list xs, and returns a sublist, containing the first n elements of xs. Figure 6.4 depicts its formalization in Isabelle/HOL: It is formalized as a primitive corecursive function, in which the syntactic structure of the definition ensures productivity (and thus well-definedness) of the function.

```
primcorec ltake :: enat ⇒ 'a llist ⇒ 'a llist
where
  n = 0 ∨ lnull xs ⟹ lnull (ltake n xs)
| lhd (ltake n xs) = lhd xs
| ltl (ltake n xs) = ltake (epred n) (ltl xs)
```

Figure 6.4: Formalization of lazy take function in Isabelle/HOL (excerpt from [Loc10]).

6.2 Formalizing Architecture Traces

In the following, we describe a possible formalization of the model presented in Chap. 2 using co-inductive lists. The following Isabelle/HOL snippet depicts the foundation of our formalization:

```
typedecl cnf
type-synonym trace = nat ⇒ cnf
consts arch:: trace set
```

First, we introduce a type constant *cnf*, which represents an architecture snapshot, i.e. the state of an architecture during system execution. An architecture trace is then formalized as a function which assigns a snapshot *cnf* to each point in time *nat*. Finally, an architecture *arch* is modeled as a set "*trace set*" of architecture traces.

As mentioned above, the interface to the model is given in terms of an Isabelle/HOL locale:

```
locale dynamic-component =
 fixes tCMP :: 'id ⇒ cnf ⇒ 'cmp (σ_(-) [0,110]60)
  and active :: 'id ⇒ cnf ⇒ bool (⌇-⌇_ [0,110]60)
```

The locale introduces two type parameters:
'id a type containing component identifiers.
'cmp a type containing component snapshots.
Moreover, it requires two function parameters:
tCMP is an operator to extract the state of a component with a certain identifier *'id* from an architecture snapshot *cnf*.
active is a predicate to assert whether a component with a certain identifier *'id* is activated within an architecture snapshot *cnf*.

The locale introduces several operators for architecture traces along with some characteristic properties thereof.

6.2.1 Behavior Projection

Perhaps the most important operator is behavior projection. Intuitively, the operator takes a component identifier c and an architecture trace t and returns a so-called *behavior trace*, i.e., a list containing all the states of component c in t. Thereby, all the time points in which component c is not activated in t are removed. The operator is formalized by combining the lazy filter function *lfilter* (described above) with the lazy map function *lmap*:

definition $proj{::}\ 'id \Rightarrow (cnf\ llist) \Rightarrow ('cmp\ llist)\ (\pi_{\_}(\text{-})\ [0{,}110]60)$
where $proj\ c = lmap\ (\lambda cnf.\ (\sigma_c(cnf))) \circ (lfilter\ (active\ c))$

First, *lfilter* is used to remove all time points in t, where c is not activated. Then, *lmap* is used to extract the state of a component out of a given architecture snapshot.

6.2.2 Number of Activations

Another useful operator for architecture traces, introduced in the last section by Def. 12, returns the number of activations of a certain component within a given architecture trace. Intuitively, the operator takes a component identifier c, a time point n, and an architecture trace t and returns the number of activations of c up to (and including) time point n. The operator is formalized by combining component projection with the lazy take function *ltake* (described above) and the lazy length function:

definition $nAct :: 'id \Rightarrow enat \Rightarrow (cnf\ llist) \Rightarrow enat\ (\langle\text{-}\ \#_{\_}\text{-}\rangle)$ **where**
$\langle c\ \#_n\ t\rangle \equiv llength\ (\pi_c(ltake\ n\ t))$

First, *ltake* is used to obtain a sublist of length n from the original architecture trace t. Then, component projection is applied to the remaining architecture trace to remove all time points in which component c is not active. What is left is a lazy list containing all the activations of c in t up to time point n and we simply return its length.

6.2.3 Least Deactivation

The following operator takes a component identifier c, an architecture trace t, and a time point n and returns the time point right after the last activation of c in t prior to n. It is introduced by Def. 8 in Chap. 2 and it is formalized using Isabelle/HOL's definite description operator $LEAST$:

definition $lNAct :: 'id \Rightarrow (nat \Rightarrow cnf) \Rightarrow nat \Rightarrow nat\ (\langle\text{-} \Leftarrow \text{-}\rangle_{\_})$
where $\langle c \Leftarrow t\rangle_n \equiv (LEAST\ n'.\ n{=}n' \lor (n'{<}n \land (\nexists k.\ k{\geq}n' \land k{<}n \land \wr c\wr_{t\ k})))$

Note that $LEAST$ returns the *least* element which satisfies a certain condition or an arbitrary element of the corresponding type if no element satisfied the condition.

6.2.4 Next Activation

Next activation is also introduced by Def. 8. It takes a component identifier c, an architecture trace t, and a time point n and returns the next point in time (including n) at which c is active in t. The formalization of this operator uses Isabelle/HOL's definite description operator *THE*:

```
definition nxtAct :: 'id ⇒ (nat ⇒ cnf) ⇒ nat ⇒ nat (⟨- → -⟩-)
 where ⟨c → t⟩n ≡ (THE n'. n'≥n ∧ ⦃c⦄t n' ∧ (∄k. k≥n ∧ k<n' ∧ ⦃c⦄t k))
```

Note that *THE* returns the *unique* element which satisfies a given condition if such an element exists or an arbitrary element of the corresponding type if no such element exists.

6.2.5 Latest Activation

In the following we describe the formalization of an operator to obtain the latest activation of a component *before* a certain point in time. Again it is introduced by Def. 8 and its formalization uses one of Isabelle/HOL's definite description operator:

```
definition latestAct :: 'id ⇒ (nat ⇒ cnf) ⇒ nat ⇒ nat (⟨- ← -⟩-)
 where latestAct c t n = (GREATEST n'. n'<n ∧ ⦃c⦄t n')
```

Note that *GREATEST* in Isabelle/HOL is the dual of *LEAST*. It returns the *greatest* element which satisfies a certain condition or an arbitrary element of the corresponding type if element satisfied the condition.

6.2.6 Last Activation

Also the last point in time at which a component is active in an architecture trace is introduced by Def. 8. It can be obtained using operator *lActive* and it is again formalized using Isabelle/HOL's *GREATEST* operator:

```
definition lActive :: 'id ⇒ (nat ⇒ cnf) ⇒ nat (⟨- ∧ -⟩)
 where ⟨c ∧ t⟩ ≡ (GREATEST i. ⦃c⦄t i)
```

6.2.7 Mapping Time Points

As discussed in the last chapter, applying behavior projection for a component c to an architecture trace t, results in a behavior trace which contains all the states of c whenever it is active in t. Thereby, the time points at which a certain state of c is available after applying projection may change (due to the deactivation of c in t). Thus, in order to map time points in between an architecture trace and the corresponding projection, we introduced two additional operators with Def. 13 and Def. 14 which are formalized as follows:

```
definition cnf2bhv :: 'id ⇒ (nat ⇒ cnf) ⇒ nat ⇒ nat (-↓-(-) [150,150,150] 110)
 where c↓t(n) ≡ the-enat(llength (πc(inf-llist t))) − 1 + (n − ⟨c ∧ t⟩)
definition bhv2cnf :: 'id ⇒ (nat ⇒ cnf) ⇒ nat ⇒ nat (-↑-(-) [150,150,150] 110)
```

```
where c↑t(n) ≡ ⟨c ∧ t⟩ + (n − (the-enat(llength (πc(inf-llist t))) − 1))
```

Note that *cnf2bhv* is used to map a given time point n for an architecture trace t to the corresponding projection $\Pi_c(t) ^\frown t'$, while *bhv2cnf* is used to map a time point n for the $\Pi_c(t) ^\frown t'$ back to the corresponding architecture trace t.

6.3 Specifying Architecture Traces

In order to specify architecture traces, we formalized our notion of architecture trace assertion (introduced in Chap. 3). To this end, we first introduced a type synonym for architecture trace assertions:

```
type-synonym cta = trace ⇒ nat ⇒ bool
```

Then, we introduced a mapping to lift an architecture assertion (Sect. 3.3.2) to a corresponding architecture trace assertion:

```
definition ca :: (cnf ⇒ bool) ⇒ cta
  where ca φ ≡ λ t n. φ (t n)
```

Finally, we defined each of the operators involved in the specification of architecture trace assertions in terms of predicate transformers, i.e., functions which take an architecture trace assertion and modify it accordingly.

6.3.1 Logical Connectives

First, we introduced definitions for the basic logical operators:

```
definition neg :: cta ⇒ cta (¬ᶜ - [19] 19)
  where ¬ᶜ γ ≡ λ t n. ¬ γ t n
definition conj :: cta ⇒ cta ⇒ cta (infixl ∧ᶜ 20)
  where γ ∧ᶜ γ' ≡ λ t n. γ t n ∧ γ' t n
definition disj :: cta ⇒ cta ⇒ cta (infixl ∨ᶜ 15)
  where γ ∨ᶜ γ' ≡ λ t n. γ t n ∨ γ' t n
definition imp :: cta ⇒ cta ⇒ cta (infixl ⟶ᶜ 10)
  where γ ⟶ᶜ γ' ≡ λ t n. γ t n ⟶ γ' t n
```

They mainly lift each corresponding HOL operator to architecture traces. In a similar way, we introduced quantifiers for architecture trace assertions:

```
definition all :: ('a ⇒ cta)
  ⇒ cta (binder ∀c 10)
  where all P ≡ λt n. (∀ y. (P y t n))
definition ex :: ('a ⇒ cta)
  ⇒ cta (binder ∃c 10)
  where ex P ≡ λt n. (∃ y. (P y t n))
```

6.3.2 Temporal Operators

Then, we introduced definitions for each temporal logic operator. Their semantics indeed resembles the traditional semantics of linear temporal logics [MP92].

Temporal logic next is implemented as a function which takes an architecture trace assertion γ and returns another architecture trace assertion which evaluates γ at the next point in time.

definition $nxt :: cta \Rightarrow cta\ (\bigcirc_c(\text{-})\ 24)$
  where $\bigcirc_c(\gamma) \equiv \lambda t\ n.\ \gamma\ t\ (Suc\ n)$

Eventually is formalized as a function which takes an architecture trace assertion γ and returns another architecture trace assertion which evaluates γ somewhere in the future:

definition $evt :: cta \Rightarrow cta\ (\Diamond_c(\text{-})\ 23)$
  where $\Diamond_c(\gamma) \equiv \lambda t\ n.\ \exists\, n' \geq n.\ \gamma\ t\ n'$

The globally operator transforms an architecture trace assertion γ to another architecture trace assertion which evaluates γ at every time in the future:

definition $glob :: cta \Rightarrow cta\ (\Box_c(\text{-})\ 22)$
  where $\Box_c(\gamma) \equiv \lambda t\ n.\ \forall\, n' \geq n.\ \gamma\ t\ n'$

Finally, until takes two architecture trace assertions γ and γ' and evaluates γ' in the future as long as γ does not hold:

definition $until :: cta \Rightarrow cta \Rightarrow cta$ $($**infixl** $\mathfrak{U}_c\ 21)$
  where $\gamma'\ \mathfrak{U}_c\ \gamma \equiv \lambda t\ n.\ \exists\, n'' \geq n.\ \gamma\ t\ n'' \wedge (\forall\, n' \geq n.\ n' < n'' \longrightarrow \gamma'\ t\ n')$

We also introduce a weaker notion of until as a combination of until and globally:

definition $wuntil :: cta \Rightarrow cta \Rightarrow cta$ $($**infixl** $\mathfrak{W}_c\ 20)$
  where $\gamma'\ \mathfrak{W}_c\ \gamma \equiv \gamma'\ \mathfrak{U}_c\ \gamma \vee^c \Box_c(\gamma')$

6.4 Formalizing the Calculus

In order to formalize the calculus presented in Chap. 5, we first formalized the notion of behavior trace assertion as described in Sect. 3.2.2.1. Then, we formalized the evaluation definition presented in Def. 15 of the last chapter. Finally, we formalized all of the rules of the calculus introduced in Chap. 5 and verified their soundness w.r.t. the introduced evaluation function.

6.4.1 Specifying Component Behavior

As described in Sect. 2, component behavior is specified using behavior trace assertions. Just as for architecture trace assertions, we start the formalization by introducing a corresponding type synonym:

type-synonym $'c\ bta = (nat \Rightarrow\ 'c) \Rightarrow nat \Rightarrow bool$

A behavior trace assertion is formalized in terms of a predicate over a behavior trace and a natural number. Thereby, the state of a component is modeled in terms of a type parameter $'c$.

Similar as for architecture trace assertions, we then introduced an operator to lift behavior assertions (Sect. 3.2.2.1) to corresponding behavior trace assertions:

definition $ba :: ('cmp \Rightarrow bool) \Rightarrow ('cmp\ bta)$
  where $ba\ \varphi \equiv \lambda\ t\ n.\ \varphi\ (t\ n)$

In addition, we also introduce an operator to lift an arbitrary HOL predicate to a corresponding behavior trace assertion:

definition $pred :: bool \Rightarrow ('cmp\ bta)$
  where $pred\ P \equiv \lambda\ t\ n.\ P$

Note that such a definition was not required for architecture trace assertions since architecture assertions can be used to lift arbitrary predicates to the level of architecture trace assertion. For behavior assertions this is not possible since they are evaluated only at time points where a component is indeed active.

Finally, we defined each of the operators used in the specification of behavior trace assertions in terms of predicate transformers, i.e., functions which take a behavior trace assertion and modify it accordingly.

6.4.1.1 Logical Connectives

Basic logical connectives are defined in a similar way as for architecture trace assertions:

definition $imp :: ('cmp\ bta) \Rightarrow ('cmp\ bta) \Rightarrow ('cmp\ bta)$ (**infixl** $\longrightarrow^b$ 10)
  where $\gamma \longrightarrow^b \gamma' \equiv \lambda\ t\ n.\ \gamma\ t\ n \longrightarrow \gamma'\ t\ n$
definition $disj :: ('cmp\ bta) \Rightarrow ('cmp\ bta) \Rightarrow ('cmp\ bta)$ (**infixl** $\vee^b$ 15)
  where $\gamma \vee^b \gamma' \equiv \lambda\ t\ n.\ \gamma\ t\ n \vee \gamma'\ t\ n$
definition $conj :: ('cmp\ bta) \Rightarrow ('cmp\ bta) \Rightarrow ('cmp\ bta)$ (**infixl** $\wedge^b$ 20)
  where $\gamma \wedge^b \gamma' \equiv \lambda\ t\ n.\ \gamma\ t\ n \wedge \gamma'\ t\ n$
definition $neg :: ('cmp\ bta) \Rightarrow ('cmp\ bta)$ ($\neg^b$ - $[19]$ 19)
  where $\neg^b\ \gamma \equiv \lambda\ t\ n.\ \neg\ \gamma\ t\ n$

Behavior assertions also support quantification over variables of a certain type. Again, Isabelle/HOL quantifiers are used to formalize quantification for behavior assertions:

definition $all :: ('a \Rightarrow ('cmp\ bta))$
  $\Rightarrow ('cmp\ bta)$ (**binder** $\forall_b$ 10)
  where $all\ P \equiv \lambda t\ n.\ (\forall\, y.\ (P\ y\ t\ n))$
definition $ex :: ('a \Rightarrow ('cmp\ bta))$
  $\Rightarrow ('cmp\ bta)$ (**binder** $\exists_b$ 10)
  where $ex\ P \equiv \lambda t\ n.\ (\exists\, y.\ (P\ y\ t\ n))$

6.4.1.2 Temporal Operators

Similar as for architecture trace assertions, we formalize temporal operators for behavior trace assertions using their traditional semantics [MP92]:

definition $nxt :: ('cmp\ bta) \Rightarrow ('cmp\ bta)\ (\bigcirc_b(\text{-})\ 24)$
  where $\bigcirc_b(\gamma) \equiv \lambda\ t\ n.\ \gamma\ t\ (Suc\ n)$
definition $evt :: ('cmp\ bta) \Rightarrow ('cmp\ bta)\ (\Diamond_b(\text{-})\ 23)$
  where $\Diamond_b(\gamma) \equiv \lambda\ t\ n.\ \exists\, n' \geq n.\ \gamma\ t\ n'$
definition $glob :: ('cmp\ bta) \Rightarrow ('cmp\ bta)\ (\Box_b(\text{-})\ 22)$
  where $\Box_b(\gamma) \equiv \lambda\ t\ n.\ \forall\, n' \geq n.\ \gamma\ t\ n'$
definition $until :: ('cmp\ bta) \Rightarrow ('cmp\ bta) \Rightarrow ('cmp\ bta)$ (**infixl** $\mathfrak{U}_b\ 21$)
  where $\gamma'\ \mathfrak{U}_b\ \gamma \equiv \lambda\ t\ n.\ \exists\, n'' \geq n.\ \gamma\ t\ n'' \wedge (\forall\, n' \geq n.\ n' < n'' \longrightarrow \gamma'\ t\ n')$
definition $wuntil :: ('cmp\ bta) \Rightarrow ('cmp\ bta) \Rightarrow ('cmp\ bta)$ (**infixl** $\mathfrak{W}_b\ 20$)
  where $\gamma'\ \mathfrak{W}_b\ \gamma \equiv \gamma'\ \mathfrak{U}_b\ \gamma \vee^b \Box_b(\gamma')$

6.4.2 Evaluation

Remember that the specification of component behavior is given in terms of behavior trace assertions, i.e., temporal logic assertions over sequences of component snapshots. As already discussed at the beginning of the previous chapter, in order to evaluate such specifications over architecture traces, we need to define how a behavior trace assertion is to be interpreted over an architecture trace in which components are subject to activation and deactivation. Therefore, we presented an alternative evaluation function (Def. 15) which allows to interpret a given behavior trace assertion over an architecture trace instead of a behavior trace. The corresponding formalization is provided by function *eval* which takes a component identifier *'id* and a behavior trace assertion and transforms it to a corresponding architecture trace assertion:

definition $eval :: 'id \Rightarrow (nat \Rightarrow cnf) \Rightarrow (nat \Rightarrow 'cmp) \Rightarrow nat$
 $\Rightarrow ((nat \Rightarrow 'cmp) \Rightarrow nat \Rightarrow bool) \Rightarrow bool$
 where $eval\ cid\ t\ t'\ n\ \gamma \equiv$
 $(\exists\, i \geq n.\ \wr cid \wr_{t\ i}) \wedge$
 $\gamma(lnth\ ((\pi_{cid}(inf\text{-}llist\ t))@_l(inf\text{-}llist\ t')))(the\text{-}enat(\langle cid\ \#_n\ inf\text{-}llist\ t\rangle)) \vee$
 $(\exists\, i.\ \wr cid \wr_{t\ i}) \wedge$
 $(\nexists\, i'.\ i' \geq n \wedge \wr cid \wr_{t\ i'}) \wedge \gamma(lnth\ ((\pi_{cid}(inf\text{-}llist\ t))@_l(inf\text{-}llist\ t')))({}_{cid}{\downarrow}_t(n)) \vee$
 $(\nexists\, i.\ \wr cid \wr_{t\ i}) \wedge \gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))@_l(inf\text{-}llist\ t')))\ n$

In order to evaluate a behavior trace assertion γ over an architecture trace t at time point n, *eval* distinguishes between three cases:

- If component *cid* is again activated in the future, γ is evaluated at the next point in time where *cid* is active in t.
- If component *cid* is not again activated in the future but it is activated at least once in t, then γ is evaluated at the point in time given by the corresponding time mapping.
- If component *cid* is never active in t, then γ is evaluated at time point n.

6.4.3 Rules of the Calculus

We can now use the evaluation function introduced above, to formalize all the rules of the calculus presented in the last chapter. Since the calculus was already discussed in the previous chapter, we are not going to discuss all of it again in this chapter. However, we want to point out that instantiating locale *dynamic-component* for the different types of components involved in an ADP results in an instantiation of all the rules of the calculus for components of that type. This way, each component type comes with its own version of the calculus which can then be used to reason about the behavior of components of that type. The following excerpt shows, for example, the formalization of the introduction rule for the next operator in the case that there exists a future activation of a component:

lemma *nxtIA*[*intro*]:
  fixes $c{::}'id$
    and $t{::}nat \Rightarrow cnf$
    and $t'{::}nat \Rightarrow 'cmp$
    and $n{::}nat$
  assumes $\exists\, i{\geq}n.\ \wr c \wr_{t\ i}$
    and $\llbracket \exists\, i{>}\langle c \rightarrow t\rangle_n.\ \wr c \wr_{t\ i} \rrbracket \Longrightarrow \exists\, n' \geq n.\ (\exists!i.\ n{\leq}i \wedge i{<}n' \wedge \wr c \wr_{t\ i}) \wedge eval\ c\ t\ t'\ n'\ \gamma$
    and $\llbracket \neg(\exists\, i{>}\langle c \rightarrow t\rangle_n.\ \wr c \wr_{t\ i}) \rrbracket \Longrightarrow eval\ c\ t\ t'\ (Suc\ \langle c \rightarrow t\rangle_n)\ \gamma$
  shows $eval\ c\ t\ t'\ n\ (\bigcirc_b(\gamma))$

Mechanized proofs of this and all remaining rules are provided in App. D.2 and available at the archive of formal proofs [Mar17a].

6.5 Creating Pattern Theories

As mentioned at the beginning, the formalization of the model, as presented in this section, can be used to support the interactive verification of ADPs. To this end, a pattern specification (in terms of the techniques presented in Sect. 3) can be systematically transferred to a corresponding Isabelle/HOL theory. Algorithm 1 describes the mapping in more detail. In general, the transformation is done in four main steps:

1. The specified FACTum datatypes are transferred to corresponding Isabelle/HOL datatypes.
2. An Isabelle locale is created for the corresponding pattern, which imports other locales for each instantiated pattern (or locale *dynamic_component* for each type of component which does not instantiate any component type from another pattern). Ports for component types are added as locale parameters and typed by the corresponding Isabelle/HOL datatypes.
3. Specifications of component behavior are added as locale assumptions, formulated in terms of behavior trace assertions (as formalized in Sect. 6.4.1), and evaluated using the evaluation function introduced in Sect. 6.4.2.
4. Activation and connection assertions are provided as locale assumptions, formulated in terms of architecture trace assertions, formalized in Sect. 6.3.

The following result guarantees soundness of Alg. 1, i.e., that the algorithm indeed preserves the semantics of a FACTum specification.

Algorithm 1 Mapping a pattern specification to an Isabelle/HOL Theory.

```
Input: A FACTum specification of ADP
    {with Datatype Spec. $DS$, Component Type Spec. $CS$, and Architecture Spec. $AS$}
Output: An Isabelle/HOL theory for the specification
 1: for all Datatypes $dt$ in $DS$ do
 2:     create Isabelle/HOL datatype specification for $dt$
 3: end for
 4: create Isabelle/HOL locale for the pattern
 5: for all Component Types $ct$ in $CS$ do
 6:     if $ct$ instantiates a component type of another pattern specification $PS$ then
 7:         import the corresponding locale for $PS$
            {requires to import the corresponing Isabelle theory}
 8:         create instance of ports/parameters according to the specified port mapping
            {the parameter for every port of $ct$ is passed to the imported locale}
 9:     else
10:         import locale “dynamic_component” of theory “Configuration_Traces”
11:     end if
12:     create instance of locale parameters $tCMP$ and $active$
13:     for all component parameters $p$ of $ct$ which are not instances do
14:         create locale parameter $par$ of the type corresponding to the type of $p$
15:         create locale assumption “$\forall x.\ \exists c.\ par(c) = x$”
            {since FACTum requires nonempty sets of components for each type}
16:     end for
        {instantiated parameters are already handled at line 8}
17:     for all ports $p$ which are not instances do
18:         create locale parameter of the type corresponding to the datatype of $p$
19:     end for
        {instantiated ports are already handled at line 8}
20:     for all behavior trace assertions $b$ of $ct$ do
21:         create locale assumption for $b$
            {use the operators and evaluation function presented in this chapter}
22:     end for
23: end for
24: for all architecture trace assertions $c$ of $AS$ do
25:     create locale assertion for $c$
        {use the operators presented in this chapter}
26: end for
```

Theorem 4 (Soundness of Alg. 1)**.** *A set of architecture traces satisfies a* FACTum *specification iff it satisfies the theory generated from the* FACTum *specification by algorithm 1.*

Although a *formal* proof for this theorem is out of the scope of this text, we provide an informal argument for it in App. E. Moreover, note that the generated theory is based on

Isabelle/HOL's implementation of architecture traces presented in this chapter. Thus, a calculus is instantiated for each component type which provides a set of rules to reason about the specification of the behavior of components of that type.

6.6 Summary

Figure 6.5 provides an overview of the results presented in this chapter. First, we provide formalizations of the model presented in Chap. 2 as well as the calculus presented in Chap. 5 in terms of two Isabelle/HOL theories which are available through the archive of formal proofs [Mar17a]:

`Configuration_Traces` imports theory `Coinductive_List` and provides a formalization of the model described in Chap. 2 in terms of co-inductive lists.

`Dynamic_Architecture_Calculus` imports theory `Configuration_Traces`, provides operators for the specification of component behavior, and implements a calculus to reason about component behavior [Mar17c] specified using these operators.

Moreover, we provide an interface to these theories in terms of an Isabelle/HOL locale [Bal04] `dynamic_component`. Finally, we provide an algorithm to map a FACTUM specification to a corresponding Isabelle/HOL theory: Thereby, locale `dynamic_component` is instantiated for every type of component involved in the pattern. Then, the behavior of each component type is specified using the specification operators provided by the corresponding instantiation. Moreover, activation and connection constraints are specified for components of the different types.

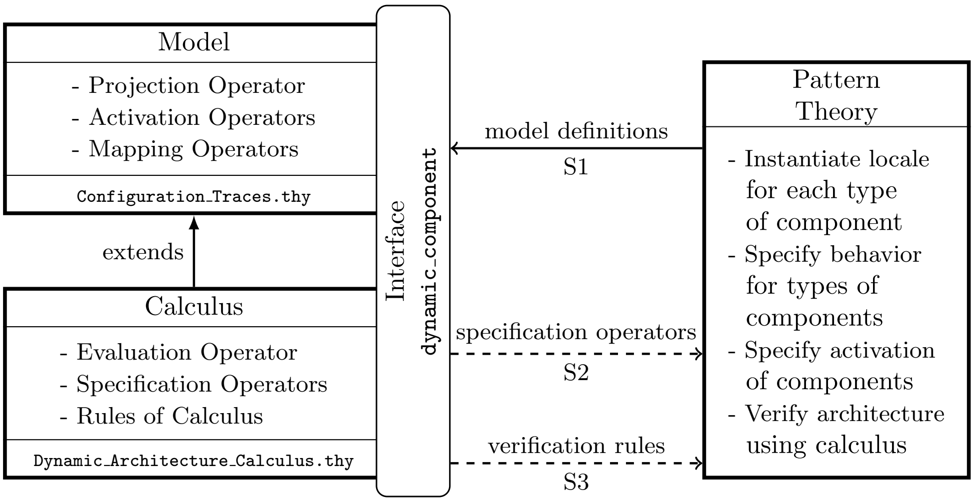


Figure 6.5: Overview of results presented in this chapter.

Part IV

Evaluation

7 Singletons, Publisher-Subscribers, and Blackboards

In the last chapters, we introduced techniques to specify ADPs and verify them by means of interactive theorem proving. In the following chapter, we demonstrate feasibility of the approach. To this end, we first specify properties for each of our running examples and verify them in Isabelle/HOL.

Verification is based on our implementation of the calculus presented in Chap. 5. Moreover, we leverage the hierarchical nature of the specifications of our running examples. Thus, results obtained for the Singleton pattern are used for the verification of the Publisher-Subscriber pattern. In addition, results obtained for both patterns, the Singleton and the Publisher-Subscriber pattern, are used for the verification of the Blackboard pattern. In total, verification consists of three theories amounting up to almost 1000 lines of Isabelle/HOL proof code.

Although the examples presented in this chapter demonstrate feasibility of the approach, their main purpose is to demonstrate the concepts and ideas of this thesis. To further evaluate the methodology on a larger scale, the next chapter presents a case study from the domain of blockchain architectures.

7.1 Singleton

First, we discuss verification of the Singleton pattern. We first present a possible guarantee for the pattern. Then, we show the Isabelle/HOL code generated from the specification of the pattern. Finally, we show how the guarantee is verified by proving the corresponding theorem.

7.1.1 Architectural Guarantee

One possible guarantee of a Singleton is that there exists indeed a *unique* component of type singleton which is always active. It it is formalized in Fig. 7.1 in terms of an architecture trace assertion (as introduced in Sect. 3.3). First, we specify two component variables for singletons: a flexible variable s and a rigid variable *the-singleton*. While flexible variables may change their value at each point in time, rigid variables are required to keep their value during the whole execution. Then, we require the existence of such a rigid singleton, which is always activated and indeed the only component of type singleton which is active at any point in time.

ASpec Guarantee_Singleton	**for** Singleton
flex *s*:	*Singleton*
rig *the-singleton*:	*Singleton*
$\exists$*the-singleton*: $\square(\wr \mathit{the\text{-}singleton} \wr \land (\wr s \wr \longrightarrow s = \mathit{the\text{-}singleton}))$	

Figure 7.1: Architectural guarantee for the Singleton pattern.

7.1.2 Mapping the Pattern Specification

Since the specification of the Singleton did not involve the specification of data types, no Isabelle datatype specification is created. However, a corresponding Isabelle locale is created from the specification of a Singleton's interface (discussed in Ex. 16). The corresponding Isabelle/HOL excerpt looks as follows:

```
locale singleton = dynamic-component cmp active
  for active :: 'id ⇒ cnf ⇒ bool (≀-≀_ [0,110]60)
  and cmp :: 'id ⇒ cnf ⇒ 'cmp (σ_(-) [0,110]60) +
```

In order to use our verification framework later on, locale *dynamic-component* must be instantiated for each type of component involved in a pattern's specification. Since the Singleton pattern consists of only one component type, a single instantiation of locale *dynamic-component* is generated. The locale requires two parameters:

- A mapping *cmp* to access a singleton component in a given architecture snapshot based on its identifier.
- A predicate *active* which checks whether a singleton component with a certain identifier is activated in an architecture snapshot.

Moreover, the specification of the singleton consists of two architectural assumptions (as described in Sec. 21), which are transferred to corresponding locale assumptions:

```
assumes alwaysActive: ⋀k. ∃ id. ≀id≀k
  and unique: ∃ id. ∀ k. ∀ id'. (≀id'≀k ⟶ id = id')
```

Locale assumption *alwaysActive* is generated from the singleton's assumption that a singleton component is always active. Locale assumption *unique* is created from the assumption that a singleton component is indeed unique.

The architectural guarantee specified for a singleton (Sec. 7.1.1) is systematically transferred to the following Isabelle/HOL theorem:

```
definition the-singleton ≡ THE id. ∀ k. ∀ id'. ≀id'≀k ⟶ id' = id

theorem ts-prop:
  fixes k::cnf
  shows ⋀id. ≀id≀k ⟹ id = the-singleton
    and ≀the-singleton≀k
```

First, Isabelle/HOL's definite description operator *THE* is used to define the *unique* singleton component. Then, a theorem is created which guarantees that a singleton is indeed unique and always activated.

7.1.3 Verification

Since the singleton is declared to be an instance of locale *dynamic-component*, all the rules of our calculus are available for singleton components and can be used for the verification of the pattern.

7.1.4 Properties from the Calculus

Figure 7.2 demonstrates the properties which are available for a singleton as part of the framework. Essentially, we get introduction and elimination rules for each of the operators used to specify component behavior. Since a singleton does not have any behavior specification, these rules are actually not used for the verification of the pattern. However, as we consider more complicated patterns, these rules turn out to be useful for verification.

baIA: $[\![\exists\, i \geq n.\ \wr c \wr_{t\ i};\ \varphi\ (\sigma_c t\ \langle c \rightarrow t\rangle_n)]\!] \Longrightarrow eval\ c\ t\ t'\ n\ (ba\ \varphi)$

baIN1: $[\![\exists\, i.\ \wr c \wr_{t\ i};\ \neg\ (\exists\, i \geq n.\ \wr c \wr_{t\ i});\ \varphi\ (t'\ (n - \langle c \wedge t\rangle - 1))]\!]$
$\Longrightarrow eval\ c\ t\ t'\ n\ (ba\ \varphi)$

baIN2: $[\![\nexists\, i.\ \wr c \wr_{t\ i};\ \varphi\ (t'\ n)]\!] \Longrightarrow eval\ c\ t\ t'\ n\ (ba\ \varphi)$

. . . Similar rules are available for each operator

Figure 7.2: Calculus instantiated for the Singleton pattern.

7.1.5 Proving the Theorem

A possible proof for theorem *ts-prop*, presented in Sec. 7.1.2, may look as follows:

```
proof –
  { fix id
    assume a1: ≀id≀_k
    have (THE id. ∀ k. ∀ id'. ≀id'≀_k ⟶ id' = id) = id
    proof (rule the-equality)
      show ∀ k id'. ≀id'≀_k ⟶ id' = id
      proof
        fix k show ∀ id'. ≀id'≀_k ⟶ id' = id
        proof
          fix id' show ≀id'≀_k ⟶ id' = id
          proof
            assume ≀id'≀_k
            from unique have ∃ id. ∀ k. ∀ id'. (≀id'≀_k ⟶ id = id') .
```

```
          then obtain i'' where ∀ k. ∀ id'. (≀id'≀_k ⟶ i'' = id') by auto
          with ‹≀id'≀_k› have id=i'' and id'=i'' using a1 by auto
          thus id' = id by simp
        qed
      qed
    qed
  next
    fix i'' show ∀ k id'. ≀id'≀_k ⟶ id' = i'' ⟹ i'' = id using a1 by auto
  qed
  hence ≀id≀_k ⟹ id = the-singleton by (simp add: the-singleton-def)
} note g1 = this
thus ⋀id. ≀id≀_k ⟹ id = the-singleton by simp

from alwaysActive obtain id where ≀id≀_k by blast
with g1 have id = the-singleton by simp
with ‹≀id≀_k› show ≀the-singleton≀_k by simp
qed
```

The proof is formulated in Isabelle's structured proof language Isabelle/Isar and resembles a normal, mathematical proof. Note, however, the reference to the two assumptions *unique* and *alwaysActive* generated from the pattern's imposed assumptions and discussed in Sec. 7.1.2.

7.2 Publisher-Subscriber

Next, we discuss the verification of the Publisher-Subscriber pattern. Again, we first present a possible guarantee for such architectures. Then, we discuss the Isabelle/HOL code generated from its specification. Finally, we discuss the verification of the corresponding theorem in Isabelle/HOL.

7.2.1 Architectural Guarantees

Since the publisher component was specified to be an instance of the Singleton pattern, the corresponding guarantee of the Singleton pattern is also available for the Publisher-Subscriber pattern. Moreover, we can use the additional assumptions imposed by the specification of the Publisher-Subscriber pattern to come up with another guarantee for the pattern. It is specified in Fig. 7.3 and guarantees that a subscriber component indeed receives all the messages for which it is subscribed.

7.2.2 Mapping Data Types

Since the specification of the Publisher-Subscriber pattern indeed contains specifications for data types, we first need to create the corresponding datatype specification in Isabelle/HOL:

```
datatype 'evt subscription = sub 'evt | unsub 'evt
```

ASpec Publisher-Subscriber **for** Publisher-Subscriber

flex *the-pb*: *Publisher*
m: `msg`
E: $\wp($`evt`$)$
rig *s'*: *Subscriber*
e: `evt`

$$\Box\Bigg(\wr s'\wr \wedge (\exists E\colon\ \text{sub } E = s'.sb \wedge e \in E)$$
$$\longrightarrow \Big(\big(\wr s'\wr \wedge (e,m) = \textit{the-pb.nt} \longrightarrow (e,m) = s'.nt\big)$$
$$\mathcal{W}\ \big(\wr s'\wr \wedge (\exists E\colon\ \text{unsub } E = s'.sb \wedge e \in E)\big)\Big)\Bigg)$$

Figure 7.3: Architectural guarantee for the Publisher-Subscriber pattern.

According to the datatype specification presented in Ex. 12, we create a parametric datatype *subscription*, which depends on a type parameter *'evt* to denote events for which subscribers can subscribe. Thereby, the elements of a subscription are defined to be either a subscription *sub* to an event *'evt*, or an unsubscription *unsub* for an event *'evt*.

7.2.3 Mapping Architectural Assumptions

The specification of the patterns interfaces (Ex. 28) are again mapped to a corresponding locale specification:

```
locale publisher-subscriber =
  pb: singleton pbactive pbcmp +
  sb: dynamic-component sbcmp sbactive
    for pbactive :: 'pid ⇒ cnf ⇒ bool
    and pbcmp :: 'pid ⇒ cnf ⇒ 'PB
    and sbactive :: 'sid ⇒ cnf ⇒ bool
    and sbcmp :: 'sid ⇒ cnf ⇒ 'SB +
```

This time, however, two interfaces are specified which requires two instantiations for the locale. Since the publisher component type is specified to be a instance of the Singleton pattern, a corresponding instantiation of the singleton locale is created. The subscriber component type does not instantiate any other component type which is why it instantiates the default locale *dynamic-component* from our framework. Note that locale instantiations are indeed transitive which means that, implicitly, also the publisher component type instantiates locale *dynamic-component*. Thus, the verification framework can also be used for publisher components, although they do not directly instantiate *dynamic-component*.

In contrast to the singleton component type, which has no specified ports, publishers as well as subscribers have ports specified for their interfaces. The port types specified in Ex. 14 are mapped to corresponding locale parameters:

```
 fixes pbsb :: 'PB ⇒ ('evt set) subscription set
   and pbnt :: 'PB ⇒ ('evt × 'msg)
   and sbnt :: 'SB ⇒ ('evt × 'msg) set
   and sbsb :: 'SB ⇒ ('evt set) subscription
```

For each port, we create a locale parameter which takes a component of the corresponding component type and returns a set of messages of the corresponding port type.

Finally, the two connection assumptions specified for the Publisher-Subscriber pattern in Ex. 22, are mapped to corresponding locale assumptions:

```
 assumes conn1: ⋀k pid. pbactive pid k
     ⟹ pbsb (pbcmp pid k) = (⋃sid∈{sid. sbactive sid k}. {sbsb (sbcmp sid k)})
 and conn2: ⋀t n n'' sid pid E e m.
   ⟦t ∈ arch; pbactive pid (t n); sbactive sid (t n); n''≥ n; e ∈ E;
   sub E = sbsb (sbcmp sid (t n));
   ∄n' E'. n' ≥ n ∧ n' ≤ n'' ∧ sbactive sid (t n') ∧
     unsub E' = sbsb (sbcmp sid (t n')) ∧ e ∈ E';
   (e, m) = pbnt (pbcmp pid (t n'')); sbactive sid (t n'')⟧
     ⟹ pbnt (pbcmp pid (t n'')) ∈ sbnt (sbcmp sid (t n''))
```

Thereby, connections between two ports is simply mapped to an equality assumption for the corresponding locale parameters. *conn1*, for example, denotes the constraint that port *sb* of a publisher component is connected to port *sb* of every active subscriber component.

7.2.4 Mapping the Guarantee

Similar as for the Singleton pattern, the architectural guarantee for Publisher-Subscriber architectures (specified in Sec. 7.2.1), is mapped to a corresponding Isabelle/HOL theorem. First, however, we introduce an abbreviation for the *unique* publisher component inherited from the singleton:

```
abbreviation the-pb :: 'pid where
the-pb ≡ pb.the-singleton
```

Then, we can finally generate the corresponding theorem:

```
theorem msgDelivery:
  fixes t n n'' sid E e m
  assumes t ∈ arch
    and sbactive sid (t n)
    and sub E = sbsb (sbcmp sid (t n))
    and n'' ≥ n
    and ∄n' E'. n' ≥ n ∧ n' ≤ n'' ∧ sbactive sid (t n') ∧ unsub E' = sbsb(sbcmp sid (t n'))
          ∧ e ∈ E'
    and e ∈ E
    and (e,m) = pbnt (pbcmp the-pb (t n''))
    and sbactive sid (t n'')
  shows (e,m) ∈ sbnt (sbcmp sid (t n''))
```

7.2.5 Publisher-Subscriber

Similar as for the Singleton pattern, our framework provides us with rules to support reasoning about component behavior. Moreover, since a publisher was declared to be an instance of a singleton, results from the Singleton pattern propagate to publishers.

7.2.5.1 Properties from the Calculus

In contrast to the Singleton pattern, there are two types of components in a Publisher-Subscriber pattern and each of these types, publishers as well as subscribers, come with their own instantiation of the calculus. Figure 7.4 depicts two of the introduction rules for basic behavior assertions which are available: one for publisher components and one for subscriber components. Note that the rules are similar, but for publisher components we use activation and selection parameters *pbactive* and *pbcmp*, while for subscribers we use *sbactive* and *sbcmp*, respectively.

pb.baIA: $[\![\exists\, i{\geq}n.\ pbactive\ c\ (t\ i);\ \varphi\ (pbcmp\ c\ (t\ (pb.nxtAct\ c\ t\ n)))]\!]$
$\Longrightarrow pb.eval\ c\ t\ t'\ n\ (pb.ba\ \varphi)$

sb.baIA: $[\![\exists\, i{\geq}n.\ sbactive\ c\ (t\ i);\ \varphi\ (sbcmp\ c\ (t\ (sb.nxtAct\ c\ t\ n)))]\!]$
$\Longrightarrow sb.eval\ c\ t\ t'\ n\ (sb.ba\ \varphi)$

. . . Similar rules are available for each operator

Figure 7.4: Calculus instantiated for the Publisher-Subscriber pattern.

7.2.5.2 Results from Pattern Instantiations

Since the Publisher-Subscriber pattern instantiates the Singleton pattern, results obtained for the singleton are automatically interpreted in the context of the Publisher-Subscriber pattern. Thus, declaring the publisher to be an instance of a singleton has two major consequences.

First, a corresponding definition of the *unique* publisher component is available:

abbreviation *the-pb* :: *'pid* **where**
the-pb ≡ *pb.the-singleton*

Essentially, *the-pb* abbreviates definition *the-singleton* introduced in Sec. 7.1.2.

Moreover, the theorem proved for singleton components is available also for components of type publisher:

pb.ts-prop (1): pbactive id k $\Longrightarrow$ *id = the-pb*
pb.ts-prop (2): pbactive the-pb k

7.2.5.3 Proving the Theorem

The proof for theorem *msgDelivery*, presented in Sec. 7.2.4 is a simple one-liner:

```
using conn1[OF pb.ts-prop(2)] .
```

It follows directly from the assumptions generated for the pattern and the guarantee inherited from the singleton.

7.3 Blackboard

Finally, we present the verification for the Blackboard pattern. Again, we first specify a possible guarantee for the pattern. Then, we present the corresponding Isabelle/HOL code and the proof of the theorem generated from the pattern's guarantee.

7.3.1 Architectural Guarantees

Again, the architectural guarantee specified for singletons (Fig. 7.1) as well as the guarantee specified for Publisher-Subscriber architectures (Fig. 7.3) are inherited for the blackboard specification. Figure 7.5 provides the specification of an additional guarantee for Blackboard architectures: If for every open subproblem, a knowledge source able to solve this problem is eventually activated (Eq. (7.1)), then, the architecture will eventually solve a given problem (Eq. (7.2)), even if no single knowledge source is able to solve the problem on its own. Note that the specification uses the concept of parametrized variables for knowledge sources. Thus, given a problem p, variable $ks_{\langle p\rangle}$ denotes a variable for a knowledge source component which is indeed able to solve problem p.

ASpec Guarantee_Blackboard **for** Blackboard

flex	*the-bb*:	BB
	ks:	KS
	P:	$\wp(\mathtt{PROB})$
rig	p:	$\mathtt{PROB}$

$$\Big(\Box(\forall p = \textit{the-bb.op}\colon \Diamond \wr ks_{\langle p\rangle} \wr)\Big) \longrightarrow \tag{7.1}$$

$$\Box\Big(\forall P\colon \big(\mathtt{sub}\ P \in \textit{the-bb.rp} \longrightarrow \forall p \in P\colon \Diamond\big((p, \textit{solve}(p)) \in \textit{the-bb.cs}\big)\Big) \tag{7.2}$$

Figure 7.5: Architectural guarantee for the Blackboard pattern.

7.3.2 Mapping Data Types

In contrast to the Publisher-Subscriber pattern, data types for the Blackboard pattern are specified axiomatically:

```
typedecl PROB
consts sb :: (PROB × PROB) set
axiomatization where sbWF: wf sb
```

```
typedecl SOL
consts solve:: PROB ⇒ SOL
```

As required by the corresponding FACTUM datatype specification (Ex. 13), the specification introduces two abstract types: problems and solutions. It then requires the existence of a well-founded relation *sb* which relates problems with corresponding subproblems. Finally, it requires the existence of a mapping which is assumed to assign the correct solution to each problem.

7.3.3 Mapping Architectural Assumptions

Again, the pattern specification is mapped to a corresponding Isabelle/HOL locale. Similar as for the Publisher-Subscriber pattern, the pattern's interfaces are used to generate a corresponding locale header:

```
locale blackboard = publisher-subscriber bbactive bbcmp ksactive kscmp bbrp bbcs kscs ksrp
  for bbactive :: 'bid ⇒ cnf ⇒ bool (≀-≀_ [0,110]60)
    and bbcmp :: 'bid ⇒ cnf ⇒ 'BB (σ_(-) [0,110]60)
    and ksactive :: 'kid ⇒ cnf ⇒ bool (≀-≀_ [0,110]60)
    and kscmp :: 'kid ⇒ cnf ⇒ 'KS (σ_(-) [0,110]60)
    and bbrp :: 'BB ⇒ (PROB set) subscription set
    and bbcs :: 'BB ⇒ (PROB × SOL)
    and kscs :: 'KS ⇒ (PROB × SOL) set
    and ksrp :: 'KS ⇒ (PROB set) subscription +
  fixes bbns :: 'BB ⇒ (PROB × SOL) set
    and ksns :: 'KS ⇒ (PROB × SOL)
    and bbop :: 'BB ⇒ PROB
    and ksop :: 'KS ⇒ PROB set
```

Since the Blackboard is specified to be an instance of the Publisher-Subscriber pattern, the locale created for the Blackboard pattern instantiates the locale of the Publisher-Subscriber pattern. The instantiation requires 8 parameters: The first 4 parameters are the usual parameters required by the framework to obtain a component from an architecture snapshot, and to check activation of a component within an architecture snapshot. In addition, we must provide an additional parameter for each port available in the specification of the Publisher-Subscriber pattern. These parameters denote ports of the Blackboard pattern which interpret the corresponding ports of the Publisher-Subscriber pattern (as specified in Ex. 29). For example, port *rp* of a blackboard corresponds to port *sb* of a publisher, port *cs* of a blackboard to port *nt* of the publisher, port *cs* of a knowledge source to port *nt* of a subscriber, and port *rp* of a knowledge source to port *sb* of a subscriber.

As a next step, we generate interface parameters and corresponding assumptions:

```
    and prob :: 'kid ⇒ PROB
  assumes
    ks1: ∀p. ∃ks. p=prob ks — Component Parameter
```

Since knowledge sources are parametrized by problems, we must generate a corresponding locale parameter which assigns a problem to each knowledge source. Moreover, we generate an assumption *ks1*, which requires the existence of at least one knowledge source for each problem (as required by the semantics of parametric component types).

Finally, we generate additional locale assumptions according to the activation and connection assumptions described in Ex. 23:

```
— Assertions about component activation.
and actks:
  ⋀t n kid p. ⟦t ∈ arch; ksactive kid (t n); p=prob kid; p∈ksop (kscmp kid (t n))⟧
  ⟹ (∃n'≥n. ksactive kid (t n') ∧ (p, solve p) = ksns (kscmp kid (t n')) ∧
  (∀n''≥n. n''<n' ⟶ ksactive kid (t n'')))
  ∨ (∀n'≥n. (ksactive kid (t n') ∧ (¬(p, solve p) = ksns (kscmp kid (t n')))))

— Assertions about connections.
and conn1: ⋀k bid. bbactive bid k
  ⟹ bbns (bbcmp bid k) = (⋃kid∈{kid. ksactive kid k}. {ksns (kscmp kid k)})
and conn2: ⋀k kid. ksactive kid k
  ⟹ ksop (kscmp kid k) = (⋃bid∈{bid. bbactive bid k}. {bbop (bbcmp bid k)})
```

In contrast to the patterns discussed so far, a Blackboard involves also the specification of component types, i.e., assumptions about component behavior. Thus, one additional locale assumption is generated for every behavior assumption presented in Ex. 19 and Ex. 20:

```
— Assertions about component behavior.
and bhvbb1: ⋀t t' bId p s. ⟦t ∈ arch⟧ ⟹ pb.eval bId t t' 0
  (pb.glob (pb.ba (λbb. (p,s)∈bbns bb)
  ⟶^p (pb.evt (pb.ba (λbb. (p,s) = bbcs bb)))))
and bhvbb2: ⋀t t' bId P q. ⟦t∈arch⟧ ⟹ pb.eval bId t t' 0
  (pb.glob (pb.ba (λbb. sub P ∈ bbrp bb ∧ q ∈ P) ⟶^p
  (pb.evt (pb.ba (λbb. q = bbop bb)))))
and bhvbb3: ⋀t t' bId p . ⟦t∈arch⟧ ⟹ pb.eval bId t t' 0
  (pb.glob (pb.ba (λbb. p = bbop(bb)) ⟶^p
  (pb.wuntil (pb.ba (λbb. p=bbop(bb))) (pb.ba (λbb. (p,solve(p)) = bbcs(bb))))))
and bhvks1: ⋀t t' kId p P. ⟦t∈arch; p = prob kId⟧ ⟹ sb.eval kId t t' 0
  (sb.glob ((sb.ba (λks. sub P = ksrp ks)) ∧^s
  (sb.all (λq. (sb.pred (q∈P)) ⟶^s (sb.evt (sb.ba (λks. (q,solve(q)) ∈ kscs ks)))))
  ⟶^s (sb.evt (sb.ba (λks. (p, solve p) = ksns ks)))))
and bhvks2: ⋀t t' kId p P q. ⟦t ∈ arch;p = prob kId⟧ ⟹ sb.eval kId t t' 0
  (sb.glob (sb.ba (λks. sub P = ksrp ks ∧ q ∈ P ⟶ (q,p) ∈ sb)))
and bhvks3: ⋀t t' kId p. ⟦t∈arch;p = prob kId⟧ ⟹ sb.eval kId t t' 0
  (sb.glob ((sb.ba (λks. p∈ksop ks)) ⟶^s (sb.evt (sb.ba (λks. (∃P. sub P = ksrp ks))))))
and bhvks4: ⋀t t' kId p P. ⟦t∈arch; p∈P⟧ ⟹ sb.eval kId t t' 0
  (sb.glob ((sb.ba (λks. sub P = ksrp ks)) ⟶^s
  (sb.wuntil (¬^s (∃_s P'. (sb.pred (p∈P') ∧^s (sb.ba (λks. unsub P' = ksrp ks)))))
  (sb.ba (λks. (p,solve p) ∈ kscs ks)))))
```

In FACTum, assumptions about component behavior are specified without considering possible activations and deactivations of a component. Thus, they cannot be simply transferred to assumptions over an architecture trace, as it was done for the mapping of architectural assumptions. Rather, our framework provides an operation *eval* which is instantiated for each component type and which can be used to specify assumptions about component behavior in a dynamic environment. Later on, our framework can be used to combine the assumptions about component activation with the assumptions about component behavior to reason about a pattern specification.

7.3.4 Mapping the Guarantee

As for the other examples, we finally generate a theorem according to the guarantee presented in Sec. 7.3.1. Again, we first introduce an abbreviation for the *unique* blackboard component inherited from the Publisher-Subscriber:

```
abbreviation the-bb ≡ the-pb
```

To foster readability, we even use Isabelle/HOL's indefinite description operator *SOME* to introduce an additional definition to denote a knowledge source with a certain property:

```
definition sKs:: PROB ⇒ 'kid where
  sKs p ≡ (SOME kid. p = prob kid)
```

Then, we can generate the corresponding Isabelle/HOL theorem:

```
theorem pSolved:
  fixes t and t'::nat ⇒'BB and t''::nat ⇒'KS
  assumes t∈arch and
    ∀n. (∃n'≥n. ksactive (sKs (bbop(bbcmp the-bb (t n)))) (t n'))
  shows
    ∀n. (∀P. (sub P ∈ bbrp(bbcmp the-bb (t n))
      ⟶ (∀p ∈ P. (∃m≥n. (p,solve(p)) = bbcs (bbcmp the-bb (t m))))))
```

7.3.5 Proving the Theorem

Finally, we discuss the verification of the theorem generated for the Blackboard pattern. Again, we get all the rules of the calculus and all the results for the Singleton pattern and Publisher-Subscriber pattern, for free.

7.3.5.1 Results from the Calculus

Similar as for the Publisher-Subscriber pattern, we get introduction and elimination rules for all the operators used in the specification of blackboards as well as knowledge sources. Figure 7.6 shows an introduction rule for basic behavior assertions instantiated for both types of components. Again, the rules are similar, except for the activation and selection parameters.

pb.baIA: $[\![\exists\, i{\geq}n.\ \wr c \wr_{t\ i};\ \varphi\ (\sigma_{c} t\ \langle c \rightarrow t\rangle_{n})]\!] \Longrightarrow pb.eval\ c\ t\ t'\ n\ (pb.ba\ \varphi)$

sb.baIA: $[\![\exists\, i{\geq}n.\ \wr c \wr_{t\ i};\ \varphi\ (\sigma_{c} t\ (sb.nxtAct\ c\ t\ n))]\!] \Longrightarrow sb.eval\ c\ t\ t'\ n\ (sb.ba\ \varphi)$

. . . Similar rules are available for each operator

Figure 7.6: Calculus instantiated for the Blackboard pattern.

7.3.5.2 Results from Pattern Instantiations

Since a Blackboard instantiates the Publisher-Subscriber pattern (and therefore implicitly also the Singleton pattern), we get all the properties verified for these patterns also for the Blackboard pattern, for free. First, we get all the results for the Singleton pattern:

abbreviation *the-bb* $\equiv$ *the-pb*

pb.ts-prop (*1*): $\wr id \wr_{k} \Longrightarrow id = \mathit{the\text{-}bb}$

pb.ts-prop (*2*): $\wr \mathit{the\text{-}bb} \wr_{k}$

Similar as for the Publisher-Subscriber, we first introduce an abbreviation *the-bb* to denote the *unique* component of type blackboard and then we get a corresponding lemma about uniqueness of a blackboard component.

In addition, we get results from the Publisher-Subscriber pattern:

msgDelivery:
$[\![t \in arch;$
$\wr sid \wr_{t\ n};$
$sub\ E = ksrp\ (\sigma_{sid} t\ n);$
$n \leq n'';$
$\nexists\, n'\ E'.\ n \leq n' \wedge n' \leq n'' \wedge \wr sid \wr_{t\ n'} \wedge unsub\ E' = ksrp\ (\sigma_{sid} t\ n') \wedge e \in E';$
$e \in E;$
$(e,\ m) = bbcs\ (\sigma_{\mathit{the\text{-}bb}} t\ n'');$
$\wr sid \wr_{t\ n''}]\!]$
$\Longrightarrow (e,\ m) \in kscs\ (\sigma_{sid} t\ n'')$

Basically, the results resemble the property verified for Publisher-Subscriber patterns (discussed in the last section) with activation and selection parameters from knowledge sources and blackboards, respectively.

7.3.5.3 Proving the Theorem

In order to prove theorem *pSolved*, presented in Sect. 7.3.4, we first prove a corresponding lemma:

lemma *pSolved-Ind*:
fixes t **and** $t'{::}nat \Rightarrow 'BB$ **and** p **and** $t''{::}nat \Rightarrow 'KS$
assumes $t{\in}arch$ **and**

```
  ∀n. (∃n'≥n. ksactive (sKs (bbop(bbcmp the-bb (t n)))) (t n'))
shows
  ∀n. (∃P. sub P ∈ bbrp(bbcmp the-bb (t n)) ∧ p ∈ P) ⟶
    (∃m≥n. (p,solve(p)) = bbcs (bbcmp the-bb (t m)))
```

The lemma can be proved by well-founded induction over the subproblem relation sb, since *sb* was assumed to be well-founded in Sec. 7.3.2. The final theorem is now a direct consequence of this lemma and can be proved in a single line:

```
using assms pSolved-Ind by blast
```

7.4 Summary

In this chapter, we presented results obtained from the interactive verification of the Singleton pattern, the Publisher-Subscriber pattern, and the Blackboard pattern. In the following, we briefly summarize the results obtained for each of the patterns.

7.4.1 Singleton

The leftmost graph in Fig. 7.7 depicts the effort for the verification of the Singleton pattern. Essentially, we have two classes of verification results for this pattern:

- A key result for the pattern is formalized by property `ts_prop`, which guarantees that, in our version of the Singleton, a singleton component is *unique* and always *active*.
- The second class of results leverages property `ts_prop` to provide rules to reason about the behavior of singleton components. Remember that, in general, reasoning about the behavior of components requires to consider activation constraints for that type of component. However, since a singleton component is always active and unique, reasoning about its behavior can be done without considering activation specifications at all.

7.4.1.1 Publisher-Subscriber

In our version of the Publisher-Subscriber pattern, we modeled a publisher component as an instance of a singleton. Thus, as already mentioned above, all the verification results for singleton components from the Singleton pattern are available for publisher components in the Publisher-Subscriber pattern. Hence, we get special rules to reason about the behavior of publisher components which we use for the verification of two additional properties for a Publisher-Subscriber architecture:

- Property `msgDelivery` provides a characteristic property for such architectures which ensures that a subscriber component indeed receives all the messages associated with events for which it is subscribed.

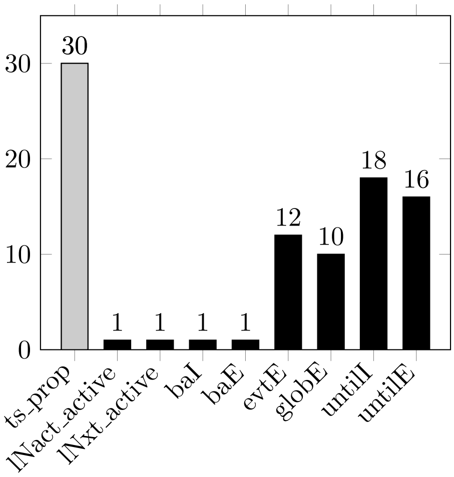


Figure 7.7: Propositions for the Singleton pattern.

- Property `conn1A` provides a more technical result to support the reasoning about Publisher-Subscriber architectures. It leverages the fact that a publisher is actually a singleton to provide an alternative version of the basic rule to reason about connected components.

As can be observed from Fig. 7.8, the proofs for both properties are simple one-liners. Note, however, that this is only due to the fact that the proofs are based on the results obtained from the Singleton pattern.

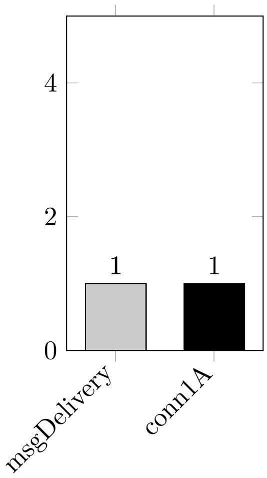


Figure 7.8: Propositions for the Publisher-Subscriber pattern.

7.4.1.2 Blackboard

We modeled the Blackboard pattern as a version of the Publisher-Subscriber pattern in which a blackboard takes the role of a publisher and the knowledge sources correspond to subscriber components. Thus, again, all the results from the Publisher-Subscriber pattern are available to support the verification of the Blackboard pattern. The additional constraints added by the Blackboard pattern can then be used to derive some additional guarantees of which one is of particular interest: Property `pSolved` guarantees that a Blackboard architecture is able to solve a given problem, provided that for each open sub-problem there exists a knowledge source which is able to address it. To prove this property, we first proved a more general result `pSolved_Ind` by induction. As shown in Fig. 7.9, the proof of it consisted of 391 lines of Isabelle/HOL code. The final property then follows directly from this lemma.

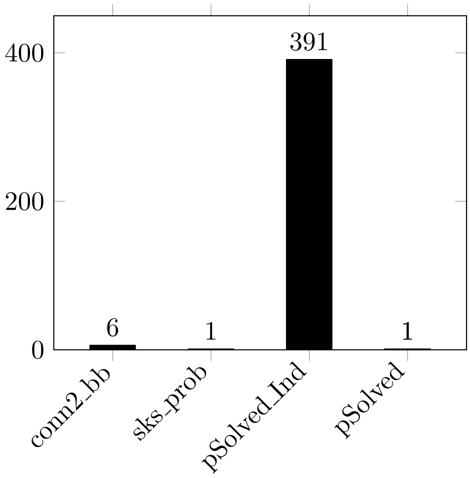


Figure 7.9: Propositions for the Blackboard pattern.

8 Verification of Blockchain Architectures

In the last chapter, we introduced FACTum and demonstrated it in terms of three running examples: the singleton, the publisher-subscriber, and the blackboard pattern. Thereby, the verification of these patterns was rather trivial and the main purpose was to demonstrate the methodology, rather than evaluating it.

In the following chapter, we apply the methodology to a larger case study to specify and verify a pattern for Blockchain architectures:

- First, we specify blockchains as parametric lists using algebraic specification techniques.
- Then, we specify two types of components: trusted nodes which follow a given consensus protocol and untrusted nodes which may deviate from the protocol.
- Finally, we specified several architectural assertions which constrain the activation of nodes and their interconnection.

Then, we systematically transfer the specification of the pattern to a corresponding Isabelle/HOL [NPW02] theory using the algorithm described in Chap. 6. Finally, we formalize the notion of “resistance to modification of blockchain entries”, transfer it to a corresponding Isabelle/HOL theorem, and prove it using the calculus presented in Chap. 5.

In total, the verification consists of two Isabelle theory files amounting to roughly 3 000 lines of Isabelle proof script. Thereby, we discovered 9 architectural assumptions required by Blockchain architectures in order to guarantee persistence of blockchain entries. While some of them are actually concerned with details of an architecture, one of the assumptions could be considered fundamental for Blockchain architectures: the requirement that the relative frequency of minings from trusted and untrusted nodes observed at every time interval is bounded by the number of confirmation blocks.

8.1 Blockchain Architectures

Blockchain architectures were first introduced with the invention of the Bitcoin cryptocurrency [Nak08]. In cryptocurrencies, a digital coin is usually passed from one owner to the next by digitally signing an electronic transaction. In order to ensure that coins are only spent once by any owner, a payee has to know whether a received coin is already spent or not at the time he receives it. This problem is known as the *double spend problem* and before the invention of blockchain, it was solved using a central, trusted identity which knew every transaction of the system and confirmed that a coin was not already spent. In an attempt to avoid such central authorities, Bitcoin proposed a system called blockchain to solve the double spend problem in a distributed, peer-to-peer network.

Thereby, the network stores a continuously growing list of persistent entries which contain the actual money transactions. The list is shared amongst all participants of the network and by inspecting it, a node can independently verify that a coin was not already spent. In this chapter, we call such a network a *Blockchain architecture* and in the following we summarize some basic concepts of such architectures as described in [Nak08].

8.1.1 Blockchain Data Structure

The term blockchain usually refers to the major data structure involved in a Blockchain architecture: a list of records aka. *blocks*. Blocks, on the other hand, contain the actual data elements, for example, money transactions in cryptocurrency applications. Blocks can be added on top of the chain and verified by a process known as *mining*. In Bitcoin, for example, mining involves the guessing of a random number (a so-called *nonce*), adding it to a candidate block and checking whether the corresponding hash exhibits a certain form (starting with a certain amount of zeros). This makes mining of a new block computationally expensive since it usually requires many guesses (and subsequent hashings) to find a number which produces the right hash. However, ensuring that a given block was indeed successfully mined remains computationally cheap (it only requires a single hashing).

8.1.2 Blockchain Architectures

In a Blockchain architecture, every node maintains a local copy of the blockchain which it exchanges with its peers. Due to the distributed nature, it may happen that two different blocks are added concurrently, resulting in two different versions of the blockchain available in the network. In order to reach a *consensus* on which version is the “right” one, a Blockchain architecture usually comes with a strategy of how to select the right version from a set of competing blockchains. This rule is applied by every *trusted* node of the network and should guarantee that the nodes eventually reach a consensus.

8.1.3 Consensus Rules

There are several different types of strategies used to reach consensus, such as proof of work [Nak08] or proof of stake [BGM16]. In the proposed pattern, we rely on the *proof of work* concept also used by Bitcoin and related applications. It is based on the observation that the number of blocks in a blockchain usually represents the amount of computing power involved to build this chain. Thus, the largest chain from a set of competing blockchains must be the one accepted by the majority of the network. Thus, if a trusted node is facing two versions of a blockchain, it is required to always choose the longer one.

8.1.4 Confirmation Blocks

In a proof-of-work network, every CPU gets one vote and majority decisions can usually only be manipulated if one entity owns more than 50% of the computing power of the network. This might not be true, however, for blocks added to the blockchain only recently. A single node may just be lucky and guess the right nonce fast, without investing a lot of computational power. In order to cope with such lucky guesses, one usually waits for some blocks to be mined on top of the block containing a certain transaction, in order to accept this transaction as completed. In Bitcoin, for example, it is suggested to wait at least 6 blocks in order to accept a transaction as completed.

8.2 Formalizing Blockchain Architectures

In the following, we present our formalization of a possible pattern for Blockchain architectures. Therefore, we first describe the involved data types. Then, we present the types of components and constraints about their behavior. Finally, we discuss additional architectural constraints about component activation and interconnection.

8.3 Data Types and Ports

As described in Sect. 8.1, a key data type in Blockchain architectures is the *blockchain* itself. In the following, we first formalize a blockchain datastructure in terms of algebraic datatypes. Then, we specify two types of ports to send and receive blockchains.

8.3.1 Blockchains

A blockchain is modeled as a parametric list where the nature of the list entries (the blocks) depends on the concrete application context of the pattern. In cryptocurrency applications, for example, a block is actually a set of transactions. In other applications, however, blocks could be of a different type.

Figure 8.1a provides a specification of blockchains in terms of an abstract data type specification [Bro96, Wir90]. First, a parametric sort $\langle\mathsf{B}\rangle\mathsf{BC}$ is introduced as a synonym for a corresponding list. Thereby, the type of blocks is denoted with type parameter B. In addition, we specify a function symbol MAX for blockchains which takes a set of blockchains and returns a blockchain with maximal length. Thus, we require two characteristic properties for MAX: Eq. (8.1) requires that a maximal blockchain of a set of blockchains BC is indeed part of BC itself. In addition, Eq. (8.2) requires that MAX is indeed maximal, i.e., that the length of every other blockchain of the corresponding set BC is less or equal to the length of MAX. Note that $MAX(BC)$ is guaranteed to exist whenever $BC \neq \emptyset$ and BC is *finite*.

8.3.2 Port Types

Figure 8.1b specifies two types of ports which can be used to exchange blockchains: *pin* for input ports and *pout* for output ports. They will be used later on for the specification of component type interfaces.

DTSpec Blockchain	**imports** $\langle \mathsf{B} \rangle \mathtt{LIST}$ **as** $\langle \mathsf{B} \rangle \mathtt{BC}$
MAX:	$\wp(\langle \mathsf{B} \rangle \mathtt{BC}) \to \langle \mathsf{B} \rangle \mathtt{BC}$
flex BC:	$\wp(\langle \mathsf{B} \rangle \mathtt{BC})$
bc:	$\mathtt{BC}$
$MAX(BC) \in BC$	(8.1)
$\forall bc \in BC\colon \#bc \leq \#MAX(BC)$	(8.2)

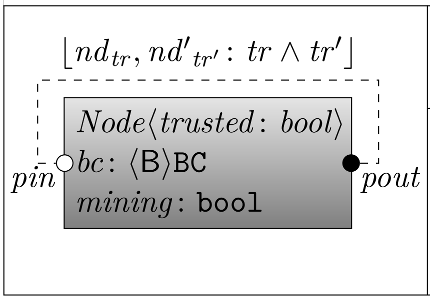
(a) Data type specification.

PSpec BPort	
pin:	$\langle \mathsf{B} \rangle \mathtt{BC}$
$pout$:	$\langle \mathsf{B} \rangle \mathtt{BC}$

(b) Port specification.

Figure 8.1: Specification of the Blockchain pattern.

Diagram Blockchain$\langle cb\colon \mathtt{NAT}\rangle$



var tn: $Node[trusted]$

un: $Node[\neg trusted]$

$$PoW \overset{\text{def}}{=} LEAST\ x\colon \forall tn\colon \wr tn \wr \longrightarrow \#tn.bc \leq x$$

$$tmining \overset{\text{def}}{=} \exists tn\colon \wr tn \wr \ \land\ tn.mining$$

$$umining \overset{\text{def}}{=} \exists un\colon \wr un \wr \ \land\ un.mining$$

Figure 8.2: Architecture diagram for Blockchain architectures.

8.4 Component Types

The components involved in a Blockchain architecture are called *nodes*. In the following, we first describe the syntactic interface of such a node component. Then, we introduce some auxiliary definitions for nodes used later on. Finally, we provide a set of characteristic properties for a node's behavior.

8.4.1 Interfaces

The architecture diagram depicted in Fig. 8.2 specifies the syntactic interface of blockchain nodes. Actually, the diagram also contains a graphical representation of a connection constraint as well as the definition of three auxiliary definitions for nodes. For now, however, we skip these additional aspects and focus on the description of the interface. We will come back to the auxiliary definitions in the next section and we will discuss the connection constraint later on in Sec. 8.5.

First of all, a node in a blockchain may either be trusted or untrusted. Therefore, a node is parametrized by a boolean value *trusted* which means that every component of type node has a constant, boolean value associated to it which determines its trustworthiness. In addition, a node has two state variables: variable *bc* keeps a local copy of the node's blockchain and variable *mining* signals the mining of a new block. Finally, a node may exchange blockchains through its input port *pin* and output port *pout*.

8.4.2 Auxiliary Definitions

To support subsequent development, the right hand side of Fig. 8.2 introduces three auxiliary definitions for nodes: trusted proof of work and trusted/untrusted mining.

Trusted proof of work. Trusted proof of work is denoted by PoW and represents the *maximal* proof of work currently available in the trusted community. Since proof of work corresponds to the length of a blockchain (Sec. 8.1), trusted proof of work is defined as the least upper bound for the length of trusted blockchains, i.e., blockchains of active and trusted nodes. Note the use of the definite description operator $LEAST$ to denote the *least* length x which is greater or equal to the length of the blockchain of every trusted and active node.

Trusted and untrusted mining. Trusted mining is a predicate denoted by *tmining* which states that at the current point in time, some trusted node was able to mine a new block. Similarly, untrusted mining states that currently an untrusted node was able to mine a new block. It is denoted by *umining*. Trusted and untrusted mining play an important role in the formalization of a fundamental property for Blockchain architectures later on.

8.4.3 Behavior

The behavior of nodes is given in terms of a set of so-called behavior trace assertions, i.e., linear temporal logic [MP92] formulæ, formulated over a node's interface[1]. Figure 8.3 depicts the corresponding specification. First, we introduce several variables to denote single blocks (b), blockchains (bc and bc'), trusted nodes (tn), untrusted nodes un, and nodes in general (nd). Note the distinction between "flexible" and "rigid" variables: while "flexible" variables may be newly interpreted at every point in time, "rigid" variables keep their value over time. Then, we require four assertions for a node's behavior: The first two assertions Eq. (8.3) and Eq. (8.4) are general properties required for trusted as well as untrusted node components. Eq. (8.3) requires that a new node is initialized by the empty blockchain while Eq. (8.4) requires that every node nd indeed always forwards a copy of its local blockchain to the network through its output port $pout$. Eq. (8.5) and Eq. (8.6), on the other hand, are specific to trusted and untrusted nodes. They are used to characterize the behavior for trusted and untrusted components and in the following they are described in more detail.

Trusted nodes. The behavior of trusted nodes is characterized by Eq. (8.5). The property essentially requires that a trusted node can only add newly mined blocks on top of a given blockchain. Moreover, it also contains the consensus rule for trusted nodes which requires that a trusted node always takes the blockchain with maximal proof of work as the current one, i.e. if a trusted node receives a blockchain on its input with

---

[1] Behavior trace assertions are described in detail in Chap. 3

BSpec Blockchain **for** *Node* **of** Blockchain

flex	b:	B
	bc':	$\langle\mathsf{B}\rangle\mathtt{BC}$
	nd:	$Node\langle tr\rangle$
rig	bc:	$\langle\mathsf{B}\rangle\mathtt{BC}$
	tn:	$Node[trusted]$
	un:	$Node[\neg trusted]$

$$nd.bc = [] \tag{8.3}$$

$$\Box(nd.pout = nd.bc) \tag{8.4}$$

$$\Box\left(bc = \begin{cases} MAX(tn.pin) & \text{if } \exists bc' \in tn.pin\colon \#bc' > \#tn.bc, \\ tn.bc & \text{else.} \end{cases} \right. \\ \left. \longrightarrow \bigcirc(\neg tn.mining \land tn.bc = bc \lor tnmining \land \exists b\colon tn.bc = bc@b) \right) \tag{8.5}$$

$$\Box\left(bc = (\varepsilon bc' \in (un.pin \cup \{un.bc\})) \right. \\ \left. \longrightarrow \bigcirc(\neg un.mining \land un.bc \sqsubseteq bc \lor un.mining \land \exists b\colon un.bc = bc@b) \right) \tag{8.6}$$

Figure 8.3: Specification of behavior for node components.

more proof of work than its own blockchain, then he will accept that blockchain as the current one.

The property actually consists of two parts. The precondition formalizes the consensus rule:

$$bc = \begin{cases} MAX(tn.pin) & \text{if } \exists bc' \in tn.pin\colon \#bc' > \#tn.bc, \\ tn.bc & \text{else.} \end{cases}$$

Since the proof of work for a blockchain is given by its length, the property fixes a blockchain *bc* which is either a maximal blockchain from the input port *pin* of a trusted node *tn* (for the case that it is strictly longer than its own blockchain), or its own blockchain *tn.bc* (for the case that no blockchain from its input is longer than its own blockchain). The implication formalizes the mining process:

$$\bigcirc(\neg tn.mining \land tn.bc = bc \lor tn.mining \land \exists b\colon tn.bc = bc@b).$$

Thereby, a trusted node *tn* may either mine a new block (*mining*), append it to *bc* and take the resulting chain as its current blockchain *tn.bc*, or *tn* may not mine any new block ($\neg$*mining*) and just set *bc* as its current blockchain *tn.bc*.

Untrusted nodes. The behavior of untrusted nodes is characterized by Eq. (8.6). Note that, compared to trusted nodes, untrusted nodes may not follow the consensus rules. Thus, while trusted nodes always take the blockchain with the most proof of work as their current blockchain, untrusted nodes may take every blockchain from its input as the current one. Moreover, in contrast to trusted nodes, untrusted nodes may also drop elements from a blockchain, thus trying to modify a blockchain's history.

Similar as for trusted nodes, the specification of the behavior for untrusted nodes consists of two parts. The precondition again fixes a blockchain *bc*:

$$bc = (\varepsilon bc' \colon\ bc' \in (un.pin \cup \{un.bc\}))$$

Note that we used Hilbert's ε operator here to denote *some* element bc' from input port *pin* or state port *bc*. The implication is similar to the implication for trusted nodes:

$$\bigcirc(\neg un.mining \wedge un.bc \sqsubseteq bc \vee un.mining \wedge \exists b\colon un.bc = bc@b)$$

Note that, due to computing restrictions, even untrusted nodes may at most mine one single block at a time. Thus, the mining case is indeed the same as for trusted nodes. The difference, however, comes with the case in which no new block is mined. While, for such a case, trusted nodes are required to take *bc* as their current blockchain, untrusted nodes may take an arbitrary prefix of *bc* as their current blockchain.

8.5 Architectural Constraints

Architectural constraints restrict activation and deactivation of components and connections between component ports [MG16a, MG16b]. They are mainly formulated in terms of architecture trace assertions, i.e., linear temporal logic formulæ, formulated over component ports[2]. Certain constraints, however, can be expressed more easily graphically by annotating the pattern's architecture diagram. In the following, we first discuss connection constraints for Blockchain architectures. Then, we present some basic activation constraints for such architectures. Finally, we conclude the section by describing a fundamental constraint for Blockchain architectures which is essential to guarantee persistence of blockchain entries.

8.5.1 Connection Constraints

Connection constraints restrict connections between component ports and therefore they affect the topology of an architecture. For our pattern of Blockchain architectures, we require a single connection constraint which is expressed graphically by an annotation of the architecture diagram depicted in Fig. 8.2. The dashed connection between a node's input and output ports expresses a conditional connection between ports *pout* and *pin* of two (possible different) components of type node. The *minimal* condition for the connection to happen is expressed by the connections annotation:

$$\lfloor nd_{tr}, nd'_{tr'} \colon tr \wedge tr' \rfloor.$$

The condition essentially requires the corresponding ports to be connected whenever two components are *trusted*. Roughly speaking, the constraint requires that every trusted node is connected to every other trusted node of the network. While this constraint is indeed a strong requirement, it is necessary to guarantee persistence of blockchain entries.

---

[2] Architecture trace assertions are described in detail in Chap. 3

ASpec Basic		**for** Blockchain
flex	bc:	$\mathtt{BC}\langle\mathsf{B}\rangle$
	nd:	$Node\langle tr\rangle$
	nd':	$Node\langle tr'\rangle$
rig	tn:	$Node[trusted]$

$$\Box\Big(finite(\{nd \mid \wr nd\wr\})\Big) \tag{8.7}$$

$$\Box\Big(\exists tn\colon \wr tn\wr \wedge \bigcirc \wr tn\wr\Big) \tag{8.8}$$

$$\Box\Big(\wr tn\wr \wedge tn.mining \longrightarrow \ominus \wr tn\wr\Big) \tag{8.9}$$

$$\Box\Big(\wr nd\wr \wedge bc \in nd.pin \longrightarrow \exists nd'\colon \wr nd'\wr \wedge nd'.bc = bc\Big) \tag{8.10}$$

Figure 8.4: Basic activation constraints for Blockchain architectures.

8.5.2 Basic Activation Constraints

Activation constraints affect the activation and deactivation of components of a certain type. We require four basic activation constraints for Blockchain architectures summarized in Fig. 8.4 and explained in more detail in the following.

Finite amount of active nodes. Our first activation property for Blockchain architectures is more of theoretical nature and restricts the number of active components at each point in time. By Eq. (8.7), we require that at each point in time, only a finite number of node components can be activated. The property should be satisfied by every architecture found in practice. However, it is needed in order to guarantee that at every point in time, a node component receives only a finite amount of blockchains which, in turn, is required to guarantee that maximal blockchains are well-defined for a component's input port.

Keeping the trusted blockchain. The second activation property we require for Blockchain architectures is needed in order to guarantee that the trusted blockchain, i.e., the blockchain accepted by trusted nodes as the "correct" one, is not lost. It is formalized by Eq. (8.8) and requires that at every point in time, there exists an active and trusted node which stays active for at least one time step. Thus, it is guaranteed that the current trusted blockchain is stored by the trusted network and does not get lost.

Mining on most recent blockchain. Another basic activation property for Blockchain architectures is needed in order to ensure that the trusted network indeed collaborates in the mining process. The property is formalized by Eq. (8.9) using the previous operator: it requires that whenever a trusted node is mining a new block, this node was active at the time point right before the mining happened. This ensures that the node had indeed access to the most recent version of the trusted blockchain and works on extending this version instead of an older version.

Closed architecture. The last basic activation property for Blockchain architectures requires such an architecture to be closed. Eq. (8.10) formalizes the property and requires that for every blockchain available at the input of any active node component at any point in time, there exists a corresponding active node component which provides the blockchain at its output. In other words, the property guarantees that every blockchain available in the architecture was built up from the network due to the mining process and not injected from the outside.

8.5.3 A Fundamental Constraint for Blockchain Architectures

In the following section, we present a fundamental constraint for Blockchain architectures. Since its specification requires to express mining frequencies, we first introduce an operator to express such frequencies in LTL. Then, we use this operator to specify the property.

Relative frequencies in LTL. In the following, we introduce an operator for LTL which can be used to express statements of the form: "for every time span in which at least x states can be observed which satisfy a certain property φ, at least y states can be observed to satisfy a certain property φ'".

Definition 16 (Weak until for relative frequencies)**.** *A trace t satisfies $\varphi \;{}_{\lceil x \rceil}\mathcal{W}_{\lfloor y \rfloor}\; \varphi'$, for state predicates φ and φ', at time point n, iff*

$$\begin{aligned}&\exists n' \geq n\colon cc(t,n,n',\varphi') \geq y \;\;\wedge\;\; (\forall n \leq i < n'\colon cc(t,n,i,\varphi) \leq x)\\ &\quad\vee\;\; (\forall n' \geq n\colon cc(t,n,n',\varphi) \leq x),\end{aligned}$$

with $cc(t,n,n',p) \;\;\overset{def}{=}\;\; |\{i' \mid i' > n \wedge i' \leq n' \wedge p(t(n))\}|.$

In the following, we provide an overview of some characteristic properties derived for the operator. The first lemma characterizes the operator for the case that the two indexes x and y are greater zero.

Lemma 1 (Indexes greater zero)**.** *Assuming $t, n \models \varphi \;{}_{\lceil x \rceil}\mathcal{W}_{\lfloor y \rfloor}\; \varphi'$, $x > 0$, and $y > 0$, then, the following holds:*

$$\begin{aligned}
\varphi(t(n)) \wedge \varphi'(t(n)) \;&\Longrightarrow\; t, n+1 \models \varphi \;{}_{\lceil x-1 \rceil}\mathcal{W}_{\lfloor y-1 \rfloor}\; \varphi',\\
\varphi(t(n)) \wedge \neg\varphi'(t(n)) \;&\Longrightarrow\; t, n+1 \models \varphi \;{}_{\lceil x-1 \rceil}\mathcal{W}_{\lfloor y \rfloor}\; \varphi',\\
\neg\varphi(t(n)) \wedge \varphi'(t(n)) \;&\Longrightarrow\; t, n+1 \models \varphi \;{}_{\lceil x \rceil}\mathcal{W}_{\lfloor y-1 \rfloor}\; \varphi',\ \textit{and}\\
\neg\varphi(t(n)) \wedge \neg\varphi'(t(n)) \;&\Longrightarrow\; t, n+1 \models \varphi \;{}_{\lceil x \rceil}\mathcal{W}_{\lfloor y \rfloor}\; \varphi'. \qquad \square
\end{aligned}$$

ASpec Blockchain **for** Blockchain

$$\square\Big(umining\ {}_{\lceil cb\rceil}\mathcal{W}_{\lfloor cb+1\rfloor}\ tmining\Big) \tag{8.11}$$

Figure 8.5: Fundamental constraint for Blockchain architectures.

Essentially, the properties state that whenever $\varphi\ {}_{\lceil x\rceil}\mathcal{W}_{\lfloor y\rfloor}\ \varphi'$ holds for some trace t at some time point n, then the indexes can be reduced by one for the next state, depending on whether φ or φ' hold at the current state.

A second lemma specifies what happens if at some point in time we reach the point where the first index reaches zero:

Lemma 2 (First index zero)**.** *Assuming* $t, n \models \varphi\ {}_{\lceil x\rceil}\mathcal{W}_{\lfloor y\rfloor}\ \varphi'$, $x = 0$, *and* $y > 0$. *Then we have:* $\neg\varphi(t(n+1))$. $\square$

The property shows that after reaching zero at the first index, it is guaranteed that property φ does not hold again as long as y remains greater zero.

Relative mining frequencies. Now, we have everything it needs in order to formalize a fundamental requirement for Blockchain architectures. It is formalized as an architecture constraint in Fig. 8.5 using the operator introduced above. Essentially, the property requires that for every time span in which we can observe a number of untrusted minings which is *greater or equal* the number of confirmation blocks, then we can also observe a number of trusted minings which is *greater* than the number of confirmation blocks. Note that this is an important requirement needed to guarantee persistence of blockchain entries. Later on, in Sect. 8.7.1, we discuss the importance of this property in more detail.

8.6 Verifying Blockchain Architectures

In the following, we verify an important property for Blockchain architectures which ensures persistence of blockchain entries.

8.6.1 Persistence of Blockchain Entries

As described in the introduction, Blockchain architectures were invented to solve the double spend problem in a distributed peer-to-peer network. In order to do so, blockchain entries, once accepted by the network, must be resistant to future modifications. This property is summarized by the following theorem:

Theorem 5 (Persistence of blockchain entries)**.** *In a Blockchain architecture, the entries of trusted blockchains which are confirmed by a number of blocks greater or equal to the number of confirmation blocks, are resistant to future modifications.*

The theorem is formally specified by the architectural assertion depicted in Fig. 8.6. Thereby, *sbc* denotes a blockchain which contains the entries supposed to be persistent.

ASpec Save **for** Blockchain

flex tn: $Node[trusted]$
un: $Node[\neg trusted]$
nd: $Node[\neg trusted]$
rig tn': $Node$
sbc: $\langle \mathsf{B}\rangle \mathtt{BC}$

$$
\begin{align}
&\Box\Bigg(\Big(\forall tn'\colon (\neg \wr tn' \wr)\ \mathcal{W}\ (\wr tn' \wr \land sbc \sqsubseteq tn'.bc)\Big) \land \tag{8.12}\\
&\quad PoW \geq \#sbc + cb \land \tag{8.13}\\
&\quad \Big(\forall un\colon \wr un \wr \longrightarrow \#un.bc < \#sbc\Big) \land \tag{8.14}\\
&\quad \ominus\boxminus\Big(\forall nd\colon \wr nd \wr \longrightarrow \#nd.bc < \#sbc \lor sbc \sqsubseteq nd.bc\Big) \land \tag{8.15}\\
&\quad \longrightarrow \quad \Box\Big(\forall tn\colon \wr tn \wr \longrightarrow sbc \sqsubseteq tn.bc\Big)\Bigg) \tag{8.16}
\end{align}
$$

Figure 8.6: Specification of persistence property for Blockchain architectures.

Eq. (8.12) - Eq. (8.15) then characterize a time point n_s for which the property actually holds.

Eq. (8.12) requires that *sbc* is indeed a prefix of the blockchain of every trusted node tn' at tn''s first activation after n_s. It basically ensures that the trusted network is initialized with blockchains extending *sbc*.

Eq. (8.13) requires the proof of work at time point n_s to be greater or equal to the length of *sbc* increased by the number of confirmation blocks *cb*. This equation is required to provide the trusted network with some lead over a potential attacker which might want to change *sbc*. Note, however, that the assumption is indeed feasible, since Thm. 5 ensures persistence only of entries which were confirmed by *cb* number of blocks.

Eq. (8.14) requires the length of the blockchain of every active and untrusted node *un* to be less than the length of *sbc*. Together with Eq. (8.15), this equation ensures that a potential attacker did not prepare a “false” blockchain before time point n_s which he could then use later on to cheat the trusted network.

Eq. (8.15) requires for every node's blockchain $nd.bc$, at every time point before n_s, that *sbc* is either a prefix of $nd.bc$ or that the length of $nd.bc$ is smaller than the length of *sbc*.

For every time point n_s for which the above conditions hold, the property depicted in Fig. 8.6 guarantees that *sbc* will always be a prefix of every trusted node's blockchain (formalized by Eq. (8.16)).

8.6.2 Verification Approach

The above property was formalized as theorem *blockchain-save* in the corresponding Isabelle/HOL theory [Mar18c]. Its proof consists of roughly 11 500 lines of *normalized* proof code. In the following, we are going to discuss the proof idea. Therefore, we first introduce an auxiliary concept: *blockchain developments*. Then we explain how this concept was used to prove the above property.

8.6.2.1 Blockchain Development

In a Blockchain architecture, at any point during the execution, the blockchain of every node has a history describing its development by prior mining activities from other nodes in the network. This is called a blockchain development and it is modeled as a sequence of blockchains. Such a development is characterized by an important property: a blockchain can grow at most by one element at each point in the development. This property has two important consequences which are discussed in the following.

Blockchain modifications. One important consequence regards the nature of modifications of blockchain entries in a development: in order to modify an entry of a blockchain, its development must first shrink the blockchain to the desired entry and then append the modified block.

Relative growth. Another important consequence regards the relative growth of two different types of developments: *trusted* and *untrusted developments*. In a *trusted development*, minings have to be done only by trusted nodes. Similarly, an *untrusted development* contains only minings from untrusted nodes. If we consider the fundamental property of blockchains described in Fig. 8.5, we can derive the following useful property: If at any point in time, the untrusted development is below the trusted one by at least cb elements, then the length of the untrusted development will never surpass the one of the trusted development.

8.6.2.2 Overview of the Proof

Basically, the proof is by induction over the time point referred to by the globally operator provided in Eq. (8.16). For each time point we then show Eq. (8.16) by contradiction: In order to violate it, there must exist an untrusted node with a blockchain larger than the blockchain of one of the trusted nodes (since only then the consensus rule would require the trusted node to take the larger one). Assuming there exists such an untrusted node, we can then construct the untrusted development of the corresponding blockchain. Moreover, we can also construct the trusted development of the trusted node's blockchain. The “blockchain modification” property for blockchain developments discussed above, now requires that at some point, the untrusted development must be below the trusted development by at least cb elements. Thus, property “relative growth”, would require that the length of the untrusted development is always less than the length of the trusted one. However, this would be in contradiction with the assumption that the blockchain of the untrusted node is smaller than the blockchain of the trusted node.

8.7 Discussion

In the following, we discuss some interesting observations about Blockchain architectures. In particular, we discuss the importance of Eq. (8.11) to guarantee Thm. 5.

8.7.1 Unbounded Untrusted Mining

First, we demonstrate why, in general, it is necessary to bound the number of subsequent minings of untrusted nodes, to guarantee persistence of blockchain entries. Therefore, we show how unbounded mining of untrusted nodes may lead to situations in which already confirmed entries of blockchains of trusted nodes may be modified in the future.

Example 30 (Modification of already confirmed blocks)**.** *Let us assume that our blockchain is storing characters A, B, C, ... Figure 8.7 depicts the development of two blockchain copies for a trusted node (solid) and an untrusted node (dashed) starting from a time point n. The blockchain of the trusted node initially (at time point n) contains four entries: A, B, C, and D. If we consider the number of confirmation blocks to be two, then we can consider blocks A and B to be persistent, since two other blocks are already mined on top of them. Since the trusted node broadcasts its copy of the blockchain to the whole network, at time point* $n+1$*, the untrusted node receives the copy and stores it. By Eq.* (8.6)*, the untrusted node may now perform one of two actions: either it removes some blocks from the top of its blockchain, or he mines a new block and appends it to its local copy of the blockchain. Lets assume, that the untrusted node first removes the top three blocks from the chain and then mines a new block* X *on top of its remaining blockchain. Thus, at time point* $n+3$*, the blockchain of the untrusted node contains two entries: A and* X*. Assuming that, in the meantime, the copy of the trusted node's blockchain did not change, the length of the untrusted blockchain is still less than the length of the trusted one. Thus, according to Eq.* (8.4)*, the trusted node would currently not accept the untrusted blockchain. Now assume that the untrusted node is able to mine three additional blocks* Y*,* Z*, and* K*, on top of its blockchain, while the trusted node was still not able to mine any single block. Note that this is a feasible assumption,*

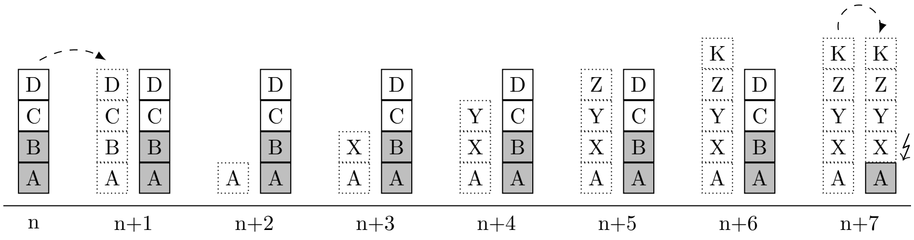


Figure 8.7: Graphical depiction of a double spend attack.

since we do not have any constraints on the number of untrusted minings. The untrusted blockchain now consists of five blocks: A, X, Y, Z, and K. If it sends its copy of the chain to the trusted node, the latter would accept it, since the proof of work (the length) of the untrusted blockchain is now larger than the proof of work of its own copy. Thus, the untrusted node was indeed able to modify entry B of the trusted blockchain, although it was originally supposed to be persistent. ☐

8.7.2 Weakening Eq. (8.11)

Example 30 shows that it is indeed necessary to constrain the number of subsequent minings of untrusted nodes, in order ensure persistence of blockchain entries. However, looking at the example, one may ask whether Eq. (8.9) is really necessary. Why not put a weaker constraint, such as requiring at least *one* trusted mining every cb untrusted minings? While this would indeed suffice to cope with situations as described in the previous example, the next example describes a situation which shows that this weaker version of Eq. (8.9) is also not sufficient to guarantee persistence of blockchain entries.

Example 31 (Modification of already confirmed blocks)**.** *Due to the introduced bound on untrusted minings, the situation described by Ex. 30 is not feasible anymore: since at time point $n+3$, two consecutive minings of untrusted nodes happened, the newly introduced bound requires mining of a trusted node to happen next. However, Fig. 8.8 depicts an alternative continuation of the development discussed in Ex. 30 which satisfies the bound and still leads to a modification of an already confirmed entry of the blockchain of the trusted node. According to the new constraint, the next block (after time point $n+3$) has to be mined by a trusted node. This is what actually happens at time point $n+4$: the trusted node mines a new block E on top of its blockchain, while the untrusted node keeps its current copy of the blockchain. However, since a trusted mining just happened, the untrusted node may now mine two additional blocks before a trusted mining is required which may lead to a situation as depicted at time point $n+6$. Since the untrusted blockchain still contains less proof of work than the trusted one, the trusted node keeps its own copy and mines an additional block F on top of it at time point $n+7$. Again, occurrence of a trusted mining now allows for two additional untrusted minings, which leads to the situation shown at $n+9$. The last steps can be repeated to finally arrive at*

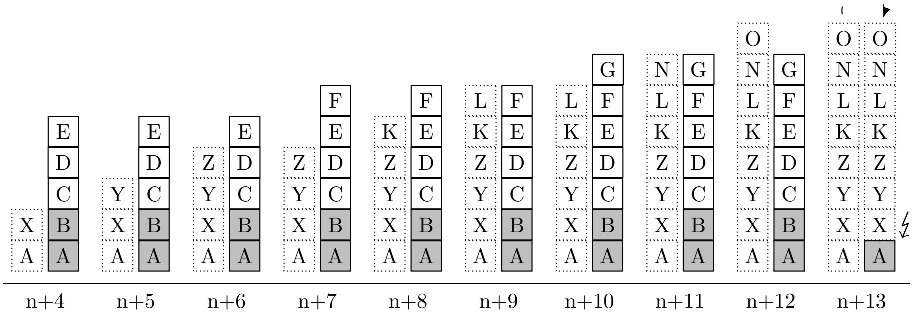


Figure 8.8: Graphical depiction of a double spend attack.

a situation where the length of the untrusted blockchain finally overtakes the length of the trusted node's blockchain (time point $n + 12$) which will then accept the untrusted blockchain as required by the consensus rule (time point $n + 13$). Thus, the first entry (block A) was modified although it was already confirmed. ☐

8.7.3 A Note on Practical Feasibility

We admit that the constraint discussed in this section may somehow be idealized and difficult to verify in a practical environment. To verify it, one would indeed need to control the mining ability of untrusted entities which is not really feasible. Similar problems arise for other constraints provided in this chapter, such as the connection constraint from Fig. 8.2 which requires trusted nodes to be always connected.

This reveals a general characteristic of Blockchain architectures: they usually don't provide any strong guarantees. Rather, their guarantees are of probabilistic nature. Nevertheless, the constraints presented in this chapter formalize key assumptions behind Blockchain architectures and they may indeed be used by an architect to analyze a given Blockchain architecture: by guessing (or even measuring) the likelihood of the properties to be true in a given context, he can make an educated guess about the likelihood that blockchain entries are indeed persistent.

8.8 Summary

To evaluate the approach on a larger case study, we formalized a pattern for Blockchain architectures based on the proof-of-work consensus algorithm [Nak08]. We then verify a characteristic property for such architectures: *persistence* of blockchain entries [KRDO17]. While a detailed discussion of the blockchain pattern is beyond the scope of this paper, in the following, we provide an overview of the required effort to verify the pattern.

The verification is split intro 3 different Isabelle/HOL theories available at [Mar18c]:

- A theory `Auxiliary` which contains some auxiliary results, such as custom induction rules.
- A theory `RF_LTL` which contains a calculus for Blockchain architectures based on counting LTL [LMP10].
- A theory `Blockchain` which is the main theory containing the actual formalization of the pattern.

Figure 8.9 depicts the effort of the corresponding verification in terms of proof code for each proposition. The key property is formalized as theorem *blockchain-save* (highlighted with gray color in the figure). Its proof is by induction and consists of roughly 300 lines of Isabelle/HOL proof code. It required to introduce two auxiliary concepts:

- a set `his` containing a blockchain's history, i.e., its state during certain points in time and

- a function `devBC`, representing a blockchain's development.

The main remaining propositions are then concerned with these two concepts:

- Lemma `his_determ_ext` shows that the history of a blockchain is deterministic, i.e., that it has a unique state at each point in time.
- Lemma `devExt_devop` proves a basic property for blockchain developments, i.a., that it can only grow by one through a mining process.
- Lemma `devExt` shows that the development of a blockchain (which is defined using its history), is indeed a well-defined function.

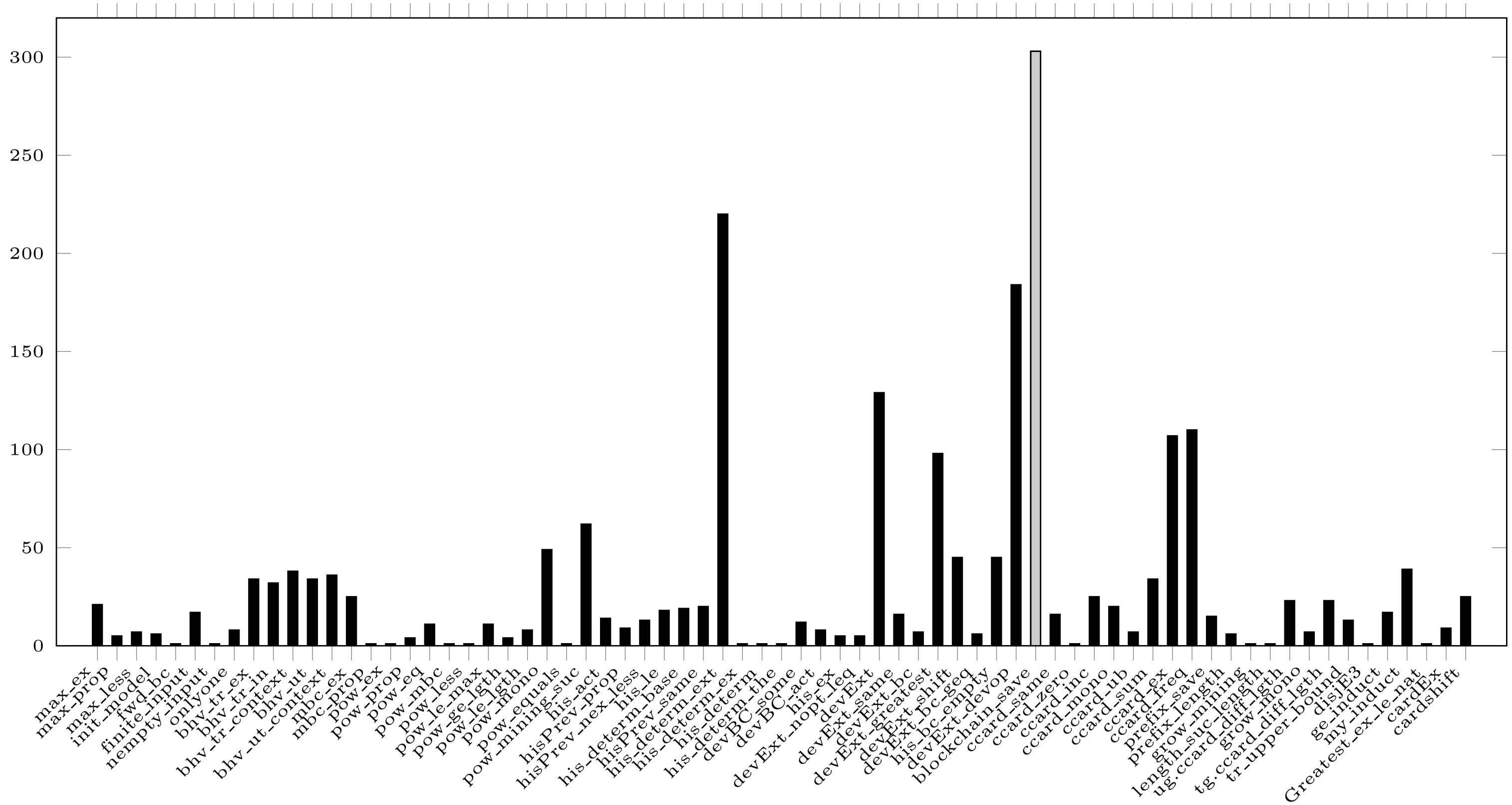


Figure 8.9: Propositions for the Blockchain pattern.

Part V

Conclusion

9 Conclusion

This thesis introduced FACTUM, a methodology for the axiomatic specification and verification of architectural design patterns (ADPs). Therefore, we first developed a model for (potentially dynamic) architectures and techniques to specify ADPs over this model. Then, we developed a calculus to support the verification of such specifications, implemented the calculus in Isabelle/HOL, and provided an algorithm to map a given specification to a corresponding Isabelle/HOL theory. To evaluate it, FACTUM was implemented in Eclipse/EMF and applied for the verification of 4 different ADPs: the Singleton, the Publisher-Subscriber, the Blackboard pattern, and a pattern for Blockchain architectures. To conclude the thesis, in the following, we first summarize the major results obtained with this thesis and discuss possible implications thereof. Then, we describe our overall research agenda and point to future work which is needed to achieve our vision.

9.1 Summary

Chap. 2 introduces our model for dynamic architectures and Chap. 3 and Chap. 4 present techniques to specify ADPs over this model. Chap. 5 then presents a calculus to reason about such specifications and in Chap. 6 we presents the formalization of the model and the calculus in Isabelle/HOL. Chap. 7 then combines all the results into an overall methodology for the interactive verification of ADPs. Finally, Chap. 8 presents a case study in which the approach is used to verify a pattern for blockchain architectures. In the following, we summarize each chapter in more detail.

9.1.1 A Model of Dynamic Architectures

Since patterns exist for static as well as for dynamic architectures, our approach is based on a model of dynamic architectures, which is described in detail in Chap. 2. Our model is a dynamic version of Broy's FOCUS model [BS01] and consists of the following main concepts:

- messages and ports (typed with sets of messages),
- interfaces consisting of input and output ports,
- a set of component types which consist of an interface, component parameters valuated with messages, and associated behavior in terms of a causal set of behavior traces, i.e., streams of snapshots of a component during execution,

- an architecture specification consisting of a set of architecture traces, i.e., streams of snapshots of an architecture during execution,
- a projection operator, which extracts the behavior of a single component out of a given architecture trace, and
- a composition operator which combines a set of component types with a given architecture specification.

9.1.2 Basic Specification Techniques

Based on the model presented in Chap. 2, we describe basic techniques for the axiomatic specification of ADPs in Chap. 3. Such a specification consist of three parts: an interface specification, a component type specification, and a specification of architectural constraints.

9.1.2.1 Interface Specification

An interface specification consists of a specification of the abstract data types used in a pattern, a set of port identifiers typed by these data types, and a set of interfaces over these ports. Data types are specified using traditional, algebraic specification techniques [Bro96], and interfaces can be specified using a graphical specification language called *architecture diagrams*.

In order to support the specification of related types of components (which is often required for the specification of ADPs), we also provide a notion of *parametrized* component types. Therefore, interfaces may contain so-called *interface parameters* which are typed by the abstract datatypes introduced for the pattern. Their semantics requires that at least one component exists for each valuation of interface parameters, which allows to introduce the notion of parametrized component variables. Such variables are guaranteed to be interpreted only by components with corresponding parameter values and thus support the specification of component types and architectural assumptions.

9.1.2.2 Component Type Specification

A component type specification consists of a set of axioms for each interface to specify assumptions about the behavior of components of a certain type. In order to specify these axioms, we introduce the notion of *behavior trace assertion*, a type of linear temporal logic with component ports as free variables.

9.1.2.3 Specifying Architectural Constraints

Architectural constraints are formulated over all interfaces to specify assumptions about the activation/deactivation of components and their connections. To specify these axioms, we introduce the notion of *architecture trace assertions*, which are again a type of linear temporal logic with special predicates to denote component activation/deactivation and connections between the ports of components.

9.1.3 Advanced Specification Techniques

In order to facilitate the specification of certain activation and connection constraints, Chap. 4 introduces various types of *annotations* for *architecture diagrams*. Moreover, a pattern specification may reuse other pattern specifications by instantiating the corresponding component types.

9.1.3.1 Annotations for Architecture Diagrams

We provide three types of annotations for architecture diagrams: *activation annotations*, *connection annotations*, and *dependencies*. In general, annotations are graphical synonyms for corresponding architectural assertions and their semantics is given by mapping them to corresponding architecture trace assertions (as introduced in Chap. 3).

Activation annotations Activation annotations allow to annotate a component type with pre- and postconditions regarding the activation and deactivation of components of that type.

Connection annotations Connection annotations allow to specify pre- and postconditions for connections by annotating the corresponding edge in an architecture diagram.

Dependencies Dependencies allow to express relationships between components of certain types by connecting the corresponding interfaces in an architecture digram. These relationships can then be used to express *relative* activation and connection conditions, i.e., conditions which depend on certain conditions from a related component.

Hierarchical Specifications In order to address the hierarchical nature of patterns, FACTUM specifications allow for hierarchical pattern specifications, i.e., patterns can be instantiated for the specification of other patterns. Pattern instantiations are expressed by annotating the interfaces of architecture diagrams by so-called *port mappings*, i.e., mappings which relate the ports of instantiated components with the ports of the corresponding instantiating component.

9.1.4 Reasoning about Pattern Specifications

To support the verification of FACTum specifications, Chap. 5 introduces a calculus to reason about FACTUM specifications. The calculus formalizes reasoning about a component type specification in the context of a corresponding component activation specification. It provides introduction and elimination rules for each operator involved in a FACTum specification and consists of roughly 35 rules. It is shown to be sound and it is implemented in Isabelle/HOL, where it can be used for the verification of pattern specifications.

9.1.5 Evaluation

In order to evaluate our approach, we implemented it in Eclipse/EMF and used it to verify properties for our running examples as well as a larger case study from the domain of Blockchain architectures.

9.1.5.1 FACTum Studio

To support an architect in the specification of ADPs, FACTUM comes with tool support in terms of a corresponding Eclipse/EMF application. The tool supports the graphical modeling of architecture diagrams (as described in Chap. 4), which can then be enriched by corresponding textual specifications. To support the textual specifications, the tool also provides rigorous type checking mechanisms for the specification of datatypes, component types, and architectural assumptions (as described in Chap. 3). Finally, the tool implements the algorithm presented in Chap. 6 to generate corresponding Isabelle/HOL theories on top of the verification framework presented in Chap. 5.

9.1.5.2 Running Examples

We demonstrated our approach by means of three running examples: the singleton, the publisher subscriber, and the blackboard pattern. For each pattern, we provide a formal specification of the pattern's assumptions and corresponding guarantees. Then, we verify each of them in Isabelle/HOL. To demonstrate hierarchical specification and verification, the publisher component is modeled as an instance of the singleton and the blackboard pattern is specified as an instance of the publisher subscriber pattern.

9.1.5.3 Case Study: Verified Blockchain Architectures

For our case study we propose a pattern for blockchain architectures based on the proof of work consensus algorithm. Therefore, we first apply the specification techniques from Chap. 3 and Chap. 4 to formalize the patterns assumptions as well as an important guarantee for blockchain architectures: persistence of blockchain entries. We then map the specification to a corresponding Isabelle/HOL theory and the guarantee to a corresponding Isabelle/HOL theorem (using the algorithm presented in Chap. 6) and verify the theorem using the calculus presented in Chap. 5. Thereby, we discover an important property for blockchain architectures which is essential to ensure its guarantee: relative mining frequencies need to be bounded by the number of confirmation blocks.

9.2 Implications

The methodology presented in this thesis can be used to formally investigate ADPs. Thereby, we address both problems with pattern specifications identified in Chap. 1.

9.2.1 Problem 1: Missing Constraints

Verifying an ADP may reveal constraints assumed by the pattern which are important to meet its guarantee, but which are not mentioned in any specification of the pattern so far. While the major part of such missing constraints is usually concerned with details of an architecture, some of them can be also of more fundamental nature. For example, in this thesis, we discover around 16 assumptions for different ADPs. While many of them are concerned with details about an architecture, two of them may be considered fundamental: The first one is assumed by blackboard architectures and requires problems to be ordered by a subproblem relation which is required to be *well-founded.* This is a fundamental constraint which needs to be ensured before applying the pattern. Otherwise, the corresponding architecture will not be able to solve certain problems and the pattern would not fulfill its purpose. A second fundamental constraint concerns relative mining frequencies in blockchain architectures. In order to apply the pattern, one needs to ensure that it will indeed be highly unlikely that the mining frequency of untrusted nodes exceeds the mining frequency of trusted nodes by the number of confirmation blocks. Otherwise, entries of a blockchain may be subject to modification by untrusted entities and the pattern would fail its guarantee.

9.2.2 Problem 2: Unnecessary Constraints

The support for verification also has the potential to uncover unnecessary constraints in a pattern specification. If certain assumptions a pattern makes about an architecture are not used for the verification of its guarantee, the corresponding constraints can be removed and the scope of the pattern is increased. For example, many descriptions of blockchain architectures require the data entries to be financial transactions with corresponding private and public keys. However, these assumptions are not required in order to guarantee persistence of entries and they unnecessarily restrict the application scope of the pattern.

Note, however, that the problem of too strong assumptions, compared to the problem of too weak assumptions, cannot be guaranteed to be solved by verifying the corresponding ADP. A proof of an architectural guarantee may indeed contain unnecessary references to architectural constraints. However, if the proof does not contain any reference to an architectural constraint, the corresponding architectural design constraint can be safely removed from a pattern's specification.

9.3 Limitations

In the following, we take a critical look at the results obtained with this thesis.

9.3.1 Non-functional Aspects

When it comes to ADPs, non-functional aspects play an important role. Many patterns are actually invented to address certain non-functional aspects, such as maintainability.

With the approach presented in this thesis it is not possible to investigate whether or not a certain pattern really satisfies certain non-functional aspects. Rather, with our approach we focus on the correct implementation of a pattern and we consider them as lemmata to support the verification of architectures using these patterns. Nevertheless, we admit that non-functional aspects play an important role and indeed a lot of research in the architecture community is devoted to this aspect. One line of research uses a quantitative approach and aims towards the development of pattern-specific cost models for certain quality attributes [KK99, Mar10]. Another line of research follows a more qualitative approach and uses so-called quality attribute scenarios to evaluate quality attributes for patterns [BCK07].

9.3.2 Target Audience

Using interactive theorem proving make the approach presented in this thesis very general and thus able to address the abstract nature of patterns. However, it makes the approach also difficult to apply, since ITP comes with a steep learning curve and is not yet well-known in the architecture community. The algorithm (and its implementation in Eclipse/EMF) as well as the calculus to support the interactive verification of patterns in Isabelle/HOL are first steps towards making the approach accessible to a broader audience. However, users still need to have some expertise in ITP to efficiently use the approach and thus it might be difficult to apply for practitioners. Thus, as of now, the target group of the approach is mainly researchers in software architectures. In the next section, however, we also provide some ideas for future work to make the approach usable also for practitioners.

9.4 Outlook

Figure 9.1 depicts our overall research agenda. We basically consider ADPs as lemmata for the verification of architectures. To this end, which we envision a *repository* containing a growing collection of verified ADPs. Researchers can connect to the repository and fill it with verification results for existing or even new ADPs. Thereby, they can leverage the hierarchical nature of patterns and verify higher-level patterns using available results from lower level patterns. When verifying an architecture, an architect can connect to the repository and verify the architecture against the assumptions provided by the ADPs. The corresponding guarantee is then automatically transferred to the architecture and can be used to support its verification.

9.5 Future Work

To achieve our vision, future work is required in at least two areas: The development of an interactive pattern verification language and the integration of our approach in current architecture verification practice.

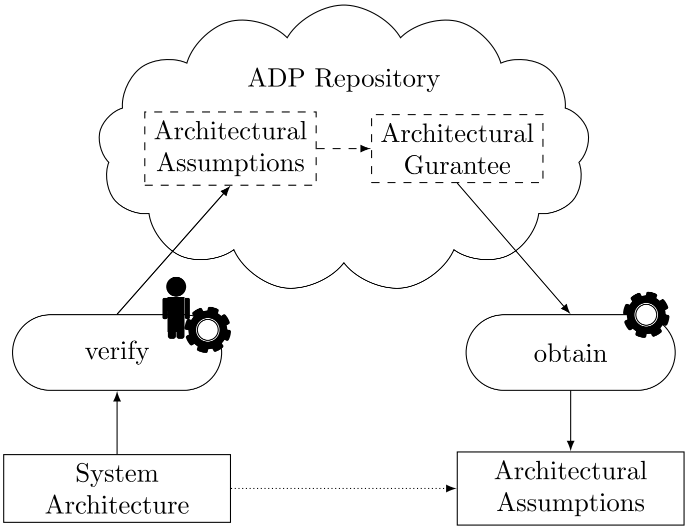


Figure 9.1: Research vision: A repository of verified ADPs.

9.5.1 Interactive Pattern Verification Language

With this thesis, we provide a first step towards interactive pattern verification: An architect can specify a pattern in Eclipse/EMF and then generate a corresponding Isabelle/HOL theory out of it. Then he can verify the pattern in Isabelle/HOL using a corresponding calculus.

However, as discussed above, architects are usually not trained in interactive theorem proving and future work should investigate possibilities to further support an architect in the verification process. A first step could be the development of a more abstract proof language which allows an architect to sketch a proof using abstractions he is familiar with, such as sequence diagrams. The abstract proof should then be translated to a corresponding Isabelle/Isar proof and verified by Isabelle.

9.5.2 Integration into Architecture Verification

Another crucial step to achieve our vision concerns the integration of verification results obtained for ADPs to support the verification of architectures. Compared to the verification of ADPs (which can be reused for different architectures), verification of architectures against ADPs has to be done for each architecture, which is why future work should investigate possibilities to automate this step.

A Conventions

A.1 Sets

Convention 1 (Natural numbers)**.** *We denote with* $\mathbb{N}$ *the set of* natural numbers*, with* $\mathbb{N}^+$ *the set of* positive *natural numbers (excluding* 0*), and with* $\mathbb{N}_\infty$ *the set of* extended *natural numbers (including* ∞*).*

Convention 2 (Powerset)**.** *We denote with* $\wp(S)$ *the* powerset *of a set* S*, i.e., the set containing all subsets of* S*.*

Convention 3 (Tuples)**.** *For an* n*-tuple* $c = (c_1, \ldots, c_n)$ *(where* $n \in \mathbb{N}^+$*), we denote by* $c_{(i)} = c_i$ *with* $1 \leq i \leq n$ *the projection to the* i*-th component of* c*.*

Convention 4 (Indexed family of sets)**.** *Given a non-empty set* I*, we denote with* $(S_i)_{i \in I}$ *a family of sets indexed by* I*, i.e., a mapping associating a set* S_i *with each element* $i \in I$*.*

A.2 Functions

Convention 5 (Functions)**.** *Given two sets* A *and* B*, we denote with* $A \to B$ *the set of functions with domain* A *and range* B*. For a function* $f\colon A \to B$ *we denote with* $\operatorname{dom}(f) \overset{def}{=} A$ *the domain of* f *and with* $\operatorname{ran}(f) \overset{def}{=} B$ *its range.*

Given four sets A, B, C, D*, we denote with* $(A \to B)\overrightarrow{\cup}(C \to D)$ *the set of all functions* $f\colon (A \cup C) \to (B \cup D)$*, such that* $\forall x \in A\colon f(x) \in B$ *and* $\forall x \in C\colon f(x) \in D$*.*

For a function $f\colon D \to R$ *and an element* $r \in R$*, we denote with* $f^{-1}(r) \overset{def}{=} \{d \in D \mid f(d) = r\}$ *the* inverse image *of* r *in* f*.*

Convention 6 (Function merge)**.** *For two functions* $f\colon A \to B$ *and* $g\colon C \to D$ *with* disjoint *domains* $A \cap C = \emptyset$*, we denote with* $f \cup g\colon A \cup C \to B \cup D$ *their merge:*

$$(f \cup g)(x) \overset{def}{=} \begin{cases} f(x) & \text{if } x \in A \ , \\ g(x) & \text{else} \ . \end{cases}$$

Convention 7 (Function update)**.** *For a function* $f\colon D \to R$ *and elements* $d \in D$ *and* $r \in R$*, we denote with* $f[d \mapsto r]\colon D \to R$ *a function which is equal to* f *but maps* d *to* r*:*

$$f[d \mapsto r](x) \overset{def}{=} \begin{cases} r & \text{if } x = d \ , \\ f(x) & \text{else} \ . \end{cases}$$

Convention 8 (Indexed family of functions)**.** *For two indexed families of functions* $F = (F_i)_{i \in I}$ *and* $F' = (F_i)_{i \in I}$*, with disjoint domains* $\operatorname{dom}(F_i) \cap \operatorname{dom}(F'_i) = \emptyset$ *for each* $i \in I$*, we denote with* $F \cup F' = (F \cup F'_i)_{i \in I}$ *a new family of functions with* $\left(F \cup F'\right)_i \overset{def}{=} F_i \cup F'_i$*.*

For an indexed family of functions $F = (F_i)_{i \in I}$*, index* $j \in I$*, function* $F_j \colon D \to R$*, elements* $d \in D$ *and* $r \in R$*, we denote by* $F[j \colon d \mapsto r]$ *an indexed family of functions where function* F_j *is updated to* $F_j[d \mapsto r]$*.*

$$F[j \colon d \mapsto r]_i \overset{def}{=} \begin{cases} F_i[d \mapsto r] & \textit{if } i = j \ , \\ F_i & \textit{else} \ . \end{cases}$$

Convention 9 (Mappings)**.** *We denote by* $[i_1, i_2, \ldots \mapsto o_1, o_2, \ldots]$ *a function which maps input* i_1 *to output* o_1*, input* i_2 *to output* o_2*, etc.*

A.3 Sequences

Convention 10 (Sequences)**.** *Given any set* E*, we denote by* $(E)^*$ *the set of all finite sequences over* E*, by* $(E)^\infty$ *the set of all infinite sequences over* E*, and by* $(E)^\omega$ *the set of all finite and infinite sequences over* E*. We use the following notations for sequences:*

- *With* $\langle\rangle$ *we denote the empty sequence.*
- *Similar as for restriction of functions, we shall use* $s|_n$ *to extract the first* n *elements of a sequence. Thereby* $s|_0 \overset{def}{=} \langle\rangle$*.*
- *For a sequence* s*, we denote by* $\#s$ *the length of* s *and with* $s\&e$ *the sequence resulting by appending element* $e \in E$ *to sequence* s*.*
- *For two sequences* s *and* s'*, we denote by* $s^\frown s'$ *the concatenation of* s *and* s'*.*
- *With* $s' \sqsubseteq s$ *we denote that* s' *is a prefix of* s*.*
- *With* $rg(s)$ *we denote the set of all elements of a given sequence* s*.*

We assume the following properties for sequences s*:*

- $\forall n \in \mathbb{N} \colon s(n) = s|_{n+1}(n)$*. (A.1)*
- $s\&e(\#s) = e$*. (A.2)*

Convention 11 (Prefix)**.** *With* $s' \sqsubseteq s$ *we denote that sequence* s' *is a prefix of* s*.*

A function $m : (E)^\omega \to (E)^\omega$ *such that* $s' \sqsubseteq s \implies m(s') \sqsubseteq m(s)$ *is called prefix-monotonic. For a prefix-monotonic function* m *we assume the following property:*

$$\forall n \in \mathbb{N} \colon m(s)|_{\#m(s|_{n+1})} = m(s|_{n+1}) \ . \tag{A.3}$$

A.4 Logics

Convention 12 (Boolean values)**.** *With true we denote logical truth and with false logical false. With* $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ *we denote the set of boolean values.*

B Proof for Thm. 1

We show that Γ holds for an architecture specification $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$ and for each component type $ct \in \mathcal{CT}$ a property γ_{ct} holds, iff Γ holds for $\otimes_{\mathcal{A}}(\mathcal{C})$ and γ_{ct} holds for the projection to every component $c \in \mathcal{C}$ of type ct in $\otimes_{\mathcal{A}}(\mathcal{C})$.

B.1 $\Longrightarrow$

Assume that $t \in \otimes_{\mathcal{A}}(\mathcal{C})$. We show

1. t fulfills Γ and
2. γ_{ct} holds for the projection to every component $c \in \mathcal{C}$ of type ct.

B.1.1 Goal 1

By Def. 11, $t \in \mathcal{A}$. Thus, by assumption, t fulfills Γ.

B.1.2 Goal 2

Again, by Def. 11, $\forall ct \in \mathcal{CT},\ c \in \mathcal{C}_{ct}\ \exists t' \subseteq (\overline{\mathsf{port}(ct)})^{\infty} \colon \Pi_c(t) ^\frown t' \in \mathsf{bhv}(ct)$. Thus, by assumption, γ_{ct} holds for the projection to every component $c \in \mathcal{C}$ of type ct.

B.2 $\Longleftarrow$

Assume that t fulfills Γ and γ_{ct} holds for the projection to every component $c \in \mathcal{C}$ of type ct. We show $t \in \otimes_{\mathcal{A}}(\mathcal{C})$. To this end, we show that

1. $t \in \mathcal{A}$ and
2. $\forall ct \in \mathcal{CT},\ c \in \mathcal{C}_{ct}\ \exists t' \subseteq (\overline{\mathsf{port}(ct)})^{\infty} \colon \Pi_c(t) ^\frown t' \in \mathsf{bhv}(ct)$.

Then, we conclude $t \in \otimes_{\mathcal{A}}(\mathcal{C})$ by Def. 11.

B.2.1 Goal 1

$t \in \mathcal{A}$ follows directly from the assumption.

B.2.2 Goal 2

Again, by assumption, γ_{ct} holds for the projection to every component $c \in \mathcal{C}$ of type ct.

C Behavior Trace Assertions

Behavior trace assertions are formulated over data type variables, i.e., variables representing messages of a certain type. Thus, given a signature $\Sigma = (S, F, B)$, we assume the existence of a family of data type variables $DV = (DV_s)_{s \in S}$ and rigid data type variables DV'. Both types of variables are interpreted over an algebra $A = ((A_s)_{s \in S}, (f^A)_{f \in F}, (p^A)_{p \in B}) \in \mathcal{A}(\Sigma)$ for signature Σ (where F^n and B^n denote all the function/predicate symbols of arity n, and sf and sp assign a tuple of sorts to each function/predicate symbol, respectively[1]). Thereby, data type variable assignments $\iota = (\iota_s)_{s \in S}$ consist of interpretations $\iota_s \colon DV_s \to A_s$, which are newly evaluated at each point in time. Rigid data type variables, on the other hand, are interpreted only once for the whole execution by a so-called rigid data type variable assignment $\iota' = (\iota'_s)_{s \in S}$. With $\mathcal{I}_A^{DV}$ we denote the set of all data type variable assignments for data type variables DV in algebra A and with $\mathcal{I'}_A^{DV'}$ the set of all rigid data type variable assignments for rigid data type variables DV' in algebra A, respectively.

C.1 Behavior terms

C.1.1 Syntax

Definition 17 (Behavior terms: syntax)**.** *The set of all* behavior terms *of sort $s \in S$ over a signature $\Sigma = (S, F, B)$, datatype variables DV, and port specification $ps = (PID, \mathsf{tp})$, is the smallest set ${}^{s}_{\Sigma}BT_{DV}(ps)$ satisfying the equations of Fig. C.1. The set of all behavior terms of all sorts is denoted by ${}_{\Sigma}BT_{DV}(ps)$.*

---

[1] For function symbols, the sort for the return type is assumed to be on position 0 of the tuple.

Behavior terms: syntax

$$
\begin{aligned}
v \in DV_s \quad &\Longrightarrow \quad \text{“}v\text{”} \in {}^{s}_{\Sigma}BT_{DV}(ps) \ , \\
p \in PID \quad &\Longrightarrow \quad \text{“}p\text{”} \in {}^{s}_{\Sigma}BT_{DV}(ps) \ [\text{for } \mathsf{tp}(p) = s] \ , \\
f \in F^0 \quad &\Longrightarrow \quad \text{“}f\text{”} \in {}^{s}_{\Sigma}BT_{DV}(ps) \ [\text{for } \mathsf{sf}(f)_{(0)} = s] \ ,
\end{aligned}
$$

$$
\left.\begin{array}{l}
f \in F^{n+1} \quad \wedge \\
\text{“}t_1\text{”} \in {}^{s_1}_{\Sigma}BT_{DV}(ps), \cdots , \\
\text{“}t_{n+1}\text{”} \in {}^{s_{n+1}}_{\Sigma}BT_{DV}(ps)
\end{array}\right\} \quad \Longrightarrow \quad
\left\{\begin{array}{l}
\text{“}f(t_1, \cdots, t_{n+1})\text{”} \in {}^{s}_{\Sigma}BT_{DV}(ps) \\
[\text{for } n \in \mathbb{N},\ \mathsf{sf}(f)_{(0)} = s, \text{ and} \\
\mathsf{sf}(f)_{(1)} = s_1, \cdots , \mathsf{sf}(f)_{(n+1)} = s_{n+1}] \ .
\end{array}\right.
$$

Figure C.1: Inductive definition of behavior terms.

Behavior terms: semantics

$$
\begin{aligned}
{}^{\iota}_{A}[\![\text{“}v\text{”}]\!]^{\delta}_{\mu} &= \iota_s(v) \text{ [for } v \in DV_s] \ , \\
{}^{\iota}_{A}[\![\text{“}p\text{”}]\!]^{\delta}_{\mu} &= \mu(\delta(p)) \text{ [for } p \in PID] \ , \\
{}^{\iota}_{A}[\![\text{“}f\text{”}]\!]^{\delta}_{\mu} &= A_f \text{ [for function symbol } f \in F^0] \ , \\
{}^{\iota}_{A}[\![\text{“}f(t_1, \cdots, t_n)\text{”}]\!]^{\delta}_{\mu} &= \begin{cases} A_f(\, {}^{\iota}_{A}[\![\text{“}t_1\text{”}]\!]^{\delta}_{\mu}, \cdots, {}^{\iota}_{A}[\![\text{“}t_n\text{”}]\!]^{\delta}_{\mu}) \\ \text{[for function symbol } f \in F^{n+1}] \ . \end{cases}
\end{aligned}
$$

Figure C.2: Recursive definition of semantic function for behavior terms.

C.1.2 Semantics

Definition 18 (Behavior terms: semantics)**.** *The semantics of behavior terms* ${}_{\Sigma}BT_{DV}(ps)$*, formulated over port specification* $ps = (PID, \mathtt{tp})$*, is defined over an algebra* $A \in \mathcal{A}(\Sigma)$ *with corresponding data type variable assignments* $\iota \in \mathcal{I}_A^{DV}$ *and a valuation* $\mu \in \overline{\mathcal{P}}$ *of a set of ports* $\mathcal{P}$ *with corresponding interpretation* $\delta\colon PID \to \mathcal{P}$ *for the port identifiers of ps. It is given by a semantic function* ${}^{\iota}_{A}[\![\_]\!]^{\delta}_{\mu}\colon \overrightarrow{\bigcup}_{s \in S}({}^{s}_{\Sigma}BT_{DV}(ps) \to A_s)$*, defined recursively by the equations provided in Fig. C.2.*

C.2 Behavior assertions

C.2.1 Syntax

Definition 19 (Behavior assertions: syntax)**.** *The set of all* behavior assertions *over a signature* $\Sigma = (S, F, B)$*, datatype variables* DV*, and port specification* $ps = (PID, \mathtt{tp})$*, is the smallest set* ${}_{\Sigma}BA_{DV}(ps)$ *satisfying the equations of Fig. C.3.*

C.2.2 Semantics

Definition 20 (Behavior assertions: semantics)**.** *The semantics of* behavior assertions ${}_{\Sigma}BA_{DV}(ps)$*, formulated over port specification* $ps = (PID, \mathtt{tp})$*, is defined over an algebra* $A \in \mathcal{A}(\Sigma)$ *with corresponding data type variable assignments* $\iota \in \mathcal{I}_A^{DV}$ *and an interpretation* $\delta\colon PID \to \mathcal{P}$ *for the port identifiers of ps with concrete ports of a set* $\mathcal{P}$*. It is given by a relation* $\models^{\delta}_{A,\iota} \subseteq \overline{\mathcal{P}} \times {}_{\Sigma}BA_{DV}(ps)$ *defined recursively by the equations provided in Fig. C.4*

C.3 Behavior trace assertions

C.3.1 Syntax

Definition 21 (Behavior trace assertions: syntax)**.** *The set of all* behavior trace assertions *over a signature* $\Sigma = (S, F, B)$*,* disjoint *sets of datatype variables* DV *and*

Behavior assertions: syntax

$$\begin{array}{rcl}
 & & \text{“}\mathit{true}\text{”} \in {}_{\Sigma}BA_{DV}(ps)\ , \\
 & & \text{“}\mathit{false}\text{”} \in {}_{\Sigma}BA_{DV}(ps)\ , \\
b \in B^{0} & \Longrightarrow & \text{“}b\text{”} \in {}_{\Sigma}BA_{DV}(ps)\ , \\
\left.\begin{array}{l} b \in B^{n+1} \quad \wedge \\ \text{“}t_1\text{”} \in {}^{s_1}_{\Sigma}BT_{DV}(ps), \cdots, \\ \text{“}t_{n+1}\text{”} \in {}^{s_{n+1}}_{\Sigma}BT_{DV}(ps) \end{array}\right\} & \Longrightarrow & \left\{\begin{array}{l} \text{“}b(t_1, \cdots, t_{n+1})\text{”} \in {}_{\Sigma}BA_{DV}(ps) \\ \text{[for } n \in \mathbb{N} \text{ and} \\ \mathsf{sp}(b)_{(1)} = s_1, \cdots, \mathsf{sp}(b)_{(n+1)} = s_{n+1}]\ , \end{array}\right. \\
\text{“}t\text{”}, \text{“}t'\text{”} \in {}^{s}_{\Sigma}BT_{DV}(ps) & \Longrightarrow & \text{“}t = t'\text{”} \in {}_{\Sigma}BA_{DV}(ps) \text{ [for some } s \in S]\ , \\
\text{“}\phi\text{”} \in {}_{\Sigma}BA_{DV}(ps) & \Longrightarrow & \text{“}\neg\phi\text{”} \in {}_{\Sigma}BA_{DV}(ps)\ , \\
\text{“}\phi\text{”}, \text{“}\phi'\text{”} \in {}_{\Sigma}BA_{DV}(ps) & \Longrightarrow & \left\{\begin{array}{l} \text{“}\phi \wedge \phi'\text{”} \in {}_{\Sigma}BA_{DV}(ps), \\ \text{“}\phi \vee \phi'\text{”} \in {}_{\Sigma}BA_{DV}(ps), \\ \text{“}\phi \longrightarrow \phi'\text{”} \in {}_{\Sigma}BA_{DV}(ps), \\ \text{“}\phi \longleftrightarrow \phi'\text{”} \in {}_{\Sigma}BA_{DV}(ps).\ , \end{array}\right. \\
\text{“}\phi\text{”} \in {}_{\Sigma}BA_{DV}(ps) \wedge x \in DV_s & \Longrightarrow & \left\{\begin{array}{l} \text{“}\forall x\colon\ \phi\text{”} \in {}_{\Sigma}BA_{DV}(ps), \\ \text{“}\exists x\colon\ \phi\text{”} \in {}_{\Sigma}BA_{DV}(ps) \text{ [for } s \in S].\ . \end{array}\right.
\end{array}$$

Figure C.3: Inductive definition of behavior assertions.

rigid datatype variables DV'*, and port specification* $ps = (PID, \mathtt{tp})$*, is the smallest set* ${}_{\Sigma}BTA^{DV'}_{DV}(ps)$ *satisfying the equations of Fig. C.5.*

C.3.2 Semantics

Definition 22 (Behavior trace assertions: semantics)**.** *The semantics of behavior trace assertions* ${}_{\Sigma}BTA^{DV'}_{DV}(ps)$*, formulated over port specification* $ps = (PID, \mathtt{tp})$*, is defined over an algebra* $A \in \mathcal{A}(\Sigma)$ *with corresponding rigid data type variable assignments* $\iota' \in \mathcal{I}'^{DV'}_{A}$ *and an interpretation* $\delta\colon PID \to \mathcal{P}$ *for the port identifiers of ps with concrete ports of a set* $\mathcal{P}$*. It is given by a relation* $\overset{\delta}{\underset{A,\iota'}{\models}} \subseteq ((\overline{\mathcal{P}})^{\infty} \times \mathbb{N}) \times {}_{\Sigma}BA_{DV}(ps)$ *defined recursively by the equations provided in Fig. C.6.*

Behavior assertions: semantics

$$
\begin{array}{rcl}
\mu \models^{\delta}_{A,\iota} \text{“}\mathit{true}\text{”} & & , \\
\neg(\mu \models^{\delta}_{A,\iota} \text{“}\mathit{false}\text{”}) & & , \\
\mu \models^{\delta}_{A,\iota} \text{“}b\text{”} & \Longleftrightarrow & A_b \text{ [for } b \in B^0\text{]} \;, \\
\mu \models^{\delta}_{A,\iota} \text{“}b(t_1, \cdots, t_n)\text{”} & \Longleftrightarrow & A_b({}^{\iota}_{A}[\![\text{“}t_1\text{”}]\!]^{\delta}_{\mu}, \cdots, {}^{\iota}_{A}[\![\text{“}t_n\text{”}]\!]^{\delta}_{\mu}) \text{ [for } b \in B^{n+1}\text{]} \;, \\
\mu \models^{\delta}_{A,\iota} \text{“}t = t'\text{”} & \Longleftrightarrow & {}^{\iota}_{A}[\![\text{“}t\text{”}]\!]^{\delta}_{\mu} = {}^{\iota}_{A}[\![\text{“}t'\text{”}]\!]^{\delta}_{\mu} \;, \\
\mu \models^{\delta}_{A,\iota} \text{“}\phi \wedge \phi & \Longleftrightarrow & \mu \models^{\delta}_{A,\iota} \text{“}\phi\text{”} \wedge \mu \models^{\delta}_{A,\iota} \text{“}\phi'\text{”} \;, \\
\mu \models^{\delta}_{A,\iota} \text{“}\phi \vee \phi'\text{”} & \Longleftrightarrow & \mu \models^{\delta}_{A,\iota} \text{“}\phi\text{”} \wedge \mu \models^{\delta}_{A,\iota} \text{“}\phi'\text{”} \;, \\
\mu \models^{\delta}_{A,\iota} \text{“}\phi \longrightarrow \phi'\text{”} & \Longleftrightarrow & \mu \models^{\delta}_{A,\iota} \text{“}\phi\text{”} \wedge \mu \models^{\delta}_{A,\iota} \text{“}\phi'\text{”} \;, \\
\mu \models^{\delta}_{A,\iota} \text{“}\phi \longleftrightarrow \phi'\text{”} & \Longleftrightarrow & \mu \models^{\delta}_{A,\iota} \text{“}\phi\text{”} \wedge \mu \models^{\delta}_{A,\iota} \text{“}\phi'\text{”} \;, \\
\mu \models^{\delta}_{A,\iota} \text{“}\exists x\colon\; \phi\text{”} & \Longleftrightarrow & \begin{cases} \exists x' \in A_s \colon \mu \models^{\delta}_{A,\iota[s\colon x \mapsto x']} \text{“}\phi\text{”} \\ \text{[for } s \in S \text{ and } x \in DV_s\text{]} \;, \end{cases} \\
\mu \models^{\delta}_{A,\iota} \text{“}\forall x\colon\; \phi\text{”} & \Longleftrightarrow & \begin{cases} \forall x' \in A_s \colon \mu \models^{\delta}_{A,\iota[s\colon x \mapsto x']} \text{“}\phi\text{”} \\ \text{[for } s \in S \text{ and } x \in DV_s\text{]} \;, \end{cases}
\end{array}
$$

Figure C.4: Recursive definition of satisfaction relation for behavior assertions.

Behavior trace assertions: syntax

$$
\begin{array}{rcl}
 & & \text{“}\mathit{true}\text{”},\; \text{“}\mathit{false}\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps), \\
\phi \in {}_{\Sigma}BA_{DV \cup DV'}(ps) & \Longrightarrow & \phi \in {}_{\Sigma}BTA^{DV'}_{DV}(ps), \\
\text{“}\gamma\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps) & \Longrightarrow & \text{“}\neg\gamma\text{”},\; \text{“}\bigcirc\gamma\text{”},\; \text{“}\Diamond\gamma\text{”},\; \text{“}\Box\gamma\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps), \\
\text{“}\gamma\text{”}, \text{“}\gamma'\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps) & \Longrightarrow & \begin{cases} \text{“}\gamma \wedge \gamma'\text{”},\; \text{“}\gamma \vee \gamma'\text{”}, \\ \text{“}\gamma \longrightarrow \gamma'\text{”},\; \text{“}(\gamma' \;\mathcal{U}\; \gamma)\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps), \end{cases} \\
\left.\begin{array}{l} x \in DV' \;\wedge \\ \text{“}\gamma\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps) \end{array}\right\} & \Longrightarrow & \begin{cases} \text{“}\forall x\colon\; \gamma\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps), \\ \text{“}\exists x\colon\; \gamma\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps). \end{cases}
\end{array}
$$

Figure C.5: Inductive definition of behavior trace assertions.

Behavior trace assertions: semantics

$$\begin{aligned}
(t,n) &\models^{\delta}_{A,\iota'} \text{``}\mathit{true}\text{''} && , \\
\neg\big((t,n) &\models^{\delta}_{A,\iota'} \text{``}\mathit{false}\text{''}\big) && , \\
(t,n) &\models^{\delta}_{A,\iota'} \phi && \iff \forall \iota \in \mathcal{I}_A^{DV} : t(n) \models_{A,\iota\cup\iota'} \phi \text{ [for } \phi \in {}_{\Sigma}BA_{DV}(ps)], \\
(t,n) &\models^{\delta}_{A,\iota'} \text{``}\bigcirc\gamma\text{''} && \iff (t,n+1) \models^{\delta}_{A,\iota'} \text{``}\gamma\text{''}, \\
(t,n) &\models^{\delta}_{A,\iota'} \text{``}\Diamond\gamma\text{''} && \iff \exists n' \geq n\colon (t,n') \models^{\delta}_{A,\iota'} \text{``}\gamma\text{''}, \\
(t,n) &\models^{\delta}_{A,\iota'} \text{``}\Box\gamma\text{''} && \iff \forall n' \geq n\colon (t,n') \models^{\delta}_{A,\iota'} \text{``}\gamma\text{''}, \\
(t,n) &\models^{\delta}_{A,\iota'} \text{``}\gamma' \,\mathcal{U}\, \gamma\text{''} && \iff \begin{cases} \exists n' \geq n\colon (t,n') \models^{\delta}_{A,\iota'} \text{``}\gamma\text{''} \ \wedge \\ \forall n \leq m < n'\colon (t,m) \models^{\delta}_{A,\iota'} \text{``}\gamma'\text{''}, \end{cases} \\
(t,n) &\models_{A,\iota'} \text{``}\exists x\colon\ \gamma\text{''} && \iff \begin{cases} \exists x' \in A_s\colon (t,n) \models^{\delta}_{A,\iota'[s\colon x\mapsto x']} \text{``}\gamma\text{''} \\ \text{[for } s \in S \text{ and } x \in DV_s] \ , \end{cases} \\
(t,n) &\models_{A,\iota'} \text{``}\forall x\colon\ \gamma\text{''} && \iff \begin{cases} \forall x' \in A_s\colon (t,n) \models^{\delta}_{A,\iota'[s\colon x\mapsto x']} \text{``}\gamma\text{''} \\ \text{[for } s \in S \text{ and } x \in DV_s] \ . \end{cases}
\end{aligned}$$

Figure C.6: Recursive definition of satisfaction relation for behavior trace assertions.

Architecture terms: syntax

$$v \in DV_s \Longrightarrow \text{“}v\text{”} \in {}^{s}_{\Sigma}CT^{DV}_{CV}(is)\ ,$$

$$v \in (CV_i)_\omega \land p \in \mathsf{port}(\mathsf{if}(i)) \Longrightarrow \begin{cases} \text{“}v.p\text{”} \in {}^{s}_{\Sigma}CT^{DV}_{CV}(is) \\ [\text{for } i \in I \text{ and } \mathsf{tp}(p) = s]\ , \end{cases}$$

$$f \in F^0 \Longrightarrow \text{“}f\text{”} \in {}^{s}_{\Sigma}CT^{DV}_{CV}(is)\ [\text{for } \mathsf{sf}(f)_{(0)} = s]\ ,$$

$$\left.\begin{array}{l} f \in F^{n+1} \quad \land \\ \text{“}t_1\text{”} \in {}^{s_1}_{\Sigma}CT^{DV}_{CV}(is), \cdots, \\ \text{“}t_{n+1}\text{”} \in {}^{s_{n+1}}_{\Sigma}CT^{DV}_{CV}(is) \end{array}\right\} \Longrightarrow \begin{cases} \text{“}f(t_1, \cdots, t_{n+1})\text{”} \in {}^{s}_{\Sigma}CT^{DV}_{CV}(is) \\ [\text{for } n \in \mathbb{N},\ \mathsf{sf}(f)_{(0)} = s, \text{ and} \\ \mathsf{sf}(f)_{(1)} = s_1, \cdots, \mathsf{sf}(f)_{(n+1)} = s_{n+1}]\ . \end{cases}$$

Figure C.7: Inductive definition of architecture terms.

C.4 Architecture Trace Assertions

In addition to variables for data types (as introduced already for behavior trace assertions), architecture trace assertions are formulated also over component variables, i.e., variables representing components of a certain type. Thus, given a signature Σ and an interface specification $is = (I, \mathsf{if})$ over port specification (PID, tp), we assume the existence of a family of component variables $CV = (CV_i)_{i \in I}$ with component variables $CV_i = \big((CV_i)_\omega\big)_{\omega\colon p \mapsto DtT_{\mathsf{tp}(p)}(\Sigma, DV)}$ for each interface $i \in I$ and each valuation of component parameters ω. In addition, we assume the existence of a corresponding family of rigid component variables CV'.

Component variables are interpreted over a family of components $\mathcal{C} = (\mathcal{C}_{ct})_{ct \in CT_{\mathcal{I}}}$ by a so-called component variable assignment $\kappa = (\kappa_i)_{i \in I}$, with $\kappa_i = \big((\kappa_i)_\omega\big)_{\omega\colon p \mapsto DtT_{\mathsf{tp}(p)}(\Sigma, DV)}$ and $\kappa_i = (\kappa_i)_\omega \colon (CV_i)_\omega \to \mathcal{C}_{\epsilon(i), \lambda p \in \mathsf{par}(\mathsf{if}(i))\colon {}_{A}[\![\omega(p)]\!]}$ (for a given interface interpretation ϵ and port interpretation δ). Again, we denote with κ' a corresponding rigid component variable assignment for rigid component variables. The set of all component variable assignments is denoted with $\mathcal{K}^{CV}_{\mathcal{C}}$ and the set of all rigid component variable assignments with $\mathcal{K'}^{CV'}_{\mathcal{C}}$.

C.4.1 Architecture Terms

C.4.1.1 Syntax

Definition 23 (Architecture terms: syntax)**.** *The set of all* architecture terms *of sort $s \in S$ over a signature $\Sigma = (S, F, B)$, interface specification $is = (I, \mathsf{if})$ over port specification (PID, tp), datatype variables DV, and component variables CV, is the smallest set ${}^{s}_{\Sigma}CT^{DV}_{CV}(is)$, satisfying the equations of Fig. C.7. The set of all architecture terms of all sorts is denoted by ${}_{\Sigma}CT^{DV}_{CV}(is)$.*

Architecture terms: semantics

$$
\begin{aligned}
{}^{\iota}_{A}[\![\text{“}v\text{”}]\!]^{\kappa}_{\delta}(as) &= \iota_s(v) \text{ [for } v \in DV_s] \ , \\
{}^{\iota}_{A}[\![\text{“}v.p\text{”}]\!]^{\kappa}_{\delta}(as) &= \begin{cases} val_{as}\Big(((\kappa_i)_{\omega}(v), (\delta(p)))\Big) \\ \text{[for } i \in I \text{ and } v \in (CV_i)_{\omega}] \ , \end{cases} \\
{}^{\iota}_{A}[\![\text{“}f\text{”}]\!]^{\kappa}_{\delta}(as) &= A_f \text{ [for function symbol } f \in F^0] \ , \\
{}^{\iota}_{A}[\![\text{“}f(t_1, \cdots, t_n)\text{”}]\!]^{\kappa}_{\delta}(as) &= \begin{cases} A_f(\,{}^{\iota}_{A}[\![\text{“}t_1\text{”}]\!]^{\kappa}_{\delta}(as), \cdots, {}^{\iota}_{A}[\![\text{“}t_n\text{”}]\!]^{\kappa}_{\delta}(as)) \\ \text{[for function symbol } f \in F^{n+1}] \ . \end{cases}
\end{aligned}
$$

Figure C.8: Recursive definition of semantic function for architecture terms.

C.4.1.2 Semantics

Definition 24 (Architecture terms: semantics)**.** *The* semantics *of architecture terms* ${}_{\Sigma}CT^{DV}_{CV}(is)$*, formulated over interface specification* $is = (I, \mathsf{if})$ *and port specification* (PID, tp)*, is defined over an algebra* $A \in \mathcal{A}(\Sigma)$ *with corresponding data type variable assignments* $\iota \in \mathcal{I}^{DV}_{A}$*, an architecture snapshot* $as \in AS^{\mathcal{C}}_{\mathcal{T}}$ *with corresponding port interpretation* $\delta\colon PID \to \mathcal{P}$ *for the port identifiers of ps, and component interpretation* $\kappa \in \mathcal{K}^{CV}_{\mathcal{C}}$*. It is given by a semantic function* ${}^{\iota}_{A}[\![\_]\!]^{\kappa}_{\delta}(as)\colon \overrightarrow{\bigcup}_{s \in S}({}^{s}_{\Sigma}CT^{DV}_{CV}(is) \to A_s)$*, defined recursively by the equations provided in Fig. C.8.*

C.4.2 Architecture Assertions

C.4.2.1 Syntax

Definition 25 (Architecture assertions: syntax)**.** *The set of all* architecture assertions *over a signature* Σ*, interface specification* $is = (I, \mathsf{if})$ *over port specification* (PID, tp)*, data type variables* DV*, and component variables* CV *is the smallest set* ${}_{\Sigma}CA^{DV}_{CV}(is)$ *satisfying the equations in Fig. C.9.*

C.4.2.2 Semantics

Definition 26 (Architecture assertion: semantics)**.** *The* semantics *of* architecture assertions ${}_{\Sigma}CA^{DV}_{CV}(is)$*, formulated over interface specification* $is = (I, \mathsf{if})$ *and port specification* (PID, tp)*, is defined over an algebra* $A \in \mathcal{A}(\Sigma)$ *with corresponding data type variable assignments* $\iota \in \mathcal{I}^{DV}_{A}$*, interface interpretation* $\epsilon\colon I \to CT_{\mathcal{I}}$*, an interpretation* $\delta\colon PID \to \mathcal{P}$ *for the port identifiers of ps, and component interpretation* $\kappa \in \mathcal{K}^{CV}_{\mathcal{C}}$*. It is given by a relation* $\models^{\epsilon,\delta,\kappa}_{A,\iota} \subseteq AS^{\mathcal{C}}_{\mathcal{T}} \times {}_{\Sigma}CA^{DV}_{CV}(is)$ *defined recursively by the equations provided in Fig. C.10*

Architecture assertions: syntax

$$
\begin{aligned}
& && \text{“}\mathit{true}\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is)\ , \\
& && \text{“}\mathit{false}\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is)\ , \\
b \in B^{0} & \implies && \text{“}b\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is)\ , \\
\left.\begin{array}{l} b \in B^{n+1} \quad \wedge \\ \text{“}t_1\text{”} \in {}^{s_1}_{\Sigma}CT^{DV}_{CV}(is), \cdots , \\ \text{“}t_{n+1}\text{”} \in {}^{s_{n+1}}_{\Sigma}CT^{DV}_{CV}(is) \end{array}\right\} & \implies && \begin{cases} \text{“}b(t_1, \cdots , t_{n+1})\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \\ [\text{for } n \in \mathbb{N} \text{ and} \\ \mathsf{sp}(b)_{(1)} = s_1, \cdots , \mathsf{sp}(b)_{(n+1)} = s_{n+1}]\ , \end{cases} \\
\text{“}t\text{”}, \text{“}t'\text{”} \in {}^{s}_{\Sigma}CT^{DV}_{CV}(is) & \implies && \text{“}t = t'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is)\ [\text{for some } s \in S]\ , \\
\text{“}\phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) & \implies && \text{“}\neg\phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is)\ , \\
\text{“}\phi\text{”}, \text{“}\phi'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) & \implies && \begin{cases} \text{“}\phi \wedge \phi'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ \text{“}\phi \vee \phi'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ \text{“}\phi \longrightarrow \phi'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ \text{“}\phi \longleftrightarrow \phi'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is).\ , \end{cases} \\
\text{“}\phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \wedge x \in DV_s & \implies && \begin{cases} \text{“}\forall x \colon \phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ \text{“}\exists x \colon \phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is)\ [\text{for } s \in S].\ , \end{cases} \\
\text{“}\phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \wedge x \in (CV_i)_{\omega} & \implies && \begin{cases} \text{“}\forall x \colon \phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ \text{“}\exists x \colon \phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is)\ [\text{for } i \in I].\ , \end{cases} \\
v \in (CV_i)_{\omega} \wedge p \in \mathsf{port}(\mathsf{if}(i)) & \implies && \text{“}\widehat{v.p}\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is)\ [\text{for } i \in I]\ , \\
v \in (CV_i)_{\omega} & \implies && \text{“}\wr v \wr\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is)\ [\text{for } i \in I]\ , \\
\left.\begin{array}{r} v \in (CV_i)_{\omega} \wedge v' \in (CV_j)_{\tau} \wedge \\ p \in \mathsf{in}(\mathsf{if}(i)) \wedge p' \in \mathsf{out}(\mathsf{if}(j)) \end{array}\right\} & \implies && \begin{cases} \text{“}v.p \rightsquigarrow v'.p'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ [\text{for } i, j \in I]\ . \end{cases}
\end{aligned}
$$

Figure C.9: Inductive definition of architecture assertions.

Architecture assertions: semantics

$$
\begin{array}{rcl}
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\mathit{true}\text{”} & & , \\
\neg(as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\mathit{false}\text{”}) & & , \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}b\text{”} & \Longleftrightarrow & A_b \ [\text{for } b \in B^0] \ , \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}b(t_1, \cdots, t_n)\text{”} & \Longleftrightarrow & A_b\big({}^{\iota}_{A}[\![\text{“}t_1\text{”}]\!]^{\kappa}_{J}(as), \cdots, {}^{\iota}_{A}[\![\text{“}t_n\text{”}]\!]^{\kappa}_{J}(as)\big) \ [\text{for } b \in B^{n+1}] \ , \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}t = t'\text{”} & \Longleftrightarrow & {}^{\iota}_{A}[\![\text{“}t\text{”}]\!]^{\kappa}_{J}(as) = {}^{\iota}_{A}[\![\text{“}t'\text{”}]\!]^{\kappa}_{J}(as) \ , \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi \wedge \phi'\text{”} & \Longleftrightarrow & as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi\text{”} \wedge as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi'\text{”} \ , \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi \vee \phi'\text{”} & \Longleftrightarrow & as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi\text{”} \vee as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi'\text{”} \ , \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi \longrightarrow \phi'\text{”} & \Longleftrightarrow & as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi\text{”} \Longrightarrow as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi'\text{”} \ , \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi \longleftrightarrow \phi'\text{”} & \Longleftrightarrow & as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi\text{”} \Longleftrightarrow as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\phi'\text{”} \ , \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\exists x\colon \phi\text{”} & \Longleftrightarrow & \begin{cases} \exists x' \in A_s\colon as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota[s\colon x \mapsto x']}{\models}} \text{“}\phi\text{”} \\ [\text{for } s \in S \text{ and } x \in DV_s] \ , \end{cases} \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\forall x\colon \phi\text{”} & \Longleftrightarrow & \begin{cases} \forall x' \in A_s\colon as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota[s\colon x \mapsto x']}{\models}} \text{“}\phi\text{”} \\ [\text{for } s \in S \text{ and } x \in DV_s] \ , \end{cases} \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\exists x\colon \phi\text{”} & \Longleftrightarrow & \begin{cases} \exists x' \in \mathcal{C}_{(\epsilon(i), \lambda p\colon \ {}_{A}[\![\omega(p)]\!])}\colon as \overset{\epsilon,\delta,\kappa[i\colon \omega \mapsto \kappa_i[\omega\colon x \mapsto x']]}{\underset{A,\iota}{\models}} \text{“}\phi\text{”} \\ [\text{for } i \in I \text{ and } x \in (CV_i)_\omega] \ , \end{cases} \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\forall x\colon \phi\text{”} & \Longleftrightarrow & \begin{cases} \forall x' \in \mathcal{C}_{(\epsilon(i), \lambda p\colon \ {}_{A}[\![\omega(p)]\!])}\colon as \overset{\epsilon,\delta,\kappa[i\colon \omega \mapsto \kappa_i[\omega\colon x \mapsto x']]}{\underset{A,\iota}{\models}} \text{“}\phi\text{”} \\ [\text{for } i \in I \text{ and } x \in (CV_i)_\omega] \ , \end{cases} \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\widehat{v.p}\text{”} & \Longleftrightarrow & \begin{cases} val_{as}\Big(\big((\kappa_i)_\omega(v), (\delta(p))\big)\Big) \neq \emptyset \\ [\text{for } i \in I,\ v \in (CV_i)_\omega, \text{ and } p \in \mathsf{port}(\mathsf{if}(i))] \ , \end{cases} \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}\wr v \wr\text{”} & \Longleftrightarrow & \begin{cases} \wr(\kappa_i)_\omega(v)\wr_{as} \\ [\text{for } i \in I, \text{ and } v \in (CV_i)_\omega] \ , \end{cases} \\
as \overset{\epsilon,\delta,\kappa}{\underset{A,\iota}{\models}} \text{“}v.p \rightsquigarrow v'.p'\text{”} & \Longleftrightarrow & \begin{cases} \Big(\big((\kappa_i)_\omega(v'), \delta(p')\big)\Big) \in CN_{as}\Big(\big((\kappa_j)_\tau(v), \delta(p)\big)\Big) \\ [\text{for } i \in I,\ v \in (CV_i)_\omega,\ p \in \mathsf{in}(\mathsf{if}(i)), \\ \ j \in I,\ v' \in (CV_j)_\omega,\ p' \in \mathsf{out}(\mathsf{if}(j))] \ . \end{cases}
\end{array}
$$

Figure C.10: Recursive definition of satisfaction relation for architecture assertions.

Architecture trace assertions: syntax

$$\begin{array}{rcl}
 & & \text{“}\mathit{true}\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)\ , \\
 & & \text{“}\mathit{false}\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)\ , \\
\phi \in {}_{\Sigma}CA^{DV\cup DV'}_{CV\cup CV'}(is) & \Longrightarrow & \phi \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)\ , \\
\text{“}\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) & \Longrightarrow & \text{“}\neg\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)\ , \\
\text{“}\gamma\text{”}, \text{“}\gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) & \Longrightarrow & \begin{cases} \text{“}\gamma \wedge \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\gamma \vee \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\gamma \longrightarrow \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\gamma \longleftrightarrow \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is). \end{cases} , \\
\text{“}\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) & \Longrightarrow & \text{“}\bigcirc\gamma\text{”},\ \text{“}\Diamond\gamma\text{”},\ \text{“}\Box\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)\ , \\
\text{“}\gamma\text{”}, \text{“}\gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) & \Longrightarrow & \begin{cases} \text{“}\gamma\ \mathcal{U}\ \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\gamma\ \mathcal{W}\ \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is). \end{cases} , \\
\left.\begin{array}{l} x \in ({DV'}_s)_\omega \quad \wedge \\ \text{“}\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \end{array}\right\} & \Longrightarrow & \begin{cases} \text{“}\forall x\colon \gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\exists x\colon \gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)\ \text{[for } s \in S\text{].} \end{cases} , \\
\left.\begin{array}{l} x \in ({CV'}_i)_\omega \quad \wedge \\ \text{“}\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \end{array}\right\} & \Longrightarrow & \begin{cases} \text{“}\forall x\colon \gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\exists x\colon \gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)\ \text{[for } i \in I\text{].} \end{cases} .
\end{array}$$

Figure C.11: Inductive definition of architecture trace assertions.

C.4.3 Architecture Trace Assertions

C.4.3.1 Syntax

Definition 27 (Architecture trace assertion: syntax)**.** *The set of all* architecture trace assertions *over signature Σ, interface specification $is = (I, \mathsf{if})$ over (PID, tp), data type variables DV, rigid data type variables DV', component variables CV, and rigid component variables CV' is the smallest set ${}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)$ satisfying the equations in Fig. C.11.*

C.4.3.2 Semantics

Definition 28 (Architecture trace assertion: semantics)**.** *The* semantics *of* architecture trace assertions ${}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)$*, formulated over interface specification $is = (I, \mathsf{if})$ and port specification (PID, tp), is defined over an algebra $A \in \mathcal{A}(\Sigma)$ with corresponding rigid data type variable assignments $\iota' \in {\mathcal{I}'}^{DV'}_{A}$, interface interpretation $\epsilon\colon I \to CT_{\mathcal{I}}$, an interpretation $\delta\colon PID \to \mathcal{P}$ for the port identifiers of ps, and rigid component interpre-*

tation $\kappa' \in \mathcal{K}'^{CV}_{\mathcal{C}}$. *It is given by a relation* $\overset{\epsilon,\delta,\kappa'}{\underset{A,\iota'}{\models}} \subseteq ((AS^{\mathcal{C}}_{\mathcal{T}})^{\infty} \times \mathbb{N}) \times {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)$ *defined recursively by the equations provided in Fig. C.12*

Architecture trace assertions: semantics

$$\begin{aligned}
&(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\mathit{true}\text{”} \quad ,\\
&\neg\big((t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\mathit{false}\text{”}\big),\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \phi \quad &\iff \quad \forall \iota \in \mathcal{I}^{DV}_{A}, \kappa \in \mathcal{K}^{CV}_{\mathcal{C}} : t(n) \models^{\epsilon,\delta,\kappa\cup\kappa'}_{A,\iota\cup\iota'} \phi \ ,\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \wedge \gamma'\text{”} \quad &\iff \quad (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \wedge (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma'\text{”} \ ,\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \vee \gamma'\text{”} \quad &\iff \quad (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \vee (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma'\text{”} \ ,\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \longrightarrow \gamma'\text{”} \quad &\iff \quad (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \implies (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma'\text{”} \ ,\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \longleftrightarrow \gamma'\text{”} \quad &\iff \quad (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \iff (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma'\text{”} \ ,\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\bigcirc\gamma\text{”} \quad &\iff \quad (t,n+1) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \ ,\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\Diamond\gamma\text{”} \quad &\iff \quad \exists n' \geq n \colon (t,n') \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \ ,\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\Box\gamma\text{”} \quad &\iff \quad \forall n' \geq n \colon (t,n') \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \ ,\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \,\mathcal{U}\, \gamma'\text{”} \quad &\iff \quad \begin{cases} \exists n' \geq n \colon (t,n') \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma'\text{”} \ \wedge \\ \forall n \leq m < n' \colon (t,m) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \ , \end{cases}\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \,\mathcal{W}\, \gamma'\text{”} \quad &\iff \quad \begin{cases} (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \,\mathcal{U}\, \gamma'\text{”} \ \vee \\ (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\Box\gamma\text{”} \ , \end{cases}\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\exists x \colon \gamma\text{”} \quad &\iff \quad \begin{cases} \exists x' \in A_s \colon (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'[s\colon x \mapsto x']} \text{“}\gamma\text{”} \\ [\text{for } s \in S \text{ and } x \in DV'_s] \ , \end{cases}\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\forall x \colon \gamma\text{”} \quad &\iff \quad \begin{cases} \forall x' \in A_s \colon (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'[s\colon x \mapsto x']} \text{“}\gamma\text{”} \\ [\text{for } s \in S \text{ and } x \in DV'_s] \ , \end{cases}\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\exists x \colon \gamma\text{”} \quad &\iff \quad \begin{cases} \exists x' \in \mathcal{C}_{(\epsilon(i),\lambda p\colon\ {}_{A}[\![\omega(p)]\!])} \colon (t,n) \models^{\epsilon,\delta,\kappa'[i\colon \omega \mapsto \kappa'_i[\omega\colon x \mapsto x']]}_{A,\iota'} \text{“}\gamma\text{”} \\ [\text{for } i \in I \text{ and } x \in (CV'_i)_\omega] \ , \end{cases}\\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\forall x \colon \gamma\text{”} \quad &\iff \quad \begin{cases} \forall x' \in \mathcal{C}_{(\epsilon(i),\lambda p\colon\ {}_{A}[\![\omega(p)]\!])} \colon (t,n) \models^{\epsilon,\delta,\kappa'[i\colon \omega \mapsto \kappa'_i[\omega\colon x \mapsto x']]}_{A,\iota'} \text{“}\gamma\text{”} \\ [\text{for } i \in I \text{ and } x \in (CV'_i)_\omega] \ . \end{cases}
\end{aligned}$$

Figure C.12: Recursive definition of satisfaction relation for architecture trace assertions.

D Remaining Rules of the Calculus

D.1 Elimination Rules for Basic Logical Operators

In the following we list elimination rules for the basic logical operators:

ImpE

$$\frac{(t,t',n) \underset{c}{\models} \text{“}\gamma \longrightarrow \gamma'\text{”}}{(t,t',n) \underset{c}{\models} \text{“}\gamma\text{”} \longrightarrow (t,t',n) \underset{c}{\models} \text{“}\gamma'\text{”}}$$

AndE

$$\frac{(t,t',n) \underset{c}{\models} \text{“}\gamma \wedge \gamma'\text{”}}{(t,t',n) \underset{c}{\models} \text{“}\gamma\text{”} \wedge (t,t',n) \underset{c}{\models} \text{“}\gamma'\text{”}}$$

OrE

$$\frac{(t,t',n) \underset{c}{\models} \text{“}\gamma \vee \gamma'\text{”}}{(t,t',n) \underset{c}{\models} \text{“}\gamma\text{”} \vee (t,t',n) \underset{c}{\models} \text{“}\gamma'\text{”}}$$

NotE

$$\frac{(t,t',n) \underset{c}{\models} \text{“}\neg\gamma\text{”}}{\neg(t,t',n) \underset{c}{\models} \text{“}\gamma\text{”}}$$

AllE

$$\frac{(t,t',n) \underset{c}{\models} \text{“}\forall x\colon\ \gamma\text{”}}{\forall x\colon (t,t',n) \underset{c}{\models} \text{“}\gamma\text{”}}$$

ExE

$$\frac{(t,t',n) \underset{c}{\models} \text{“}\exists x\colon\ \gamma\text{”}}{\exists x\colon (t,t',n) \underset{c}{\models} \text{“}\gamma\text{”}}$$

D.2 Elimination of Behavior Assertions

The first case describes elimination for situations in which a component is guaranteed to be activated sometimes in the future:

AssE$_{\mathrm{a}}$

$$\frac{(t,t',n) \underset{c}{\models} \phi}{\mathsf{val}(c) \cup \left(\lambda p \in \mathsf{port}(c)\colon\ \mathit{val}_{t(c \xrightarrow{n} t)}(c,p)\right) \ \models\ \phi}\ \exists \underset{\cdot}{i} \geq n\colon \wr c \wr_{t(i)}$$

The rule for such cases allows us to eliminate a basic BA ϕ and conclude that ϕ holds at the very *next* point in time where component c is active.

The next rule deals with the case in which a component was sometimes active, but is not activated again in the future:

AssE$_{\mathrm{n1}}$

$$\frac{(t,t',n) \underset{c}{\models} \phi}{t'(n - \mathit{last}(c,t) - 1)\ \models\ \phi}\ \exists i\colon \wr c \wr_{t(i)} \wedge \nexists \underset{\cdot}{i} \geq n\colon \wr c \wr_{t(i)}$$

The rule for this case allows us to conclude that a BA ϕ holds at a certain point in time for continuation t'. Again, the corresponding time point is calculated as the difference of n and the last time component c was activated.

Finally, we provide a rule for the case in which a component is never activated:

AssE$_{\mathrm{n2}}$

$$\frac{(t, t', n) \models_{c} \phi}{t'(n) \ \models \ \phi} \ \nexists i\colon \wr c \wr_{t(i)}$$

For such cases, we may eliminate ϕ and conclude that ϕ holds at n for continuation t'.

The following theory formalizes configuration traces [MG16a, MG16b] as a model for dynamic architectures. Since configuration traces may be finite as well as infinite, the theory depends on Lochbihler's theory of co-inductive lists [Loc10].

```
theory Configuration-Traces
  imports Coinductive.Coinductive-List
begin
```

In the following we first provide some preliminary results for natural numbers, extended natural numbers, and lazy lists. Then, we introduce a locale @textdynamic_architectures which introduces basic definitions and corresponding properties for dynamic architectures.

D.3 Natural Numbers

We provide one additional property for natural numbers.

```
lemma boundedGreatest:
  assumes P (i::nat)
    and ∀ n' > n. ¬ P n'
  shows ∃ i'≤n. P i' ∧ (∀ n'. P n' ⟶ n'≤i')
proof –
  have P (i::nat) ⟹ n≥i ⟹ ∀ n' > n. ¬ P n' ⟹ (∃ i'≤n. P i' ∧ (∀ n'≤n. P n' ⟶ n'≤i'))
  proof (induction n)
    case 0
    then show ?case by auto
  next
    case (Suc n)
    then show ?case
    proof cases
      assume i = Suc n
      then show ?thesis using Suc.prems by auto
    next
      assume ¬(i = Suc n)
      thus ?thesis
      proof cases
        assume P (Suc n)
        thus ?thesis by auto
      next
        assume ¬ P (Suc n)
        with Suc.prems have ∀ n' > n. ¬ P n' using Suc-lessI by blast
        moreover from ‹¬(i = Suc n)› have i ≤ n and P i using Suc.prems by auto
        ultimately obtain i' where i'≤n ∧ P i' ∧ (∀ n'≤n. P n' ⟶ n' ≤ i')
          using Suc.IH by blast
        hence i' ≤ n and P i' and (∀ n'≤n. P n' ⟶ n' ≤ i') by auto
        thus ?thesis by (metis le-SucI le-Suc-eq)
      qed
    qed
  qed
```

```
  moreover have n≥i
  proof (rule ccontr)
    assume ¬ (n ≥ i)
    hence n < i by arith
    thus False using assms by blast
  qed
  ultimately obtain i' where i'≤n and P i' and ∀n'≤n. P n' ⟶ n' ≤ i' using assms by
blast
  with assms have ∀n'. P n' ⟶ n' ≤ i' using not-le-imp-less by blast
  with ‹i' ≤ n› and ‹P i'› show ?thesis by auto
qed
```

D.4 Extended Natural Numbers

We provide one simple property for the *strict* order over extended natural numbers.

```
lemma enat-min:
  assumes m ≥ enat n'
    and enat n < m − enat n'
  shows enat n + enat n' < m
 using assms by (metis add.commute enat.simps(3) enat-add-mono enat-add-sub-same le-iff-add)
```

D.5 Lazy Lists

In the following we provide some additional notation and properties for lazy lists.

```
notation LNil ([]_l)
notation LCons (infixl #_l 60)
notation lappend (infixl @_l 60)

lemma lnth-lappend[simp]:
  assumes lfinite xs
    and ¬ lnull ys
  shows lnth (xs @_l ys) (the-enat (llength xs)) = lhd ys
proof −
  from assms have ∃k. llength xs = enat k using lfinite-conv-llength-enat by auto
  then obtain k where llength xs = enat k by blast
  hence lnth (xs @_l ys) (the-enat (llength xs)) = lnth ys 0
    using lnth-lappend2[of xs k k ys] by simp
  with assms show ?thesis using lnth-0-conv-lhd by simp
qed

lemma lfilter-ltake:
  assumes ∀(n::nat)≤llength xs. n≥i ⟶ (¬ P (lnth xs n))
  shows lfilter P xs = lfilter P (ltake i xs)
proof −
  have lfilter P xs = lfilter P ((ltake i xs) @_l (ldrop i xs))
    using lappend-ltake-ldrop[of (enat i) xs] by simp
  hence lfilter P xs = (lfilter P ((ltake i) xs)) @_l (lfilter P (ldrop i xs)) by simp
```

```
  show ?thesis
  proof cases
    assume enat i ≤ llength xs

    have ∀x<llength (ldrop i xs). ¬ P (lnth (ldrop i xs) x)
    proof (rule allI)
      fix x show enat x < llength (ldrop (enat i) xs) ⟶ ¬ P (lnth (ldrop (enat i) xs) x)
      proof
        assume enat x < llength (ldrop (enat i) xs)
        moreover have llength (ldrop (enat i) xs) = llength xs − enat i
          using llength-ldrop[of enat i] by simp
        ultimately have enat x < llength xs − enat i by simp
        with ‹enat i ≤ llength xs› have enat x + enat i < llength xs
          using enat-min[of i llength xs x] by simp
        moreover have enat i + enat x = enat x + enat i by simp
        ultimately have enat i + enat x < llength xs by arith
        hence i + x < llength xs by simp
        hence lnth (ldrop i xs) x = lnth xs (x + the-enat i) using lnth-ldrop by simp
        moreover have x + i ≥ i by simp
        with assms ‹i + x < llength xs› have ¬ P (lnth xs (x + the-enat i))
          by (simp add: assms(1) add.commute)
        ultimately show ¬ P (lnth (ldrop i xs) x) using assms by simp
      qed
    qed
    hence lfilter P (ldrop i xs) = []_l by (metis diverge-lfilter-LNil in-lset-conv-lnth)
    with ‹lfilter P xs = (lfilter P ((ltake i) xs)) @_l (lfilter P (ldrop i xs))›
      show lfilter P xs = lfilter P (ltake i xs) by simp
  next
    assume ¬ enat i ≤ llength xs
    hence enat i > llength xs by simp
    hence ldrop i xs = []_l by simp
    hence lfilter P (ldrop i xs) = []_l using lfilter-LNil[of P] by arith
    with ‹lfilter P xs = (lfilter P ((ltake i) xs)) @_l (lfilter P (ldrop i xs))›
      show lfilter P xs = lfilter P (ltake i xs) by simp
  qed
qed

lemma lfilter-lfinite[simp]:
  assumes lfinite (lfilter P t)
    and ¬ lfinite t
  shows ∃n. ∀n'>n. ¬ P (lnth t n')
proof −
  from assms have finite {n. enat n < llength t ∧ P (lnth t n)} using lfinite-lfilter by auto
  then obtain k where sset:
    {n. enat n < llength t ∧ P (lnth t n)} ⊆ {n. n<k ∧ enat n < llength t ∧ P (lnth t n)}
    using finite-nat-bounded[of {n. enat n < llength t ∧ P (lnth t n)}] by auto
  show ?thesis
  proof (rule ccontr)
    assume ¬(∃n. ∀n'>n. ¬ P (lnth t n'))
```

hence $\forall n.\ \exists n'{>}n.\ P\ (\mathit{lnth}\ t\ n')$ **by** *simp*
then obtain n' **where** $n'{>}k$ **and** $P\ (\mathit{lnth}\ t\ n')$ **by** *auto*
moreover from $\langle\neg\ \mathit{lfinite}\ t\rangle$ **have** $n' < \mathit{llength}\ t$ **by** (*simp add*: *not-lfinite-llength*)
ultimately have $n' \notin \{n.\ n{<}k \wedge \mathit{enat}\ n < \mathit{llength}\ t \wedge P\ (\mathit{lnth}\ t\ n)\}$ **and**
$n'{\in}\{n.\ \mathit{enat}\ n < \mathit{llength}\ t \wedge P\ (\mathit{lnth}\ t\ n)\}$ **by** *auto*
with *sset* **show** *False* **by** *auto*
qed
qed

D.6 A Model of Dynamic Architectures

In the following we formalize dynamic architectures in terms of configuration traces, i.e., sequences of architecture configurations. Moreover, we introduce definitions for operations to support the specification of configuration traces.

typedecl *cnf*
type-synonym $\mathit{trace} = \mathit{nat} \Rightarrow \mathit{cnf}$
consts *arch*:: *trace set*

D.6.1 Implication

definition *imp* :: $((\mathit{nat} \Rightarrow \mathit{cnf}) \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}) \Rightarrow ((\mathit{nat} \Rightarrow \mathit{cnf}) \Rightarrow \mathit{nat} \Rightarrow \mathit{bool})$
$\Rightarrow ((\mathit{nat} \Rightarrow \mathit{cnf}) \Rightarrow \mathit{nat} \Rightarrow \mathit{bool})$ (**infixl** $\longrightarrow^c$ *10*)
where $\gamma \longrightarrow^c \gamma' \equiv \lambda\ t\ n.\ \gamma\ t\ n \longrightarrow \gamma'\ t\ n$

declare *imp-def*[*simp*]

lemma *impI*[*intro*!]:
fixes $t\ n$
assumes $\gamma\ t\ n \Longrightarrow \gamma'\ t\ n$
shows $(\gamma \longrightarrow^c \gamma')\ t\ n$ **using** *assms* **by** *simp*

lemma *impE*[*elim*!]:
fixes $t\ n$
assumes $(\gamma \longrightarrow^c \gamma')\ t\ n$ **and** $\gamma\ t\ n$ **and** $\gamma'\ t\ n \Longrightarrow \gamma''\ t\ n$
shows $\gamma''\ t\ n$ **using** *assms* **by** *simp*

D.6.2 Disjunction

definition *disj* :: $((\mathit{nat} \Rightarrow \mathit{cnf}) \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}) \Rightarrow ((\mathit{nat} \Rightarrow \mathit{cnf}) \Rightarrow \mathit{nat} \Rightarrow \mathit{bool})$
$\Rightarrow ((\mathit{nat} \Rightarrow \mathit{cnf}) \Rightarrow \mathit{nat} \Rightarrow \mathit{bool})$ (**infixl** $\vee^c$ *15*)
where $\gamma \vee^c \gamma' \equiv \lambda\ t\ n.\ \gamma\ t\ n \vee \gamma'\ t\ n$

declare *disj-def*[*simp*]

lemma *disjI1*[*intro*]:
assumes $\gamma\ t\ n$
shows $(\gamma \vee^c \gamma')\ t\ n$ **using** *assms* **by** *simp*

lemma *disjI2*[*intro*]:
  assumes $\gamma' \ t \ n$
  shows $(\gamma \vee^c \gamma') \ t \ n$ **using** *assms* **by** *simp*

lemma *disjE*[*elim*!]:
  assumes $(\gamma \vee^c \gamma') \ t \ n$
    and $\gamma \ t \ n \Longrightarrow \gamma'' \ t \ n$
    and $\gamma' \ t \ n \Longrightarrow \gamma'' \ t \ n$
  shows $\gamma'' \ t \ n$ **using** *assms* **by** *auto*

D.6.3 Conjunction

definition *conj* :: $((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool) \Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$
  $\Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$ (**infixl** $\wedge^c$ *20*)
  where $\gamma \wedge^c \gamma' \equiv \lambda \ t \ n. \ \gamma \ t \ n \wedge \gamma' \ t \ n$

declare *conj-def*[*simp*]

lemma *conjI*[*intro*!]:
  fixes n
  assumes $\gamma \ t \ n$ **and** $\gamma' \ t \ n$
  shows $(\gamma \wedge^c \gamma') \ t \ n$ **using** *assms* **by** *simp*

lemma *conjE*[*elim*!]:
  fixes n
  assumes $(\gamma \wedge^c \gamma') \ t \ n$ **and** $\gamma \ t \ n \Longrightarrow \gamma' \ t \ n \Longrightarrow \gamma'' \ t \ n$
  shows $\gamma'' \ t \ n$ **using** *assms* **by** *simp*

D.6.4 Negation

definition *not* :: $((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool) \Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$ ($\neg^c$ - [*19*] *19*)
  where $\neg^c \ \gamma \equiv \lambda \ t \ n. \ \neg \ \gamma \ t \ n$

declare *not-def*[*simp*]

lemma *notI*[*intro*!]:
  assumes $\gamma \ t \ n \Longrightarrow False$
  shows $(\neg^c \ \gamma) \ t \ n$ **using** *assms* **by** *auto*

lemma *notE*[*elim*!]:
  assumes $(\neg^c \ \gamma) \ t \ n$
    and $\gamma \ t \ n$
  shows $\gamma' \ t \ n$ **using** *assms* **by** *simp*

D.6.5 Quantifiers

definition *all* :: $('a \Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool))$
  $\Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$ (**binder** $\forall_c$ *10*)
  where $all \ P \equiv \lambda t \ n. \ (\forall \, y. \ (P \ y \ t \ n))$

declare *all-def*[*simp*]

lemma *allI*[*intro*!]:
assumes $\bigwedge x.\ \gamma\ x\ t\ n$
shows $(\forall_c x.\ \gamma\ x)\ t\ n$ **using** *assms* **by** *simp*

lemma *allE*[*elim*!]:
fixes n
assumes $(\forall_c x.\ \gamma\ x)\ t\ n$ **and** $\gamma\ x\ t\ n \Longrightarrow \gamma'\ t\ n$
shows $\gamma'\ t\ n$ **using** *assms* **by** *simp*

definition *ex* :: $('a \Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool))$
$\Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$ (**binder** $\exists_c$ *10*)
where $ex\ P \equiv \lambda t\ n.\ (\exists\, y.\ (P\ y\ t\ n))$

declare *ex-def*[*simp*]

lemma *exI*[*intro*!]:
assumes $\gamma\ x\ t\ n$
shows $(\exists_c x.\ \gamma\ x)\ t\ n$ **using** *assms HOL.exI* **by** *simp*

lemma *exE*[*elim*!]:
assumes $(\exists_c x.\ \gamma\ x)\ t\ n$ **and** $\bigwedge x.\ \gamma\ x\ t\ n \Longrightarrow \gamma'\ t\ n$
shows $\gamma'\ t\ n$ **using** *assms HOL.exE* **by** *auto*

D.6.6 Atomic Assertions

First we provide rules for basic behavior assertions.

definition *ca* :: $(cnf \Rightarrow bool) \Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$
where $ca\ \varphi \equiv \lambda\ t\ n.\ \varphi\ (t\ n)$

lemma *caI*[*intro*]:
fixes n
assumes $\varphi\ (t\ n)$
shows $(ca\ \varphi)\ t\ n$ **using** *assms ca-def* **by** *simp*

lemma *caE*[*elim*]:
fixes n
assumes $(ca\ \varphi)\ t\ n$
shows $\varphi\ (t\ n)$ **using** *assms ca-def* **by** *simp*

D.6.7 Next Operator

definition *nxt* :: $((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool) \Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$ ($\bigcirc_c$(-) *24*)
where $\bigcirc_c(\gamma) \equiv \lambda(t::(nat \Rightarrow cnf))\ n.\ \gamma\ t\ (Suc\ n)$

D.6.8 Eventually Operator

definition *evt* :: $((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool) \Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$ ($\Diamond_c$(-) *23*)

where $\Diamond_c(\gamma) \equiv \lambda(t::(nat \Rightarrow cnf))\ n.\ \exists\, n' \geq n.\ \gamma\ t\ n'$

D.6.9 Globally Operator

definition *glob* :: $((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool) \Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$ $(\Box_c(\text{-})\ 22)$
where $\Box_c(\gamma) \equiv \lambda(t::(nat \Rightarrow cnf))\ n.\ \forall\, n' \geq n.\ \gamma\ t\ n'$

lemma *globI*[*intro*!]:
fixes n'
assumes $\forall\, n \geq n'.\ \gamma\ t\ n$
shows $(\Box_c(\gamma))\ t\ n'$ **using** *assms glob-def* **by** *simp*

lemma *globE*[*elim*!]:
fixes $n\ n'$
assumes $(\Box_c(\gamma))\ t\ n$ **and** $n' \geq n$
shows $\gamma\ t\ n'$ **using** *assms glob-def* **by** *simp*

D.6.10 Until Operator

definition *until* :: $((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool) \Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$
$\Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$ (**infixl** $\mathfrak{U}_c$ *21*)
where $\gamma' \mathfrak{U}_c\ \gamma \equiv \lambda(t::(nat \Rightarrow cnf))\ n.\ \exists\, n'' \geq n.\ \gamma\ t\ n'' \wedge (\forall\, n' \geq n.\ n' < n'' \longrightarrow \gamma'\ t\ n')$

lemma *untilI*[*intro*]:
fixes n
assumes $\exists\, n'' \geq n.\ \gamma\ t\ n'' \wedge (\forall\, n' \geq n.\ n' < n'' \longrightarrow \gamma'\ t\ n')$
shows $(\gamma' \mathfrak{U}_c\ \gamma)\ t\ n$ **using** *assms until-def* **by** *simp*

lemma *untilE*[*elim*]:
fixes n
assumes $(\gamma' \mathfrak{U}_c\ \gamma)\ t\ n$
shows $\exists\, n'' \geq n.\ \gamma\ t\ n'' \wedge (\forall\, n' \geq n.\ n' < n'' \longrightarrow \gamma'\ t\ n')$ **using** *assms until-def* **by** *simp*

D.6.11 Weak Until

definition *wuntil* :: $((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool) \Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$
$\Rightarrow ((nat \Rightarrow cnf) \Rightarrow nat \Rightarrow bool)$ (**infixl** $\mathfrak{W}_c$ *20*)
where $\gamma' \mathfrak{W}_c\ \gamma \equiv \gamma' \mathfrak{U}_c\ \gamma \vee^c \Box_c(\gamma')$

lemma *wUntilI*[*intro*]:
fixes n
assumes $(\exists\, n'' \geq n.\ \gamma\ t\ n'' \wedge (\forall\, n' \geq n.\ n' < n'' \longrightarrow \gamma'\ t\ n')) \vee (\forall\, n' \geq n.\ \gamma'\ t\ n')$
shows $(\gamma' \mathfrak{W}_c\ \gamma)\ t\ n$ **using** *assms wuntil-def* **by** *auto*

lemma *wUntilE*[*elim*]:
fixes $n\ n'$
assumes $(\gamma' \mathfrak{W}_c\ \gamma)\ t\ n$
shows $(\exists\, n'' \geq n.\ \gamma\ t\ n'' \wedge (\forall\, n' \geq n.\ n' < n'' \longrightarrow \gamma'\ t\ n')) \vee (\forall\, n' \geq n.\ \gamma'\ t\ n')$
proof –
from *assms* **have** $(\gamma' \mathfrak{U}_c\ \gamma \vee^c \Box_c(\gamma'))\ t\ n$ **using** *wuntil-def* **by** *simp*

```
  hence (γ′ 𝔘_c γ) t n ∨ (□_c(γ′)) t n by simp
  thus ?thesis
  proof
    assume (γ′ 𝔘_c γ) t n
    hence ∃ n″≥n. γ t n″ ∧ (∀ n′≥n. n′ < n″ ⟶ γ′ t n′) by auto
    thus ?thesis by auto
  next
    assume (□_cγ′) t n
    hence ∀ n′≥n. γ′ t n′ by auto
    thus ?thesis by auto
  qed
qed

lemma wUntil-Glob:
  assumes (γ′ 𝔚_c γ) t n
    and (□_c(γ′ ⟶^c γ″)) t n
  shows (γ″ 𝔚_c γ) t n
proof
  from assms(1) have (∃ n″≥n. γ t n″ ∧ (∀ n′≥n. n′ < n″ ⟶ γ′ t n′)) ∨ (∀ n′≥n. γ′ t n′)
    using wUntilE by simp
  thus (∃ n″≥n. γ t n″ ∧ (∀ n′≥n. n′ < n″ ⟶ γ″ t n′)) ∨ (∀ n′≥n. γ″ t n′)
  proof
    assume ∃ n″≥n. γ t n″ ∧ (∀ n′≥n. n′ < n″ ⟶ γ′ t n′)
    show (∃ n″≥n. γ t n″ ∧ (∀ n′≥n. n′ < n″ ⟶ γ″ t n′)) ∨ (∀ n′≥n. γ″ t n′)
    proof −
      from ‹∃ n″≥n. γ t n″ ∧ (∀ n′≥n. n′ < n″ ⟶ γ′ t n′)› obtain n″
        where n″≥n and γ t n″ and a1: ∀ n′≥n. n′ < n″ ⟶ γ′ t n′ by auto
      moreover have ∀ n′≥n. n′ < n″ ⟶ γ″ t n′
      proof
        fix n′
        show n′≥n ⟶ n′< n″ ⟶ γ″ t n′
        proof (rule HOL.impI[OF HOL.impI])
          assume n′≥n and n′<n″
          with assms(2) have (γ′ ⟶^c γ″) t n′ using globE by simp
          hence γ′ t n′ ⟶ γ″ t n′ using impE by auto
          moreover from a1 ‹n′≥n› ‹n′<n″› have γ′ t n′ by simp
          ultimately show γ″ t n′ by simp
        qed
      qed
      ultimately show ?thesis by auto
    qed
  next
    assume a1: ∀ n′≥n. γ′ t n′
    have ∀ n′≥n. γ″ t n′
    proof
      fix n′
      show n′≥n ⟶ γ″ t n′
      proof
        assume n′≥n
```

```
        with assms(2) have (γ' ⟶ᶜ γ'') t n' using globE by simp
        hence γ' t n' ⟶ γ'' t n' using impE by auto
        moreover from a1 ‹n'≥n› have γ' t n' by simp
        ultimately show γ'' t n' by simp
      qed
    qed
    thus (∃n''≥n. γ t n'' ∧ (∀n'≥n. n' < n'' ⟶ γ'' t n')) ∨ (∀n'≥n. γ'' t n') by simp
  qed
qed
```

D.7 Dynamic Components

To support the specification of patterns over dynamic architectures we provide a locale for dynamic components. It takes the following type parameters:

- id: a type for component identifiers
- cmp: a type for components
- cnf: a type for architecture configurations

```
locale dynamic-component =
  fixes tCMP :: 'id ⇒ cnf ⇒ 'cmp (σ_(-) [0,110]60)
    and active :: 'id ⇒ cnf ⇒ bool (≀-≀_ [0,110]60)
begin
```

The locale requires two parameters:

- *tCMP* is an operator to obtain a component with a certain identifier from an architecture configuration.
- *active* is a predicate to assert whether a certain component is activated within an architecture configuration.

The locale provides some general properties about its parameters and introduces six important operators over configuration traces:

- An operator to extract the behavior of a certain component out of a given configuration trace.
- An operator to obtain the number of activations of a certain component within a given configuration trace.
- An operator to obtain the least point in time (before a certain point in time) from which on a certain component is not activated anymore.
- An operator to obtain the latest point in time where a certain component was activated.

- Two operators to map time-points between configuration traces and behavior traces.

Moreover, the locale provides several properties about the operators and their relationships.

lemma *nact-active*:
  fixes $t::nat \Rightarrow cnf$
    and $n::nat$
    and n''
    and id
  assumes $\wr id \wr_{t\ n}$
    and $n'' \geq n$
    and $\neg\ (\exists\, n' \geq n.\ n' < n'' \wedge \wr id \wr_{t\ n'})$
  shows $n=n''$
  using *assms le-eq-less-or-eq* **by** *auto*

lemma *nact-exists*:
  fixes $t::nat \Rightarrow cnf$
  assumes $\exists\, i \geq n.\ \wr c \wr_{t\ i}$
  shows $\exists\, i \geq n.\ \wr c \wr_{t\ i} \wedge (\nexists\, k.\ n \leq k \wedge k < i \wedge \wr c \wr_{t\ k})$
proof −
  let $?L = LEAST\ i.\ (i \geq n \wedge \wr c \wr_{t\ i})$
  from *assms* **have** $?L \geq n \wedge \wr c \wr_{t\ ?L}$ **using** $LeastI[of\ \lambda x::nat.\ (x \geq n \wedge \wr c \wr_{t\ x})]$ **by** *auto*
  moreover have $\nexists\, k.\ n \leq k \wedge k < ?L \wedge \wr c \wr_{t\ k}$ **using** *not-less-Least* **by** *auto*
  ultimately show *?thesis* **by** *blast*
qed

lemma *lActive-least*:
  assumes $\exists\, i \geq n.\ i < llength\ t \wedge \wr c \wr_{lnth\ t\ i}$
  shows $\exists\, i \geq n.\ (i < llength\ t \wedge \wr c \wr_{lnth\ t\ i} \wedge (\nexists\, k.\ n \leq k \wedge k < i \wedge k < llength\ t \wedge \wr c \wr_{lnth\ t\ k}))$
proof −
  let $?L = LEAST\ i.\ (i \geq n \wedge i < llength\ t \wedge \wr c \wr_{lnth\ t\ i})$
  from *assms* **have** $?L \geq n \wedge\ ?L < llength\ t \wedge \wr c \wr_{lnth\ t\ ?L}$
    using $LeastI[of\ \lambda x::nat.(x \geq n \wedge x < llength\ t \wedge \wr c \wr_{lnth\ t\ x})]$ **by** *auto*
  moreover have $\nexists\, k.\ n \leq k \wedge k < llength\ t \wedge k < ?L \wedge \wr c \wr_{lnth\ t\ k}$ **using** *not-less-Least* **by** *auto*
  ultimately show *?thesis* **by** *blast*
qed

D.8 Projection

In the following we introduce an operator which extracts the behavior of a certain component out of a given configuration trace.

definition $proj::\ 'id \Rightarrow (cnf\ llist) \Rightarrow ('cmp\ llist)\ (\pi_{\_}(\text{-})\ [0{,}110]60)$
  where $proj\ c = lmap\ (\lambda cnf.\ (\sigma_{c}(cnf))) \circ (lfilter\ (active\ c))$

lemma *proj-lnil* [*simp*,*intro*]:
  $\pi_{c}([]_{l}) = []_{l}$ **using** *proj-def* **by** *simp*

lemma *proj-lnull* [*simp*]:
$\pi_c(t) = []_l \longleftrightarrow (\forall k \in \mathit{lset}\ t.\ \neg \wr c \wr_k)$
proof
assume $\pi_c(t) = []_l$
hence *lfilter* (*active c*) $t = []_l$ **using** *proj-def lmap-eq-LNil* **by** *auto*
thus $\forall k \in \mathit{lset}\ t.\ \neg \wr c \wr_k$ **using** *lfilter-eq-LNil*[*of active c*] **by** *simp*
next
assume $\forall k{\in}\mathit{lset}\ t.\ \neg \wr c \wr_k$
hence *lfilter* (*active c*) $t = []_l$ **by** *simp*
thus $\pi_c(t) = []_l$ **using** *proj-def* **by** *simp*
qed

lemma *proj-LCons* [*simp*]:
$\pi_i(x \#_l xs) = (\mathit{if}\ \wr i \wr_x\ \mathit{then}\ (\sigma_i(x)) \#_l (\pi_i(xs))\ \mathit{else}\ \pi_i(xs))$
using *proj-def* **by** *simp*

lemma *proj-llength*[*simp*]:
$\mathit{llength}\ (\pi_c(t)) \leq \mathit{llength}\ t$
using *llength-lfilter-ile proj-def* **by** *simp*

lemma *proj-ltake*:
assumes $\forall (n'::\mathit{nat}){\leq}\mathit{llength}\ t.\ n'{\geq}n \longrightarrow (\neg \wr c \wr_{\mathit{lnth}\ t\ n'})$
shows $\pi_c(t) = \pi_c(\mathit{ltake}\ n\ t)$ **using** *lfilter-ltake proj-def assms* **by** (*metis comp-apply*)

lemma *proj-finite-bound*:
assumes *lfinite* (π_c(*inf-llist t*))
shows $\exists n.\ \forall n'{>}n.\ \neg \wr c \wr_{t\ n'}$
using *assms lfilter-lfinite*[*of active c inf-llist t*] *proj-def* **by** *simp*

D.8.1 Monotonicity and Continuity

lemma *proj-mcont*:
shows *mcont lSup lprefix lSup lprefix* (*proj c*)
proof –
have *mcont lSup lprefix lSup lprefix* ($\lambda x.$ *lmap* ($\lambda cnf.\ \sigma_c(cnf)$) (*lfilter* (*active c*) *x*))
by *simp*
moreover have ($\lambda x.$ *lmap* ($\lambda cnf.\ \sigma_c(cnf)$) (*lfilter* (*active c*) *x*)) =
lmap ($\lambda cnf.\ \sigma_c(cnf)$) $\circ$ *lfilter* (*active c*) **by** *auto*
ultimately show *?thesis* **using** *proj-def* **by** *simp*
qed

lemma *proj-mcont2mcont*:
assumes *mcont lub ord lSup lprefix f*
shows *mcont lub ord lSup lprefix* ($\lambda x.\ \pi_c(f\ x)$)
proof –
have *mcont lSup lprefix lSup lprefix* (*proj c*) **using** *proj-mcont* **by** *simp*
with *assms* **show** *?thesis* **using** *llist.mcont2mcont*[*of lSup lprefix proj c*] **by** *simp*
qed

```
lemma proj-mono-prefix[simp]:
  assumes lprefix t t′
  shows lprefix (π_c(t)) (π_c(t′))
proof −
  from assms have lprefix (lfilter (active c) t) (lfilter (active c) t′) using lprefix-lfilterI by simp
  hence lprefix (lmap (λcnf. σ_c(cnf)) (lfilter (active c) t))
    (lmap (λcnf. σ_c(cnf)) (lfilter (active c) t′)) using lmap-lprefix by simp
  thus ?thesis using proj-def by simp
qed
```

D.8.2 Finiteness

```
lemma proj-finite[simp]:
  assumes lfinite t
  shows lfinite (π_c(t))
  using assms proj-def by simp

lemma proj-finite2:
  assumes ∀(n′::nat)≤llength t. n′≥n ⟶ (¬ ⦃c⦄_lnth t n′)
  shows lfinite (π_c(t)) using assms proj-ltake proj-finite by simp

lemma proj-append-lfinite[simp]:
  fixes t t′
  assumes lfinite t
  shows π_c(t @_l t′) = (π_c(t)) @_l (π_c(t′)) (is ?lhs=?rhs)
proof −
  have ?lhs = (lmap (λcnf. σ_c(cnf)) ∘ (lfilter (active c))) (t @_l t′) using proj-def by simp
  also have ... = lmap (λcnf. σ_c(cnf)) (lfilter (active c) (t @_l t′)) by simp
  also from assms have ... = lmap (λcnf. σ_c(cnf))
    ((lfilter (active c) t) @_l (lfilter (active c) t′)) by simp
  also have ... = (@_l) (lmap (λcnf. σ_c(cnf)) (lfilter (active c) t))
    (lmap (λcnf. σ_c(cnf)) (lfilter (active c) t′)) using lmap-lappend-distrib by simp
  also have ... = ?rhs using proj-def by simp
  finally show ?thesis .
qed

lemma proj-one:
  assumes ∃i. i < llength t ∧ ⦃c⦄_lnth t i
  shows llength (π_c(t)) ≥ 1
proof −
  from assms have ∃x∈lset t. ⦃c⦄_x using lset-conv-lnth by force
  hence ¬ lfilter (λk. ⦃c⦄_k) t = []_l using lfilter-eq-LNil[of (λk. ⦃c⦄_k)] by blast
  hence ¬ π_c(t) = []_l using proj-def by fastforce
  thus ?thesis by (simp add: ileI1 lnull-def one-eSuc)
qed
```

D.8.3 Projection not Active

```
lemma proj-not-active[simp]:
  assumes enat n < llength t
```

```
    and ¬ ⦃c⦄_lnth t n
  shows π_c(ltake (Suc n) t) = π_c(ltake n t) (is ?lhs = ?rhs)
proof −
  from assms have ltake (enat (Suc n)) t = (ltake (enat n) t) @_l ((lnth t n) #_l []_l)
    using ltake-Suc-conv-snoc-lnth by blast
  hence ?lhs = π_c((ltake (enat n) t) @_l ((lnth t n) #_l []_l)) by simp
  moreover have ... = (π_c(ltake (enat n) t)) @_l (π_c((lnth t n) #_l []_l)) by simp
  moreover from assms have π_c((lnth t n) #_l []_l) = []_l by simp
  ultimately show ?thesis by simp
qed

lemma proj-not-active-same:
  assumes enat n ≤ (n'::enat)
      and ¬ lfinite t ∨ n'−1 < llength t
      and ∄ k. k≥n ∧ k<n' ∧ k < llength t ∧ ⦃c⦄_lnth t k
    shows π_c(ltake n' t) = π_c(ltake n t)
proof −
  have π_c(ltake (n + (n' − n)) t) = π_c((ltake n t) @_l (ltake (n'−n) (ldrop n t)))
    by (simp add: ltake-plus-conv-lappend)
  hence π_c(ltake (n + (n' − n)) t) =
    (π_c(ltake n t)) @_l (π_c(ltake (n'−n) (ldrop n t))) by simp
  moreover have π_c(ltake (n'−n) (ldrop n t)) = []_l
  proof −
    have ∀ k∈{lnth (ltake (n' − enat n) (ldrop (enat n) t)) na |
      na. enat na < llength (ltake (n' − enat n) (ldrop (enat n) t))}. ¬ ⦃c⦄_k
    proof
      fix k assume k∈{lnth (ltake (n' − enat n) (ldrop (enat n) t)) na |
        na. enat na < llength (ltake (n' − enat n) (ldrop (enat n) t))}
      then obtain k' where enat k' < llength (ltake (n' − enat n) (ldrop (enat n) t))
        and k=lnth (ltake (n' − enat n) (ldrop (enat n) t)) k' by auto
      have enat (k' + n) < llength t
      proof −
        from ‹enat k' < llength (ltake (n' − enat n) (ldrop (enat n) t))›
          have enat k' < n'−n by simp
        hence enat k' + n < n' using assms(1) enat-min by auto
        show ?thesis
        proof cases
          assume lfinite t
          with ‹¬ lfinite t ∨ n'−1 < llength t› have n'−1<llength t by simp
          hence n'< eSuc (llength t) by (metis eSuc-minus-1 enat-minus-mono1 leD leI)
          hence n'≤ llength t using eSuc-ile-mono ileI1 by blast
          with ‹enat k' + n < n'› show ?thesis by (simp add: add.commute)
        next
          assume ¬ lfinite t
          hence llength t = ∞ using not-lfinite-llength by auto
          thus ?thesis by simp
        qed
      qed
      moreover have k = lnth t (k' + n)
```

```
      proof -
        from ‹enat k' < llength (ltake (n' - enat n) (ldrop (enat n) t))›
          have enat k'<n' - enat n by auto
        hence lnth (ltake (n' - enat n) (ldrop (enat n) t)) k' = lnth (ldrop (enat n) t) k'
          using lnth-ltake[of k' n' - enat n] by simp
        with ‹enat (k' + n) < llength t› show ?thesis using lnth-ldrop[of n k' t ]
         using ‹k = lnth (ltake (n' - enat n) (ldrop (enat n) t)) k'› by (simp add: add.commute)
      qed
      moreover from ‹enat n ≤ (n'::enat)› have k' + the-enat n≥n by auto
      moreover from ‹enat k' < llength (ltake (n' - enat n) (ldrop (enat n) t))›
        have k' + n<n' using assms(1) enat-min by auto
      ultimately show ¬ ⦃c⦄_k using ‹∄k. k≥n ∧ k<n' ∧ k < llength t ∧ ⦃c⦄_(lnth t k)› by simp
    qed
    hence ∀k∈lset (ltake (n'-n) (ldrop n t)). ¬ ⦃c⦄_k
      using lset-conv-lnth[of (ltake (n' - enat n) (ldrop (enat n) t))] by simp
    thus ?thesis using proj-lnull by auto
  qed
  moreover from assms have n + (n' - n) = n'
   by (meson enat.distinct(1) enat-add-sub-same enat-diff-cancel-left enat-le-plus-same(1) less-imp-le)
  ultimately show ?thesis by simp
qed
```

D.8.4 Projection Active

```
lemma proj-active[simp]:
  assumes enat i < llength t ⦃c⦄_(lnth t i)
  shows π_c(ltake (Suc i) t) = (π_c(ltake i t)) @_l ((σ_c(lnth t i)) #_l []_l) (is ?lhs = ?rhs)
proof -
  from assms have ltake (enat (Suc i)) t = (ltake (enat i) t) @_l ((lnth t i) #_l []_l)
    using ltake-Suc-conv-snoc-lnth by blast
  hence ?lhs = π_c((ltake (enat i) t) @_l ((lnth t i) #_l []_l)) by simp
  moreover have ... = (π_c(ltake (enat i) t)) @_l (π_c((lnth t i) #_l []_l)) by simp
  moreover from assms have π_c((lnth t i) #_l []_l) = (σ_c(lnth t i)) #_l []_l by simp
  ultimately show ?thesis by simp
qed
```

```
lemma proj-active-append:
  assumes a1: (n::nat) ≤ i
      and a2: enat i < (n'::enat)
      and a3: ¬ lfinite t ∨ n'-1 < llength t
      and a4: ⦃c⦄_(lnth t i)
      and ∀i'. (n ≤ i' ∧ enat i'<n' ∧ i' < llength t ∧ ⦃c⦄_(lnth t i')) ⟶ (i' = i)
    shows π_c(ltake n' t) = (π_c(ltake n t)) @_l ((σ_c(lnth t i)) #_l []_l) (is ?lhs = ?rhs)
proof -
  have ?lhs = π_c(ltake (Suc i) t)
  proof -
    from a2 have Suc i ≤ n' by (simp add: Suc-ile-eq)
    moreover from a3 have ¬ lfinite t ∨ n'-1 < llength t by simp
    moreover have ∄k. enat k≥enat (Suc i) ∧ k<n' ∧ k < llength t ∧ ⦃c⦄_(lnth t k)
```

```
    proof
      assume ∃k. enat k≥enat (Suc i) ∧ k<n' ∧ k < llength t ∧ ⦃c⦄⇘lnth t k⇙
      then obtain k where enat k≥enat (Suc i) and k<n' and k < llength t and ⦃c⦄⇘lnth t k⇙
        by blast
      moreover from ‹enat k≥enat (Suc i)› have enat k≥n
        using assms by (meson dual-order.trans enat-ord-simps(1) le-SucI)
      ultimately have enat k=enat i using assms using enat-ord-simps(1) by blast
      with ‹enat k≥enat (Suc i)› show False by simp
    qed
    ultimately show ?thesis using proj-not-active-same[of Suc i n' t c] by simp
  qed
  also have … = (π⇘c⇙(ltake i t)) @⇩l ((σ⇘c⇙(lnth t i)) #⇩l []⇩l)
  proof –
    have i < llength t
    proof cases
      assume lfinite t
      with a3 have n'−1 < llength t by simp
      hence n' ≤ llength t by (metis eSuc-minus-1 enat-minus-mono1 ileI1 not-le)
      with a2 show enat i < llength t by simp
    next
      assume ¬ lfinite t
      thus ?thesis by (metis enat-ord-code(4) llength-eq-infty-conv-lfinite)
    qed
    with a4 show ?thesis by simp
  qed
  also have … = ?rhs
  proof –
    from a1 have enat n ≤ enat i by simp
    moreover from a2 a3 have ¬ lfinite t ∨ enat i−1 < llength t
      using enat-minus-mono1 less-imp-le order.strict-trans1 by blast
    moreover have ∄k. k≥n ∧ enat k<enat i ∧ enat k < llength t ∧ ⦃c⦄⇘lnth t k⇙
    proof
      assume ∃k. k≥n ∧ enat k<enat i ∧ enat k < llength t ∧ ⦃c⦄⇘lnth t k⇙
      then obtain k where k≥n and enat k<enat i and enat k < llength t and ⦃c⦄⇘lnth t k⇙
        by blast
      moreover from ‹enat k<enat i› have enat k<n' using assms dual-order.strict-trans
        by blast
      ultimately have enat k=enat i using assms by simp
      with ‹enat k<enat i› show False by simp
    qed
    ultimately show ?thesis using proj-not-active-same[of n i t c] by simp
  qed
  finally show ?thesis by simp
qed
```

D.8.5 Same and not Same

```
lemma proj-same-not-active:
  assumes n ≤ n'
```

```
    and enat (n'−1) < llength t
    and π_c(ltake n' t) = π_c(ltake n t)
  shows ∄k. k≥n ∧ k<n' ∧ ⌇c⌇_lnth t k
proof
  assume ∃k. k≥n ∧ k<n' ∧ ⌇c⌇_lnth t k
  then obtain i where i≥n and i<n' and ⌇c⌇_lnth t i by blast
  moreover from ‹enat (n'−1)<llength t› and ‹i<n'› have i<llength t
    by (metis diff-Suc-1 dual-order.strict-trans enat-ord-simps(2) lessE)
  ultimately have π_c(ltake (Suc i) t) =
    (π_c(ltake i t)) @_l ((σ_c(lnth t i)) #_l []_l) by simp
  moreover from ‹i<n'› have Suc i ≤ n' by simp
    hence lprefix(π_c(ltake (Suc i) t)) (π_c(ltake n' t)) by simp
    then obtain tl where π_c(ltake n' t) = (π_c(ltake (Suc i) t)) @_l tl
      using lprefix-conv-lappend by auto
  moreover from ‹n≤i› have lprefix(π_c(ltake n t)) (π_c(ltake i t)) by simp
    hence lprefix(π_c(ltake n t)) (π_c(ltake i t)) by simp
    then obtain hd where π_c(ltake i t) = (π_c(ltake n t)) @_l hd
      using lprefix-conv-lappend by auto
  ultimately have π_c(ltake n' t) =
    (((π_c(ltake n t)) @_l hd) @_l ((σ_c(lnth t i)) #_l []_l)) @_l tl by simp
  also have ... = ((π_c(ltake n t)) @_l hd) @_l ((σ_c(lnth t i)) #_l tl)
    using lappend-snocL1-conv-LCons2[of (π_c(ltake n t)) @_l hd σ_c(lnth t i)] by simp
  also have ... = (π_c(ltake n t)) @_l (hd @_l ((σ_c(lnth t i)) #_l tl))
    using lappend-assoc by auto
  also have π_c(ltake n' t) = (π_c(ltake n' t)) @_l []_l by simp
  finally have (π_c(ltake n' t)) @_l []_l = (π_c(ltake n t)) @_l (hd @_l ((σ_c(lnth t i)) #_l tl)) .
  moreover from assms(3) have llength (π_c(ltake n' t)) = llength (π_c(ltake n t)) by simp
  ultimately have lfinite (π_c(ltake n' t)) ⟶ []_l = hd @_l ((σ_c(lnth t i)) #_l tl)
    using assms(3) lappend-eq-lappend-conv[of π_c(ltake n' t) π_c(ltake n t) []_l] by simp
  moreover have lfinite (π_c(ltake n' t)) by simp
  ultimately have []_l = hd @_l ((σ_c(lnth t i)) #_l tl) by simp
  hence (σ_c(lnth t i)) #_l tl = []_l using LNil-eq-lappend-iff by auto
  thus False by simp
qed

lemma proj-not-same-active:
  assumes enat n ≤ (n'::enat)
    and (¬ lfinite t) ∨ n'−1 < llength t
    and ¬(π_c(ltake n' t) = π_c(ltake n t))
  shows ∃k. k≥n ∧ k<n' ∧ enat k < llength t ∧ ⌇c⌇_lnth t k
proof (rule ccontr)
  assume ¬(∃k. k≥n ∧ k<n' ∧ enat k < llength t ∧ ⌇c⌇_lnth t k)
  have π_c(ltake n' t) = π_c(ltake (enat n) t)
  proof cases
    assume lfinite t
    hence llength t≠∞ by (simp add: lfinite-llength-enat)
    hence enat (the-enat (llength t)) = llength t by auto
    with assms ‹¬ (∃k≥n. k < n' ∧ enat k < llength t ∧ ⌇c⌇_lnth t k)›
      show ?thesis using proj-not-active-same[of n n' t c] by simp
```

```
  next
    assume ¬ lfinite t
    with assms ‹¬ (∃k≥n. k < n' ∧ enat k < llength t ∧ ≀c≀_{lnth t k})›
      show ?thesis using proj-not-active-same[of n n' t c] by simp
  qed
  with assms show False by simp
qed
```

D.9 Activations

We also introduce an operator to obtain the number of activations of a certain component within a given configuration trace.

```
definition nAct :: 'id ⇒ enat ⇒ (cnf llist) ⇒ enat (⟨- #_- -⟩) where
⟨c #_n t⟩ ≡ llength (π_c(ltake n t))

lemma nAct-0[simp]:
  ⟨c #_0 t⟩ = 0 by (simp add: nAct-def)

lemma nAct-NIL[simp]:
  ⟨c #_n []_l⟩ = 0 by (simp add: nAct-def)

lemma nAct-Null:
  assumes llength t ≥ n
      and ⟨c #_n t⟩ = 0
    shows ∀i<n. ¬ ≀c≀_{lnth t i}
proof –
  from assms have lnull (π_c(ltake n t)) using nAct-def lnull-def by simp
  hence π_c(ltake n t) = []_l using lnull-def by blast
  hence (∀k∈lset (ltake n t). ¬ ≀c≀_k) by simp
  show ?thesis
  proof (rule ccontr)
    assume ¬ (∀i<n. ¬ ≀c≀_{lnth t i})
    then obtain i where i<n and ≀c≀_{lnth t i} by blast
    moreover have enat i < llength (ltake n t)  ∧ lnth (ltake n t) i = (lnth t i)
    proof
      from ‹llength t ≥ n› have n = min n (llength t) using min.orderE by auto
      hence llength (ltake n t) = n by simp
      with ‹i<n› show enat i < llength (ltake n t) by auto
      from ‹i<n› show lnth (ltake n t) i = (lnth t i) using lnth-ltake by auto
    qed
    hence (lnth t i ∈ lset (ltake n t)) using in-lset-conv-lnth[of lnth t i ltake n t] by blast
    ultimately show False using ‹(∀k∈lset (ltake n t). ¬ ≀c≀_k)› by simp
  qed
qed

lemma nAct-ge-one[simp]:
  assumes llength t ≥ n
      and i < n
```

and $\wr c \wr_{lnth\ t\ i}$
shows $\langle c\ \#_n\ t\rangle \geq$ *enat 1*
proof (*rule ccontr*)
assume $\neg\ (\langle c\ \#_n\ t\rangle \geq$ *enat 1*)
hence $\langle c\ \#_n\ t\rangle <$ *enat 1* **by** *simp*
hence $\langle c\ \#_n\ t\rangle < 1$ **using** *enat-1* **by** *simp*
hence $\langle c\ \#_n\ t\rangle = 0$ **using** *Suc-ile-eq* ‹¬ *enat 1* $\leq \langle c\ \#_n\ t\rangle$› *zero-enat-def* **by** *auto*
with ‹*llength t* $\geq n$› **have** $\forall\, i{<}n.\ \neg\ \wr c \wr_{lnth\ t\ i}$ **using** *nAct-Null* **by** *simp*
with *assms* **show** *False* **by** *simp*
qed

lemma *nAct-finite*[*simp*]:
assumes $n \neq \infty$
shows $\exists\, n'.\ \langle c\ \#_n\ t\rangle =$ *enat* n'
proof –
from *assms* **have** *lfinite* (*ltake n t*) **by** *simp*
hence *lfinite* (π_c(*ltake n t*)) **by** *simp*
hence $\exists\, n'.$ *llength* (π_c(*ltake n t*)) = *enat* n'
using *lfinite-llength-enat*[*of* π_c(*ltake n t*)] **by** *simp*
thus *?thesis* **using** *nAct-def* **by** *simp*
qed

lemma *nAct-enat-the-nat*[*simp*]:
assumes $n \neq \infty$
shows *enat* (*the-enat* ($\langle c\ \#_n\ t\rangle$)) = $\langle c\ \#_n\ t\rangle$
proof –
from *assms* **have** $\langle c\ \#_n\ t\rangle \neq \infty$ **by** *simp*
thus *?thesis* **using** *enat-the-enat* **by** *simp*
qed

D.9.1 Monotonicity and Continuity

lemma *nAct-mcont*:
shows *mcont lSup lprefix Sup* ($\leq$) (*nAct c n*)
proof –
have *mcont lSup lprefix lSup lprefix* (*ltake n*) **by** *simp*
hence *mcont lSup lprefix lSup lprefix* ($\lambda t.\ \pi_c$(*ltake n t*))
using *proj-mcont2mcont*[*of lSup lprefix* (*ltake n*)] **by** *simp*
hence *mcont lSup lprefix Sup* ($\leq$) ($\lambda t.$ *llength* (π_c(*ltake n t*))) **by** *simp*
moreover have *nAct c n* = ($\lambda t.$ *llength* (π_c(*ltake n t*))) **using** *nAct-def* **by** *auto*
ultimately show *?thesis* **by** *simp*
qed

lemma *nAct-mono*:
assumes $n \leq n'$
shows $\langle c\ \#_n\ t\rangle \leq \langle c\ \#_{n'}\ t\rangle$
proof –
from *assms* **have** *lprefix* (*ltake n t*) (*ltake* n' *t*) **by** *simp*
hence *lprefix* (π_c(*ltake n t*)) (π_c(*ltake* n' *t*)) **by** *simp*

hence $llength\ (\pi_c(ltake\ n\ t)) \leq llength\ (\pi_c(ltake\ n'\ t))$
using *lprefix-llength-le*[*of* $(\pi_c(ltake\ n\ t))$] **by** *simp*
thus *?thesis* **using** *nAct-def* **by** *simp*
qed

lemma *nAct-strict-mono-back*:
assumes $\langle c\ \#_n\ t\rangle < \langle c\ \#_{n'}\ t\rangle$
shows $n < n'$
proof (*rule ccontr*)
assume $\neg\ n{<}n'$
hence $n{\geq}n'$ **by** *simp*
hence $\langle c\ \#_n\ t\rangle \geq \langle c\ \#_{n'}\ t\rangle$ **using** *nAct-mono* **by** *simp*
thus *False* **using** *assms* **by** *simp*
qed

D.9.2 Not Active

lemma *nAct-not-active*[*simp*]:
fixes n::*nat*
and n'::*nat*
and t::(*cnf llist*)
and c::*'id*
assumes $enat\ i < llength\ t$
and $\neg\ \wr c \wr_{lnth\ t\ i}$
shows $\langle c\ \#_{Suc\ i}\ t\rangle = \langle c\ \#_i\ t\rangle$
proof −
from *assms* **have** $\pi_c(ltake\ (Suc\ i)\ t) = \pi_c(ltake\ i\ t)$ **by** *simp*
hence $llength\ (\pi_c(ltake\ (enat\ (Suc\ i))\ t)) = llength\ (\pi_c(ltake\ i\ t))$ **by** *simp*
moreover have $llength\ (\pi_c(ltake\ i\ t)) \neq \infty$
using *llength-eq-infty-conv-lfinite*[*of* $\pi_c(ltake\ (enat\ i)\ t)$] **by** *simp*
ultimately have $llength\ (\pi_c(ltake\ (Suc\ i)\ t)) = llength\ (\pi_c(ltake\ i\ t))$
using *the-enat-eSuc* **by** *simp*
with *nAct-def* **show** *?thesis* **by** *simp*
qed

lemma *nAct-not-active-same*:
assumes $enat\ n \leq (n'{::}enat)$
and $n'{-}1 < llength\ t$
and $\nexists k.\ enat\ k{\geq}n \wedge k{<}n' \wedge \wr c \wr_{lnth\ t\ k}$
shows $\langle c\ \#_{n'}\ t\rangle = \langle c\ \#_n\ t\rangle$
using *assms proj-not-active-same nAct-def* **by** *simp*

D.9.3 Active

lemma *nAct-active*[*simp*]:
fixes n::*nat*
and n'::*nat*
and t::(*cnf llist*)
and c::*'id*
assumes $enat\ i < llength\ t$

and $\wr c \wr_{lnth\ t\ i}$
shows $\langle c\ \#_{Suc\ i}\ t\rangle = eSuc\ (\langle c\ \#_i\ t\rangle)$
proof –
from *assms* **have** $\pi_c(ltake\ (Suc\ i)\ t) =$
$(\pi_c(ltake\ i\ t))\ @_l\ ((\sigma_c(lnth\ t\ i))\ \#_l\ []_l)$ **by** *simp*
hence $llength\ (\pi_c(ltake\ (enat\ (Suc\ i))\ t)) = eSuc\ (llength\ (\pi_c(ltake\ i\ t)))$
using *plus-1-eSuc one-eSuc* **by** *simp*
moreover have $llength\ (\pi_c(ltake\ i\ t)) \neq \infty$
using *llength-eq-infty-conv-lfinite*[*of* $\pi_c(ltake\ (enat\ i)\ t)$] **by** *simp*
ultimately have $llength\ (\pi_c(ltake\ (Suc\ i)\ t)) = eSuc\ (llength\ (\pi_c(ltake\ i\ t)))$
using *the-enat-eSuc* **by** *simp*
with *nAct-def* **show** *?thesis* **by** *simp*
qed

lemma *nAct-active-suc*:
fixes $n::nat$
and $n'::enat$
and $t::(cnf\ llist)$
and $c::'id$
assumes $\neg\ lfinite\ t \vee n'{-}1 < llength\ t$
and $n \leq i$
and $enat\ i < n'$
and $\wr c \wr_{lnth\ t\ i}$
and $\forall i'.\ (n \leq i' \wedge enat\ i'{<}n' \wedge i' < llength\ t \wedge \wr c \wr_{lnth\ t\ i'}) \longrightarrow (i' = i)$
shows $\langle c\ \#_{n'}\ t\rangle = eSuc\ (\langle c\ \#_n\ t\rangle)$
proof –
from *assms* **have** $\pi_c(ltake\ n'\ t) = (\pi_c(ltake\ (enat\ n)\ t))\ @_l\ ((\sigma_c(lnth\ t\ i))\ \#_l\ []_l)$
using *proj-active-append*[*of n i n' t c*] **by** *blast*
moreover have $llength\ ((\pi_c(ltake\ (enat\ n)\ t))\ @_l\ ((\sigma_c(lnth\ t\ i))\ \#_l\ []_l)) =$
$eSuc\ (llength\ (\pi_c(ltake\ (enat\ n)\ t)))$ **using** *one-eSuc eSuc-plus-1* **by** *simp*
ultimately show *?thesis* **using** *nAct-def* **by** *simp*
qed

lemma *nAct-less*:
assumes $enat\ k < llength\ t$
and $n \leq k$
and $k < (n'::enat)$
and $\wr c \wr_{lnth\ t\ k}$
shows $\langle c\ \#_n\ t\rangle < \langle c\ \#_{n'}\ t\rangle$
proof –
have $\langle c\ \#_k\ t\rangle \neq \infty$ **by** *simp*
then obtain *en* **where** *en-def*: $\langle c\ \#_k\ t\rangle = enat\ en$ **by** *blast*
moreover have $eSuc\ (enat\ en) \leq \langle c\ \#_{n'}\ t\rangle$
proof –
from *assms* **have** $Suc\ k \leq n'$ **using** *Suc-ile-eq* **by** *simp*
hence $\langle c\ \#_{Suc\ k}\ t\rangle \leq \langle c\ \#_{n'}\ t\rangle$ **using** *nAct-mono* **by** *simp*
moreover from *assms* **have** $\langle c\ \#_{Suc\ k}\ t\rangle = eSuc\ (\langle c\ \#_k\ t\rangle)$ **by** *simp*
ultimately have $eSuc\ (\langle c\ \#_k\ t\rangle) \leq \langle c\ \#_{n'}\ t\rangle$ **by** *simp*
thus *?thesis* **using** *en-def* **by** *simp*

qed
moreover have $\textit{enat en} < \textit{eSuc}\ (\textit{enat en})$ **by** *simp*
ultimately have $\textit{enat en} < \langle c \#_{n'}\ t\rangle$ **using** *less-le-trans*[*of enat en eSuc* (*enat en*)] **by** *simp*
moreover have $\langle c \#_{n}\ t\rangle \le \textit{enat en}$
proof −
from *assms* **have** $\langle c \#_{n}\ t\rangle \le \langle c \#_{k}\ t\rangle$ **using** *nAct-mono* **by** *simp*
thus *?thesis* **using** *en-def* **by** *simp*
qed
ultimately show *?thesis* **using** *le-less-trans*[*of* $\langle c \#_{n}\ t\rangle$] **by** *simp*
qed

lemma *nAct-less-active*:
assumes $n' - 1 < \textit{llength}\ t$
and $\langle c \#_{\textit{enat}\ n}\ t\rangle < \langle c \#_{n'}\ t\rangle$
shows $\exists\, i{\ge}n.\ i{<}n' \wedge \wr c\wr_{\textit{lnth}\ t\ i}$
proof (*rule ccontr*)
assume $\neg\ (\exists\, i{\ge}n.\ i{<}n' \wedge \wr c\wr_{\textit{lnth}\ t\ i})$
moreover have $\textit{enat}\ n \le n'$ **using** *assms*(*2*) *less-imp-le nAct-strict-mono-back* **by** *blast*
ultimately have $\langle c \#_{n}\ t\rangle = \langle c \#_{n'}\ t\rangle$ **using** ‹$n' - 1 < \textit{llength}\ t$› *nAct-not-active-same*
by *simp*
thus *False* **using** *assms* **by** *simp*
qed

D.9.4 Same and Not Same

lemma *nAct-same-not-active*:
assumes $\langle c \#_{n'}\ \textit{inf-llist}\ t\rangle = \langle c \#_{n}\ \textit{inf-llist}\ t\rangle$
shows $\forall\, k{\ge}n.\ k{<}n' \longrightarrow \neg\ \wr c\wr_{t\ k}$
proof (*rule ccontr*)
assume $\neg(\forall\, k{\ge}n.\ k{<}n' \longrightarrow \neg\ \wr c\wr_{t\ k})$
then obtain k **where** $k{\ge}n$ **and** $k{<}n'$ **and** $\wr c\wr_{t\ k}$ **by** *blast*
hence $\langle c \#_{\textit{Suc}\ k}\ \textit{inf-llist}\ t\rangle = \textit{eSuc}\ (\langle c \#_{k}\ \textit{inf-llist}\ t\rangle)$ **by** *simp*
moreover have $\langle c \#_{k}\ \textit{inf-llist}\ t\rangle{\neq}\infty$ **by** *simp*
ultimately have $\langle c \#_{k}\ \textit{inf-llist}\ t\rangle < \langle c \#_{\textit{Suc}\ k}\ \textit{inf-llist}\ t\rangle$ **by** *fastforce*
moreover from ‹$n{\le}k$› **have** $\langle c \#_{n}\ \textit{inf-llist}\ t\rangle \le \langle c \#_{k}\ \textit{inf-llist}\ t\rangle$ **using** *nAct-mono* **by** *simp*
moreover from ‹$k{<}n'$› **have** $\textit{Suc}\ k \le n'$ **by** (*simp add*: *Suc-ile-eq*)
hence $\langle c \#_{\textit{Suc}\ k}\ \textit{inf-llist}\ t\rangle \le \langle c \#_{n'}\ \textit{inf-llist}\ t\rangle$ **using** *nAct-mono* **by** *simp*
ultimately show *False* **using** *assms* **by** *simp*
qed

lemma *nAct-not-same-active*:
assumes $\langle c \#_{\textit{enat}\ n}\ t\rangle < \langle c \#_{n'}\ t\rangle$
and $\neg\ \textit{lfinite}\ t \vee n' - 1 < \textit{llength}\ t$
shows $\exists\,(i{::}\textit{nat}){\ge}n.\ \textit{enat}\ i{<}\ n' \wedge i{<}\textit{llength}\ t \wedge \wr c\wr_{\textit{lnth}\ t\ i}$
proof −
from *assms* **have** $\textit{llength}(\pi_{c}(\textit{ltake}\ n\ t)) < \textit{llength}\ (\pi_{c}(\textit{ltake}\ n'\ t))$ **using** *nAct-def* **by** *simp*
hence $\pi_{c}(\textit{ltake}\ n'\ t) \neq \pi_{c}(\textit{ltake}\ n\ t)$ **by** *auto*
moreover from *assms* **have** $\textit{enat}\ n < n'$ **using** *nAct-strict-mono-back*[*of c enat n*] **by** *simp*
ultimately show *?thesis* **using** *proj-not-same-active*[*of n n' t c*] *assms* **by** *simp*

```
qed

lemma nAct-less-llength-active:
  assumes x < llength (π_{c}(t))
    and enat x = ⟨c #_{enat n'} t⟩
  shows ∃ (i::nat)≥n'. i<llength t ∧ ⦃c⦄_{lnth t i}
proof –
  have llength(π_{c}(ltake n' t)) < llength (π_{c}(t)) using assms(1) assms(2) nAct-def by auto
  hence llength(π_{c}(ltake n' t)) < llength (π_{c}(ltake (llength t) t)) by (simp add: ltake-all)
  hence ⟨c #_{enat n'} t⟩ < ⟨c #_{llength t} t⟩ using nAct-def by simp
  moreover have ¬ lfinite t ∨ llength t – 1 < llength t
  proof (rule Meson.imp-to-disjD[OF HOL.impI])
    assume lfinite t
    hence llength t ≠ ∞ by (simp add: llength-eq-infty-conv-lfinite)
    moreover have llength t>0
    proof –
      from ‹x < llength (π_{c}(t))› have llength (π_{c}(t))>0 by auto
      thus ?thesis using proj-llength Orderings.order-class.order.strict-trans2 by blast
    qed
    ultimately show llength t – 1 < llength t by (metis One-nat-def ‹lfinite t› diff-Suc-less
      enat-ord-simps(2) idiff-enat-enat lfinite-conv-llength-enat one-enat-def zero-enat-def)
  qed
  ultimately show ?thesis using nAct-not-same-active[of c n' t llength t] by simp
qed

lemma nAct-exists:
  assumes x < llength (π_{c}(t))
  shows ∃ (n'::nat). enat x = ⟨c #_{n'} t⟩
proof –
  have x < llength (π_{c}(t)) ⟶ (∃ (n'::nat). enat x = ⟨c #_{n'} t⟩)
  proof (induction x)
    case 0
    thus ?case by (metis nAct-0 zero-enat-def)
  next
    case (Suc x)
    show ?case
    proof
      assume Suc x < llength (π_{c}(t))
      hence x < llength (π_{c}(t)) using Suc-ile-eq less-imp-le by auto
      with Suc.IH obtain n' where enat x = ⟨c #_{enat n'} t⟩ by blast
      with ‹x < llength (π_{c}(t))› have ∃ i≥n'. i < llength t ∧ ⦃c⦄_{lnth t i}
        using nAct-less-llength-active[of x c t n'] by simp
      then obtain i where i≥n' and i<llength t and ⦃c⦄_{lnth t i}
        and ∄ k. n'≤k ∧ k<i ∧ k<llength t ∧ ⦃c⦄_{lnth t k} using lActive-least[of n' t c] by auto
      moreover from ‹i<llength t› have ¬ lfinite t ∨ enat (Suc i) – 1 < llength t
        by (simp add: one-enat-def)
      moreover have enat i < enat (Suc i) by simp
      moreover have ∀ i'. (n' ≤ i' ∧ enat i'<enat (Suc i) ∧ i'<llength t ∧ ⦃c⦄_{lnth t i'})
        ⟶ (i' = i)
```

proof (*rule HOL.impI*[*THEN HOL.allI*])
fix i' **assume** $n' \leq i' \wedge enat\ i'<enat\ (Suc\ i) \wedge i'<llength\ t \wedge \wr c\wr_{lnth\ t\ i'}$
with ‹$\nexists k.\ n'\leq k \wedge k<i \wedge k<llength\ t \wedge \wr c\wr_{lnth\ t\ k}$› **show** $i'=i$ **by** *fastforce*
qed
ultimately have $\langle c\ \#_{Suc\ i}\ t\rangle = eSuc\ (\langle c\ \#_{n'}\ t\rangle)$
using *nAct-active-suc*[*of t Suc i n′ i c*] **by** *simp*
with ‹$enat\ x = \langle c\ \#_{enat\ n'}\ t\rangle$› **have** $\langle c\ \#_{Suc\ i}\ t\rangle = eSuc\ (enat\ x)$ **by** *simp*
thus $\exists n'.\ enat\ (Suc\ x) = \langle c\ \#_{enat\ n'}\ t\rangle$ **by** (*metis eSuc-enat*)
qed
qed
with *assms* **show** *?thesis* **by** *simp*
qed

D.10 Projection and Activation

In the following we provide some properties about the relationship between the projection and activations operator.

lemma *nAct-le-proj*:
$\langle c\ \#_n\ t\rangle \leq llength\ (\pi_c(t))$
proof –
from *nAct-def* **have** $\langle c\ \#_n\ t\rangle = llength\ (\pi_c(ltake\ n\ t))$ **by** *simp*
moreover have $llength\ (\pi_c(ltake\ n\ t)) \leq llength\ (\pi_c(t))$
proof –
have *lprefix* (*ltake n t*) *t* **by** *simp*
hence $lprefix\ (\pi_c(ltake\ n\ t))\ (\pi_c(t))$ **by** *simp*
hence $llength\ (\pi_c(ltake\ n\ t)) \leq llength\ (\pi_c(t))$ **using** *lprefix-llength-le* **by** *blast*
thus *?thesis* **by** *auto*
qed
thus *?thesis* **using** *nAct-def* **by** *simp*
qed

lemma *proj-nAct*:
assumes (*enat n < llength t*)
shows $\pi_c(ltake\ n\ t) = ltake\ (\langle c\ \#_n\ t\rangle)\ (\pi_c(t))$ (**is** *?lhs = ?rhs*)
proof –
have $?lhs = ltake\ (llength\ (\pi_c(ltake\ n\ t)))\ (\pi_c(ltake\ n\ t))$
using *ltake-all*[*of* $\pi_c(ltake\ n\ t)\ llength\ (\pi_c(ltake\ n\ t))$] **by** *simp*
also have $\ldots = ltake\ (llength\ (\pi_c(ltake\ n\ t)))\ ((\pi_c(ltake\ n\ t))\ @_l\ (\pi_c(ldrop\ n\ t)))$
using *ltake-lappend1*[*of* $llength\ (\pi_c(ltake\ (enat\ n)\ t))\ \pi_c(ltake\ n\ t)\ (\pi_c(ldrop\ n\ t))$] **by** *simp*
also have $\ldots = ltake\ (\langle c\ \#_n\ t\rangle)\ ((\pi_c(ltake\ n\ t))\ @_l\ (\pi_c(ldrop\ n\ t)))$ **using** *nAct-def* **by** *simp*

also have $\ldots = ltake\ (\langle c\ \#_n\ t\rangle)\ (\pi_c((ltake\ (enat\ n)\ t)\ @_l\ (ldrop\ n\ t)))$ **by** *simp*
also have $\ldots = ltake\ (\langle c\ \#_n\ t\rangle)\ (\pi_c(t))$ **using** *lappend-ltake-ldrop*[*of n t*] **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *proj-active-nth*:
assumes $enat\ (Suc\ i) < llength\ t\ \wr c\wr_{lnth\ t\ i}$

```
  shows lnth (π_c(t)) (the-enat (⟨c #_i t⟩)) = σ_c(lnth t i)
proof −
  from assms have enat i < llength t using Suc-ile-eq[of i llength t] by auto
  with assms have π_c(ltake (Suc i) t) = (π_c(ltake i t)) @_l ((σ_c(lnth t i)) #_l []_l) by simp
  moreover have lnth ((π_c(ltake i t)) @_l ((σ_c(lnth t i)) #_l []_l))
    (the-enat (llength (π_c(ltake i t)))) = σ_c(lnth t i)
  proof −
    have ¬ lnull ((σ_c(lnth t i)) #_l []_l) by simp
    moreover have lfinite (π_c(ltake i t)) by simp
    ultimately have lnth ((π_c(ltake i t)) @_l ((σ_c(lnth t i)) #_l []_l))
      (the-enat (llength (π_c(ltake i t)))) = lhd ((σ_c(lnth t i)) #_l []_l) by simp
    also have ... = σ_c(lnth t i) by simp
    finally show lnth ((π_c(ltake i t)) @_l ((σ_c(lnth t i)) #_l []_l))
      (the-enat (llength (π_c(ltake i t)))) = σ_c(lnth t i) by simp
  qed
  ultimately have σ_c(lnth t i) = lnth (π_c(ltake (Suc i) t))
    (the-enat (llength (π_c(ltake i t)))) by simp
  also have ... = lnth (π_c(ltake (Suc i) t)) (the-enat (⟨c #_i t⟩)) using nAct-def by simp
  also have ... = lnth (ltake (⟨c #_Suc i t⟩) (π_c(t))) (the-enat (⟨c #_i t⟩))
    using proj-nAct[of Suc i t c] assms by simp
  also have ... = lnth (π_c(t)) (the-enat (⟨c #_i t⟩))
  proof −
    from assms have ⟨c #_Suc i t⟩ = eSuc (⟨c #_i t⟩) using ‹enat i < llength t› by simp
    moreover have ⟨c #_i t⟩ < eSuc (⟨c #_i t⟩)
      using iless-Suc-eq[of the-enat (⟨c #_enat i t⟩)] by simp
    ultimately have ⟨c #_i t⟩ < (⟨c #_Suc i t⟩) by simp
    hence enat (the-enat (⟨c #_Suc i t⟩)) > enat (the-enat (⟨c #_i t⟩)) by simp
    thus ?thesis using lnth-ltake[of the-enat (⟨c #_i t⟩) the-enat (⟨c #_enat (Suc i) t⟩) π_c(t)]
      by simp
  qed
  finally show ?thesis ..
qed

lemma nAct-eq-proj:
  assumes ¬(∃ i≥n. §c§_lnth t i)
  shows ⟨c #_n t⟩ = llength (π_c(t)) (is ?lhs = ?rhs)
proof −
  from nAct-def have ?lhs = llength (π_c(ltake n t)) by simp
  moreover from assms have ∀ (n'::nat)≤llength t. n'≥n ⟶ (¬ §c§_lnth t n') by simp
  hence π_c(t) = π_c(ltake n t) using proj-ltake by simp
  ultimately show ?thesis by simp
qed

lemma nAct-llength-proj:
  assumes ∃ i≥n. §c§_t i
  shows llength (π_c(inf-llist t)) ≥ eSuc (⟨c #_n inf-llist t⟩)
proof −
  from ‹∃ i≥n. §c§_t i› obtain i where i≥n and §c§_t i
    and ¬ (∃ k≥n. k < i ∧ k < llength (inf-llist t) ∧ §c§_t k)
```

```
    using lActive-least[of n inf-llist t c] by auto
  moreover have llength (πc(inf-llist t)) ≥ ⟨c #Suc i inf-llist t⟩ using nAct-le-proj by simp
  moreover have eSuc (⟨c #n inf-llist t⟩) = ⟨c #Suc i inf-llist t⟩
  proof –
    have enat (Suc i) < llength (inf-llist t) by simp
    moreover have i < Suc i by simp
    moreover from ‹¬ (∃k≥n. k < i ∧ k < llength (inf-llist t) ∧ ⦃c⦄t k)›
      have ∀i'. n ≤ i' ∧ i' < Suc i ∧ ⦃c⦄lnth (inf-llist t) i' ⟶ i' = i by fastforce
    ultimately show ?thesis using nAct-active-suc ‹i≥n› ‹⦃c⦄t i› by simp
  qed
  ultimately show ?thesis by simp
qed
```

D.11 Least not Active

In the following, we introduce an operator to obtain the least point in time before a certain point in time where a component was deactivated.

```
definition lNAct :: 'id ⇒ (nat ⇒ cnf) ⇒ nat ⇒ nat (⟨- ⇐ -⟩-)
  where ⟨c ⇐ t⟩n ≡ (LEAST n'. n=n' ∨ (n'<n ∧ (∄k. k≥n' ∧ k<n ∧ ⦃c⦄t k)))

lemma lNact0[simp]:
  ⟨c ⇐ t⟩0 = 0
  by (simp add: lNAct-def)

lemma lNact-least:
  assumes n=n' ∨ n'<n ∧ (∄k. k≥n' ∧ k<n ∧ ⦃c⦄t k)
  shows ⟨c ⇐ t⟩n ≤ n'
using Least-le[of λn'. n=n' ∨ (n'<n ∧ (∄k. k≥n' ∧ k<n ∧ ⦃c⦄t k)) n'] lNAct-def using assms
by auto

lemma lNAct-ex: ⟨c ⇐ t⟩n=n ∨ ⟨c ⇐ t⟩n<n ∧ (∄k. k≥⟨c ⇐ t⟩n ∧ k<n ∧ ⦃c⦄t k)
proof –
  let ?P=λn'. n=n' ∨ n'<n ∧ (∄k. k≥n' ∧ k<n ∧ ⦃c⦄t k)
  from lNAct-def have ⟨c ⇐ t⟩n = (LEAST n'. ?P n') by simp
  moreover have ?P n by simp
  with LeastI have ?P (LEAST n'. ?P n') .
  ultimately show ?thesis by auto
qed

lemma lNact-notActive:
  fixes c t n k
  assumes k≥⟨c ⇐ t⟩n
    and k<n
  shows ¬⦃c⦄t k
  by (metis assms lNAct-ex leD)

lemma lNactGe:
  fixes c t n n'
```

```
  assumes n' ≥ ⟨c ⇐ t⟩_n
    and ⦃c⦄_t n'
  shows n' ≥ n
  using assms lNact-notActive leI by blast

lemma lNactLe[simp]:
  fixes n n'
  shows ⟨c ⇐ t⟩_n ≤ n
  using lNAct-ex less-or-eq-imp-le by blast

lemma lNactLe-nact:
  fixes n n'
  assumes n'=n ∨ (n'<n ∧ (∄k. k≥n' ∧ k<n ∧ ⦃c⦄_t k))
  shows ⟨c ⇐ t⟩_n ≤ n'
  using assms lNAct-def Least-le[of λn'. n=n' ∨ (n'<n ∧ (∄k. k≥n' ∧ k<n ∧ ⦃c⦄_t k))] by auto

lemma lNact-active:
  fixes cid t n
  assumes ∀k<n. ⦃cid⦄_t k
  shows ⟨cid ⇐ t⟩_n = n
  using assms lNAct-ex by blast

lemma nAct-mono-back:
  fixes c t and n and n'
  assumes ⟨c #_n' inf-llist t⟩ ≥ ⟨c #_n inf-llist t⟩
  shows n'≥⟨c ⇐ t⟩_n
proof cases
  assume ⟨c #_n' inf-llist t⟩ = ⟨c #_n inf-llist t⟩
  thus ?thesis
  proof cases
    assume n'≥n
    thus ?thesis using lNactLe by (metis HOL.no-atp(11))
  next
    assume ¬ n'≥n
    hence n'<n by simp
    with ‹⟨c #_n' inf-llist t⟩ = ⟨c #_n inf-llist t⟩› have ∄k. k≥n' ∧ k<n ∧ ⦃c⦄_t k
      by (metis enat-ord-simps(1) enat-ord-simps(2) nAct-same-not-active)
    thus ?thesis using lNactLe-nact by (simp add: ‹n' < n›)
  qed
next
  assume ¬⟨c #_n' inf-llist t⟩ = ⟨c #_n inf-llist t⟩
  with assms have ⟨c #_enat n' inf-llist t⟩ > ⟨c #_enat n inf-llist t⟩ by simp
  hence n' > n using nAct-strict-mono-back[of c enat n inf-llist t enat n'] by simp
  thus ?thesis by (meson dual-order.strict-implies-order lNactLe le-trans)
qed

lemma nAct-mono-lNact:
  assumes ⟨c ⇐ t⟩_n ≤ n'
  shows ⟨c #_n inf-llist t⟩ ≤ ⟨c #_n' inf-llist t⟩
```

proof –
  have $\nexists k.\ k\geq\langle c \Leftarrow t\rangle_{n} \wedge k<n \wedge \wr c\wr_{t\ k}$ **using** *lNact-notActive* **by** *auto*
  moreover have $enat\ n - 1 < llength\ (inf\text{-}llist\ t)$ **by** (*simp add*: *one-enat-def*)
  moreover from ‹$\langle c \Leftarrow t\rangle_{n} \leq n'$› **have** $enat\ \langle c \Leftarrow t\rangle_{n} \leq enat\ n$ **by** *simp*
  ultimately have $\langle c\ \#_{n}\ inf\text{-}llist\ t\rangle = \langle c\ \#_{\langle c \Leftarrow t\rangle_{n}}\ inf\text{-}llist\ t\rangle$
    using *nAct-not-active-same* **by** *simp*
  thus *?thesis* **using** *nAct-mono assms* **by** *simp*
qed

D.12 Next Active

In the following, we introduce an operator to obtain the next point in time when a component is activated.

definition $nxtAct :: 'id \Rightarrow (nat \Rightarrow cnf) \Rightarrow nat \Rightarrow nat\ (\langle\text{-} \rightarrow \text{-}\rangle_{\text{-}})$
  where $\langle c \rightarrow t\rangle_{n} \equiv (THE\ n'.\ n'\geq n \wedge \wr c\wr_{t\ n'} \wedge (\nexists k.\ k\geq n \wedge k<n' \wedge \wr c\wr_{t\ k}))$

lemma *nxtActI*:
  fixes $n::nat$
    and $t::nat \Rightarrow cnf$
    and $c::'id$
  assumes $\exists\, i\geq n.\ \wr c\wr_{t\ i}$
  shows $\langle c \rightarrow t\rangle_{n} \geq n \wedge \wr c\wr_{t\ \langle c \rightarrow t\rangle_{n}} \wedge (\nexists k.\ k\geq n \wedge k<\langle c \rightarrow t\rangle_{n} \wedge \wr c\wr_{t\ k})$
proof –
  let $?P = THE\ n'.\ n'\geq n \wedge \wr c\wr_{t\ n'} \wedge (\nexists k.\ k\geq n \wedge k<n' \wedge \wr c\wr_{t\ k})$
  from *assms* **obtain** i **where** $i\geq n \wedge \wr c\wr_{t\ i} \wedge (\nexists k.\ k\geq n \wedge k<i \wedge \wr c\wr_{t\ k})$
    using *lActive-least*[*of n inf-llist t c*] **by** *auto*
  moreover have $(\bigwedge x.\ n \leq x \wedge \wr c\wr_{t\ x} \wedge \neg\, (\exists\, k\geq n.\ k < x \wedge \wr c\wr_{t\ k}) \Longrightarrow x = i)$
  proof –
    fix x **assume** $n \leq x \wedge \wr c\wr_{t\ x} \wedge \neg\, (\exists\, k\geq n.\ k < x \wedge \wr c\wr_{t\ k})$
    show $x = i$
    proof (*rule ccontr*)
      assume $\neg\, (x = i)$
      thus *False* **using** ‹$i\geq n \wedge \wr c\wr_{t\ i} \wedge (\nexists k.\ k\geq n \wedge k<i \wedge \wr c\wr_{t\ k})$›
        ‹$n \leq x \wedge \wr c\wr_{t\ x} \wedge \neg\, (\exists\, k\geq n.\ k < x \wedge \wr c\wr_{t\ k})$› **by** *fastforce*
    qed
  qed
  ultimately have $(?P) \geq n \wedge \wr c\wr_{t\ (?P)} \wedge (\nexists k.\ k\geq n \wedge k<?P \wedge \wr c\wr_{t\ k})$
    using *theI*[*of* $\lambda n'.\ n'\geq n \wedge \wr c\wr_{t\ n'} \wedge (\nexists k.\ k\geq n \wedge k<n' \wedge \wr c\wr_{t\ k})$] **by** *blast*
  thus *?thesis* **using** *nxtAct-def*[*of c t n*] **by** *metis*
qed

lemma *nxtActLe*:
  fixes $n\ n'$
  assumes $\exists\, i\geq n.\ \wr c\wr_{t\ i}$
  shows $n \leq \langle c \rightarrow t\rangle_{n}$
  by (*simp add*: *assms nxtActI*)

lemma *nxtAct-eq*:

```
  assumes n'≥n
    and ⦃c⦄_{t n'}
    and ∀ n''≥n. n''<n' ⟶ ¬ ⦃c⦄_{t n''}
  shows n' = ⟨c → t⟩_n
  by (metis assms(1) assms(2) assms(3) nxtActI linorder-neqE-nat nxtActLe)

lemma nxtAct-active:
  fixes i::nat
    and t::nat ⇒ cnf
    and c::'id
  assumes ⦃c⦄_{t i}
  shows ⟨c → t⟩_i = i by (metis assms le-eq-less-or-eq nxtActI)

lemma nxtActive-no-active:
  assumes ∃!i. i≥n ∧ ⦃c⦄_{t i}
  shows ¬ (∃ i'≥Suc ⟨c → t⟩_n. ⦃c⦄_{t i'})
proof
  assume ∃ i'≥Suc ⟨c → t⟩_n. ⦃c⦄_{t i'}
  then obtain i' where i'≥Suc ⟨c → t⟩_n and ⦃c⦄_{t i'} by auto
  moreover from assms(1) have ⟨c → t⟩_n≥n using nxtActI by auto
  ultimately have i'≥n and ⦃c⦄_{t i'} and i'≠⟨c → t⟩_n by auto
  moreover from assms(1) have ⦃c⦄_{t ⟨c → t⟩_n} and ⟨c → t⟩_n≥n using nxtActI by auto
  ultimately show False using assms(1) by auto
qed

lemma nxt-geq-lNact[simp]:
  assumes ∃ i≥n. ⦃c⦄_{t i}
  shows ⟨c → t⟩_n≥⟨c ⇐ t⟩_n
proof -
  from assms have n ≤ ⟨c → t⟩_n using nxtActLe by simp
  moreover have ⟨c ⇐ t⟩_n≤n by simp
  ultimately show ?thesis by arith
qed

lemma active-geq-nxtAct:
  assumes ⦃c⦄_{t i}
    and the-enat (⟨c #_i inf-llist t⟩)≥the-enat (⟨c #_n inf-llist t⟩)
  shows i≥⟨c → t⟩_n
proof cases
  assume ⟨c #_i inf-llist t⟩=⟨c #_n inf-llist t⟩
  show ?thesis
  proof (rule ccontr)
    assume ¬ i≥⟨c → t⟩_n
    hence i<⟨c → t⟩_n by simp
    with ‹⟨c #_i inf-llist t⟩=⟨c #_n inf-llist t⟩› have ¬ (∃ k≥i. k < n ∧ ⦃c⦄_{t k})
      by (metis enat-ord-simps(1) leD leI nAct-same-not-active)
    moreover have ¬ (∃ k≥n. k <⟨c → t⟩_n ∧ ⦃c⦄_{t k}) using nxtActI by blast
    ultimately have ¬ (∃ k≥i. k <⟨c → t⟩_n ∧ ⦃c⦄_{t k}) by auto
    with ‹i<⟨c → t⟩_n› show False using ‹⦃c⦄_{t i}› by simp
```

```
  qed
next
  assume ¬⟨c #_i inf-llist t⟩=⟨c #_n inf-llist t⟩
  moreover from ‹the-enat (⟨c #_i inf-llist t⟩)≥the-enat (⟨c #_n inf-llist t⟩)›
  have ⟨c #_i inf-llist t⟩≥⟨c #_n inf-llist t⟩
    by (metis enat.distinct(2) enat-ord-simps(1) nAct-enat-the-nat)
  ultimately have ⟨c #_i inf-llist t⟩>⟨c #_n inf-llist t⟩ by simp
  hence i>n using nAct-strict-mono-back[of c n inf-llist t i] by simp
  with ‹≀c≀_t i› show ?thesis by (meson dual-order.strict-implies-order leI nxtActI)
qed

lemma nAct-same:
  assumes ⟨c ⇐ t⟩_n ≤ n' and n' ≤ ⟨c → t⟩_n
  shows the-enat (⟨c #_enat n' inf-llist t⟩) = the-enat (⟨c #_enat n inf-llist t⟩)
proof cases
  assume n ≤ n'
  moreover have n' − 1 < llength (inf-llist t) by simp
  moreover have ¬ (∃ i≥n. i <n' ∧ ≀c≀_t i) by (meson assms(2) less-le-trans nxtActI)
  ultimately show ?thesis using nAct-not-active-same by (simp add: one-enat-def)
next
  assume ¬ n ≤ n'
  hence n' < n by simp
  moreover have n − 1 < llength (inf-llist t) by simp
  moreover have ¬ (∃ i≥n'. i < n ∧ ≀c≀_t i)
    by (metis ‹¬ n ≤ n'› assms(1) dual-order.trans lNAct-ex)
  ultimately show ?thesis using nAct-not-active-same[of n' n] by (simp add: one-enat-def)
qed

lemma nAct-mono-nxtAct:
  assumes ∃ i≥n. ≀c≀_t i
    and ⟨c → t⟩_n ≤ n'
  shows ⟨c #_n inf-llist t⟩ ≤ ⟨c #_n' inf-llist t⟩
proof −
  from assms have ⟨c #_⟨c → t⟩_n inf-llist t⟩ ≤ ⟨c #_n' inf-llist t⟩
    using nAct-mono assms by simp
  moreover have ⟨c #_⟨c → t⟩_n inf-llist t⟩=⟨c #_n inf-llist t⟩
  proof −
    from assms have ∄ k. k≥n ∧ k<⟨c → t⟩_n ∧ ≀c≀_t k and n ≤ ⟨c → t⟩_n
      using nxtActI by auto
    moreover have enat ⟨c → t⟩_n − 1 < llength (inf-llist t) by (simp add: one-enat-def)
    ultimately show ?thesis using nAct-not-active-same[of n ⟨c → t⟩_n] by auto
  qed
  ultimately show ?thesis by simp
qed
```

D.13 Latest Activation

In the following, we introduce an operator to obtain the latest point in time when a component is activated.

abbreviation *latestAct-cond*:: *'id* $\Rightarrow$ *trace* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
  where *latestAct-cond c t n n'* $\equiv n'<n \wedge \wr c \wr_{t\ n'}$

definition *latestAct*:: *'id* $\Rightarrow$ *trace* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $(\langle \text{-} \leftarrow \text{-}\rangle_{\text{-}})$
  where *latestAct c t n* $=$ (*GREATEST n'*. *latestAct-cond c t n n'*)

lemma *latestActEx*:
  assumes $\exists\, n'<n.\ \wr nid \wr_{t\ n'}$
  shows $\exists\, n'$. *latestAct-cond nid t n n'* $\wedge$ ($\forall\, n''$. *latestAct-cond nid t n n''* $\longrightarrow n'' \leq n'$)
proof –
  from *assms* **obtain** n' **where** *latestAct-cond nid t n n'* **by** *auto*
  moreover have $\forall\, n''>n.\ \neg$ *latestAct-cond nid t n n''* **by** *simp*
  ultimately obtain n' **where** *latestAct-cond nid t n n'* $\wedge$
    ($\forall\, n''$. *latestAct-cond nid t n n''* $\longrightarrow n'' \leq n'$)
    using *boundedGreatest*[*of latestAct-cond nid t n n'*] **by** *blast*
  thus *?thesis* **..**
qed

lemma *latestAct-prop*:
  assumes $\exists\, n'<n.\ \wr nid \wr_{t\ n'}$
  shows $\wr nid \wr_{t\ (latestAct\ nid\ t\ n)}$ **and** *latestAct nid t n*$<n$
proof –
  from *assms latestActEx* **have**
    latestAct-cond nid t n (*GREATEST x*. *latestAct-cond nid t n x*)
    using *GreatestI-ex-nat*[*of latestAct-cond nid t n*] **by** *blast*
  thus $\wr nid \wr_{t\ \langle nid \leftarrow t\rangle_n}$ **and** *latestAct nid t n*$<n$ **using** *latestAct-def* **by** *auto*
qed

lemma *latestAct-less*:
  assumes *latestAct-cond nid t n n'*
  shows $n' \leq \langle nid \leftarrow t\rangle_n$
proof –
  from *assms latestActEx* **have** $n' \leq$ (*GREATEST x*. *latestAct-cond nid t n x*)
    using *Greatest-le-nat*[*of latestAct-cond nid t n*] **by** *blast*
  thus *?thesis* **using** *latestAct-def* **by** *auto*
qed

lemma *latestActNxt*:
  assumes $\exists\, n'<n.\ \wr nid \wr_{t\ n'}$
  shows $\langle nid \rightarrow t\rangle_{\langle nid \leftarrow t\rangle_n} = \langle nid \leftarrow t\rangle_n$
  using *assms latestAct-prop*(*1*) *nxtAct-active* **by** *auto*

lemma *latestActNxtAct*:
  assumes $\exists\, n'\geq n.\ \wr tid \wr_{t\ n'}$

and $\exists\, n'{<}n.\ \wr tid \wr_{t\ n'}$
shows $\langle tid \rightarrow t\rangle_n > \langle tid \leftarrow t\rangle_n$
by (*meson assms latestAct-prop*(*2*) *less-le-trans nxtActI zero-le*)

lemma *latestActless*:
assumes $\exists\, n'{\geq}n_s.\ n'{<}n \wedge \wr nid \wr_{t\ n'}$
shows $\langle nid \leftarrow t\rangle_n{\geq}n_s$
by (*meson assms dual-order.trans latestAct-less*)

lemma *latestActEq*:
fixes *nid*::*'id*
assumes $\wr nid \wr_{t\ n'}$ **and** $\neg(\exists\, n''{>}n'.\ n''{<}n \wedge \wr nid \wr_{t\ n'})$ **and** $n'{<}n$
shows $\langle nid \leftarrow t\rangle_n = n'$
using *latestAct-def*
proof
have (*GREATEST n'. latestAct-cond nid t n n'*) = *n'*
proof (*rule Greatest-equality*[*of latestAct-cond nid t n n'*])
from *assms*(*1*) *assms* (*3*) **show** *latestAct-cond nid t n n'* **by** *simp*
next
fix *y* **assume** *latestAct-cond nid t n y*
hence $\wr nid \wr_{t\ y}$ **and** $y{<}n$ **by** *auto*
thus $y \leq n'$ **using** *assms*(*1*) *assms* (*2*) *leI* **by** *blast*
qed
thus *n'* = (*GREATEST n'. latestAct-cond nid t n n'*) **by** *simp*
qed

D.14 Last Activation

In the following we introduce an operator to obtain the latest point in time where a certain component was activated within a certain configuration trace.

definition *lActive* :: $'id \Rightarrow (nat \Rightarrow cnf) \Rightarrow nat$ ($\langle$- $\wedge$ -$\rangle$)
where $\langle c \wedge t\rangle \equiv (GREATEST\ i.\ \wr c \wr_{t\ i})$

lemma *lActive-active*:
assumes $\wr c \wr_{t\ i}$
and $\forall\, n' > n.\ \neg\ (\wr c \wr_{t\ n'})$
shows $\wr c \wr_t\ (\langle c \wedge t\rangle)$
proof –
from *assms* **obtain** i' **where** $\wr c \wr_{t\ i'}$ **and** $(\forall\, y.\ \wr c \wr_{t\ y} \longrightarrow y \leq i')$
using *boundedGreatest*[*of* $\lambda i'.\ \wr c \wr_{t\ i'}$ *i n*] **by** *blast*
thus *?thesis* **using** *lActive-def Nat.GreatestI-nat*[*of* $\lambda i'.\ \wr c \wr_{t\ i'}$] **by** *simp*
qed

lemma *lActive-less*:
assumes $\wr c \wr_{t\ i}$
and $\forall\, n' > n.\ \neg\ (\wr c \wr_{t\ n'})$
shows $\langle c \wedge t\rangle \leq n$
proof (*rule ccontr*)

assume $\neg \langle c \wedge t\rangle \leq n$
hence $\langle c \wedge t\rangle > n$ **by** *simp*
moreover from *assms* **have** $\wr c\wr_{t\ (\langle c \wedge t\rangle)}$ **using** *lActive-active* **by** *simp*
ultimately show *False* **using** *assms* **by** *simp*
qed

lemma *lActive-greatest*:
assumes $\wr c\wr_{t\ i}$
and $\forall\, n' > n.\ \neg\ (\wr c\wr_{t\ n'})$
shows $i \leq \langle c \wedge t\rangle$
proof –
from *assms* **obtain** i' **where** $\wr c\wr_{t\ i'}$ **and** $(\forall\, y.\ \wr c\wr_{t\ y} \longrightarrow y \leq i')$
using *boundedGreatest*[*of* $\lambda i'.\ \wr c\wr_{t\ i'}\ i\ n$] **by** *blast*
with *assms* **show** *?thesis* **using** *lActive-def Nat.Greatest-le-nat*[*of* $\lambda i'.\ \wr c\wr_{t\ i'}\ i$] **by** *simp*
qed

lemma *lActive-greater-active*:
assumes $n > \langle c \wedge t\rangle$
and $\forall\, n'' > n'.\ \neg\ \wr c\wr_{t\ n''}$
shows $\neg\ \wr c\wr_{t\ n}$
proof (*rule ccontr*)
assume $\neg\ \neg\ \wr c\wr_{t\ n}$
with ‹$\forall\, n'' > n'.\ \neg\ \wr c\wr_{t\ n''}$› **have** $n \leq \langle c \wedge t\rangle$ **using** *lActive-greatest* **by** *simp*
thus *False* **using** *assms* **by** *simp*
qed

lemma *lActive-greater-active-all*:
assumes $\forall\, n'' > n'.\ \neg\ \wr c\wr_{t\ n''}$
shows $\neg(\exists\, n > \langle c \wedge t\rangle.\ \wr c\wr_{t\ n})$
proof (*rule ccontr*)
assume $\neg\neg(\exists\, n > \langle c \wedge t\rangle.\ \wr c\wr_{t\ n})$
then obtain n **where** $n{>}\langle c \wedge t\rangle$ **and** $\wr c\wr_{t\ n}$ **by** *blast*
with ‹$\forall\, n'' > n'.\ \neg\ (\wr c\wr_{t\ n''})$› **have** $\neg\ \wr c\wr_{t\ n}$ **using** *lActive-greater-active* **by** *simp*
with ‹$\wr c\wr_{t\ n}$› **show** *False* **by** *simp*
qed

lemma *lActive-equality*:
assumes $\wr c\wr_{t\ i}$
and $(\bigwedge x.\ \wr c\wr_{t\ x} \Longrightarrow x \leq i)$
shows $\langle c \wedge t\rangle = i$ **unfolding** *lActive-def*
using *assms Greatest-equality*[*of* $\lambda i'.\ \wr c\wr_{t\ i'}$] **by** *simp*

lemma *nxtActive-lactive*:
assumes $\exists\, i{\geq}n.\ \wr c\wr_{t\ i}$
and $\neg\ (\exists\, i{>}\langle c \rightarrow t\rangle_n.\ \wr c\wr_{t\ i})$
shows $\langle c \rightarrow t\rangle_n{=}\langle c \wedge t\rangle$
proof –
from *assms*(*1*) **have** $\wr c\wr_{t\ \langle c \rightarrow t\rangle_n}$ **using** *nxtActI* **by** *auto*
moreover from *assms* **have** $\neg\ (\exists\, i'{\geq}Suc\ \langle c \rightarrow t\rangle_n.\ \wr c\wr_{t\ i'})$

using *nxtActive-no-active* **by** *simp*
hence $(\bigwedge x.\ \wr c \wr_{t\ x} \Longrightarrow x \leq \langle c \rightarrow t\rangle_{n})$ **using** *not-less-eq-eq* **by** *auto*
ultimately show *?thesis* **using** $‹\neg\ (\exists\, i' \geq Suc\ \langle c \rightarrow t\rangle_{n}.\ \wr c \wr_{t\ i'})›$ *lActive-equality* **by** *simp*
qed

D.15 Mapping Time Points

In the following we introduce two operators to map time-points between configuration traces and behavior traces.

D.15.1 Configuration Trace to Behavior Trace

First we provide an operator which maps a point in time of a configuration trace to the corresponding point in time of a behavior trace.

definition *cnf2bhv* :: $'id \Rightarrow (nat \Rightarrow cnf) \Rightarrow nat \Rightarrow nat$ $({}_{\_}\downarrow_{\_}(\_)\ [150,150,150]\ 110)$
where ${}_{c}\downarrow_{t}(n) \equiv the\text{-}enat(llength\ (\pi_{c}(inf\text{-}llist\ t))) - 1 + (n - \langle c \wedge t\rangle)$

lemma *cnf2bhv-mono*:
assumes $n' \geq n$
shows ${}_{c}\downarrow_{t}(n') \geq {}_{c}\downarrow_{t}(n)$
by (*simp add*: *assms cnf2bhv-def diff-le-mono*)

lemma *cnf2bhv-mono-strict*:
assumes $n \geq \langle c \wedge t\rangle$ **and** $n' > n$
shows ${}_{c}\downarrow_{t}(n') > {}_{c}\downarrow_{t}(n)$
using *assms cnf2bhv-def* **by** *auto*

Note that the functions are nat, that means that also in the case the difference is negative they will return a 0!

lemma *cnf2bhv-ge-llength*[*simp*]:
assumes $n \geq \langle c \wedge t\rangle$
shows ${}_{c}\downarrow_{t}(n) \geq the\text{-}enat(llength\ (\pi_{c}(inf\text{-}llist\ t))) - 1$
using *assms cnf2bhv-def* **by** *simp*

lemma *cnf2bhv-greater-llength*[*simp*]:
assumes $n > \langle c \wedge t\rangle$
shows ${}_{c}\downarrow_{t}(n) > the\text{-}enat(llength\ (\pi_{c}(inf\text{-}llist\ t))) - 1$
using *assms cnf2bhv-def* **by** *simp*

lemma *cnf2bhv-suc*[*simp*]:
assumes $n \geq \langle c \wedge t\rangle$
shows ${}_{c}\downarrow_{t}(Suc\ n) = Suc\ ({}_{c}\downarrow_{t}(n))$
using *assms cnf2bhv-def* **by** *simp*

lemma *cnf2bhv-lActive*[*simp*]:
shows ${}_{c}\downarrow_{t}(\langle c \wedge t\rangle) = the\text{-}enat(llength\ (\pi_{c}(inf\text{-}llist\ t))) - 1$
using *cnf2bhv-def* **by** *simp*

lemma *cnf2bhv-lnth-lappend*:
assumes *act*: $\exists\, i.\ \wr c \wr_{t\ i}$
and *nAct*: $\nexists\, i.\ i \geq n \wedge \wr c \wr_{t\ i}$
shows $\textit{lnth}\ ((\pi_c(\textit{inf-llist}\ t))\ @_l\ (\textit{inf-llist}\ t'))\ ({}_c{\downarrow}_t(n)) = \textit{lnth}\ (\textit{inf-llist}\ t')\ (n - \langle c \wedge t\rangle - \mathit{1})$
(**is** $\textit{?lhs} = \textit{?rhs}$)
proof $-$
from *nAct* **have** $\textit{lfinite}\ (\pi_c(\textit{inf-llist}\ t))$ **using** *proj-finite2* **by** *auto*
then obtain k **where** *k-def*: $\textit{llength}\ (\pi_c(\textit{inf-llist}\ t)) = \textit{enat}\ k$
using *lfinite-llength-enat* **by** *blast*
moreover have $k \leq {}_c{\downarrow}_t(n)$
proof $-$
from *nAct* **have** $\nexists\, i.\ i > n - \mathit{1} \wedge \wr c \wr_{t\ i}$ **by** *simp*
with *act* **have** $\langle c \wedge t\rangle \leq n - \mathit{1}$ **using** *lActive-less* **by** *auto*
moreover have $n > \mathit{0}$ **using** *act nAct* **by** *auto*
ultimately have $\langle c \wedge t\rangle < n$ **by** *simp*
hence $\textit{the-enat}\ (\textit{llength}\ (\pi_c\textit{inf-llist}\ t)) - \mathit{1} < {}_c{\downarrow}_t(n)$ **using** *cnf2bhv-greater-llength* **by** *simp*
with *k-def* **show** *?thesis* **by** *simp*
qed
ultimately have $\textit{?lhs} = \textit{lnth}\ (\textit{inf-llist}\ t')\ ({}_c{\downarrow}_t(n) - k)$ **using** *lnth-lappend2* **by** *blast*
moreover have ${}_c{\downarrow}_t(n) - k = n - \langle c \wedge t\rangle - \mathit{1}$
proof $-$
from *cnf2bhv-def* **have**
${}_c{\downarrow}_t(n) - k = \textit{the-enat}\ (\textit{llength}\ (\pi_c\textit{inf-llist}\ t)) - \mathit{1} + (n - \langle c \wedge t\rangle) - k$ **by** *simp*
also have $\ldots = \textit{the-enat}\ (\textit{llength}\ (\pi_c\textit{inf-llist}\ t)) - \mathit{1} + (n - \langle c \wedge t\rangle) -$
$\textit{the-enat}\ (\textit{llength}\ (\pi_c(\textit{inf-llist}\ t)))$ **using** *k-def* **by** *simp*
also have $\ldots = \textit{the-enat}\ (\textit{llength}\ (\pi_c\textit{inf-llist}\ t)) + (n - \langle c \wedge t\rangle) - \mathit{1} -$
$\textit{the-enat}\ (\textit{llength}\ (\pi_c(\textit{inf-llist}\ t)))$
proof $-$
have $\exists\, i.\ \textit{enat}\ i < \textit{llength}\ (\textit{inf-llist}\ t) \wedge \wr c \wr_{\textit{lnth}\ (\textit{inf-llist}\ t)\ i}$ **by** (*simp add*: *act*)
hence $\textit{llength}\ (\pi_c\textit{inf-llist}\ t) \geq \mathit{1}$ **using** *proj-one* **by** *simp*
moreover from *k-def* **have** $\textit{llength}\ (\pi_c\textit{inf-llist}\ t) \neq \infty$ **by** *simp*
ultimately have $\textit{the-enat}\ (\textit{llength}\ (\pi_c\textit{inf-llist}\ t)) \geq \mathit{1}$ **by** (*simp add*: *k-def one-enat-def*)
thus *?thesis* **by** *simp*
qed
also have $\ldots = \textit{the-enat}\ (\textit{llength}\ (\pi_c\textit{inf-llist}\ t)) + (n - \langle c \wedge t\rangle) -$
$\textit{the-enat}\ (\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - \mathit{1}$ **by** *simp*
also have $\ldots = n - \langle c \wedge t\rangle - \mathit{1}$ **by** *simp*
finally show *?thesis* **.**
qed
ultimately show *?thesis* **by** *simp*
qed

lemma *nAct-cnf2proj-Suc-dist*:
assumes $\exists\, i \geq n.\ \wr c \wr_{t\ i}$
and $\neg(\exists\, i > \langle c \rightarrow t\rangle_n.\ \wr c \wr_{t\ i})$
shows $\textit{Suc}\ (\textit{the-enat}\ \langle c\ \#_{\textit{enat}\ n}\textit{inf-llist}\ t\rangle) = {}_c{\downarrow}_t(\textit{Suc}\ \langle c \rightarrow t\rangle_n)$
proof $-$
have $\textit{the-enat}\ \langle c\ \#_{\textit{enat}\ n}\textit{inf-llist}\ t\rangle = {}_c{\downarrow}_t(\langle c \rightarrow t\rangle_n)$ (**is** $\textit{?LHS} = \textit{?RHS}$)

proof –
from *assms* **have** $?RHS = the\text{-}enat(llength\ (\pi_c(inf\text{-}llist\ t))) - 1$
using *nxtActive-lactive*$[of\ n\ c\ t]$ **by** *simp*
also have $llength\ (\pi_c(inf\text{-}llist\ t)) = eSuc\ (\langle c\ \#_{\langle c \rightarrow t\rangle_n}\ inf\text{-}llist\ t\rangle)$
proof –
from *assms* **have** $\neg\ (\exists\, i' \geq Suc\ (\langle c \rightarrow t\rangle_n).\ \wr c \wr_{t\ i'})$ **using** *nxtActive-no-active* **by** *simp*
hence $\langle c\ \#_{Suc\ (\langle c \rightarrow t\rangle_n)}\ inf\text{-}llist\ t\rangle = llength\ (\pi_c(inf\text{-}llist\ t))$
using *nAct-eq-proj*$[of\ Suc\ (\langle c \rightarrow t\rangle_n)\ c\ inf\text{-}llist\ t]$ **by** *simp*
moreover from *assms*(1) **have** $\wr c \wr_{t\ (\langle c \rightarrow t\rangle_n)}$ **using** *nxtActI* **by** *blast*
hence $\langle c\ \#_{Suc\ (\langle c \rightarrow t\rangle_n)}\ inf\text{-}llist\ t\rangle = eSuc\ (\langle c\ \#_{\langle c \rightarrow t\rangle_n}\ inf\text{-}llist\ t\rangle)$ **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
also have $the\text{-}enat(eSuc\ (\langle c\ \#_{\langle c \rightarrow t\rangle_n}\ inf\text{-}llist\ t\rangle)) - 1 = (\langle c\ \#_{\langle c \rightarrow t\rangle_n}\ inf\text{-}llist\ t\rangle)$
proof –
have $\langle c\ \#_{\langle c \rightarrow t\rangle_n}\ inf\text{-}llist\ t\rangle \neq \infty$ **by** *simp*
hence $the\text{-}enat(eSuc\ (\langle c\ \#_{\langle c \rightarrow t\rangle_n}\ inf\text{-}llist\ t\rangle)) = Suc(the\text{-}enat(\langle c\ \#_{\langle c \rightarrow t\rangle_n}\ inf\text{-}llist\ t\rangle))$
using *the-enat-eSuc* **by** *simp*
thus *?thesis* **by** *simp*
qed
also have $\ldots = ?LHS$
proof –
have $enat\ \langle c \rightarrow t\rangle_n - 1 < llength\ (inf\text{-}llist\ t)$ **by** $(simp\ add\colon one\text{-}enat\text{-}def)$
moreover from *assms*(1) **have** $\langle c \rightarrow t\rangle_n \geq n$ **and**
$\nexists k.\ enat\ n \leq enat\ k \wedge enat\ k < enat\ \langle c \rightarrow t\rangle_n \wedge \wr c \wr_{lnth\ (inf\text{-}llist\ t)\ k}$
using *nxtActI* **by** *auto*
ultimately have $\langle c\ \#_{enat\ \langle c \rightarrow t\rangle_n} inf\text{-}llist\ t\rangle = \langle c\ \#_{enat\ n} inf\text{-}llist\ t\rangle$
using *nAct-not-active-same*$[of\ n\ \langle c \rightarrow t\rangle_n\ inf\text{-}llist\ t\ c]$ **by** *simp*
moreover have $\langle c\ \#_{enat\ n} inf\text{-}llist\ t\rangle \neq \infty$ **by** *simp*
ultimately show *?thesis* **by** *auto*
qed
finally show *?thesis* **by** *fastforce*
qed
moreover from *assms* **have** $\langle c \rightarrow t\rangle_n = \langle c \wedge t\rangle$ **using** *nxtActive-lactive* **by** *simp*
hence $Suc\ ({}_c{\downarrow}_t(\langle c \rightarrow t\rangle_n)) = {}_c{\downarrow}_t(Suc\ \langle c \rightarrow t\rangle_n)$
using *cnf2bhv-suc*[**where** $n = \langle c \rightarrow t\rangle_n$] **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

D.15.2 Behavior Trace to Configuration Trace

Next we define an operator to map a point in time of a behavior trace back to a corresponding point in time for a configuration trace.

definition *bhv2cnf* $::\ 'id \Rightarrow (nat \Rightarrow cnf) \Rightarrow nat \Rightarrow nat\ ({}_\text{-}{\uparrow}_\text{-}(\text{-})\ [150,150,150]\ 110)$
where ${}_c{\uparrow}_t(n) \equiv \langle c \wedge t\rangle + (n - (the\text{-}enat(llength\ (\pi_c(inf\text{-}llist\ t))) - 1))$

lemma *bhv2cnf-mono*:
assumes $n' \geq n$
shows ${}_c{\uparrow}_t(n') \geq {}_c{\uparrow}_t(n)$

by (*simp add*: *assms bhv2cnf-def diff-le-mono*)

lemma *bhv2cnf-mono-strict*:
assumes $n' > n$
and $n \geq$ *the-enat* (*llength* (π_c(*inf-llist t*))) $-$ *1*
shows ${}_{c}{\uparrow}_{t}(n') > {}_{c}{\uparrow}_{t}(n)$
using *assms bhv2cnf-def* **by** *auto*

Note that the functions are nat, that means that also in the case the difference is negative they will return a 0!

lemma *bhv2cnf-ge-lActive*[*simp*]:
shows ${}_{c}{\uparrow}_{t}(n) \geq \langle c \wedge t\rangle$
using *bhv2cnf-def* **by** *simp*

lemma *bhv2cnf-greater-lActive*[*simp*]:
assumes $n >$ *the-enat*(*llength* (π_c(*inf-llist t*))) $-$ *1*
shows ${}_{c}{\uparrow}_{t}(n) > \langle c \wedge t\rangle$
using *assms bhv2cnf-def* **by** *simp*

lemma *bhv2cnf-lActive*[*simp*]:
assumes $\exists\, i.\ \wr c \wr_{t\ i}$
and *lfinite* (π_c(*inf-llist t*))
shows ${}_{c}{\uparrow}_{t}$(*the-enat*(*llength* (π_c(*inf-llist t*)))) = *Suc* ($\langle c \wedge t\rangle$)
proof $-$
from *assms* **have** π_c(*inf-llist t*)$\neq [\,]_l$ **by** *simp*
hence *llength* (π_c(*inf-llist t*)) $>$ *0* **by** (*simp add*: *lnull-def*)
moreover from ‹*lfinite* (π_c(*inf-llist t*))› **have** *llength* (π_c(*inf-llist t*)) $\neq \infty$
using *llength-eq-infty-conv-lfinite* **by** *auto*
ultimately have *the-enat*(*llength* (π_c(*inf-llist t*))) $>$ *0* **using** *enat-0-iff*(*1*) **by** *fastforce*
hence *the-enat*(*llength* (π_c(*inf-llist t*))) $-$ (*the-enat*(*llength* (π_c(*inf-llist t*))) $-$ *1*) = *1*
by *simp*
thus *?thesis* **using** *bhv2cnf-def* **by** *simp*
qed

D.15.3 Relating the Mappings

In the following we provide some properties about the relationship between the two mapping operators.

lemma *bhv2cnf-cnf2bhv*:
assumes $n \geq \langle c \wedge t\rangle$
shows ${}_{c}{\uparrow}_{t}({}_{c}{\downarrow}_{t}(n)) = n$ (**is** *?lhs* = *?rhs*)
proof $-$
have *?lhs* = $\langle c \wedge t\rangle$ + ((${}_{c}{\downarrow}_{t}(n)$) $-$ (*the-enat*(*llength* (π_c(*inf-llist t*))) $-$ *1*))
using *bhv2cnf-def* **by** *simp*
also have ... = $\langle c \wedge t\rangle$ + (((*the-enat* (*llength* (π_c(*inf-llist t*)))) $-$ *1* + ($n - \langle c \wedge t\rangle$)) $-$
(*the-enat* (*llength* (π_c(*inf-llist t*))) $-$ *1*)) **using** *cnf2bhv-def* **by** *simp*
also have (*the-enat*(*llength* (π_c(*inf-llist t*)))) $-$ *1* + (n $-$ ($\langle c \wedge t\rangle$)) $-$
(*the-enat* (*llength* (π_c(*inf-llist t*))) $-$ *1*) = (*the-enat*(*llength* (π_c(*inf-llist t*)))) $-$ *1* $-$
((*the-enat* (*llength* (π_c(*inf-llist t*)))) $-$ *1*) + (n $-$ ($\langle c \wedge t\rangle$)) **by** *simp*

also have $\ldots = n - (\langle c \wedge t\rangle)$ **by** *simp*
also have $(\langle c \wedge t\rangle) + (n - (\langle c \wedge t\rangle)) = (\langle c \wedge t\rangle) + n - \langle c \wedge t\rangle$ **using** *assms* **by** *simp*
also have $\ldots =$ *?rhs* **by** *simp*
finally show *?thesis* **.**
qed

lemma *cnf2bhv-bhv2cnf*:
assumes $n \geq \textit{the-enat}\ (\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1$
shows ${}_{c}{\downarrow}_{t}({}_{c}{\uparrow}_{t}(n)) = n$ (**is** *?lhs = ?rhs*)
proof −
have *?lhs* $= \textit{the-enat}(\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1 + (({}_{c}{\uparrow}_{t}(n)) - (\langle c \wedge t\rangle))$
using *cnf2bhv-def* **by** *simp*
also have $\ldots = \textit{the-enat}(\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1 + (\langle c \wedge t\rangle +$
$(n - (\textit{the-enat}(\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1)) - (\langle c \wedge t\rangle))$ **using** *bhv2cnf-def* **by** *simp*
also have $\langle c \wedge t\rangle + (n - (\textit{the-enat}(\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1)) - (\langle c \wedge t\rangle) =$
$\langle c \wedge t\rangle - (\langle c \wedge t\rangle) + (n - (\textit{the-enat}(\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1))$ **by** *simp*
also have $\ldots = n - (\textit{the-enat}(\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1)$ **by** *simp*
also have
$\textit{the-enat}\ (\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1 + (n - (\textit{the-enat}\ (\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1)) =$
$n - (\textit{the-enat}\ (\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1) + (\textit{the-enat}\ (\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1)$
by *simp*
also have $\ldots = n + ((\textit{the-enat}\ (\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1) -$
$(\textit{the-enat}\ (\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1))$ **using** *assms* **by** *simp*
also have $\ldots =$ *?rhs* **by** *simp*
finally show *?thesis* **.**
qed

lemma *p2c-mono-c2p*:
assumes $n \geq \langle c \wedge t\rangle$
and $n' \geq {}_{c}{\downarrow}_{t}(n)$
shows ${}_{c}{\uparrow}_{t}(n') \geq n$
proof −
from ‹$n' \geq {}_{c}{\downarrow}_{t}(n)$› **have** ${}_{c}{\uparrow}_{t}(n') \geq {}_{c}{\uparrow}_{t}({}_{c}{\downarrow}_{t}(n))$ **using** *bhv2cnf-mono* **by** *simp*
thus *?thesis* **using** *bhv2cnf-cnf2bhv* ‹$n \geq \langle c \wedge t\rangle$› **by** *simp*
qed

lemma *p2c-mono-c2p-strict*:
assumes $n \geq \langle c \wedge t\rangle$
and $n < {}_{c}{\uparrow}_{t}(n')$
shows ${}_{c}{\downarrow}_{t}(n) < n'$
proof (*rule ccontr*)
assume $\neg\ ({}_{c}{\downarrow}_{t}(n) < n')$
hence ${}_{c}{\downarrow}_{t}(n) \geq n'$ **by** *simp*
with ‹$n \geq \langle c \wedge t\rangle$› **have** ${}_{c}{\uparrow}_{t}(\textit{nat}\ ({}_{c}{\downarrow}_{t}(n))) \geq {}_{c}{\uparrow}_{t}(n')$
using *bhv2cnf-mono* **by** *simp*
hence $\neg({}_{c}{\uparrow}_{t}(\textit{nat}\ ({}_{c}{\downarrow}_{t}(n))) < {}_{c}{\uparrow}_{t}(n'))$ **by** *simp*
with ‹$n \geq \langle c \wedge t\rangle$› **have** $\neg(n < {}_{c}{\uparrow}_{t}(n'))$
using *bhv2cnf-cnf2bhv* **by** *simp*
with *assms* **show** *False* **by** *simp*

```
qed

lemma c2p-mono-p2c:
  assumes n ≥ the-enat (llength (πc(inf-llist t))) − 1
      and n′ ≥ c↑t(n)
    shows c↓t(n′) ≥ n
proof −
  from ‹n′ ≥ c↑t(n)› have c↓t(n′) ≥ c↓t(c↑t(n)) using cnf2bhv-mono by simp
  thus ?thesis using cnf2bhv-bhv2cnf ‹n ≥ the-enat (llength (πc(inf-llist t))) − 1› by simp
qed

lemma c2p-mono-p2c-strict:
  assumes n ≥ the-enat (llength (πc(inf-llist t))) − 1
      and n<c↓t(n′)
  shows c↑t(n) < n′
proof (rule ccontr)
  assume ¬ (c↑t(n) < n′)
  hence c↑t(n) ≥ n′ by simp
  with ‹n ≥ the-enat (llength (πc(inf-llist t))) − 1› have c↓t(nat (c↑t(n))) ≥ c↓t(n′)
    using cnf2bhv-mono by simp
  hence ¬(c↓t(nat (c↑t(n))) < c↓t(n′)) by simp
  with ‹n ≥ the-enat (llength (πc(inf-llist t))) − 1› have  ¬(n < c↓t(n′))
    using cnf2bhv-bhv2cnf by simp
  with assms show False by simp
qed

end

end
```

The following theory formalizes our calculus for dynamic architectures [Mar17b, Mar17c] and verifies its soundness. The calculus allows to reason about temporal-logic specifications of component behavior in a dynamic setting. The theory is based on our theory of configuration traces and introduces the notion of behavior trace assertion to specify component behavior in a dynamic setting.

theory *Dynamic-Architecture-Calculus*
  imports *Configuration-Traces*
begin

D.16 Extended Natural Numbers

We first provide one additional property for extended natural numbers.

lemma *the-enat-mono*[*simp*]:
  assumes $m \neq \infty$
    and $n \leq m$
  shows $\mathit{the\text{-}enat}\ n \leq \mathit{the\text{-}enat}\ m$
  using *assms*(*1*) *assms*(*2*) *enat-ile* **by** *fastforce*

D.17 Lazy Lists

Finally, we provide an additional property for lazy lists.

lemma *llength-geq-enat-lfiniteD*: $\mathit{llength}\ xs \leq \mathit{enat}\ n \Longrightarrow \mathit{lfinite}\ xs$
  using *not-lfinite-llength* **by** *force*

context *dynamic-component*
begin

D.18 Dynamic Evaluation of Temporal Operators

In the following we introduce a function to evaluate a behavior trace assertion over a given configuration trace.

definition $\mathit{eval}{::}\ 'id \Rightarrow (nat \Rightarrow cnf) \Rightarrow (nat \Rightarrow 'cmp) \Rightarrow nat$
  $\Rightarrow ((nat \Rightarrow 'cmp) \Rightarrow nat \Rightarrow bool) \Rightarrow bool$
  where $\mathit{eval}\ cid\ t\ t'\ n\ \gamma \equiv$
    $(\exists\, i{\geq}n.\ \wr cid \wr_{t\ i}) \wedge \gamma\ (\mathit{lnth}\ ((\pi_{cid}(\mathit{inf\text{-}llist}\ t))\ @_l\ (\mathit{inf\text{-}llist}\ t')))\ (\mathit{the\text{-}enat}(\langle cid\ \#_n\ \mathit{inf\text{-}llist}\ t\rangle))$
  $\vee$
    $(\exists\, i.\ \wr cid \wr_{t\ i}) \wedge (\nexists\, i'.\ i'{\geq}n \wedge \wr cid \wr_{t\ i'}) \wedge \gamma\ (\mathit{lnth}\ ((\pi_{cid}(\mathit{inf\text{-}llist}\ t))\ @_l\ (\mathit{inf\text{-}llist}\ t')))\ ({}_{cid}{\downarrow}_t(n))$
  $\vee$
    $(\nexists\, i.\ \wr cid \wr_{t\ i}) \wedge \gamma\ (\mathit{lnth}\ ((\pi_{cid}(\mathit{inf\text{-}llist}\ t))\ @_l\ (\mathit{inf\text{-}llist}\ t')))\ n$

eval takes a component identifier *cid*, a configuration trace t, a behavior trace t', and point in time n and evaluates behavior trace assertion γ as follows:

- If component *cid* is again activated in the future, γ is evaluated at the next point in time where *cid* is active in t.

- If component *cid* is not again activated in the future but it is activated at least once in t, then γ is evaluated at the point in time given by ${}_{cid}{\downarrow}_{t}n$.
- If component *cid* is never active in t, then γ is evaluated at time point n.

The following proposition evaluates definition *eval* by showing that a behavior trace assertion γ holds over configuration trace t and continuation t' whenever it holds for the concatenation of the corresponding projection with t'.

```
proposition eval-corr:
  $eval\ cid\ t\ t'\ 0\ \gamma \longleftrightarrow \gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ 0$
proof
  assume $eval\ cid\ t\ t'\ 0\ \gamma$
  with eval-def have $(\exists\, i{\geq}0.\ \wr cid\wr_{t\ i}) \wedge$
  $\gamma\ (lnth\ (\pi_{cid} inf\text{-}llist\ t\ @_l\ inf\text{-}llist\ t'))\ (the\text{-}enat\ \langle cid\ \#_{enat\ 0} inf\text{-}llist\ t\rangle) \vee$
  $(\exists\, i.\ \wr cid\wr_{t\ i}) \wedge \neg\ (\exists\, i'{\geq}0.\ \wr cid\wr_{t\ i'}) \wedge \gamma\ (lnth\ (\pi_{cid} inf\text{-}llist\ t\ @_l\ inf\text{-}llist\ t'))\ ({}_{cid}{\downarrow}_{t}0) \vee$
  $(\nexists\, i.\ \wr cid\wr_{t\ i}) \wedge \gamma\ (lnth\ (\pi_{cid} inf\text{-}llist\ t\ @_l\ inf\text{-}llist\ t'))\ 0$ by simp
  thus $\gamma\ (lnth\ (\pi_{cid} inf\text{-}llist\ t\ @_l\ inf\text{-}llist\ t'))\ 0$
  proof
    assume
      $(\exists\, i{\geq}0.\ \wr cid\wr_{t\ i}) \wedge \gamma\ (lnth\ (\pi_{cid} inf\text{-}llist\ t\ @_l\ inf\text{-}llist\ t'))\ (the\text{-}enat\ \langle cid\ \#_{enat\ 0} inf\text{-}llist\ t\rangle)$
    moreover have $the\text{-}enat\ \langle cid\ \#_{enat\ 0} inf\text{-}llist\ t\rangle = 0$ using zero-enat-def by auto
    ultimately show ?thesis by simp
  next
    assume $(\exists\, i.\ \wr cid\wr_{t\ i}) \wedge \neg\ (\exists\, i'{\geq}0.\ \wr cid\wr_{t\ i'}) \wedge \gamma\ (lnth\ (\pi_{cid} inf\text{-}llist\ t\ @_l\ inf\text{-}llist\ t'))\ ({}_{cid}{\downarrow}_{t}0)$
      $\vee\ (\nexists\, i.\ \wr cid\wr_{t\ i}) \wedge \gamma\ (lnth\ (\pi_{cid} inf\text{-}llist\ t\ @_l\ inf\text{-}llist\ t'))\ 0$
    thus ?thesis by auto
  qed
next
  assume $\gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ 0$
  show $eval\ cid\ t\ t'\ 0\ \gamma$
  proof cases
    assume $\exists\, i.\ \wr cid\wr_{t\ i}$
    hence $\exists\, i{\geq}0.\ \wr cid\wr_{t\ i}$ by simp
    moreover from ‹$\gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ 0$› have
      $\gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ (the\text{-}enat(\langle cid\ \#_{enat\ 0}\ inf\text{-}llist\ t\rangle))$
      using zero-enat-def by auto
    ultimately show ?thesis using eval-def by simp
  next
    assume $\nexists\, i.\ \wr cid\wr_{t\ i}$
    with ‹$\gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ 0$› show ?thesis using eval-def by simp
  qed
qed
```

D.18.1 Simplification Rules

```
lemma validCI-act[simp]:
  assumes $\exists\, i{\geq}n.\ \wr cid\wr_{t\ i}$
    and $\gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ (the\text{-}enat(\langle cid\ \#_{n}\ inf\text{-}llist\ t\rangle))$
  shows $eval\ cid\ t\ t'\ n\ \gamma$
```

using *assms eval-def* **by** *simp*

lemma *validCI-cont*[*simp*]:
assumes $\exists\, i.\ \wr cid \wr_{t\ i}$
and $\nexists\, i'.\ i' \geq n \wedge \wr cid \wr_{t\ i'}$
and $\gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ ({}_{cid}\downarrow_t(n))$
shows $eval\ cid\ t\ t'\ n\ \gamma$
using *assms eval-def* **by** *simp*

lemma *validCI-not-act*[*simp*]:
assumes $\nexists\, i.\ \wr cid \wr_{t\ i}$
and $\gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ n$
shows $eval\ cid\ t\ t'\ n\ \gamma$
using *assms eval-def* **by** *simp*

lemma *validCE-act*[*simp*]:
assumes $\exists\, i \geq n.\ \wr cid \wr_{t\ i}$
and $eval\ cid\ t\ t'\ n\ \gamma$
shows $\gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ (the\text{-}enat(\langle cid\ \#_n\ inf\text{-}llist\ t\rangle))$
using *assms eval-def* **by** *auto*

lemma *validCE-cont*[*simp*]:
assumes $\exists\, i.\ \wr cid \wr_{t\ i}$
and $\nexists\, i'.\ i' \geq n \wedge \wr cid \wr_{t\ i'}$
and $eval\ cid\ t\ t'\ n\ \gamma$
shows $\gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ ({}_{cid}\downarrow_t(n))$
using *assms eval-def* **by** *auto*

lemma *validCE-not-act*[*simp*]:
assumes $\nexists\, i.\ \wr cid \wr_{t\ i}$
and $eval\ cid\ t\ t'\ n\ \gamma$
shows $\gamma\ (lnth\ ((\pi_{cid}(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ n$
using *assms eval-def* **by** *auto*

D.18.2 No Activations

proposition *validity1*:
assumes $n \leq n'$
and $\exists\, i \geq n'.\ \wr c \wr_{t\ i}$
and $\forall\, k \geq n.\ k < n' \longrightarrow \neg\ \wr c \wr_{t\ k}$
shows $eval\ c\ t\ t'\ n\ \gamma \Longrightarrow eval\ c\ t\ t'\ n'\ \gamma$
proof –
assume $eval\ c\ t\ t'\ n\ \gamma$
moreover from *assms* **have** $\exists\, i \geq n.\ \wr c \wr_{t\ i}$ **by** (*meson order.trans*)
ultimately have $\gamma\ (lnth\ ((\pi_c(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ (the\text{-}enat\ (\langle c\ \#_{enat\ n}\ inf\text{-}llist\ t\rangle))$
using *validCE-act* **by** *blast*
moreover have $enat\ n' - 1 < llength\ (inf\text{-}llist\ t)$ **by** (*simp add*: *one-enat-def*)
with *assms* **have** $the\text{-}enat\ (\langle c\ \#_{enat\ n}\ inf\text{-}llist\ t\rangle) = the\text{-}enat\ (\langle c\ \#_{enat\ n'}\ inf\text{-}llist\ t\rangle)$
using *nAct-not-active-same*[*of n n' inf-llist t c*] **by** *simp*

ultimately have γ (*lnth* ((π_c(*inf-llist t*)) $@_l$ (*inf-llist t'*))) (*the-enat* ($\langle c \#_{enat\ n'}$ *inf-llist t*$\rangle$))
  by *simp*
with *assms* **show** *?thesis* **using** *validCI-act* **by** *blast*
qed

proposition *validity2*:
  assumes $n \leq n'$
    and $\exists\, i \geq n'.\ \wr c \wr_{t\ i}$
    and $\forall\, k \geq n.\ k < n' \longrightarrow \neg\ \wr c \wr_{t\ k}$
  shows $eval\ c\ t\ t'\ n'\ \gamma \Longrightarrow eval\ c\ t\ t'\ n\ \gamma$
proof –
  assume $eval\ c\ t\ t'\ n'\ \gamma$
  with ‹$\exists\, i \geq n'.\ \wr c \wr_{t\ i}$›
    have γ (*lnth* ((π_c(*inf-llist t*)) $@_l$ (*inf-llist t'*))) (*the-enat* ($\langle c \#_{enat\ n'}$ *inf-llist t*$\rangle$))
    using *validCE-act* **by** *blast*
  moreover have *enat* $n' - 1 <$ *llength* (*inf-llist t*) **by** (*simp add*: *one-enat-def*)
  with *assms* **have** *the-enat* ($\langle c \#_{enat\ n}$ *inf-llist t*$\rangle$) = *the-enat* ($\langle c \#_{enat\ n'}$ *inf-llist t*$\rangle$)
    using *nAct-not-active-same* **by** *simp*
  ultimately have γ (*lnth* ((π_c(*inf-llist t*)) $@_l$ (*inf-llist t'*))) (*the-enat* ($\langle c \#_{enat\ n}$ *inf-llist t*$\rangle$))
    by *simp*
  moreover from *assms* **have** $\exists\, i \geq n.\ \wr c \wr_{t\ i}$ **by** (*meson order.trans*)
  ultimately show *?thesis* **using** *validCI-act* **by** *blast*
qed

D.19 Basic Operators

In the following we introduce some basic operators for behavior trace assertions.

D.19.1 Predicates

Every predicate can be transformed to a behavior trace assertion.

definition *pred* :: *bool* $\Rightarrow$ ((*nat* $\Rightarrow$ *'cmp*) $\Rightarrow$ *nat* $\Rightarrow$ *bool*)
  where *pred* $P \equiv \lambda\ t\ n.\ P$

lemma *predI*[*intro*]:
  fixes $cid\ t\ t'\ n\ P$
  assumes P
  shows *eval cid t t' n* (*pred P*)
proof *cases*
  assume ($\exists\, i.\ \wr cid \wr_{t\ i}$)
  show *?thesis*
  proof *cases*
    assume $\exists\, i \geq n.\ \wr cid \wr_{t\ i}$
    with *assms* **show** *?thesis* **using** *eval-def pred-def* **by** *auto*
  next
    assume $\neg$ ($\exists\, i \geq n.\ \wr cid \wr_{t\ i}$)
    with *assms* **show** *?thesis* **using** *eval-def pred-def* **by** *auto*
  qed

```
next
  assume ¬(∃ i. ≀cid≀_t i)
  with assms show ?thesis using eval-def pred-def by auto
qed

lemma predE[elim]:
  fixes cid t t' n P
  assumes eval cid t t' n (pred P)
  shows P
proof cases
  assume (∃ i. ≀cid≀_t i)
  show ?thesis
  proof cases
    assume ∃ i≥n. ≀cid≀_t i
    with assms show ?thesis using eval-def pred-def by auto
  next
    assume ¬ (∃ i≥n. ≀cid≀_t i)
    with assms show ?thesis using eval-def pred-def by auto
  qed
next
  assume ¬(∃ i. ≀cid≀_t i)
  with assms show ?thesis using eval-def pred-def by auto
qed
```

D.19.2 True and False

```
definition true :: (nat ⇒ 'cmp) ⇒ nat ⇒ bool
  where true ≡ λt n. HOL.True

definition false :: (nat ⇒ 'cmp) ⇒ nat ⇒ bool
  where false ≡ λt n. HOL.False
```

D.19.3 Implication

```
definition imp :: ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool)
  ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) (infixl ⟶^b 10)
  where γ ⟶^b γ' ≡ λ t n. γ t n ⟶ γ' t n

lemma impI[intro!]:
  assumes eval cid t t' n γ ⟶ eval cid t t' n γ'
  shows eval cid t t' n (γ ⟶^b γ')
proof cases
  assume ∃ i. ≀cid≀_t i
  show ?thesis
  proof cases
    assume ∃ i≥n. ≀cid≀_t i
    with ‹eval cid t t' n γ ⟶ eval cid t t' n γ'›
      have γ (lnth (π_cid inf-llist t @_l inf-llist t')) (the-enat ⟨cid #_enat n inf-llist t⟩)
      ⟶ γ' (lnth (π_cid inf-llist t @_l inf-llist t')) (the-enat ⟨cid #_enat n inf-llist t⟩)
      using eval-def by blast
```

```
    with ‹∃ i≥n. ≀cid≀_t i› have eval cid t t′ n (λt n. γ t n ⟶ γ′ t n)
      using validCI-act[where γ=λ t n. γ t n ⟶ γ′ t n] by blast
    thus ?thesis using imp-def by simp
  next
    assume ¬ (∃ i≥n. ≀cid≀_t i)
    with ‹∃ i. ≀cid≀_t i› ‹eval cid t t′ n γ ⟶ eval cid t t′ n γ′›
      have γ (lnth (π_cid inf-llist t @_l inf-llist t′)) (_cid↓_t n)
      ⟶ γ′ (lnth (π_cid inf-llist t @_l inf-llist t′)) (_cid↓_t n) using eval-def by blast
    with ‹∃ i. ≀cid≀_t i› ‹¬ (∃ i≥n. ≀cid≀_t i)› have eval cid t t′ n (λt n. γ t n ⟶ γ′ t n)
      using validCI-cont[where γ=λ t n. γ t n ⟶ γ′ t n] by blast
    thus ?thesis using imp-def by simp
  qed
next
  assume ¬(∃ i. ≀cid≀_t i)
  with ‹eval cid t t′ n γ ⟶ eval cid t t′ n γ′›
    have γ (lnth (π_cid inf-llist t @_l inf-llist t′)) n ⟶ γ′ (lnth (π_cid inf-llist t @_l inf-llist t′)) n
    using eval-def by blast
  with ‹¬(∃ i. ≀cid≀_t i)› have eval cid t t′ n (λt n. γ t n ⟶ γ′ t n)
    using validCI-not-act[where γ=λ t n. γ t n ⟶ γ′ t n] by blast
  thus ?thesis using imp-def by simp
qed

lemma impE[elim!]:
  assumes eval cid t t′ n (γ ⟶^b γ′)
  shows eval cid t t′ n γ ⟶ eval cid t t′ n γ′
proof cases
  assume (∃ i. ≀cid≀_t i)
  show ?thesis
  proof cases
    assume ∃ i≥n. ≀cid≀_t i
    moreover from ‹eval cid t t′ n (γ ⟶^b γ′)› have eval cid t t′ n (λt n. γ t n ⟶ γ′ t n)
      using imp-def by simp
    ultimately have γ (lnth (π_cid inf-llist t @_l inf-llist t′)) (the-enat ⟨cid #_enat n inf-llist t⟩)
      ⟶ γ′ (lnth (π_cid inf-llist t @_l inf-llist t′)) (the-enat ⟨cid #_enat n inf-llist t⟩)
      using validCE-act[where γ=λ t n. γ t n ⟶ γ′ t n] by blast
    with ‹∃ i≥n. ≀cid≀_t i› show ?thesis using eval-def by blast
  next
    assume ¬ (∃ i≥n. ≀cid≀_t i)
    moreover from ‹eval cid t t′ n (γ ⟶^b γ′)› have eval cid t t′ n (λt n. γ t n ⟶ γ′ t n)
      using imp-def by simp
    ultimately have γ (lnth (π_cid inf-llist t @_l inf-llist t′)) (_cid↓_t n)
      ⟶ γ′ (lnth (π_cid inf-llist t @_l inf-llist t′)) (_cid↓_t n)
      using validCE-cont[where γ=λ t n. γ t n ⟶ γ′ t n] ‹∃ i. ≀cid≀_t i› by blast
    with ‹¬ (∃ i≥n. ≀cid≀_t i)› ‹∃ i. ≀cid≀_t i› show ?thesis using eval-def by blast
  qed
next
  assume ¬(∃ i. ≀cid≀_t i)
  moreover from ‹eval cid t t′ n (γ ⟶^b γ′)› have eval cid t t′ n (λt n. γ t n ⟶ γ′ t n)
    using imp-def by simp
```

```
  ultimately have γ (lnth (π_cid inf-llist t @_l inf-llist t')) n
    ⟶ γ' (lnth (π_cid inf-llist t @_l inf-llist t')) n
    using validCE-not-act[where γ=λ t n. γ t n ⟶ γ' t n] by blast
  with ‹¬(∃ i. ⦚cid⦚_t i)› show ?thesis using eval-def by blast
qed
```

D.19.4 Disjunction

```
definition disj :: ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool)
  ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) (infixl ∨^b 15)
  where γ ∨^b γ' ≡ λ t n. γ t n ∨ γ' t n

lemma disjI[intro!]:
  assumes eval cid t t' n γ ∨ eval cid t t' n γ'
  shows eval cid t t' n (γ ∨^b γ')
proof cases
  assume ∃ i. ⦚cid⦚_t i
  show ?thesis
  proof cases
    assume ∃ i≥n. ⦚cid⦚_t i
    with ‹eval cid t t' n γ ∨ eval cid t t' n γ'›
      have γ (lnth (π_cid inf-llist t @_l inf-llist t')) (the-enat ⟨cid #_enat n inf-llist t⟩)
      ∨ γ' (lnth (π_cid inf-llist t @_l inf-llist t')) (the-enat ⟨cid #_enat n inf-llist t⟩)
      using eval-def by blast
    with ‹∃ i≥n. ⦚cid⦚_t i› have eval cid t t' n (λt n. γ t n ∨ γ' t n)
      using validCI-act[where γ=λ t n. γ t n ∨ γ' t n] by blast
    thus ?thesis using disj-def by simp
  next
    assume ¬ (∃ i≥n. ⦚cid⦚_t i)
    with ‹∃ i. ⦚cid⦚_t i› ‹eval cid t t' n γ ∨ eval cid t t' n γ'›
      have γ (lnth (π_cid inf-llist t @_l inf-llist t')) (_cid↓_t n)
      ∨ γ' (lnth (π_cid inf-llist t @_l inf-llist t')) (_cid↓_t n) using eval-def by blast
    with ‹∃ i. ⦚cid⦚_t i› ‹¬ (∃ i≥n. ⦚cid⦚_t i)› have eval cid t t' n (λt n. γ t n ∨ γ' t n)
      using validCI-cont[where γ=λ t n. γ t n ∨ γ' t n] by blast
    thus ?thesis using disj-def by simp
  qed
next
  assume ¬(∃ i. ⦚cid⦚_t i)
  with ‹eval cid t t' n γ ∨ eval cid t t' n γ'›
    have γ (lnth (π_cid inf-llist t @_l inf-llist t')) n ∨ γ' (lnth (π_cid inf-llist t @_l inf-llist t')) n
    using eval-def by blast
  with ‹¬(∃ i. ⦚cid⦚_t i)› have eval cid t t' n (λt n. γ t n ∨ γ' t n)
    using validCI-not-act[where γ=λ t n. γ t n ∨ γ' t n] by blast
  thus ?thesis using disj-def by simp
qed

lemma disjE[elim!]:
  assumes eval cid t t' n (γ ∨^b γ')
  shows eval cid t t' n γ ∨ eval cid t t' n γ'
```

```
proof cases
  assume (∃ i. ⌇cid⌇_t i)
  show ?thesis
  proof cases
    assume ∃ i≥n. ⌇cid⌇_t i
    moreover from ‹eval cid t t′ n (γ ∨^b γ′)› have eval cid t t′ n (λt n. γ t n ∨ γ′ t n)
      using disj-def by simp
    ultimately have γ (lnth (π_cid inf-llist t @_l inf-llist t′)) (the-enat ⟨cid #_enat n inf-llist t⟩)
      ∨ γ′ (lnth (π_cid inf-llist t @_l inf-llist t′)) (the-enat ⟨cid #_enat n inf-llist t⟩)
      using validCE-act[where γ=λ t n. γ t n ∨ γ′ t n] by blast
    with ‹∃ i≥n. ⌇cid⌇_t i› show ?thesis
      using validCI-act[of n cid t γ t′] validCI-act[of n cid t γ′ t′] by blast
  next
    assume ¬ (∃ i≥n. ⌇cid⌇_t i)
    moreover from ‹eval cid t t′ n (γ ∨^b γ′)› have eval cid t t′ n (λt n. γ t n ∨ γ′ t n)
      using disj-def by simp
    ultimately have γ (lnth (π_cid inf-llist t @_l inf-llist t′)) (_cid↓_t n)
      ∨ γ′ (lnth (π_cid inf-llist t @_l inf-llist t′)) (_cid↓_t n)
      using validCE-cont[where γ=λ t n. γ t n ∨ γ′ t n] ‹∃ i. ⌇cid⌇_t i› by blast
    with ‹¬ (∃ i≥n. ⌇cid⌇_t i)› ‹∃ i. ⌇cid⌇_t i› show ?thesis
      using validCI-cont[of cid t n γ t′] validCI-cont[of cid t n γ′ t′] by blast
  qed
next
  assume ¬(∃ i. ⌇cid⌇_t i)
  moreover from ‹eval cid t t′ n (γ ∨^b γ′)› have eval cid t t′ n (λt n. γ t n ∨ γ′ t n)
    using disj-def by simp
  ultimately have γ (lnth (π_cid inf-llist t @_l inf-llist t′)) n
    ∨ γ′ (lnth (π_cid inf-llist t @_l inf-llist t′)) n
    using validCE-not-act[where γ=λ t n. γ t n ∨ γ′ t n] by blast
  with ‹¬(∃ i. ⌇cid⌇_t i)› show ?thesis
    using validCI-not-act[of cid t γ t′ n] validCI-not-act[of cid t γ′ t′ n] by blast
qed
```

D.19.5 Conjunction

```
definition conj :: ((nat ⇒ ′cmp) ⇒ nat ⇒ bool) ⇒ ((nat ⇒ ′cmp) ⇒ nat ⇒ bool)
  ⇒ ((nat ⇒ ′cmp) ⇒ nat ⇒ bool) (infixl ∧^b 20)
  where γ ∧^b γ′ ≡ λ t n. γ t n ∧ γ′ t n

lemma conjI[intro!]:
  assumes eval cid t t′ n γ ∧ eval cid t t′ n γ′
  shows eval cid t t′ n (γ ∧^b γ′)
proof cases
  assume ∃ i. ⌇cid⌇_t i
  show ?thesis
  proof cases
    assume ∃ i≥n. ⌇cid⌇_t i
    with ‹eval cid t t′ n γ ∧ eval cid t t′ n γ′›
      have γ (lnth (π_cid inf-llist t @_l inf-llist t′)) (the-enat ⟨cid #_enat n inf-llist t⟩)
```

```
      ∧ γ' (lnth (π_cid inf-llist t @_l inf-llist t')) (the-enat ⟨cid #_enat n inf-llist t⟩))
      using eval-def by blast
    with ‹∃ i≥n. ≀cid≀_t i› have eval cid t t' n (λt n. γ t n ∧ γ' t n)
      using validCI-act[where γ=λ t n. γ t n ∧ γ' t n] by blast
    thus ?thesis using conj-def by simp
  next
    assume ¬ (∃ i≥n. ≀cid≀_t i)
    with ‹∃ i. ≀cid≀_t i› ‹eval cid t t' n γ ∧ eval cid t t' n γ'›
      have γ (lnth (π_cid inf-llist t @_l inf-llist t')) (_cid↓_t n)
      ∧ γ' (lnth (π_cid inf-llist t @_l inf-llist t')) (_cid↓_t n) using eval-def by blast
    with ‹∃ i. ≀cid≀_t i› ‹¬ (∃ i≥n. ≀cid≀_t i)› have eval cid t t' n (λt n. γ t n ∧ γ' t n)
      using validCI-cont[where γ=λ t n. γ t n ∧ γ' t n] by blast
    thus ?thesis using conj-def by simp
  qed
next
  assume ¬(∃ i. ≀cid≀_t i)
  with ‹eval cid t t' n γ ∧ eval cid t t' n γ'› have γ (lnth (π_cid inf-llist t @_l inf-llist t')) n
    ∧ γ' (lnth (π_cid inf-llist t @_l inf-llist t')) n using eval-def by blast
  with ‹¬(∃ i. ≀cid≀_t i)› have eval cid t t' n (λt n. γ t n ∧ γ' t n)
    using validCI-not-act[where γ=λ t n. γ t n ∧ γ' t n] by blast
  thus ?thesis using conj-def by simp
qed

lemma conjE[elim!]:
  assumes eval cid t t' n (γ ∧^b γ')
  shows eval cid t t' n γ ∧ eval cid t t' n γ'
proof cases
  assume (∃ i. ≀cid≀_t i)
  show ?thesis
  proof cases
    assume ∃ i≥n. ≀cid≀_t i
    moreover from ‹eval cid t t' n (γ ∧^b γ')› have eval cid t t' n (λt n. γ t n ∧ γ' t n)
      using conj-def by simp
    ultimately have γ (lnth (π_cid inf-llist t @_l inf-llist t')) (the-enat ⟨cid #_enat n inf-llist t⟩))
      ∧ γ' (lnth (π_cid inf-llist t @_l inf-llist t')) (the-enat ⟨cid #_enat n inf-llist t⟩))
      using validCE-act[where γ=λ t n. γ t n ∧ γ' t n] by blast
    with ‹∃ i≥n. ≀cid≀_t i› show ?thesis using eval-def by blast
  next
    assume ¬ (∃ i≥n. ≀cid≀_t i)
    moreover from ‹eval cid t t' n (γ ∧^b γ')› have eval cid t t' n (λt n. γ t n ∧ γ' t n)
      using conj-def by simp
    ultimately have γ (lnth (π_cid inf-llist t @_l inf-llist t')) (_cid↓_t n)
      ∧ γ' (lnth (π_cid inf-llist t @_l inf-llist t')) (_cid↓_t n)
      using validCE-cont[where γ=λ t n. γ t n ∧ γ' t n] ‹∃ i. ≀cid≀_t i› by blast
    with ‹¬ (∃ i≥n. ≀cid≀_t i)› ‹∃ i. ≀cid≀_t i› show ?thesis using eval-def by blast
  qed
next
  assume ¬(∃ i. ≀cid≀_t i)
  moreover from ‹eval cid t t' n (γ ∧^b γ')› have eval cid t t' n (λt n. γ t n ∧ γ' t n)
```

```
    using conj-def by simp
  ultimately have γ (lnth (π_cid inf-llist t @_l inf-llist t')) n ∧ γ' (lnth (π_cid inf-llist t @_l inf-llist
t')) n
    using validCE-not-act[where γ=λ t n. γ t n ∧ γ' t n] by blast
  with ‹¬(∃ i. ⸾cid⸾_t i)› show ?thesis using eval-def by blast
qed
```

D.19.6 Negation

```
definition not ::
  ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) (¬^b - [19] 19)
  where ¬^b γ ≡ λ t n. ¬ γ t n

lemma notI[intro!]:
  assumes ¬ eval cid t t' n γ
  shows eval cid t t' n (¬^b γ)
proof cases
  assume ∃ i. ⸾cid⸾_t i
  show ?thesis
  proof cases
    assume ∃ i≥n. ⸾cid⸾_t i
    with ‹¬ eval cid t t' n γ›
      have ¬ γ (lnth (π_cid inf-llist t @_l inf-llist t')) (the-enat ⟨cid #_enat n inf-llist t⟩)
      using eval-def by blast
    with ‹∃ i≥n. ⸾cid⸾_t i› have eval cid t t' n (λt n. ¬ γ t n)
      using validCI-act[where γ=λ t n. ¬ γ t n] by blast
    thus ?thesis using not-def by simp
  next
    assume ¬ (∃ i≥n. ⸾cid⸾_t i)
    with ‹∃ i. ⸾cid⸾_t i› ‹¬ eval cid t t' n γ›
      have ¬ γ (lnth (π_cid inf-llist t @_l inf-llist t')) (_cid↓_t n) using eval-def by blast
    with ‹∃ i. ⸾cid⸾_t i› ‹¬ (∃ i≥n. ⸾cid⸾_t i)› have eval cid t t' n (λt n. ¬ γ t n)
      using validCI-cont[where γ=λ t n. ¬ γ t n] by blast
    thus ?thesis using not-def by simp
  qed
next
  assume ¬(∃ i. ⸾cid⸾_t i)
  with ‹¬ eval cid t t' n γ› have ¬ γ (lnth (π_cid inf-llist t @_l inf-llist t')) n
    using eval-def by blast
  with ‹¬(∃ i. ⸾cid⸾_t i)› have eval cid t t' n (λt n. ¬ γ t n)
    using validCI-not-act[where γ=λ t n. ¬ γ t n] by blast
  thus ?thesis using not-def by simp
qed

lemma notE[elim!]:
  assumes eval cid t t' n (¬^b γ)
  shows ¬ eval cid t t' n γ
proof cases
  assume (∃ i. ⸾cid⸾_t i)
```

```
  show ?thesis
  proof cases
    assume ∃ i≥n. ⦃cid⦄_t i
    moreover from ‹eval cid t t' n (¬^b γ)› have eval cid t t' n (λt n. ¬ γ t n)
      using not-def by simp
    ultimately have ¬ γ (lnth (π_cid inf-llist t @_l inf-llist t')) (the-enat ⟨cid #_enat n inf-llist t⟩)
      using validCE-act[where γ=λ t n. ¬ γ t n] by blast
    with ‹∃ i≥n. ⦃cid⦄_t i› show ?thesis using eval-def by blast
  next
    assume ¬ (∃ i≥n. ⦃cid⦄_t i)
    moreover from ‹eval cid t t' n (¬^b γ)› have eval cid t t' n (λt n. ¬ γ t n)
      using not-def by simp
    ultimately have ¬ γ (lnth (π_cid inf-llist t @_l inf-llist t')) (_cid↓_t n)
      using validCE-cont[where γ=λ t n. ¬ γ t n] ‹∃ i. ⦃cid⦄_t i› by blast
    with ‹¬ (∃ i≥n. ⦃cid⦄_t i)› ‹∃ i. ⦃cid⦄_t i› show ?thesis using eval-def by blast
  qed
next
  assume ¬(∃ i. ⦃cid⦄_t i)
  moreover from ‹eval cid t t' n (¬^b γ)› have eval cid t t' n (λt n. ¬ γ t n)
    using not-def by simp
  ultimately have ¬ γ (lnth (π_cid inf-llist t @_l inf-llist t')) n
    using validCE-not-act[where γ=λ t n. ¬ γ t n] by blast
  with ‹¬(∃ i. ⦃cid⦄_t i)› show ?thesis using eval-def by blast
qed
```

D.19.7 Quantifiers

```
definition all :: ('a ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool))
  ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) (binder ∀_b 10)
  where all P ≡ λt n. (∀ y. (P y t n))

lemma allI[intro!]:
  assumes ∀ p. eval cid t t' n (γ p)
  shows eval cid t t' n (all (λp. γ p))
proof cases
  assume ∃ i. ⦃cid⦄_t i
  show ?thesis
  proof cases
    assume ∃ i≥n. ⦃cid⦄_t i
    with ‹∀ p. eval cid t t' n (γ p)›
    have ∀ p. (γ p) (lnth (π_cid inf-llist t @_l inf-llist t')) (the-enat ⟨cid #_enat n inf-llist t⟩)
      using eval-def by blast
    with ‹∃ i≥n. ⦃cid⦄_t i› have eval cid t t' n (λt n. (∀ y. (γ y t n)))
      using validCI-act[where γ=λt n. (∀ y. (γ y t n))] by blast
    thus ?thesis using all-def[of γ] by auto
  next
    assume ¬ (∃ i≥n. ⦃cid⦄_t i)
    with ‹∃ i. ⦃cid⦄_t i› ‹∀ p. eval cid t t' n (γ p)›
      have ∀ p. (γ p) (lnth (π_cid inf-llist t @_l inf-llist t')) (_cid↓_t n)
```

```
      using eval-def by blast
    with ‹∃ i. ⦚cid⦚_t i› ‹¬ (∃ i≥n. ⦚cid⦚_t i)› have eval cid t t′ n (λt n. (∀ y. (γ y t n)))
      using validCI-cont[where γ=λt n. (∀ y. (γ y t n))] by blast
    thus ?thesis using all-def[of γ] by auto
  qed
next
  assume ¬(∃ i. ⦚cid⦚_t i)
  with ‹∀ p. eval cid t t′ n (γ p)› have ∀ p. (γ p) (lnth (π_cid inf-llist t @_l inf-llist t′)) n
    using eval-def by blast
  with ‹¬(∃ i. ⦚cid⦚_t i)› have eval cid t t′ n (λt n. (∀ y. (γ y t n)))
    using validCI-not-act[where γ=λt n. (∀ y. (γ y t n))] by blast
  thus ?thesis using all-def[of γ] by auto
qed

lemma allE[elim!]:
  assumes eval cid t t′ n (all (λp. γ p))
  shows ∀ p. eval cid t t′ n (γ p)
proof cases
  assume (∃ i. ⦚cid⦚_t i)
  show ?thesis
  proof cases
    assume ∃ i≥n. ⦚cid⦚_t i
    moreover from ‹eval cid t t′ n (all (λp. γ p))› have eval cid t t′ n (λt n. (∀ y. (γ y t n)))
      using all-def[of γ] by auto
    ultimately have
      ∀ p. (γ p) (lnth (π_cid inf-llist t @_l inf-llist t′)) (the-enat ⟨cid #_enat n inf-llist t⟩)
      using validCE-act[where γ=λt n. (∀ y. (γ y t n))] by blast
    with ‹∃ i≥n. ⦚cid⦚_t i› show ?thesis using eval-def by blast
  next
    assume ¬ (∃ i≥n. ⦚cid⦚_t i)
    moreover from ‹eval cid t t′ n (all (λp. γ p))› have eval cid t t′ n (λt n. (∀ y. (γ y t n)))
      using all-def[of γ] by auto
    ultimately have ∀ p. (γ p) (lnth (π_cid inf-llist t @_l inf-llist t′)) (_cid↓_t n)
      using validCE-cont[where γ=λt n. (∀ y. (γ y t n))] ‹∃ i. ⦚cid⦚_t i› by blast
    with ‹¬ (∃ i≥n. ⦚cid⦚_t i)› ‹∃ i. ⦚cid⦚_t i› show ?thesis using eval-def by blast
  qed
next
  assume ¬(∃ i. ⦚cid⦚_t i)
  moreover from ‹eval cid t t′ n (all (λp. γ p))› have eval cid t t′ n (λt n. (∀ y. (γ y t n)))
    using all-def[of γ] by auto
  ultimately have ∀ p. (γ p) (lnth (π_cid inf-llist t @_l inf-llist t′)) n
    using validCE-not-act[where γ=λt n. (∀ y. (γ y t n))] by blast
  with ‹¬(∃ i. ⦚cid⦚_t i)› show ?thesis using eval-def by blast
qed

definition ex :: ('a ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool))
  ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) (binder ∃_b 10)
  where ex P ≡ λt n. (∃ y. (P y t n))
```

```
lemma exI[intro!]:
  assumes ∃p. eval cid t t' n (γ p)
  shows eval cid t t' n ($\exists_b$p. γ p)
proof cases
  assume ∃i. $\wr cid \wr_{t\ i}$
  show ?thesis
  proof cases
    assume ∃i≥n. $\wr cid \wr_{t\ i}$
    with ‹∃p. eval cid t t' n (γ p)›
      have ∃p. (γ p) (lnth ($\pi_{cid}$inf-llist t $@_l$ inf-llist t')) (the-enat ⟨cid $\#_{enat\ n}$inf-llist t⟩)
      using eval-def by blast
    with ‹∃i≥n. $\wr cid \wr_{t\ i}$› have eval cid t t' n (λt n. (∃y. (γ y t n)))
      using validCI-act[where γ=λt n. (∃y. (γ y t n))] by blast
    thus ?thesis using ex-def[of γ] by auto
  next
    assume ¬ (∃i≥n. $\wr cid \wr_{t\ i}$)
    with ‹∃i. $\wr cid \wr_{t\ i}$› ‹∃p. eval cid t t' n (γ p)›
      have ∃p. (γ p) (lnth ($\pi_{cid}$inf-llist t $@_l$ inf-llist t')) ($_{cid}\downarrow_t n$) using eval-def by blast
    with ‹∃i. $\wr cid \wr_{t\ i}$› ‹¬ (∃i≥n. $\wr cid \wr_{t\ i}$)› have eval cid t t' n (λt n. (∃y. (γ y t n)))
      using validCI-cont[where γ=λt n. (∃y. (γ y t n))] by blast
    thus ?thesis using ex-def[of γ] by auto
  qed
next
  assume ¬(∃i. $\wr cid \wr_{t\ i}$)
  with ‹∃p. eval cid t t' n (γ p)› have ∃p. (γ p) (lnth ($\pi_{cid}$inf-llist t $@_l$ inf-llist t')) n
    using eval-def by blast
  with ‹¬(∃i. $\wr cid \wr_{t\ i}$)› have eval cid t t' n (λt n. (∃y. (γ y t n)))
    using validCI-not-act[where γ=λt n. (∃y. (γ y t n))] by blast
  thus ?thesis using ex-def[of γ] by auto
qed

lemma exE[elim!]:
  assumes eval cid t t' n ($\exists_b$p. γ p)
  shows ∃p. eval cid t t' n (γ p)
proof cases
  assume (∃i. $\wr cid \wr_{t\ i}$)
  show ?thesis
  proof cases
    assume ∃i≥n. $\wr cid \wr_{t\ i}$
    moreover from ‹eval cid t t' n (ex (λp. γ p))› have eval cid t t' n (λt n. (∃y. (γ y t n)))
      using ex-def[of γ] by auto
    ultimately have
      ∃p. (γ p) (lnth ($\pi_{cid}$inf-llist t $@_l$ inf-llist t')) (the-enat ⟨cid $\#_{enat\ n}$inf-llist t⟩)
      using validCE-act[where γ=λt n. (∃y. (γ y t n))] by blast
    with ‹∃i≥n. $\wr cid \wr_{t\ i}$› show ?thesis using eval-def by blast
  next
    assume ¬ (∃i≥n. $\wr cid \wr_{t\ i}$)
    moreover from ‹eval cid t t' n ($\exists_b$p. γ p)› have eval cid t t' n (λt n. (∃y. (γ y t n)))
      using ex-def[of γ] by auto
```

```
    ultimately have ∃p. (γ p) (lnth (π_cid inf-llist t @_l inf-llist t')) (_cid↓_t n)
      using validCE-cont[where γ=λt n. (∃y. (γ y t n))] ‹∃i. ≀cid≀_t i› by blast
    with ‹¬ (∃i≥n. ≀cid≀_t i)› ‹∃i. ≀cid≀_t i› show ?thesis using eval-def by blast
  qed
next
  assume ¬(∃i. ≀cid≀_t i)
  moreover from ‹eval cid t t' n (∃_b p. γ p)› have eval cid t t' n (λt n. (∃y. (γ y t n)))
    using ex-def[of γ] by auto
  ultimately have ∃p. (γ p) (lnth (π_cid inf-llist t @_l inf-llist t')) n
    using validCE-not-act[where γ=λt n. (∃y. (γ y t n))] by blast
  with ‹¬(∃i. ≀cid≀_t i)› show ?thesis using eval-def by blast
qed
```

D.20 Temporal Operators

We are now able to formalize all the rules of the calculus presented in [Mar17c].

D.20.1 Behavior Assertions

First we provide rules for basic behavior assertions.

```
definition ba :: ('cmp ⇒ bool) ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool)
  where ba φ ≡ λ t n. φ (t n)

lemma baIA[intro]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
  assumes ∃i≥n. ≀c≀_t i
    and φ (σ_c(t ⟨c → t⟩_n))
  shows eval c t t' n (ba φ)
proof –
  from assms have φ (σ_c(t ⟨c → t⟩_n)) by simp
  moreover have σ_c(t ⟨c → t⟩_n) = lnth (π_c(inf-llist t)) (the-enat (⟨c #_⟨c → t⟩_n inf-llist t⟩))
  proof –
    have enat (Suc ⟨c → t⟩_n) < llength (inf-llist t) using enat-ord-code by simp
    moreover from assms have ≀c≀_t (⟨c → t⟩_n) using nxtActI by simp
    hence ≀c≀_lnth (inf-llist t) ⟨c → t⟩_n by simp
    ultimately show ?thesis using proj-active-nth by simp
  qed
  ultimately have φ (lnth (π_c(inf-llist t)) (the-enat(⟨c #_⟨c → t⟩_n inf-llist t⟩))) by simp
  moreover have ⟨c #_n inf-llist t⟩ = ⟨c #_⟨c → t⟩_n inf-llist t⟩
  proof –
    from assms have ∄k. n≤k ∧ k<⟨c → t⟩_n ∧ ≀c≀_t k using nxtActI by simp
    hence ¬ (∃k≥n. k < ⟨c → t⟩_n ∧ ≀c≀_lnth (inf-llist t) k) by simp
    moreover have enat ⟨c → t⟩_n – 1 < llength (inf-llist t) by (simp add: one-enat-def)
    moreover from assms have ⟨c → t⟩_n≥n using nxtActI by simp
```

ultimately show *?thesis* **using** *nAct-not-active-same*[*of n* $\langle c \rightarrow t\rangle_n$ *inf-llist t c*] **by** *simp*
qed
ultimately have φ (*lnth* (π_c(*inf-llist t*)) (*the-enat*($\langle c\ \#_n$ *inf-llist t*$\rangle$))) **by** *simp*
moreover have *enat* (*the-enat* ($\langle c\ \#_{enat\ n}$ *inf-llist t*$\rangle$)) $<$ *llength* (π_c(*inf-llist t*))
proof $-$
have *ltake* ∞ (*inf-llist t*) $=$ (*inf-llist t*) **using** *ltake-all*[*of inf-llist t*] **by** *simp*
hence *llength* (π_c(*inf-llist t*)) $=$ $\langle c\ \#_\infty$ *inf-llist t*$\rangle$ **using** *nAct-def* **by** *simp*
moreover have $\langle c\ \#_{enat\ n}$ *inf-llist t*$\rangle$ $<$ $\langle c\ \#_\infty$ *inf-llist t*$\rangle$
proof $-$
have *enat* $\langle c \rightarrow t\rangle_n$ $<$ *llength* (*inf-llist t*) **by** *simp*
moreover from *assms* **have** $\langle c \rightarrow t\rangle_n \geq n$ **and** $⦃c⦄_{t\ (\langle c \rightarrow t\rangle_n)}$ **using** *nxtActI* **by** *auto*
ultimately show *?thesis* **using** *nAct-less*[*of* $\langle c \rightarrow t\rangle_n$ *inf-llist t n* ∞] **by** *simp*
qed
ultimately show *?thesis* **by** *simp*
qed
hence *lnth* (π_c(*inf-llist t*)) (*the-enat* ($\langle c\ \#_n$ *inf-llist t*$\rangle$)) $=$
lnth ((π_c(*inf-llist t*)) $@_l$ (*inf-llist t'*)) (*the-enat* ($\langle c\ \#_n$ *inf-llist t*$\rangle$))
using *lnth-lappend1*[*of the-enat* ($\langle c\ \#_{enat\ n}$ *inf-llist t*$\rangle$) π_c(*inf-llist t*) *inf-llist t'*] **by** *simp*
ultimately have φ (*lnth* ((π_c(*inf-llist t*)) $@_l$ (*inf-llist t'*)) (*the-enat*($\langle c\ \#_n$ *inf-llist t*$\rangle$)))
by *simp*
hence φ (*lnth* ((π_c(*inf-llist t*)) $@_l$ (*inf-llist t'*)) (*the-enat* ($\langle c\ \#_n$ *inf-llist t*$\rangle$))) **by** *simp*
moreover from *assms* **have** $\langle c \rightarrow t\rangle_n \geq n$ **and** $⦃c⦄_{t\ (\langle c \rightarrow t\rangle_n)}$ **using** *nxtActI* **by** *auto*
ultimately have ($\exists\, i \geq$ *snd* (*t*, *n*). $⦃c⦄_{fst\ (t,\ n)\ i}$) $\wedge$
φ (*lnth* ((π_c(*inf-llist* (*fst* (*t*, *n*)))) $@_l$ (*inf-llist t'*))
(*the-enat* ($\langle c\ \#_{the\text{-}enat\ (snd\ (t,n))}$ *inf-llist* (*fst* (*t*, *n*))$\rangle$)))) **by** *auto*
thus *?thesis* **using** *ba-def* **by** *simp*
qed

lemma *baIN1*[*intro*]:
fixes *c*::*'id*
and *t*::*nat* $\Rightarrow$ *cnf*
and *t'*::*nat* $\Rightarrow$ *'cmp*
and *n*::*nat*
assumes *act*: $\exists\, i.\ ⦃c⦄_{t\ i}$
and *nAct*: $\nexists\, i.\ i \geq n \wedge ⦃c⦄_{t\ i}$
and *al*: φ (*t'* (*n* $-$ $\langle c \wedge t\rangle$ $-$ *1*))
shows *eval c t t' n* (*ba* φ)
proof $-$
have *t'* (*n* $-$ $\langle c \wedge t\rangle$ $-$ *1*) $=$ *lnth* (*inf-llist t'*) (*n* $-$ $\langle c \wedge t\rangle$ $-$ *1*) **by** *simp*
moreover have $\ldots$ $=$ *lnth* ((π_c(*inf-llist t*)) $@_l$ (*inf-llist t'*)) ($c{\downarrow}_t(n)$)
using *act nAct cnf2bhv-lnth-lappend* **by** *simp*
ultimately have φ (*lnth* ((π_c(*inf-llist t*)) $@_l$ (*inf-llist t'*)) ($c{\downarrow}_t(n)$)) **using** *al* **by** *simp*
with *act nAct* **show** *?thesis* **using** *ba-def* **by** *simp*
qed

lemma *baIN2*[*intro*]:
fixes *c*::*'id*
and *t*::*nat* $\Rightarrow$ *cnf*
and *t'*::*nat* $\Rightarrow$ *'cmp*

```
    and n::nat
  assumes nAct: ∄ i. ⌇c⌇_t i
    and al: φ (t' n)
  shows eval c t t' n (ba φ)
proof –
  have t' n = lnth (inf-llist t') n by simp
  moreover have ... = lnth ((π_c(inf-llist t)) @_l (inf-llist t')) n
  proof –
    from nAct have π_c(inf-llist t) = []_l by simp
    hence (π_c(inf-llist t)) @_l (inf-llist t') = inf-llist t' by (simp add: ‹π_cinf-llist t = []_l›)
    thus ?thesis by simp
  qed
  ultimately have φ (lnth ((π_c(inf-llist t)) @_l (inf-llist t')) n) using al by simp
  hence φ (lnth ((π_c(inf-llist t)) @_l (inf-llist t')) n) by simp
  with nAct show ?thesis using ba-def by simp
qed

lemma baIANow[intro]:
  fixes t n c φ
  assumes φ (σ_c(t n))
    and ⌇c⌇_t n
  shows eval c t t' n (ba φ)
proof –
  from assms have φ(σ_c(t ⟨c → t⟩_n)) using nxtAct-active by simp
  with assms show ?thesis using baIA by blast
qed

lemma baEA[elim]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
    and i::nat
  assumes ∃ i≥n. ⌇c⌇_t i
    and eval c t t' n (ba φ)
  shows φ (σ_c(t ⟨c → t⟩_n))
proof –
  from ‹eval c t t' n (ba φ)› have eval c t t' n (λ t n. φ (t n)) using ba-def by simp
  moreover from assms have ⟨c → t⟩_n≥n and ⌇c⌇_t (⟨c → t⟩_n)
    using nxtActI[of n c t] by auto
  ultimately have φ (lnth ((π_c(inf-llist t)) @_l (inf-llist t')) (the-enat (⟨c #_n inf-llist t⟩)))
    using validCE-act by blast
  hence φ (lnth ((π_c(inf-llist t)) @_l (inf-llist t')) (the-enat (⟨c #_n inf-llist t⟩))) by simp
  moreover have enat (the-enat (⟨c #_enat n inf-llist t⟩)) < llength (π_c(inf-llist t))
  proof –
    have ltake ∞ (inf-llist t) = (inf-llist t) using ltake-all[of inf-llist t] by simp
    hence llength (π_c(inf-llist t)) = ⟨c #_∞ inf-llist t⟩ using nAct-def by simp
    moreover have ⟨c #_enat n inf-llist t⟩ < ⟨c #_∞ inf-llist t⟩
    proof –
```

```
      have enat ⟨c → t⟩_n < llength (inf-llist t) by simp
      with ‹⟨c → t⟩_n≥n› ‹⦃c⦄_t ⟨c → t⟩_n› show ?thesis using nAct-less by simp
    qed
    ultimately show ?thesis by simp
  qed
  hence lnth ((π_c(inf-llist t)) @_l (inf-llist t′)) (the-enat (⟨c #_n inf-llist t⟩)) =
    lnth (π_c(inf-llist t)) (the-enat (⟨c #_n inf-llist t⟩))
    using lnth-lappend1[of the-enat (⟨c #_enat n inf-llist t⟩) π_c(inf-llist t) inf-llist t′] by simp
  ultimately have φ (lnth (π_c(inf-llist t)) (the-enat (⟨c #_n inf-llist t⟩))) by simp
  moreover have ⟨c #_n inf-llist t⟩ = ⟨c #_⟨c → t⟩_n inf-llist t⟩
  proof –
    from assms have ∄k. n≤k ∧ k<⟨c → t⟩_n ∧ ⦃c⦄_t k using nxtActI[of n c t] by auto
    hence ¬ (∃k≥n. k < ⟨c → t⟩_n ∧ ⦃c⦄_lnth (inf-llist t) k) by simp
    moreover have enat ⟨c → t⟩_n − 1 < llength (inf-llist t) by (simp add: one-enat-def)
    ultimately show ?thesis using ‹⟨c → t⟩_n≥n› nAct-not-active-same by simp
  qed
  moreover have σ_c(t ⟨c → t⟩_n) = lnth (π_c(inf-llist t)) (the-enat (⟨c #_⟨c → t⟩_n inf-llist t⟩))
  proof –
    have enat (Suc i) < llength (inf-llist t) using enat-ord-code by simp
    moreover from ‹⦃c⦄_t ⟨c → t⟩_n› have ⦃c⦄_lnth (inf-llist t) ⟨c → t⟩_n by simp
    ultimately show ?thesis using proj-active-nth by simp
  qed
  ultimately show ?thesis by simp
qed

lemma baEN1[elim]:
  fixes c::′id
    and t::nat ⇒ cnf
    and t′::nat ⇒ ′cmp
    and n::nat
  assumes act: ∃i. ⦃c⦄_t i
    and nAct: ∄i. i≥n ∧ ⦃c⦄_t i
    and al: eval c t t′ n (ba φ)
  shows φ (t′ (n − ⟨c ∧ t⟩ − 1))
proof –
  from al have φ (lnth ((π_c(inf-llist t)) @_l (inf-llist t′)) (c↓_t(n)))
    using act nAct validCE-cont ba-def by metis
  hence φ (lnth ((π_c(inf-llist t)) @_l (inf-llist t′)) (c↓_t(n))) by simp
  moreover have
    lnth ((π_c(inf-llist t)) @_l (inf-llist t′)) (c↓_t(n)) = lnth (inf-llist t′) (n − ⟨c ∧ t⟩ − 1)
    using act nAct cnf2bhv-lnth-lappend by simp
  moreover have ... = t′ (n − ⟨c ∧ t⟩ − 1) by simp
  ultimately show ?thesis by simp
qed

lemma baEN2[elim]:
  fixes c::′id
    and t::nat ⇒ cnf
    and t′::nat ⇒ ′cmp
```

```
    and n::nat
  assumes nAct: $\nexists i.\ \wr c\wr_{t\ i}$
    and al: eval c t t' n (ba $\varphi$)
  shows $\varphi$ (t' n)
proof –
  from al have $\varphi$ (lnth (($\pi_c$(inf-llist t)) $@_l$ (inf-llist t')) n)
    using nAct validCE-not-act ba-def by metis
  hence $\varphi$ (lnth (($\pi_c$(inf-llist t)) $@_l$ (inf-llist t')) n) by simp
  moreover have lnth (($\pi_c$(inf-llist t)) $@_l$ (inf-llist t')) n = lnth (inf-llist t') n
  proof –
    from nAct have $\pi_c$(inf-llist t) = $[]_l$ by simp
    hence ($\pi_c$(inf-llist t)) $@_l$ (inf-llist t') = inf-llist t' by (simp add: ‹$\pi_c$inf-llist t = $[]_l$›)
    thus ?thesis by simp
  qed
  moreover have ... = t' n by simp
  ultimately show ?thesis by simp
qed

lemma baEANow[elim]:
  fixes t n c $\varphi$
  assumes eval c t t' n (ba $\varphi$)
    and $\wr c\wr_{t\ n}$
  shows $\varphi$ ($\sigma_c$(t n))
proof –
  from assms have $\varphi(\sigma_c(t\ \langle c \rightarrow t\rangle_n))$ using baEA by blast
  with assms show ?thesis using nxtAct-active by simp
qed
```

D.20.2 Next Operator

```
definition nxt :: ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) ($\bigcirc_b$(-) 24)
  where $\bigcirc_b(\gamma) \equiv \lambda\ t\ n.\ \gamma\ t\ (Suc\ n)$

lemma nxtIA[intro]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
  assumes $\exists\, i\geq n.\ \wr c\wr_{t\ i}$
    and $[\![\exists\, i>\langle c \rightarrow t\rangle_n.\ \wr c\wr_{t\ i}]\!] \Longrightarrow \exists\, n' \geq n.\ (\exists!i.\ n\leq i \wedge i<n' \wedge \wr c\wr_{t\ i}) \wedge eval\ c\ t\ t'\ n'\ \gamma$
    and $[\![\neg(\exists\, i>\langle c \rightarrow t\rangle_n.\ \wr c\wr_{t\ i})]\!] \Longrightarrow eval\ c\ t\ t'\ (Suc\ \langle c \rightarrow t\rangle_n)\ \gamma$
  shows eval c t t' n ($\bigcirc_b(\gamma)$)
proof (cases)
  assume $\exists\, i>\langle c \rightarrow t\rangle_n.\ \wr c\wr_{t\ i}$
  with assms(2) obtain n' where $n'\geq n$ and $\exists!i.\ n\leq i \wedge i<n' \wedge \wr c\wr_{t\ i}$ and eval c t t' n' $\gamma$
    by blast
  moreover from assms(1) have $\wr c\wr_{t\ \langle c \rightarrow t\rangle_n}$ and $\langle c \rightarrow t\rangle_n\geq n$ using nxtActI by auto
  ultimately have $\exists\, i'\geq n'.\ \wr c\wr_{t\ i'}$
    by (metis ‹$\exists\, i>\langle c \rightarrow t\rangle_n.\ \wr c\wr_{t\ i}$› dual-order.strict-trans2 leI nat-less-le)
```

```
  with ‹eval c t t' n' γ›
  have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat (⟨c #_enat n' inf-llist t⟩))
    using validCE-act by blast
  moreover have the-enat(⟨c #_n' inf-llist t⟩) = Suc (the-enat (⟨c #_n inf-llist t⟩))
  proof –
    from ‹∃!i. n≤i ∧ i<n' ∧ ⦃c⦄_t i› obtain i where n≤i and i<n' and ⦃c⦄_t i
      and ∀ i'. n≤i' ∧ i'<n' ∧ ⦃c⦄_t i' ⟶ i'=i by blast
    moreover have n' − 1 < llength (inf-llist t) by simp
    ultimately have the-enat(⟨c #_n' inf-llist t⟩) = the-enat(eSuc (⟨c #_n inf-llist t⟩))
      using nAct-active-suc[of inf-llist t n' n i c]  by (simp add: ‹n ≤ i›)
    moreover have ⟨c #_i inf-llist t⟩ ≠ ∞ by simp
    ultimately show ?thesis using the-enat-eSuc by simp
  qed
  ultimately have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (Suc (the-enat (⟨c #_n inf-llist t⟩)))
    by simp
  with assms have eval c t t' n (λt n. γ t (Suc n))
    using validCI-act[of n c t λt n. γ t (Suc n) t'] by blast
  thus ?thesis using nxt-def by simp
next
  assume ¬ (∃ i>⟨c → t⟩_n. ⦃c⦄_t i)
  with assms(3) have eval c t t' (Suc ⟨c → t⟩_n) γ by simp
  moreover from ‹¬ (∃ i>⟨c → t⟩_n. ⦃c⦄_t i)› have ¬ (∃ i≥Suc ⟨c → t⟩_n. ⦃c⦄_t i) by simp
  ultimately have γ (lnth (π_c inf-llist t @_l inf-llist t')) (c↓_t(Suc ⟨c → t⟩_n))
    using assms(1) validCE-cont[of c t Suc ⟨c → t⟩_n t' γ] by blast
  moreover from assms(1) ‹¬ (∃ i>⟨c → t⟩_n. ⦃c⦄_t i)›
    have Suc (the-enat ⟨c #_enat n inf-llist t⟩) = c↓_t(Suc ⟨c → t⟩_n)
    using nAct-cnf2proj-Suc-dist by simp
  ultimately have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (Suc (the-enat (⟨c #_n inf-llist t⟩)))
    by simp
  moreover from assms(1) have ⦃c⦄_t ⟨c → t⟩_n and ⟨c → t⟩_n ≥ n using nxtActI by auto
  ultimately have eval c t t' n (λt n. γ t (Suc n))
    using validCI-act[of n c t λt n. γ t (Suc n) t'] by blast
  with ‹⦃c⦄_t ⟨c → t⟩_n› ‹¬ (∃ i'≥Suc ⟨c → t⟩_n. ⦃c⦄_t i')› show ?thesis using nxt-def by simp
qed

lemma nxtIN[intro]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
  assumes ¬(∃ i≥n. ⦃c⦄_t i)
    and eval c t t' (Suc n) γ
  shows eval c t t' n (○_b(γ))
proof cases
  assume ∃ i. ⦃c⦄_t i
  moreover from ‹¬ (∃ i≥n. ⦃c⦄_t i)› have ¬ (∃ i≥Suc n. ⦃c⦄_t i) by simp
  ultimately have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (c↓_t(Suc n))
    using validCE-cont ‹eval c t t' (Suc n) γ› by blast
  with ‹∃ i. ⦃c⦄_t i› have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (Suc (c↓_t(n)))
```

```
    using ‹¬(∃ i≥n. ≀c≀_{t i})› lActive-less by auto
  with ‹¬(∃ i≥n. ≀c≀_{t i})› ‹∃ i. ≀c≀_{t i}› have eval c t t′ n (λt n. γ t (Suc n))
    using validCI-cont[where γ=(λt n. γ t (Suc n))] by simp
  thus ?thesis using nxt-def by simp
next
  assume ¬(∃ i. ≀c≀_{t i})
  with assms have γ (lnth (π_c inf-llist t @_l inf-llist t′)) (Suc n) using validCE-not-act by blast
  with ‹¬(∃ i. ≀c≀_{t i})› have eval c t t′ n (λt n. γ t (Suc n))
    using validCI-not-act[where γ=(λt n. γ t (Suc n))] by blast
  thus ?thesis using nxt-def by simp
qed

lemma nxtEA1[elim]:
  fixes c::′id
    and t::nat ⇒ cnf
    and t′::nat ⇒ ′cmp
    and n::nat
  assumes ∃ i>⟨c → t⟩_n. ≀c≀_{t i}
    and eval c t t′ n (○_b(γ))
    and n′≥n
    and ∃!i. i≥n ∧ i<n′ ∧ ≀c≀_{t i}
  shows eval c t t′ n′ γ
proof −
  from ‹eval c t t′ n (○_b(γ))› have eval c t t′ n (λt n. γ t (Suc n)) using nxt-def by simp
  moreover from assms(4) obtain i where i≥n and i<n′ and ≀c≀_{t i}
    and ∀ i′. n≤i′ ∧ i′<n′ ∧ ≀c≀_{t i′} ⟶ i′=i by blast
  ultimately have γ (lnth (π_c inf-llist t @_l inf-llist t′)) (Suc (the-enat ⟨c #_{enat n} inf-llist t⟩)))
    using validCE-act[of n c t t′ λt n. γ t (Suc n)] by blast
  moreover have the-enat(⟨c #_{n′} inf-llist t⟩) = Suc (the-enat (⟨c #_n inf-llist t⟩))
  proof −
    have n′ − 1 < llength (inf-llist t) by simp
    with ‹i<n′› and ‹≀c≀_{t i}› and ‹∀ i′. n≤i′ ∧ i′<n′ ∧ ≀c≀_{t i′} ⟶ i′=i›
      have the-enat(⟨c #_{n′} inf-llist t⟩) = the-enat(eSuc (⟨c #_n inf-llist t⟩))
      using nAct-active-suc[of inf-llist t n′ n i c]  by (simp add: ‹n ≤ i›)
    moreover have ⟨c #_i inf-llist t⟩ ≠ ∞ by simp
    ultimately show ?thesis using the-enat-eSuc by simp
  qed
  ultimately have γ (lnth ((π_c inf-llist t) @_l inf-llist t′)) (the-enat (⟨c #_{n′} inf-llist t⟩)) by simp
  moreover have ∃ i′≥n′. ≀c≀_{t i′}
  proof −
    from assms(4) have ⟨c → t⟩_n≥n and ≀c≀_{t ⟨c → t⟩_n} using nxtActI by auto
    with ‹∀ i′. n≤i′ ∧ i′<n′ ∧ ≀c≀_{t i′} ⟶ i′=i› show ?thesis
      using assms(1) by (metis leI le-trans less-le)
  qed
  ultimately show ?thesis using validCI-act by blast
qed

lemma nxtEA2[elim]:
  fixes c::′id
```

```
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
    and i
  assumes ∃ i≥n. ⦃c⦄_t i and ¬(∃ i>⟨c → t⟩_n. ⦃c⦄_t i)
    and eval c t t' n (○_b(γ))
  shows eval c t t' (Suc ⟨c → t⟩_n) γ
proof –
  from ‹eval c t t' n (○_b(γ))› have eval c t t' n (λt n. γ t (Suc n)) using nxt-def by simp
  with assms(1) have
    γ (lnth (π_c inf-llist t @_l inf-llist t')) (Suc (the-enat ⟨c #_enat n inf-llist t⟩))
    using validCE-act[of n c t t' λt n. γ t (Suc n)] by blast
  moreover from assms(1) assms(2) have
    Suc (the-enat ⟨c #_enat n inf-llist t⟩)=_c↓_t(Suc ⟨c → t⟩_n)
    using nAct-cnf2proj-Suc-dist by simp
  ultimately have γ (lnth (π_c inf-llist t @_l inf-llist t')) (_c↓_t(Suc ⟨c → t⟩_n)) by simp
  moreover from assms(1) assms(2) have ¬(∃ i'≥Suc ⟨c → t⟩_n. ⦃c⦄_t i')
    using nxtActive-no-active by simp
  ultimately show ?thesis using validCI-cont[where n=Suc ⟨c → t⟩_n] assms(1) by blast
qed

lemma nxtEN[elim]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
  assumes ¬(∃ i≥n. ⦃c⦄_t i)
    and eval c t t' n (○_b(γ))
  shows eval c t t' (Suc n) γ
proof cases
  assume ∃ i. ⦃c⦄_t i
  moreover from ‹eval c t t' n (○_b(γ))› have eval c t t' n (λt n. γ t (Suc n))
    using nxt-def by simp
  ultimately have γ (lnth (π_c inf-llist t @_l inf-llist t')) (Suc (_c↓_t n))
    using ‹¬(∃ i≥n. ⦃c⦄_t i)› validCE-cont[where γ=(λt n. γ t (Suc n))] by simp
  hence γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (_c↓_t(Suc n))
    using ‹∃ i. ⦃c⦄_t i› assms(1) lActive-less by auto
  moreover from ‹¬ (∃ i≥n. ⦃c⦄_t i)› have ¬ (∃ i≥Suc n. ⦃c⦄_t i) by simp
  ultimately show ?thesis using validCI-cont[where n=Suc n] ‹∃ i. ⦃c⦄_t i› by blast
next
  assume ¬(∃ i. ⦃c⦄_t i)
  moreover from ‹eval c t t' n (○_b(γ))› have eval c t t' n (λt n. γ t (Suc n))
    using nxt-def by simp
  ultimately have γ (lnth (π_c inf-llist t @_l inf-llist t')) (Suc n)
    using ‹¬(∃ i. ⦃c⦄_t i)› validCE-not-act[where γ=(λt n. γ t (Suc n))] by blast
  with ‹¬(∃ i. ⦃c⦄_t i)› show ?thesis using validCI-not-act[of c t γ t' Suc n] by blast
qed
```

D.20.3 Eventually Operator

```
definition evt :: ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) (◇_b(-) 23)
  where ◇_b(γ) ≡ λ t n. ∃ n'≥n. γ t n'

lemma evtIA[intro]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
    and n'::nat
  assumes ∃ i≥n. ≀c≀_t i
    and n'≥⟨c ⇐ t⟩_n
    and ⟦∃ i≥n'. ≀c≀_t i⟧ ⟹ ∃ n''≥⟨c ⇐ t⟩_n'. n''≤ ⟨c → t⟩_n' ∧ eval c t t' n'' γ
    and ⟦¬(∃ i≥n'. ≀c≀_t i)⟧ ⟹ eval c t t' n' γ
  shows eval c t t' n (◇_b(γ))
proof cases  assume ∃ i'≥n'. ≀c≀_t i'
  with assms(3) obtain n'' where n'' ≥⟨c ⇐ t⟩_n' and n''≤ ⟨c → t⟩_n' and eval c t t' n'' γ
by auto
  hence ∃ i'≥n''. ≀c≀_t i' using ‹∃ i'≥n'. ≀c≀_t i'› nxtActI by blast
  with ‹eval c t t' n'' γ› have
    γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat (⟨c #_n'' inf-llist t⟩))
    using validCE-act by blast
  moreover have the-enat (⟨c #_n inf-llist t⟩) ≤ the-enat (⟨c #_n'' inf-llist t⟩)
  proof –
    from ‹⟨c ⇐ t⟩_n'≤n''› have ⟨c #_n' inf-llist t⟩ ≤ ⟨c #_n'' inf-llist t⟩
      using nAct-mono-lNact by simp
    moreover from ‹n'≥⟨c ⇐ t⟩_n› have ⟨c #_n inf-llist t⟩ ≤ ⟨c #_n' inf-llist t⟩
      using nAct-mono-lNact by simp
    moreover have ⟨c #_n' inf-llist t⟩ ≠ ∞ by simp
    ultimately show ?thesis by simp
  qed
  moreover have ∃ i'≥n. ≀c≀_t i'
  proof –
    from ‹∃ i'≥n'. ≀c≀_t i'› obtain i' where i'≥n' and ≀c≀_t i' by blast
    with ‹n'≥⟨c ⇐ t⟩_n› have i'≥ n using lNactGe le-trans by blast
    with ‹≀c≀_t i'› show ?thesis by blast
  qed
  ultimately have eval c t t' n (λt n. ∃ n'≥n. γ t n')
    using validCI-act[where γ=(λt n. ∃ n'≥n. γ t n')] by blast
  thus ?thesis using evt-def by simp
next
  assume ¬(∃ i'≥n'. ≀c≀_t i')
  with ‹(∃ i≥n. ≀c≀_t i)› have n' ≥ ⟨c ∧ t⟩ using lActive-less by auto
  hence _c↓_t(n') ≥ the-enat (llength (π_c(inf-llist t))) – 1 using cnf2bhv-ge-llength by simp
  moreover have the-enat(llength (π_c(inf-llist t))) – 1 ≥ the-enat(⟨c #_n inf-llist t⟩)
  proof –
    from ‹∃ i≥n. ≀c≀_t i› have llength (π_c(inf-llist t)) ≥ eSuc (⟨c #_n inf-llist t⟩)
      using nAct-llength-proj by simp
    moreover from ‹¬(∃ i'≥n'. ≀c≀_t i')› have lfinite (π_c(inf-llist t))
```

```
      using proj-finite2[of inf-llist t] by simp
    hence llength (π_c(inf-llist t))≠∞ using llength-eq-infty-conv-lfinite by auto
    ultimately have the-enat (llength (π_c(inf-llist t))) ≥ the-enat(eSuc (⟨c #_n inf-llist t⟩))
      by simp
    moreover have ⟨c #_n inf-llist t⟩≠∞ by simp
    ultimately have the-enat (llength (π_c(inf-llist t))) ≥ Suc (the-enat (⟨c #_n inf-llist t⟩))
      using the-enat-eSuc by simp
    thus ?thesis by simp
  qed
  ultimately have c↓_t(n') ≥ the-enat (⟨c #_n inf-llist t⟩) by simp
  moreover from ‹¬(∃ i'≥n'. ⦃c⦄_{t i'})› have eval c t t' n' γ using assms(4) by simp
    with ‹∃ i≥n. ⦃c⦄_{t i}› ‹¬(∃ i'≥n'. ⦃c⦄_{t i'})›
    have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (c↓_t(n')) using validCE-cont by blast
  ultimately have eval c t t' n (λt n. ∃ n'≥n. γ t n')
    using ‹∃ i≥n. ⦃c⦄_{t i}› validCI-act[where γ=(λt n. ∃ n'≥n. γ t n')] by blast
  thus ?thesis using evt-def by simp
qed

lemma evtIN[intro]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
    and n'::nat
  assumes ¬(∃ i≥n. ⦃c⦄_{t i})
    and n'≥n
    and eval c t t' n' γ
  shows eval c t t' n (◇_b(γ))
proof cases
  assume ∃ i. ⦃c⦄_{t i}
  moreover from assms(1) assms(2) have ¬(∃ i'≥n'. ⦃c⦄_{t i'}) by simp
  ultimately have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (c↓_t(n'))
    using validCE-cont[of c t n' t' γ] ‹eval c t t' n' γ› by blast
  moreover from ‹n'≥n› have c↓_t(n') ≥ c↓_t(n) using cnf2bhv-mono by simp
  ultimately have eval c t t' n (λt n. ∃ n'≥n. γ t n')
    using validCI-cont[where γ=(λt n. ∃ n'≥n. γ t n')] ‹∃ i. ⦃c⦄_{t i}› ‹¬(∃ i≥n. ⦃c⦄_{t i})› by blast
  thus ?thesis using evt-def by simp
next
  assume ¬(∃ i. ⦃c⦄_{t i})
  with assms have γ (lnth (π_cinf-llist t @_l inf-llist t')) n' using validCE-not-act by blast
  with ‹¬(∃ i. ⦃c⦄_{t i})› have eval c t t' n (λt n. ∃ n'≥n. γ t n')
    using validCI-not-act[where γ=λt n. ∃ n'≥n. γ t n'] ‹n'≥n› by blast
  thus ?thesis using evt-def by simp
qed

lemma evtEA[elim]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
```

```
    and n::nat
  assumes ∃ i≥n. ⦃c⦄_t i
    and eval c t t' n (◇_b(γ))
  shows ∃ n'≥⟨c → t⟩_n.
          (∃ i≥n'. ⦃c⦄_t i ∧ (∀ n''≥ ⟨c ⇐ t⟩_n'. n''≤⟨c → t⟩_n' ⟶ eval c t t' n'' γ)) ∨
          (¬(∃ i≥n'. ⦃c⦄_t i) ∧ eval c t t' n' γ)
proof −
  from ‹eval c t t' n (◇_b(γ))› have eval c t t' n (λt n. ∃ n'≥n. γ t n') using evt-def by simp
  with ‹∃ i≥n. ⦃c⦄_t i›
    have ∃ n'≥the-enat ⟨c #_enat n inf-llist t⟩. γ (lnth (π_c inf-llist t @_l inf-llist t')) n'
    using validCE-act[where γ=λt n. ∃ n'≥n. γ t n'] by blast
  then obtain x where x≥the-enat (⟨c #_n inf-llist t⟩) and
    γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) x by auto
  thus ?thesis
  proof (cases)
    assume x ≥ llength (π_c(inf-llist t))
    moreover from ‹(x ≥ llength (π_c(inf-llist t)))› have llength (π_c(inf-llist t))≠∞
      by (metis infinity-ileE)
    moreover from ‹∃ i≥n. ⦃c⦄_t i› have llength (π_c(inf-llist t))≥1
      using proj-one[of inf-llist t] by auto
    ultimately have the-enat (llength (π_c(inf-llist t))) − 1 < x
      by (metis One-nat-def Suc-ile-eq antisym-conv2 diff-Suc-less enat-ord-simps(2)
          enat-the-enat less-imp-diff-less one-enat-def)
    hence x = _c↓_t(_c↑_t(x)) using cnf2bhv-bhv2cnf by simp
    with ‹γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) x›
      have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (_c↓_t(_c↑_t(x))) by simp
    moreover have ¬(∃ i≥_c↑_t(x). ⦃c⦄_t i)
    proof −
      from ‹x ≥ llength (π_c(inf-llist t))› have lfinite (π_c(inf-llist t))
        using llength-geq-enat-lfiniteD[of π_c(inf-llist t) x] by simp
      then obtain z where ∀ n''>z. ¬ ⦃c⦄_t n'' using proj-finite-bound by blast
      moreover from ‹the-enat (llength (π_c(inf-llist t))) − 1 < x› have ⟨c ∧ t⟩ < _c↑_t(x)
        using bhv2cnf-greater-lActive by simp
      ultimately show ?thesis using lActive-greater-active-all by simp
    qed
    ultimately have eval c t t' (_c↑_tx) γ
      using ‹∃ i≥n. ⦃c⦄_t i› validCI-cont[of c t _c↑_t(x)] by blast
    moreover have _c↑_t(x) ≥ ⟨c → t⟩_n
    proof −
      from ‹x ≥ llength (π_c(inf-llist t))› have lfinite (π_c(inf-llist t))
        using llength-geq-enat-lfiniteD[of π_c(inf-llist t) x] by simp
      then obtain z where ∀ n''>z. ¬ ⦃c⦄_t n'' using proj-finite-bound by blast
      moreover from ‹∃ i≥n. ⦃c⦄_t i› have ⦃c⦄_t ⟨c → t⟩_n using nxtActI by simp
      ultimately have ⟨c ∧ t⟩≥⟨c → t⟩_n using lActive-greatest by fastforce
      moreover have _c↑_t(x) ≥ ⟨c ∧ t⟩ by simp
      ultimately show _c↑_t(x) ≥ ⟨c → t⟩_n by arith
    qed
    ultimately show ?thesis using ‹¬(∃ i≥_c↑_t(x). ⦃c⦄_t i)› by blast
  next
```

```
    assume ¬(x ≥ llength (π_c(inf-llist t)))
    hence x<llength (π_c(inf-llist t)) by simp
    then obtain n'::nat where x=⟨c #_n' inf-llist t⟩ using nAct-exists by blast
    with ‹enat x < llength (π_c(inf-llist t))› have ∃ i≥n'. ⦃c⦄_{t i}
      using nAct-less-llength-active by force
    then obtain i where i≥n' and ⦃c⦄_{t i} and ¬ (∃ k≥n'. k < i ∧ ⦃c⦄_{t k})
      using nact-exists by blast
    moreover have (∀ n''≥ ⟨c ⇐ t⟩_i. n''≤⟨c → t⟩_i ⟶ eval c t t' n'' γ)
    proof
      fix n'' show ⟨c ⇐ t⟩_i ≤ n'' ⟶ n'' ≤ ⟨c → t⟩_i ⟶ eval c t t' n'' γ
      proof(rule HOL.impI[OF HOL.impI])
        assume ⟨c ⇐ t⟩_i ≤ n'' and n'' ≤ ⟨c → t⟩_i
        hence the-enat (⟨c #_{enat i} inf-llist t⟩) = the-enat (⟨c #_{enat n''} inf-llist t⟩)
          using nAct-same by simp
        moreover from ‹⦃c⦄_{t i}› have ⦃c⦄_{t ⟨c → t⟩_i} using nxtActI by auto
        with ‹n'' ≤ ⟨c → t⟩_i› have ∃ i≥n''. ⦃c⦄_{t i} using dual-order.strict-implies-order by auto
        moreover have
          γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat (⟨c #_{enat i} inf-llist t⟩))
        proof −
          have enat i − 1 < llength (inf-llist t) by (simp add: one-enat-def)
          with ‹x=⟨c #_n' inf-llist t⟩› ‹i≥n'› ‹¬ (∃ k≥n'. k < i ∧ ⦃c⦄_{t k})›
            have x=⟨c #_i inf-llist t⟩
            using one-enat-def nAct-not-active-same by simp
          moreover have ⟨c #_i inf-llist t⟩≠∞ by simp
          ultimately have x=the-enat(⟨c #_i inf-llist t⟩) by fastforce
          thus ?thesis using ‹γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) x› by blast
        qed
        with ‹the-enat (⟨c #_{enat i} inf-llist t⟩) = the-enat (⟨c #_{enat n''} inf-llist t⟩)› have
          γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat (⟨c #_{enat n''} inf-llist t⟩)) by simp
        ultimately show eval c t t' n'' γ using validCI-act by blast
      qed
    qed
    moreover have i≥⟨c → t⟩_n
    proof −
      have enat i − 1 < llength (inf-llist t) by (simp add: one-enat-def)
      with ‹x=⟨c #_n' inf-llist t⟩› ‹i≥n'› ‹¬ (∃ k≥n'. k < i ∧ ⦃c⦄_{t k})› have x=⟨c #_i inf-llist t⟩
        using one-enat-def nAct-not-active-same by simp
      moreover have ⟨c #_i inf-llist t⟩≠∞ by simp
      ultimately have x=the-enat(⟨c #_i inf-llist t⟩) by fastforce
      with ‹x≥the-enat (⟨c #_n inf-llist t⟩)›
        have the-enat (⟨c #_i inf-llist t⟩)≥the-enat (⟨c #_n inf-llist t⟩) by simp
      with ‹⦃c⦄_{t i}› show ?thesis using active-geq-nxtAct by simp
    qed
    ultimately show ?thesis using ‹⦃c⦄_{t i}› by auto
  qed
qed

lemma evtEN[elim]:
  fixes c::'id
```

```
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
    and n'::nat
  assumes ¬(∃ i≥n. ⌇c⌇_t i)
    and eval c t t' n (◇_b(γ))
  shows ∃ n'≥n. eval c t t' n' γ
proof cases
  assume ∃ i. ⌇c⌇_t i
  moreover from ‹eval c t t' n (◇_b(γ))› have eval c t t' n (λt n. ∃ n'≥n. γ t n')
    using evt-def by simp
  ultimately have ∃ n'≥_c↓_t n. γ (lnth (π_c inf-llist t @_l inf-llist t')) n'
    using validCE-cont[where γ=(λt n. ∃ n'≥n. γ t n')] ‹¬(∃ i≥n. ⌇c⌇_t i)› by blast
  then obtain x where x≥_c↓_t(n) and  γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) x by auto
  moreover have the-enat (llength (π_c(inf-llist t))) − 1 < x
  proof −
    have ⟨c ∧ t⟩ < n
    proof (rule ccontr)
      assume ¬⟨c ∧ t⟩ < n
      hence ⟨c ∧ t⟩ ≥ n by simp
      moreover from ‹∃ i. ⌇c⌇_t i› ‹¬ (∃ i≥n. ⌇c⌇_t i)› have ⌇c⌇_t ⟨c ∧ t⟩
        using lActive-active less-or-eq-imp-le by blast
      ultimately show False using ‹¬ (∃ i≥n. ⌇c⌇_t i)› by simp
    qed
   hence the-enat (llength (π_c(inf-llist t))) − 1 < _c↓_t(n) using cnf2bhv-greater-llength by simp
    with ‹x≥_c↓_t(n)› show ?thesis by simp
  qed
  hence x = _c↓_t(_c↑_t(x)) using cnf2bhv-bhv2cnf by simp
  ultimately have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (_c↓_t(_c↑_t(x))) by simp
  moreover from ‹¬(∃ i≥n. ⌇c⌇_t i)› have ¬(∃ i≥_c↑_t(x). ⌇c⌇_t i)
  proof −
    from ‹¬(∃ i≥n. ⌇c⌇_t i)› have lfinite (π_c(inf-llist t)) using proj-finite2 by simp
    then obtain z where ∀ n''>z. ¬ ⌇c⌇_t n'' using proj-finite-bound by blast
    moreover from ‹the-enat (llength (π_c(inf-llist t))) − 1 < x› have ⟨c ∧ t⟩ < _c↑_t(x)
      using bhv2cnf-greater-lActive by simp
    ultimately show ?thesis using lActive-greater-active-all by simp
  qed
  ultimately have eval c t t' (_c↑_t x) γ
    using validCI-cont[of c t _c↑_t(x) γ] ‹∃ i. ⌇c⌇_t i› by blast
  moreover from ‹∃ i. ⌇c⌇_t i› ‹¬(∃ i≥n. ⌇c⌇_t i)› have ⟨c ∧ t⟩ ≤ n
    using lActive-less[of c t - n] by auto
  with ‹x≥_c↓_t(n)› have n ≤ _c↑_t(x) using p2c-mono-c2p by blast
  ultimately show ?thesis by auto
next
  assume ¬(∃ i. ⌇c⌇_t i)
  moreover from ‹eval c t t' n (◇_b(γ))› have eval c t t' n (λt n. ∃ n'≥n. γ t n')
    using evt-def by simp
  ultimately obtain n' where n'≥n and γ (lnth (π_c inf-llist t @_l inf-llist t')) n'
    using ‹¬(∃ i. ⌇c⌇_t i)› validCE-not-act[where γ=λt n. ∃ n'≥n. γ t n'] by blast
```

with $‹\neg(\exists\, i.\ \wr c\wr_{t\ i})›$ **show** *?thesis* **using** $\mathit{validCI\text{-}not\text{-}act}[\mathit{of}\ c\ t\ \gamma\ t'\ n']$ **by** *blast*
qed

D.20.4 Globally Operator

definition $\mathit{glob} :: ((\mathit{nat} \Rightarrow \mathit{'cmp}) \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}) \Rightarrow ((\mathit{nat} \Rightarrow \mathit{'cmp}) \Rightarrow \mathit{nat} \Rightarrow \mathit{bool})\ (\Box_b(\text{-})\ 22)$
where $\Box_b(\gamma) \equiv \lambda\ t\ n.\ \forall\, n'{\geq}n.\ \gamma\ t\ n'$

lemma $\mathit{globIA}[\mathit{intro}]$:
fixes $c{::}\mathit{'id}$
and $t{::}\mathit{nat} \Rightarrow \mathit{cnf}$
and $t'{::}\mathit{nat} \Rightarrow \mathit{'cmp}$
and $n{::}\mathit{nat}$
assumes $\exists\, i{\geq}n.\ \wr c\wr_{t\ i}$
and $\bigwedge n'.\ [\![\exists\, i{\geq}n'.\ \wr c\wr_{t\ i};\ n'{\geq}\langle c \rightarrow t\rangle_n]\!] \Longrightarrow$
$\exists\, n''{\geq}\langle c \Leftarrow t\rangle_{n'}.\ n''{\leq}\langle c \rightarrow t\rangle_{n'} \wedge \mathit{eval}\ c\ t\ t'\ n''\ \gamma$
and $\bigwedge n'.\ [\![\neg(\exists\, i{\geq}n'.\ \wr c\wr_{t\ i});\ n'{\geq}\langle c \rightarrow t\rangle_n]\!] \Longrightarrow \mathit{eval}\ c\ t\ t'\ n'\ \gamma$
shows $\mathit{eval}\ c\ t\ t'\ n\ (\Box_b(\gamma))$
proof −
have $\forall\, n'{\geq}\mathit{the\text{-}enat}\ \langle c\ \#_{\mathit{enat}\ n}\mathit{inf\text{-}llist}\ t\rangle.\ \gamma\ (\mathit{lnth}\ (\pi_c\mathit{inf\text{-}llist}\ t\ @_l\ \mathit{inf\text{-}llist}\ t'))\ n'$
proof
fix $x{::}\mathit{nat}$ **show**
$x{\geq}\mathit{the\text{-}enat}\ (\langle c\ \#_n\ \mathit{inf\text{-}llist}\ t\rangle) \longrightarrow \gamma\ (\mathit{lnth}\ (\pi_c\mathit{inf\text{-}llist}\ t\ @_l\ \mathit{inf\text{-}llist}\ t'))\ x$
proof
assume $x{\geq}\mathit{the\text{-}enat}\ (\langle c\ \#_n\ \mathit{inf\text{-}llist}\ t\rangle)$
show $\gamma\ (\mathit{lnth}\ ((\pi_c(\mathit{inf\text{-}llist}\ t))\ @_l\ (\mathit{inf\text{-}llist}\ t')))\ x$
proof (*cases*)
assume $(x \geq \mathit{llength}\ (\pi_c(\mathit{inf\text{-}llist}\ t)))$
hence $\mathit{lfinite}\ (\pi_c(\mathit{inf\text{-}llist}\ t))$
using $\mathit{llength\text{-}geq\text{-}enat\text{-}lfiniteD}[\mathit{of}\ \pi_c(\mathit{inf\text{-}llist}\ t)\ x]$ **by** *simp*
then obtain z **where** $\forall\, n''{>}z.\ \neg\ \wr c\wr_{t\ n''}$ **using** *proj-finite-bound* **by** *blast*
moreover have $\wr c\wr_{t\ \langle c \rightarrow t\rangle_n}$ **by** (*simp add*: $‹\exists\, i{\geq}n.\ \wr c\wr_{t\ i}›$ *nxtActI*)
ultimately have $\langle c \wedge t\rangle{\geq}\langle c \rightarrow t\rangle_n$ **using** $\mathit{lActive\text{-}greatest}[\mathit{of}\ c\ t\ \langle c \rightarrow t\rangle_n]$ **by** *blast*
moreover have $_c{\uparrow}_t(x) \geq \langle c \wedge t\rangle$ **by** *simp*
ultimately have $_c{\uparrow}_t(x) \geq \langle c \rightarrow t\rangle_n$ **by** *arith*
moreover have $\neg\ (\exists\, i'{\geq}{_c{\uparrow}_t}(x).\ \wr c\wr_{t\ i'})$
proof −
from $‹\mathit{lfinite}\ (\pi_c(\mathit{inf\text{-}llist}\ t))›\ ‹\exists\, i{\geq}n.\ \wr c\wr_{t\ i}›$
have $_c{\uparrow}_t(\mathit{the\text{-}enat}\ (\mathit{llength}\ (\pi_c(\mathit{inf\text{-}llist}\ t)))) = \mathit{Suc}\ (\langle c \wedge t\rangle)$
using *bhv2cnf-lActive* **by** *blast*
moreover from $‹(x \geq \mathit{llength}\ (\pi_c(\mathit{inf\text{-}llist}\ t)))›$
have $x \geq \mathit{the\text{-}enat}(\mathit{llength}\ (\pi_c(\mathit{inf\text{-}llist}\ t)))$
using *the-enat-mono* **by** *fastforce*
hence $_c{\uparrow}_t(x) \geq {_c{\uparrow}_t}(\mathit{the\text{-}enat}\ (\mathit{llength}\ (\pi_c(\mathit{inf\text{-}llist}\ t))))$
using $\mathit{bhv2cnf\text{-}mono}[\mathit{of}\ \mathit{the\text{-}enat}\ (\mathit{llength}\ (\pi_c(\mathit{inf\text{-}llist}\ t)))\ x]$ **by** *simp*
ultimately have $_c{\uparrow}_t(x) \geq \mathit{Suc}\ (\langle c \wedge t\rangle)$ **by** *simp*
hence $_c{\uparrow}_t(x) > \langle c \wedge t\rangle$ **by** *simp*
with $‹\forall\, n''{>}z.\ \neg\ \wr c\wr_{t\ n''}›$ **show** *?thesis* **using** *lActive-greater-active-all* **by** *simp*
qed

```
        ultimately have eval c t t' (c↑_t(x)) γ using assms(3) by simp
        hence γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (c↓_t(c↑_t(x)))
          using validCE-cont[of c t c↑_t(x) t' γ] ‹∃ i≥n. ≀c≀_t i› ‹¬ (∃ i'≥c↑_t(x). ≀c≀_t i')› by blast
        moreover from ‹(x ≥ llength (π_c(inf-llist t)))›
          have (enat x ≥ llength (π_c(inf-llist t))) by auto
        with ‹lfinite (π_c(inf-llist t))› have llength (π_c(inf-llist t))≠∞
          using llength-eq-infty-conv-lfinite by auto
        with ‹(x ≥ llength (π_c(inf-llist t)))›
          have the-enat(llength (π_c(inf-llist t))) − 1 ≤ x by auto
        ultimately show ?thesis using cnf2bhv-bhv2cnf[of c t x] by simp
      next
        assume ¬(x ≥ llength (π_c(inf-llist t)))
        hence x<llength (π_c(inf-llist t)) by simp
        then obtain n'::nat where x=⟨c #_n' inf-llist t⟩ using nAct-exists by blast
        moreover from ‹enat x < llength (π_c(inf-llist t))› ‹enat x = ⟨c #_enat n' inf-llist t⟩›
          have ∃ i≥n'. ≀c≀_t i using nAct-less-llength-active by force
        then obtain i where i≥n' and ≀c≀_t i and ¬ (∃ k≥n'. k < i ∧ ≀c≀_t k)
          using nact-exists by blast
        moreover have enat i − 1 < llength (inf-llist t) by (simp add: one-enat-def)
        ultimately have x=⟨c #_i inf-llist t⟩ using one-enat-def nAct-not-active-same by simp
        moreover have ⟨c #_i inf-llist t⟩≠∞ by simp
        ultimately have x=the-enat(⟨c #_i inf-llist t⟩) by fastforce
        from ‹x≥the-enat (⟨c #_n inf-llist t⟩)› ‹x=the-enat(⟨c #_i inf-llist t⟩)›
        have the-enat (⟨c #_i inf-llist t⟩)≥the-enat (⟨c #_n inf-llist t⟩) by simp
        with ‹≀c≀_t i› have i≥⟨c → t⟩_n using active-geq-nxtAct by simp
        moreover from ‹x=⟨c #_i inf-llist t⟩› ‹x < llength (π_c(inf-llist t))›
          have ∃ i'. i ≤ enat i' ∧ ≀c≀_t i' using nAct-less-llength-active[of x c inf-llist t i] by simp
        hence ∃ i'≥i. ≀c≀_t i' by simp
        ultimately obtain n'' where eval c t t' n'' γ and n''≥⟨c ⇐ t⟩_i and n''≤⟨c → t⟩_i
          using assms(2) by blast
        moreover have ∃ i'≥n''. ≀c≀_t i'
          using ‹≀c≀_t i› ‹n''≤⟨c → t⟩_i› less-or-eq-imp-le nxtAct-active by auto
        ultimately have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat (⟨c #_n'' inf-llist t⟩))
          using validCE-act[of n'' c t t' γ] by blast
        moreover from ‹n''≥⟨c ⇐ t⟩_i› and ‹n''≤⟨c → t⟩_i›
          have the-enat (⟨c #_n'' inf-llist t⟩)=the-enat (⟨c #_i inf-llist t⟩)
          using nAct-same by simp
        hence the-enat (⟨c #_n'' inf-llist t⟩) = x
          by (simp add: ‹x = the-enat ⟨c #_enat i inf-llist t⟩›)
        ultimately have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat x) by simp
        thus ?thesis by simp
      qed
    qed
  qed
  with ‹∃ i≥n. ≀c≀_t i› have eval c t t' n (λt n. ∀ n'≥n. γ t n')
    using validCI-act[of n c t λ t n. ∀ n'≥n. γ t n' t'] by blast
  thus ?thesis using glob-def by simp
qed
```

```
lemma globIN[intro]:
  fixes $c::'id$
    and $t::nat \Rightarrow cnf$
    and $t'::nat \Rightarrow 'cmp$
    and $n::nat$
  assumes $\neg(\exists\, i{\geq}n.\ \wr c\wr_{t\ i})$
    and $\bigwedge n'.\ n'{\geq}n \Longrightarrow eval\ c\ t\ t'\ n'\ \gamma$
  shows $eval\ c\ t\ t'\ n\ (\Box_b(\gamma))$
proof cases
  assume $\exists\, i.\ \wr c\wr_{t\ i}$
  from ‹$\neg(\exists\, i{\geq}n.\ \wr c\wr_{t\ i})$› have $lfinite\ (\pi_c(inf\text{-}llist\ t))$ using proj-finite2 by simp
  then obtain $z$ where $\forall\, n''{>}z.\ \neg\ \wr c\wr_{t\ n''}$ using proj-finite-bound by blast

  have $\forall\, x::nat{\geq}\ {}_{c}{\downarrow}_{t}(n).\ \gamma\ (lnth\ (\pi_c inf\text{-}llist\ t\ @_l\ inf\text{-}llist\ t'))\ x$
  proof
    fix $x::nat$ show $(x{\geq}_{c}{\downarrow}_{t}(n)) \longrightarrow \gamma\ (lnth\ (\pi_c inf\text{-}llist\ t\ @_l\ inf\text{-}llist\ t'))\ x$
    proof
      assume $x{\geq}_{c}{\downarrow}_{t}(n)$
      moreover from ‹$\neg(\exists\, i{\geq}n.\ \wr c\wr_{t\ i})$› have $\langle c \wedge t\rangle \leq n$
        using ‹$\exists\, i.\ \wr c\wr_{t\ i}$› lActive-less by auto
      ultimately have ${}_{c}{\uparrow}_{t}(x) \geq n$ using p2c-mono-c2p by simp
      with assms have $eval\ c\ t\ t'\ ({}_{c}{\uparrow}_{t}(x))\ \gamma$ by simp
      moreover have $\neg\ (\exists\, i'{\geq}_{c}{\uparrow}_{t}(x).\ \wr c\wr_{t\ i'})$
      proof -
        from ‹$lfinite\ (\pi_c(inf\text{-}llist\ t))$› ‹$\exists\, i.\ \wr c\wr_{t\ i}$›
          have ${}_{c}{\uparrow}_{t}(the\text{-}enat\ (llength\ (\pi_c(inf\text{-}llist\ t)))) = Suc\ (\langle c \wedge t\rangle)$
          using bhv2cnf-lActive by blast
        moreover from ‹$\neg(\exists\, i{\geq}n.\ \wr c\wr_{t\ i})$› have $n{>}\langle c \wedge t\rangle$
          by (meson ‹$\exists\, i.\ \wr c\wr_{t\ i}$› lActive-active leI le-eq-less-or-eq)
        hence $n{\geq}Suc\ (\langle c \wedge t\rangle)$ by simp
        with ‹$n{\geq}Suc(\langle c \wedge t\rangle)$› ‹${}_{c}{\uparrow}_{t}(x) \geq n$› have ${}_{c}{\uparrow}_{t}(x) \geq Suc\ (\langle c \wedge t\rangle)$ by simp
        hence ${}_{c}{\uparrow}_{t}(x) > \langle c \wedge t\rangle$ by simp
        with ‹$\forall\, n''{>}z.\ \neg\ \wr c\wr_{t\ n''}$› show ?thesis using lActive-greater-active-all by simp
      qed
      ultimately have $\gamma\ (lnth\ ((\pi_c(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ ({}_{c}{\downarrow}_{t}({}_{c}{\uparrow}_{t}(x)))$
        using validCE-cont[of $c\ t\ {}_{c}{\uparrow}_{t}(x)\ t'\ \gamma$] ‹$\exists\, i.\ \wr c\wr_{t\ i}$› by blast
      moreover have $x \geq the\text{-}enat\ (llength\ (\pi_c(inf\text{-}llist\ t))) - 1$
        using ‹${}_{c}{\downarrow}_{t}(n) \leq x$› cnf2bhv-def by auto
      ultimately show $\gamma\ (lnth\ ((\pi_c(inf\text{-}llist\ t))\ @_l\ (inf\text{-}llist\ t')))\ x$
        using cnf2bhv-bhv2cnf by simp
    qed
  qed
  with ‹$\exists\, i.\ \wr c\wr_{t\ i}$› ‹$\neg(\exists\, i{\geq}n.\ \wr c\wr_{t\ i})$› have $eval\ c\ t\ t'\ n\ (\lambda t\ n.\ \forall\, n'{\geq}n.\ \gamma\ t\ n')$
    using validCI-cont[of $c\ t\ n\ \lambda\ t\ n.\ \forall\, n'{\geq}n.\ \gamma\ t\ n'\ t'$] by simp
  thus ?thesis using glob-def by simp
next
  assume $\neg(\exists\, i.\ \wr c\wr_{t\ i})$
  with assms have $\forall\, n'{\geq}n.\ \gamma\ (lnth\ (\pi_c inf\text{-}llist\ t\ @_l\ inf\text{-}llist\ t'))\ n'$
    using validCE-not-act by blast
```

```
  with ‹¬(∃ i. ≀c≀_t i)› have eval c t t' n (λt n. ∀ n'≥n. γ t n')
    using validCI-not-act[where γ=λ t n. ∀ n'≥n. γ t n'] by blast
  thus ?thesis using glob-def by simp
qed

lemma globEA[elim]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
    and n'::nat
  assumes ∃ i≥n. ≀c≀_t i
    and eval c t t' n (□_b(γ))
    and n'≥⟨c ⇐ t⟩_n
  shows eval c t t' n' γ
proof (cases)
  assume ∃ i≥n'. ≀c≀_t i
  with ‹n'≥⟨c ⇐ t⟩_n› have the-enat (⟨c #_n' inf-llist t⟩) ≥ the-enat (⟨c #_n inf-llist t⟩)
    using nAct-mono-lNact ‹∃ i≥n. ≀c≀_t i› by simp
  moreover from ‹eval c t t' n (□_b(γ))› have eval c t t' n (λt n. ∀ n'≥n. γ t n')
    using glob-def by simp
  hence ∀ x≥the-enat ⟨c #_enat n inf-llist t⟩. γ (lnth (π_c inf-llist t @_l inf-llist t')) x
    using validCE-act ‹∃ i≥n. ≀c≀_t i› by blast
  ultimately have
    γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat (⟨c #_n' inf-llist t⟩)) by simp
  with ‹∃ i≥n'. ≀c≀_t i› show ?thesis using validCI-act by blast
next
  assume ¬(∃ i≥n'. ≀c≀_t i)
  from ‹eval c t t' n (□_b(γ))› have eval c t t' n (λt n. ∀ n'≥n. γ t n') using glob-def by simp
  hence ∀ x≥the-enat ⟨c #_enat n inf-llist t⟩. γ (lnth (π_c inf-llist t @_l inf-llist t')) x
    using validCE-act ‹∃ i≥n. ≀c≀_t i› by blast
  moreover have _c↓_t(n') ≥ the-enat (⟨c #_n inf-llist t⟩)
  proof –
    have ⟨c #_n inf-llist t⟩≤llength (π_c(inf-llist t)) using nAct-le-proj by metis
    moreover from ‹¬ (∃ i≥n'. ≀c≀_t i)› have llength (π_c(inf-llist t))≠∞
      by (metis llength-eq-infty-conv-lfinite lnth-inf-llist proj-finite2)
    ultimately have the-enat(⟨c #_n inf-llist t⟩)≤the-enat(llength (π_c(inf-llist t))) by simp
    moreover from ‹∃ i≥n. ≀c≀_t i› ‹¬ (∃ i≥n'. ≀c≀_t i)› have n'>⟨c ∧ t⟩
      using lActive-active by (meson leI le-eq-less-or-eq)
    hence _c↓_t(n') > the-enat (llength (π_c(inf-llist t))) – 1
      using cnf2bhv-greater-llength by simp
    ultimately show ?thesis by simp
  qed
  ultimately have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (_c↓_t(n')) by simp
  with ‹∃ i≥n. ≀c≀_t i› ‹¬(∃ i≥n'. ≀c≀_t i)› show ?thesis using validCI-cont by blast
qed

lemma globEANow:
  fixes c t t' n i γ
```

```
  assumes n ≤ i
    and ⦃c⦄_{t i}
    and eval c t t' n (□_b γ)
  shows eval c t t' i γ
proof −
  from ‹⦃c⦄_{t i}› ‹n ≤ i› have ∃ i≥n. ⦃c⦄_{t i} by auto
  moreover from ‹n ≤ i› have ⟨c ⇐ t⟩_n ≤ i using dual-order.trans lNactLe by blast
  ultimately show ?thesis using globEA[of n c t t' γ i] ‹eval c t t' n (□_b γ)› by simp
qed

lemma globEN[elim]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
    and n'::nat
  assumes ¬(∃ i≥n. ⦃c⦄_{t i})
    and eval c t t' n (□_b(γ))
    and n'≥n
  shows eval c t t' n' γ
proof cases
  assume ∃ i. ⦃c⦄_{t i}
  moreover from ‹eval c t t' n (□_b(γ))› have eval c t t' n (λt n. ∀ n'≥n. γ t n')
    using glob-def by simp
  ultimately have ∀ x≥c↓_t n. γ (lnth (π_c inf-llist t @_l inf-llist t')) x
    using validCE-cont[of c t n t' λt n. ∀ n'≥n. γ t n'] ‹¬(∃ i≥n. ⦃c⦄_{t i})› by blast
  moreover from ‹n'≥n› have c↓_t(n') ≥ c↓_t(n) using cnf2bhv-mono by simp
  ultimately have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (c↓_t(n')) by simp
  moreover from ‹¬(∃ i≥n. ⦃c⦄_{t i})› ‹n'≥n› have ¬(∃ i≥n'. ⦃c⦄_{t i}) by simp
  ultimately show ?thesis using validCI-cont ‹∃ i. ⦃c⦄_{t i}› by blast
next
  assume ¬(∃ i. ⦃c⦄_{t i})
  moreover from ‹eval c t t' n (□_b(γ))› have eval c t t' n (λt n. ∀ n'≥n. γ t n')
    using glob-def by simp
  ultimately have ∀ n'≥n. γ (lnth (π_c inf-llist t @_l inf-llist t')) n'
    using ‹¬(∃ i. ⦃c⦄_{t i})› validCE-not-act[where γ=λt n. ∀ n'≥n. γ t n'] by blast
  with ‹¬(∃ i. ⦃c⦄_{t i})› ‹n'≥n› show ?thesis using validCI-not-act by blast
qed
```

D.20.5 Until Operator

```
definition until :: ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool)
  ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) (infixl 𝔘_b 21)
  where γ' 𝔘_b γ ≡ λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n')

lemma untilIA[intro]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
```

```
    and n::nat
    and n'::nat
  assumes ∃ i≥n. ≀c≀_t i
    and n'≥⟨c ⇐ t⟩_n
    and ⟦∃ i≥n'. ≀c≀_t i⟧ ⟹ ∃ n''≥⟨c ⇐ t⟩_n'. n''≤ ⟨c → t⟩_n' ∧ eval c t t' n'' γ ∧
      (∀ n'''≥⟨c → t⟩_n. n'''< ⟨c ⇐ t⟩_n''
      ⟶ (∃ n''''≥⟨c ⇐ t⟩_n'''. n''''≤ ⟨c → t⟩_n''' ∧ eval c t t' n'''' γ'))
    and ⟦¬(∃ i≥n'. ≀c≀_t i)⟧ ⟹ eval c t t' n' γ ∧
      (∀ n''≥⟨c → t⟩_n. n''< n'
      ⟶ ((∃ i≥n''. ≀c≀_t i) ∧ (∃ n'''≥⟨c ⇐ t⟩_n''. n'''≤ ⟨c → t⟩_n'' ∧ eval c t t' n''' γ')) ∨
      (¬(∃ i≥n''. ≀c≀_t i) ∧ eval c t t' n'' γ'))
  shows eval c t t' n (γ' 𝔘_b γ)
proof cases
  assume ∃ i'≥n'. ≀c≀_t i'
   with assms(3) obtain n'' where n''≥⟨c ⇐ t⟩_n' and n''≤ ⟨c → t⟩_n' and eval c t t' n'' γ
and
    a1: ∀ n'''≥⟨c → t⟩_n. n'''< ⟨c ⇐ t⟩_n''
    ⟶ (∃ n''''≥⟨c ⇐ t⟩_n'''. n''''≤ ⟨c → t⟩_n''' ∧ eval c t t' n'''' γ') by blast
  hence ∃ i'≥n''. ≀c≀_t i' using ‹∃ i'≥n'. ≀c≀_t i'› nxtActI by blast
  with ‹eval c t t' n'' γ› have
    γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat (⟨c #_n'' inf-llist t⟩))
    using validCE-act by blast
  moreover have the-enat (⟨c #_n inf-llist t⟩) ≤ the-enat (⟨c #_n'' inf-llist t⟩)
  proof –
    from ‹⟨c ⇐ t⟩_n≤n'› have ⟨c #_n inf-llist t⟩ ≤ ⟨c #_n' inf-llist t⟩
      using nAct-mono-lNact by simp
    moreover from ‹⟨c ⇐ t⟩_n'≤n''› have ⟨c #_n' inf-llist t⟩ ≤ ⟨c #_n'' inf-llist t⟩
      using nAct-mono-lNact by simp
    ultimately have ⟨c #_n inf-llist t⟩ ≤ ⟨c #_n'' inf-llist t⟩ by simp
    moreover have ⟨c #_n' inf-llist t⟩ ≠ ∞ by simp
    ultimately show ?thesis by simp
  qed
  moreover have ∃ i'≥n. ≀c≀_t i'
  proof –
    from ‹∃ i'≥n'. ≀c≀_t i'› obtain i' where i'≥n' and ≀c≀_t i' by blast
    with ‹n'≥⟨c ⇐ t⟩_n› have i'≥ n using lNactGe le-trans by blast
    with ‹≀c≀_t i'› show ?thesis by blast
  qed
  moreover have ∀ n'≥the-enat ⟨c #_n inf-llist t⟩. n' < (the-enat ⟨c #_enat n'' inf-llist t⟩)
    ⟶ γ' (lnth (π_c inf-llist t @_l inf-llist t')) n'
  proof
    fix x::nat show x≥the-enat (⟨c #_n inf-llist t⟩)
      ⟶ x < (the-enat ⟨c #_enat n'' inf-llist t⟩) ⟶ γ' (lnth (π_c inf-llist t @_l inf-llist t')) x
    proof (rule HOL.impI[OF HOL.impI])
      assume x≥the-enat (⟨c #_n inf-llist t⟩) and x < (the-enat ⟨c #_enat n'' inf-llist t⟩)
      moreover have the-enat (⟨c #_enat n'' inf-llist t⟩) = ⟨c #_enat n'' inf-llist t⟩ by simp
      ultimately have x<llength (π_c(inf-llist t)) using nAct-le-proj[of c n'' inf-llist t]
        by (metis enat-ord-simps(2) less-le-trans)
      hence x<llength (π_c(inf-llist t)) by simp
```

then obtain n'::*nat* **where** $x{=}\langle c\ \#_{n'}\ \textit{inf-llist}\ t\rangle$ **using** *nAct-exists* **by** *blast*
moreover from ‹$\textit{enat}\ x < \textit{llength}\ (\pi_c(\textit{inf-llist}\ t))$› ‹$\textit{enat}\ x = \langle c\ \#_{\textit{enat}\ n'}\ \textit{inf-llist}\ t\rangle$›
have $\exists\, i{\ge}n'.\ \wr c\wr_{t\ i}$ **using** *nAct-less-llength-active* **by** *force*
then obtain i **where** $i{\ge}n'$ **and** $\wr c\wr_{t\ i}$ **and** $\neg\ (\exists\, k{\ge}n'.\ k < i \wedge \wr c\wr_{t\ k})$
using *nact-exists* **by** *blast*
moreover have $\textit{enat}\ i - 1 < \textit{llength}\ (\textit{inf-llist}\ t)$ **by** (*simp add*: *one-enat-def*)
ultimately have $x{=}\langle c\ \#_i\ \textit{inf-llist}\ t\rangle$ **using** *one-enat-def nAct-not-active-same* **by** *simp*
moreover have $\langle c\ \#_i\ \textit{inf-llist}\ t\rangle{\neq}\infty$ **by** *simp*
ultimately have $x{=}\textit{the-enat}(\langle c\ \#_i\ \textit{inf-llist}\ t\rangle)$ **by** *fastforce*
from ‹$x{\ge}\textit{the-enat}\ (\langle c\ \#_n\ \textit{inf-llist}\ t\rangle)$› ‹$x{=}\textit{the-enat}(\langle c\ \#_i\ \textit{inf-llist}\ t\rangle)$›
have $\textit{the-enat}\ (\langle c\ \#_i\ \textit{inf-llist}\ t\rangle){\ge}\textit{the-enat}\ (\langle c\ \#_n\ \textit{inf-llist}\ t\rangle)$ **by** *simp*
with ‹$\wr c\wr_{t\ i}$› **have** $i{\ge}\langle c \rightarrow t\rangle_n$ **using** *active-geq-nxtAct* **by** *simp*
moreover have $i < \langle c \Leftarrow t\rangle_{n''}$
proof –
have $\textit{the-enat}\ \langle c\ \#_{\textit{enat}\ n''}\textit{inf-llist}\ t\rangle = \langle c\ \#_{\textit{enat}\ n''}\textit{inf-llist}\ t\rangle$ **by** *simp*
with ‹$x < (\textit{the-enat}\ \langle c\ \#_{\textit{enat}\ n''}\textit{inf-llist}\ t\rangle)$› **and** ‹$x{=}\langle c\ \#_i\ \textit{inf-llist}\ t\rangle$› **have**
$\langle c\ \#_i\ \textit{inf-llist}\ t\rangle{<}\langle c\ \#_{n''}\ \textit{inf-llist}\ t\rangle$ **by** (*metis enat-ord-simps*(*2*))
hence $i{<}n''$ **using** *nAct-strict-mono-back*[*of* $c\ i$ *inf-llist* $t\ n''$] **by** *auto*
with ‹$\wr c\wr_{t\ i}$› **show** *?thesis* **using** *lNact-notActive leI* **by** *blast*
qed
ultimately obtain n'' **where** $\textit{eval}\ c\ t\ t'\ n''\ \gamma'$ **and** $n''{\ge}\langle c \Leftarrow t\rangle_i$ **and** $n''{\le}\langle c \rightarrow t\rangle_i$
using *a1* **by** *auto*
moreover have $\exists\, i'{\ge}n''.\ \wr c\wr_{t\ i'}$
using ‹$\wr c\wr_{t\ i}$› ‹$n''{\le}\langle c \rightarrow t\rangle_i$› *less-or-eq-imp-le nxtAct-active* **by** *auto*
ultimately have
$\gamma'\ (\textit{lnth}\ ((\pi_c(\textit{inf-llist}\ t))\ @_l\ (\textit{inf-llist}\ t')))\ (\textit{the-enat}\ (\langle c\ \#_{n''}\ \textit{inf-llist}\ t\rangle))$
using *validCE-act*[*of* $n''\ c\ t\ t'\ \gamma'$] **by** *blast*
moreover from ‹$n''{\ge}\langle c \Leftarrow t\rangle_i$› **and** ‹$n''{\le}\langle c \rightarrow t\rangle_i$›
have $\textit{the-enat}\ (\langle c\ \#_{n''}\ \textit{inf-llist}\ t\rangle){=}\textit{the-enat}\ (\langle c\ \#_i\ \textit{inf-llist}\ t\rangle)$ **using** *nAct-same* **by** *simp*
hence $\textit{the-enat}\ (\langle c\ \#_{n''}\ \textit{inf-llist}\ t\rangle) = x$
by (*simp add*: ‹$x = \textit{the-enat}\ \langle c\ \#_{\textit{enat}\ i}\textit{inf-llist}\ t\rangle$›)
ultimately show $\gamma'\ (\textit{lnth}\ ((\pi_c(\textit{inf-llist}\ t))\ @_l\ (\textit{inf-llist}\ t')))\ x$ **by** *simp*
qed
qed
ultimately have $\textit{eval}\ c\ t\ t'\ n\ (\lambda\ t\ n.\ \exists\, n''{\ge}n.\ \gamma\ t\ n'' \wedge (\forall\, n'{\ge}n.\ n' < n'' \longrightarrow \gamma'\ t\ n'))$
using *validCI-act*[**where** $\gamma{=}\lambda\ t\ n.\ \exists\, n''{\ge}n.\ \gamma\ t\ n'' \wedge (\forall\, n'{\ge}n.\ n' < n'' \longrightarrow \gamma'\ t\ n')$] **by** *blast*
thus *?thesis* **using** *until-def* **by** *simp*
next
assume $\neg(\exists\, i'{\ge}n'.\ \wr c\wr_{t\ i'})$
with *assms*(*4*) **have** $\textit{eval}\ c\ t\ t'\ n'\ \gamma$ **and** *a2*: $\forall\, n''{\ge}\langle c \rightarrow t\rangle_n.\ n''{<}\ n'$
$\longrightarrow ((\exists\, i{\ge}n''.\ \wr c\wr_{t\ i}) \wedge (\exists\, n'''{\ge}\langle c \Leftarrow t\rangle_{n''}.\ n'''{\le}\ \langle c \rightarrow t\rangle_{n''} \wedge \textit{eval}\ c\ t\ t'\ n'''\ \gamma')) \vee$
$(\neg(\exists\, i{\ge}n''.\ \wr c\wr_{t\ i}) \wedge \textit{eval}\ c\ t\ t'\ n''\ \gamma')$ **by** *auto*
with ‹$\neg(\exists\, i'{\ge}n'.\ \wr c\wr_{t\ i'})$› ‹$\textit{eval}\ c\ t\ t'\ n'\ \gamma$› ‹$\exists\, i{\ge}n.\ \wr c\wr_{t\ i}$› **have**
$\gamma\ (\textit{lnth}\ ((\pi_c(\textit{inf-llist}\ t))\ @_l\ (\textit{inf-llist}\ t')))\ ({}_c{\downarrow}_t(n'))$ **using** *validCE-cont* **by** *blast*
moreover have ${}_c{\downarrow}_t(n') \ge \textit{the-enat}\ (\langle c\ \#_n\ \textit{inf-llist}\ t\rangle)$
proof –
from ‹$(\exists\, i{\ge}n.\ \wr c\wr_{t\ i})$› ‹$\neg(\exists\, i'{\ge}n'.\ \wr c\wr_{t\ i'})$› **have** $n' \ge \langle c \wedge t\rangle$ **using** *lActive-less* **by** *auto*
hence ${}_c{\downarrow}_t(n') \ge \textit{the-enat}\ (\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1$ **using** *cnf2bhv-ge-llength* **by** *simp*
moreover have $\textit{the-enat}(\textit{llength}\ (\pi_c(\textit{inf-llist}\ t))) - 1 \ge \textit{the-enat}(\langle c\ \#_n\ \textit{inf-llist}\ t\rangle)$

```
  proof –
    from ‹∃ i≥n. ≀c≀_t_i› have llength (π_c(inf-llist t)) ≥ eSuc (⟨c #_n inf-llist t⟩)
      using nAct-llength-proj by simp
    moreover from ‹¬(∃ i'≥n'. ≀c≀_t_i')› have lfinite (π_c(inf-llist t))
      using proj-finite2[of inf-llist t] by simp
    hence llength (π_c(inf-llist t))≠∞ using llength-eq-infty-conv-lfinite by auto
    ultimately have the-enat (llength (π_c(inf-llist t))) ≥ the-enat(eSuc (⟨c #_n inf-llist t⟩))
      by simp
    moreover have ⟨c #_n inf-llist t⟩≠∞ by simp
    ultimately have the-enat (llength (π_c(inf-llist t))) ≥ Suc (the-enat (⟨c #_n inf-llist t⟩))
      using the-enat-eSuc by simp
    thus ?thesis by simp
  qed
  ultimately show ?thesis by simp
qed
moreover have ∀ x≥the-enat ⟨c #_n inf-llist t⟩. x < (c↓_t(n'))
  ⟶ γ' (lnth (π_c inf-llist t @_l inf-llist t')) x
proof
  fix x::nat show
    x≥the-enat ⟨c #_n inf-llist t⟩ ⟶ x < (c↓_t(n')) ⟶ γ' (lnth (π_c inf-llist t @_l inf-llist t')) x
  proof (rule HOL.impI[OF HOL.impI])
    assume x≥the-enat ⟨c #_n inf-llist t⟩ and x < (c↓_t(n'))
    show γ' (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) x
    proof (cases)
      assume (x ≥ llength (π_c(inf-llist t)))
      hence lfinite (π_c(inf-llist t))
        using llength-geq-enat-lfiniteD[of π_c(inf-llist t) x] by simp
      then obtain z where ∀ n''>z. ¬ ≀c≀_t_n'' using proj-finite-bound by blast
      moreover have ≀c≀_t_⟨c → t⟩_n by (simp add: ‹∃ i≥n. ≀c≀_t_i› nxtActI)
      ultimately have ⟨c ∧ t⟩≥⟨c → t⟩_n using lActive-greatest[of c t ⟨c → t⟩_n] by blast
      moreover have c↑_t(x) ≥ ⟨c ∧ t⟩ by simp
      ultimately have c↑_t(x) ≥ ⟨c → t⟩_n by arith
      moreover have ¬ (∃ i'≥c↑_t(x). ≀c≀_t_i')
      proof –
        from ‹lfinite (π_c(inf-llist t))› ‹∃ i≥n. ≀c≀_t_i›
          have c↑_t(the-enat (llength (π_c(inf-llist t)))) = Suc (⟨c ∧ t⟩)
          using bhv2cnf-lActive by blast
        moreover from ‹(x ≥ llength (π_c(inf-llist t)))›
          have x ≥ the-enat(llength (π_c(inf-llist t)))
          using the-enat-mono by fastforce
        hence c↑_t(x) ≥ c↑_t(the-enat (llength (π_c(inf-llist t))))
          using bhv2cnf-mono[of the-enat (llength (π_c(inf-llist t))) x] by simp
        ultimately have c↑_t(x) ≥ Suc (⟨c ∧ t⟩) by simp
        hence c↑_t(x) > ⟨c ∧ t⟩ by simp
        with ‹∀ n''>z. ¬ ≀c≀_t_n''› show ?thesis using lActive-greater-active-all by simp
      qed
      moreover have c↑_t x < n'
      proof –
        from ‹lfinite (π_c(inf-llist t))›
```

```
      have llength (π_c inf-llist t) = the-enat (llength (π_c inf-llist t))
      by (simp add: enat-the-enat llength-eq-infty-conv-lfinite)
    with ‹x ≥ llength (π_c(inf-llist t))› have x≥the-enat (llength (π_c inf-llist t))
      using enat-ord-simps(1) by fastforce
    moreover from ‹∃ i≥n. ≀c≀_t_i› have llength (π_c inf-llist t)≥1 using proj-one by force
    ultimately have the-enat (llength (π_c inf-llist t)) − 1 ≤ x by simp
    with ‹x < (c↓_t(n'))› show ?thesis using c2p-mono-p2c-strict by simp
  qed
  ultimately have eval c t t' (c↑_t(x)) γ' using a2 by blast
  hence γ' (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (c↓_t(c↑_t(x)))
    using validCE-cont[of c t c↑_t(x) t' γ'] ‹∃ i≥n. ≀c≀_t_i› ‹¬ (∃ i'≥c↑_t(x). ≀c≀_t_i')› by blast
  moreover from ‹(x ≥ llength (π_c(inf-llist t)))›
    have (enat x ≥ llength (π_c(inf-llist t))) by auto
  with ‹lfinite (π_c(inf-llist t))› have llength (π_c(inf-llist t))≠∞
    using llength-eq-infty-conv-lfinite by auto
  with ‹(x ≥ llength (π_c(inf-llist t)))›
    have the-enat(llength (π_c(inf-llist t))) − 1 ≤ x by auto
  ultimately show ?thesis using cnf2bhv-bhv2cnf[of c t x] by simp
next
  assume ¬(x ≥ llength (π_c(inf-llist t)))
  hence x<llength (π_c(inf-llist t)) by simp
  then obtain n''::nat where x=⟨c #_n'' inf-llist t⟩ using nAct-exists by blast
  moreover from ‹enat x < llength (π_c(inf-llist t))› ‹enat x = ⟨c #_enat n'' inf-llist t⟩›
    have ∃ i≥n''. ≀c≀_t_i using nAct-less-llength-active by force
  then obtain i where i≥n'' and ≀c≀_t_i and ¬ (∃ k≥n''. k < i ∧ ≀c≀_t_k)
    using nact-exists by blast
  moreover have enat i − 1 < llength (inf-llist t) by (simp add: one-enat-def)
  ultimately have x=⟨c #_i inf-llist t⟩ using one-enat-def nAct-not-active-same by simp
  moreover have ⟨c #_i inf-llist t⟩≠∞ by simp
  ultimately have x=the-enat(⟨c #_i inf-llist t⟩) by fastforce
  from ‹x≥the-enat (⟨c #_n inf-llist t⟩)› ‹x=the-enat(⟨c #_i inf-llist t⟩)›
  have the-enat (⟨c #_i inf-llist t⟩)≥the-enat (⟨c #_n inf-llist t⟩) by simp
  with ‹≀c≀_t_i› have i≥⟨c → t⟩_n using active-geq-nxtAct by simp
  moreover from ‹x=⟨c #_i inf-llist t⟩› ‹x < llength (π_c(inf-llist t))›
    have ∃ i'. i ≤ enat i' ∧ ≀c≀_t_i' using nAct-less-llength-active[of x c inf-llist t i] by simp
  hence ∃ i'≥i. ≀c≀_t_i' by simp
  moreover have i<n'
  proof −
    from ‹∃ i≥n. ≀c≀_t_i› ‹¬(∃ i'≥n'. ≀c≀_t_i')› have n'≥⟨c ∧ t⟩ using lActive-less by auto
    hence c↓_t(n')≥the-enat(llength (π_c(inf-llist t))) − 1 using cnf2bhv-ge-llength by simp
    with ‹x<llength (π_c(inf-llist t))› show ?thesis
      using ‹¬ (∃ i'≥n'. ≀c≀_t_i')› ‹≀c≀_t_i› le-neq-implies-less nat-le-linear by blast
  qed
  ultimately obtain n''' where eval c t t' n''' γ' and n'''≥⟨c ⇐ t⟩_i and n'''≤⟨c → t⟩_i
    using a2 by blast
  moreover from ‹≀c≀_t_i› have ≀c≀_t_⟨c → t⟩_i using nxtActI by auto
  with ‹n'''≤⟨c → t⟩_i› have ∃ i'≥n'''. ≀c≀_t_i' using less-or-eq-imp-le by blast
  ultimately have
    γ' (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat (⟨c #_n''' inf-llist t⟩))
```

```
          using validCE-act[of n''' c t t' γ'] by blast
        moreover from ‹n'''≥⟨c ⇐ t⟩_i› and ‹n'''≤⟨c → t⟩_i›
          have the-enat (⟨c #_n''' inf-llist t⟩)=the-enat (⟨c #_i inf-llist t⟩)
          using nAct-same by simp
        hence the-enat (⟨c #_n''' inf-llist t⟩) = x
          by (simp add: ‹x = the-enat ⟨c #_enat i inf-llist t⟩›)
        ultimately have γ' (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat x) by simp
        thus ?thesis by simp
      qed
    qed
  qed
  ultimately have eval c t t' n (λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n'))
    using ‹∃ i≥n. ⌇c⌇_t i›
    validCI-act[of n c t λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n') t']
    by blast
  thus ?thesis using until-def by simp
qed

lemma untilIN[intro]:
  fixes c::'id
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
    and n'::nat
  assumes ¬(∃ i≥n. ⌇c⌇_t i)
    and n'≥n
    and eval c t t' n' γ
    and a1: ⋀n''. ⟦n≤n''; n''<n'⟧ ⟹ eval c t t' n'' γ'
  shows eval c t t' n (γ' 𝔘_b γ)
proof cases
  assume ∃ i. ⌇c⌇_t i
  moreover from assms(1) assms(2) have ¬(∃ i'≥n'. ⌇c⌇_t i') by simp
  ultimately have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (_c↓_t(n'))
    using validCE-cont[of c t n' t' γ] ‹eval c t t' n' γ› by blast
  moreover from ‹n'≥n› have _c↓_t(n') ≥ _c↓_t(n) using cnf2bhv-mono by simp
  moreover have ∀ x::nat≥ _c↓_t(n). x<_c↓_t(n') ⟶ γ' (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) x
  proof (rule HOL.allI[OF HOL.impI[OF HOL.impI]])
    fix x assume x≥_c↓_t(n) and x<_c↓_t(n')

    from ‹¬(∃ i≥n. ⌇c⌇_t i)› have ⟨c ∧ t⟩ ≤ n using ‹∃ i. ⌇c⌇_t i› lActive-less by auto
    with ‹x≥_c↓_t(n)› have _c↑_t(x) ≥ n using p2c-mono-c2p by simp
    moreover from ‹⟨c ∧ t⟩ ≤ n› ‹_c↓_t(n) ≤ x› have x ≥ the-enat (llength (π_c(inf-llist t))) − 1
      using cnf2bhv-ge-llength dual-order.trans by blast
    with ‹x<_c↓_t(n')› have _c↑_t(x) < n' using c2p-mono-p2c-strict[of c t x n'] by simp
    moreover from ‹¬ (∃ i≥n. ⌇c⌇_t i)› ‹_c↑_t(x) ≥ n› have ¬ (∃ i''≥_c↑_t(x). ⌇c⌇_t i'') by auto
    ultimately have eval c t t' (_c↑_t(x)) γ' using a1[of _c↑_t(x)] by simp
    with ‹¬ (∃ i''≥_c↑_tx. ⌇c⌇_t i'')›
      have γ' (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (_c↓_t(_c↑_t(x)))
      using validCE-cont[of c t _c↑_t(x) t' γ'] ‹∃ i. ⌇c⌇_t i› by blast
```

```
    moreover have x ≥ the-enat (llength (π_c(inf-llist t))) − 1
      using ‹c↓_t(n) ≤ x› cnf2bhv-def by auto
    ultimately show γ' (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (x)
      using cnf2bhv-bhv2cnf by simp
  qed
  ultimately have eval c t t' n (λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n'))
    using validCI-cont[of c t n λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n') t']
    ‹∃ i. ⦃c⦄_{t i}› ‹¬(∃ i'≥n. ⦃c⦄_{t i'})› by blast
  thus ?thesis using until-def by simp
next
  assume ¬(∃ i. ⦃c⦄_{t i})
  with assms have ∃ n''≥n. γ (lnth (π_c inf-llist t @_l inf-llist t')) n'' ∧
    (∀ n'≥n. n' < n'' ⟶ γ' (lnth (π_c inf-llist t @_l inf-llist t')) n')
    using validCE-not-act by blast
  with ‹¬(∃ i. ⦃c⦄_{t i})›
    have eval c t t' n (λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n'))
    using validCI-not-act[where γ=λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n')]
    by blast
  thus ?thesis using until-def by simp
qed

lemma untilEA[elim]:
  fixes n::nat
    and n'::nat
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and c::'id
  assumes ∃ i≥n. ⦃c⦄_{t i}
    and eval c t t' n (γ' 𝔘_b γ)
  shows ∃ n'≥⟨c → t⟩_n.
    ((∃ i≥n'. ⦃c⦄_{t i}) ∧ (∀ n''≥ ⟨c ⇐ t⟩_{n'}. n''≤⟨c → t⟩_{n'} ⟶ eval c t t' n'' γ)
      ∧ (∀ n''≥⟨c ⇐ t⟩_n. n'' < ⟨c ⇐ t⟩_{n'} ⟶ eval c t t' n'' γ') ∨
    (¬(∃ i≥n'. ⦃c⦄_{t i})) ∧ eval c t t' n' γ ∧ (∀ n''≥⟨c ⇐ t⟩_n. n'' < n' ⟶ eval c t t' n'' γ'))
proof −
  from ‹eval c t t' n (γ' 𝔘_b γ)›
    have eval c t t' n (λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n')) using until-def
by simp
  with ‹∃ i≥n. ⦃c⦄_{t i}› obtain x
    where x≥the-enat ⟨c #_{enat n} inf-llist t⟩ and γ (lnth (π_c inf-llist t @_l inf-llist t')) x and
    a1: ∀ x'≥the-enat ⟨c #_{enat n} inf-llist t⟩. x' < x ⟶ γ' (lnth (π_c inf-llist t @_l inf-llist t')) x'
   using validCE-act[where γ=λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n')] by blast
  thus ?thesis
  proof (cases)
    assume x ≥ llength (π_c(inf-llist t))
    moreover from ‹(x ≥ llength (π_c(inf-llist t)))› have llength (π_c(inf-llist t))≠∞
      by (metis infinity-ileE)
    moreover from ‹∃ i≥n. ⦃c⦄_{t i}› have llength (π_c(inf-llist t))≥1
      using proj-one[of inf-llist t] by auto
    ultimately have the-enat (llength (π_c(inf-llist t))) − 1 < x
```

by (*metis One-nat-def Suc-ile-eq antisym-conv2 diff-Suc-less enat-ord-simps*(*2*)
enat-the-enat less-imp-diff-less one-enat-def)
hence $x = {}_{c}{\downarrow}_{t}({}_{c}{\uparrow}_{t}(x))$ **using** *cnf2bhv-bhv2cnf* **by** *simp*
with ‹γ (*lnth* ((π_c(*inf-llist t*)) $@_l$ (*inf-llist t'*))) x›
have γ (*lnth* ((π_c(*inf-llist t*)) $@_l$ (*inf-llist t'*))) (${}_{c}{\downarrow}_{t}({}_{c}{\uparrow}_{t}(x))$) **by** *simp*
moreover have $\neg(\exists\, i \geq {}_{c}{\uparrow}_{t}(x).\ \{c\}_{t\ i})$
proof –
from ‹$x \geq$ *llength* (π_c(*inf-llist t*))› **have** *lfinite* (π_c(*inf-llist t*))
using *llength-geq-enat-lfiniteD*[*of* π_c(*inf-llist t*) x] **by** *simp*
then obtain z **where** $\forall\, n''{>}z.\ \neg\ \{c\}_{t\ n''}$ **using** *proj-finite-bound* **by** *blast*
moreover from ‹*the-enat* (*llength* (π_c(*inf-llist t*))) $-$ *1* $< x$› **have** $\langle c \wedge t\rangle < {}_{c}{\uparrow}_{t}(x)$
using *bhv2cnf-greater-lActive* **by** *simp*
ultimately show *?thesis* **using** *lActive-greater-active-all* **by** *simp*
qed
ultimately have *eval c t t'* (${}_{c}{\uparrow}_{t}x$) γ
using ‹$\exists\, i \geq n.\ \{c\}_{t\ i}$› *validCI-cont*[*of c t* ${}_{c}{\uparrow}_{t}(x)$] **by** *blast*
moreover have ${}_{c}{\uparrow}_{t}(x) \geq \langle c \rightarrow t\rangle_n$
proof –
from ‹$x \geq$ *llength* (π_c(*inf-llist t*))› **have** *lfinite* (π_c(*inf-llist t*))
using *llength-geq-enat-lfiniteD*[*of* π_c(*inf-llist t*) x] **by** *simp*
then obtain z **where** $\forall\, n''{>}z.\ \neg\ \{c\}_{t\ n''}$ **using** *proj-finite-bound* **by** *blast*
moreover from ‹$\exists\, i \geq n.\ \{c\}_{t\ i}$› **have** $\{c\}_{t\ \langle c \rightarrow t\rangle_n}$ **using** *nxtActI* **by** *simp*
ultimately have $\langle c \wedge t\rangle \geq \langle c \rightarrow t\rangle_n$ **using** *lActive-greatest* **by** *fastforce*
moreover have ${}_{c}{\uparrow}_{t}(x) \geq \langle c \wedge t\rangle$ **by** *simp*
ultimately show ${}_{c}{\uparrow}_{t}(x) \geq \langle c \rightarrow t\rangle_n$ **by** *arith*
qed
moreover have $\forall\, n'' \geq \langle c \Leftarrow t\rangle_n.\ n'' < ({}_{c}{\uparrow}_{t}x) \longrightarrow$ *eval c t t' n''* γ'
proof
fix n'' **show** $\langle c \Leftarrow t\rangle_n \leq n'' \longrightarrow n'' < {}_{c}{\uparrow}_{t}x \longrightarrow$ *eval c t t' n''* γ'
proof (*rule HOL.impI*[*OF HOL.impI*])
assume $\langle c \Leftarrow t\rangle_n \leq n''$ **and** $n'' < {}_{c}{\uparrow}_{t}x$
show *eval c t t' n''* γ'
proof *cases*
assume $\exists\, i \geq n''.\ \{c\}_{t\ i}$
with ‹$n'' \geq \langle c \Leftarrow t\rangle_n$› **have** *the-enat* ($\langle c \ \#_{n''}$ *inf-llist t*$\rangle$) $\geq$ *the-enat* ($\langle c \ \#_{n}$ *inf-llist t*$\rangle$)
using *nAct-mono-lNact* ‹$\exists\, i \geq n.\ \{c\}_{t\ i}$› **by** *simp*
moreover have *the-enat* ($\langle c \ \#_{n''}$ *inf-llist t*$\rangle$)$<x$
proof –
from ‹$\exists\, i \geq n''.\ \{c\}_{t\ i}$› **have** *eSuc* $\langle c \ \#_{enat\ n''}$*inf-llist t*$\rangle \leq$ *llength* (π_c*inf-llist t*)
using *nAct-llength-proj* **by** *auto*
with ‹$x \geq$ *llength* (π_c(*inf-llist t*))› **have** *eSuc* $\langle c \ \#_{enat\ n''}$*inf-llist t*$\rangle \leq x$ **by** *simp*
moreover have $\langle c \ \#_{enat\ n''}$*inf-llist t*$\rangle \neq \infty$ **by** *simp*
ultimately have *Suc* (*the-enat*($\langle c \ \#_{enat\ n''}$*inf-llist t*$\rangle$)) $\leq x$
by (*metis enat.distinct*(*2*) *the-enat.simps the-enat-eSuc the-enat-mono*)
thus *?thesis* **by** *simp*
qed
ultimately have
γ' (*lnth* ((π_c(*inf-llist t*)) $@_l$ (*inf-llist t'*))) (*the-enat* ($\langle c \ \#_{n''}$ *inf-llist t*$\rangle$))
using *a1* **by** *auto*

```
        with ‹∃i≥n''. ≀c≀t i› show ?thesis using validCI-act by blast
      next
        assume ¬(∃i≥n''. ≀c≀t i)
        moreover have c↓t(n'') ≥ the-enat (⟨c #n inf-llist t⟩)
        proof −
          have ⟨c #n inf-llist t⟩≤llength (πc(inf-llist t)) using nAct-le-proj by metis
          moreover from ‹¬ (∃i≥n''. ≀c≀t i)› have llength (πc(inf-llist t))≠∞
            by (metis llength-eq-infty-conv-lfinite lnth-inf-llist proj-finite2)
          ultimately have the-enat(⟨c #n inf-llist t⟩)≤the-enat(llength (πc(inf-llist t)))
            by simp
          moreover from ‹∃i≥n. ≀c≀t i› ‹¬ (∃i≥n''. ≀c≀t i)› have n''>⟨c ∧ t⟩
            using lActive-active by (meson leI le-eq-less-or-eq)
          hence c↓t(n'') > the-enat (llength (πc(inf-llist t))) − 1
            using cnf2bhv-greater-llength by simp
          ultimately show ?thesis by simp
        qed
        moreover from ‹¬(∃i≥n''. ≀c≀t i)› have ⟨c ∧ t⟩ ≤ n''
          using assms(1) lActive-less by auto
          with ‹n'' < c↑tx› have c↓t(n'')<x using p2c-mono-c2p-strict by simp
        ultimately have γ' (lnth ((πc(inf-llist t)) @l (inf-llist t'))) (c↓t(n''))
          using a1 by auto
       with ‹∃i≥n. ≀c≀t i› ‹¬(∃i≥n''. ≀c≀t i)› show ?thesis using validCI-cont by blast
      qed
    qed
  qed
  ultimately show ?thesis using ‹¬(∃i≥c↑t(x). ≀c≀t i)› by blast
next
  assume ¬(x ≥ llength (πc(inf-llist t)))
  hence x<llength (πc(inf-llist t)) by simp
  then obtain n'::nat where x=⟨c #n' inf-llist t⟩ using nAct-exists by blast
  with ‹enat x < llength (πc(inf-llist t))› have ∃i≥n'. ≀c≀t i
    using nAct-less-llength-active by force
  then obtain i where i≥n' and ≀c≀t i and ¬ (∃k≥n'. k < i ∧ ≀c≀t k)
    using nact-exists by blast
  moreover have (∀n''≥ ⟨c ⇐ t⟩i. n''≤⟨c → t⟩i ⟶ eval c t t' n'' γ)
  proof
    fix n'' show ⟨c ⇐ t⟩i ≤ n'' ⟶ n'' ≤ ⟨c → t⟩i ⟶ eval c t t' n'' γ
    proof(rule HOL.impI[OF HOL.impI])
      assume ⟨c ⇐ t⟩i ≤ n'' and n'' ≤ ⟨c → t⟩i
      hence the-enat (⟨c #enat i inf-llist t⟩) = the-enat (⟨c #enat n'' inf-llist t⟩)
        using nAct-same by simp
      moreover from ‹≀c≀t i› have ≀c≀t ⟨c → t⟩i using nxtActI by auto
      with ‹n'' ≤ ⟨c → t⟩i› have ∃i≥n''. ≀c≀t i using dual-order.strict-implies-order by auto
      moreover have
        γ (lnth ((πc(inf-llist t)) @l (inf-llist t'))) (the-enat (⟨c #enat i inf-llist t⟩))
      proof −
        have enat i − 1 < llength (inf-llist t) by (simp add: one-enat-def)
        with ‹x=⟨c #n' inf-llist t⟩› ‹i≥n'› ‹¬ (∃k≥n'. k < i ∧ ≀c≀t k)›
          have x=⟨c #i inf-llist t⟩
```

```
        using one-enat-def nAct-not-active-same by simp
      moreover have ⟨c #_i inf-llist t⟩≠∞ by simp
      ultimately have x=the-enat(⟨c #_i inf-llist t⟩) by fastforce
      thus ?thesis using ‹γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) x› by blast
    qed
    with ‹the-enat (⟨c #_enat i inf-llist t⟩) = the-enat (⟨c #_enat n'' inf-llist t⟩)› have
      γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat (⟨c #_enat n'' inf-llist t⟩)) by simp
    ultimately show eval c t t' n'' γ using validCI-act by blast
  qed
qed
moreover have i≥⟨c → t⟩_n
proof −
  have enat i − 1 < llength (inf-llist t) by (simp add: one-enat-def)
  with ‹x=⟨c #_n' inf-llist t⟩› ‹i≥n'› ‹¬ (∃k≥n'. k < i ∧ ≀c≀_t k)› have x=⟨c #_i inf-llist t⟩
    using one-enat-def nAct-not-active-same by simp
  moreover have ⟨c #_i inf-llist t⟩≠∞ by simp
  ultimately have x=the-enat(⟨c #_i inf-llist t⟩) by fastforce
  with ‹x≥the-enat (⟨c #_n inf-llist t⟩)›
    have the-enat (⟨c #_i inf-llist t⟩)≥the-enat (⟨c #_n inf-llist t⟩) by simp
  with ‹≀c≀_t i› show ?thesis using active-geq-nxtAct by simp
qed
moreover have ∀n''≥⟨c ⇐ t⟩_n. n'' < ⟨c ⇐ t⟩_i ⟶ eval c t t' n'' γ'
proof
  fix n'' show ⟨c ⇐ t⟩_n ≤ n'' ⟶ n'' < ⟨c ⇐ t⟩_i ⟶ eval c t t' n'' γ'
  proof (rule HOL.impI[OF HOL.impI])
    assume ⟨c ⇐ t⟩_n ≤ n'' and n'' < ⟨c ⇐ t⟩_i
    moreover have ⟨c ⇐ t⟩_i≤i by simp
    ultimately have ∃i≥n''. ≀c≀_t i using ‹≀c≀_t i› by (meson less-le less-le-trans)
    with ‹n''≥⟨c ⇐ t⟩_n› have the-enat (⟨c #_n'' inf-llist t⟩) ≥ the-enat (⟨c #_n inf-llist t⟩)
      using nAct-mono-lNact ‹∃i≥n. ≀c≀_t i› by simp
    moreover have the-enat (⟨c #_n'' inf-llist t⟩) < x
    proof −
      from ‹n'' < ⟨c ⇐ t⟩_i› ‹⟨c ⇐ t⟩_i ≤ i› have n'' < i
        using dual-order.strict-trans1 by arith
      with ‹n'' < ⟨c ⇐ t⟩_i› have ∃i'≥n''. i' < i ∧ ≀c≀_t i'
        using lNact-least[of i n''] by fastforce
      hence ⟨c #_n'' inf-llist t⟩ < ⟨c #_i inf-llist t⟩ using nAct-less by auto
      moreover have enat i − 1 < llength (inf-llist t) by (simp add: one-enat-def)
      with ‹x=⟨c #_n' inf-llist t⟩› ‹i≥n'› ‹¬ (∃k≥n'. k < i ∧ ≀c≀_t k)›
        have x=⟨c #_i inf-llist t⟩
        using one-enat-def nAct-not-active-same by simp
      moreover have ⟨c #_n'' inf-llist t⟩≠∞ by simp
      ultimately show ?thesis by (metis enat-ord-simps(2) enat-the-enat)
    qed
    ultimately have
      γ' (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (the-enat (⟨c #_n'' inf-llist t⟩))
      using a1 by auto
    with ‹∃i≥n''. ≀c≀_t i› show eval c t t' n'' γ' using validCI-act by blast
  qed
```

```
    qed
    ultimately show ?thesis using ‹≀c≀_t i› by auto
  qed
qed

lemma untilEN[elim]:
  fixes n::nat
    and n'::nat
    and t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and c::'id
  assumes ∄ i. i≥n ∧ ≀c≀_t i
    and eval c t t' n (γ' 𝔘_b γ)
  shows ∃ n'≥n. eval c t t' n' γ ∧
    (∀ n''≥n. n'' < n' ⟶ eval c t t' n'' γ')
proof cases
  assume ∃ i. ≀c≀_t i
  moreover from ‹eval c t t' n (γ' 𝔘_b γ)›
    have eval c t t' n (λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n'))
    using until-def by simp
  ultimately have ∃ n''≥_c↓_t(n). γ (lnth (π_c inf-llist t @_l inf-llist t')) n'' ∧
    (∀ n'≥_c↓_t(n). n' < n'' ⟶ γ' (lnth (π_c inf-llist t @_l inf-llist t')) n')
    using validCE-cont[where γ=λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n')]
    ‹∄ i. i≥n ∧ ≀c≀_t i› by blast
  then obtain x where x≥_c↓_t(n) and γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) x
    and ∀ x'≥_c↓_t(n). x'<x ⟶ γ' (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) x' by auto
  moreover from ‹¬(∃ i≥n. ≀c≀_t i)› have the-enat (llength (π_c(inf-llist t))) − 1 < x
  proof −
    have ⟨c ∧ t⟩ < n
    proof (rule ccontr)
      assume ¬⟨c ∧ t⟩ < n
      hence ⟨c ∧ t⟩ ≥ n by simp
      moreover from ‹∃ i. ≀c≀_t i› ‹¬ (∃ i≥n. ≀c≀_t i)› have ≀c≀_t ⟨c ∧ t⟩
        using lActive-active less-or-eq-imp-le by blast
      ultimately show False using ‹¬ (∃ i≥n. ≀c≀_t i)› by simp
    qed
    hence the-enat (llength (π_c(inf-llist t))) − 1 < _c↓_t(n)
      using cnf2bhv-greater-llength by simp
    with ‹x≥_c↓_t(n)› show ?thesis by simp
  qed
  hence x = _c↓_t(_c↑_t(x)) using cnf2bhv-bhv2cnf by simp
  ultimately have γ (lnth ((π_c(inf-llist t)) @_l (inf-llist t'))) (_c↓_t(_c↑_t(x))) by simp
  moreover from ‹¬(∃ i≥n. ≀c≀_t i)› have ¬(∃ i≥_c↑_t(x). ≀c≀_t i)
  proof −
    from ‹¬(∃ i≥n. ≀c≀_t i)› have lfinite (π_c(inf-llist t)) using proj-finite2 by simp
    then obtain z where ∀ n''>z. ¬ ≀c≀_t n'' using proj-finite-bound by blast
    moreover from ‹the-enat (llength (π_c(inf-llist t))) − 1 < x› have ⟨c ∧ t⟩ < _c↑_t(x)
      using bhv2cnf-greater-lActive by simp
    ultimately show ?thesis using lActive-greater-active-all by simp
```

```
  qed
  ultimately have eval c t t' (c↑t(x)) γ using validCI-cont ‹∃ i. ≀c≀t i› by blast
  moreover from ‹∃ i. ≀c≀t i› ‹¬(∃ i≥n. ≀c≀t i)› have ⟨c ∧ t⟩ ≤ n
    using lActive-less[of c t - n] by auto
  with ‹x≥c↓t(n)› have n ≤ c↑t(x) using p2c-mono-c2p by blast
  moreover have ∀ n''≥n. n'' < c↑t(x) ⟶ eval c t t' n'' γ'
  proof (rule HOL.allI[OF HOL.impI[OF HOL.impI]])
    fix n'' assume n ≤ n'' and n'' < c↑t(x)
    hence c↓t(n'')≥c↓t(n) using cnf2bhv-mono by simp
    moreover have n''<c↑t(x) by (simp add: ‹n'' < c↑tx›)
    with ‹⟨c ∧ t⟩ ≤ n› ‹n ≤ n''› have c↓t(n'')<c↓t(c↑t(x))
      using cnf2bhv-mono-strict by simp
    with ‹x = c↓t(c↑t(x))› have c↓t(n'')< x by simp
    ultimately have γ' (lnth ((πc(inf-llist t)) @l (inf-llist t'))) (c↓t(n''))
      using ‹∀ x'≥c↓t(n). x'<x ⟶ γ' (lnth ((πc(inf-llist t)) @l (inf-llist t'))) x'› by simp
    moreover from ‹n ≤ n''› have ∄ i. i≥n'' ∧ ≀c≀t i using ‹∄ i. i≥n ∧ ≀c≀t i› by simp
    ultimately show eval c t t' n'' γ' using validCI-cont using ‹∃ i. ≀c≀t i› by blast
  qed
  ultimately show ?thesis by auto
next
  assume ¬(∃ i. ≀c≀t i)
  moreover from ‹eval c t t' n (γ' 𝔘b γ)›
    have eval c t t' n (λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n'))
    using until-def by simp
  ultimately have ∃ n''≥n. γ (lnth (πcinf-llist t @l inf-llist t')) n''
    ∧ (∀ n'≥n. n' < n'' ⟶ γ' (lnth (πcinf-llist t @l inf-llist t')) n') using ‹¬(∃ i. ≀c≀t i)›
    validCE-not-act[where γ=λ t n. ∃ n''≥n. γ t n'' ∧ (∀ n'≥n. n' < n'' ⟶ γ' t n')] by blast
  with ‹¬(∃ i. ≀c≀t i)› show ?thesis using validCI-not-act by blast
qed
```

D.20.6 Weak Until

```
definition wuntil :: ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool)
  ⇒ ((nat ⇒ 'cmp) ⇒ nat ⇒ bool) (infixl 𝔚b 20)
  where γ' 𝔚b γ ≡ γ' 𝔘b γ ∨^b □b(γ')

end

end
```

D.21 Proof of Completeness

Assume $(t, t', n)\underset{c}{\models}\gamma$. We show by structural induction over γ, that $(t, t', n)\underset{c}{\models}\gamma$ can be derived using the rules presented in Sect. 5.

Case γ is a basic behavior assertion "ϕ": Since $(t, t', n)\underset{c}{\models}\gamma$ conclude $\Big(\exists i \geq n \colon \wr c \wr_{t(i)} \wedge (\Pi_c(t)^\frown t', \#_c^n(t)) \models \text{“}\phi\text{”}\Big) \vee \Big(\exists i \colon \wr c \wr_{t(i)} \wedge (\nexists i \geq n \colon \wr c \wr_{t(i)}) \wedge (\Pi_c(t)^\frown t', {}_c{\Downarrow}_t(n)) \models \text{“}\phi\text{”}\Big) \vee \Big(\nexists i \colon \wr c \wr_{t(i)} \wedge (t', n) \models \text{“}\phi\text{”}\Big)$ by Def. 15.

- Case $\exists i \geq n \colon \wr c \wr_{t(i)} \wedge (\Pi_c(t)^\frown t', \#_c^n(t)) \models \text{“}\phi\text{”}$: Since $\exists i \geq n \colon \wr c \wr_{t(i)}$ conclude $\Pi_c(t)^\frown t'(\#_c^n(t)) = \mathsf{val}(c) \cup \big(\lambda p \in \mathsf{port}(c) \colon \mathit{val}_{t(c \xrightarrow{n} t)}(c, p)\big)$. Thus, since ϕ is a basic behavior assertion and $(\Pi_c(t)^\frown t', \#_c^n(t)) \models \text{“}\phi\text{”}$ conclude $\mathsf{val}(c) \cup \big(\lambda p \in \mathsf{port}(c) \colon \mathit{val}_{t(c \xrightarrow{n} t)}(c, p)\big) \models \text{“}\phi\text{”}$. Thus, since $\exists i \geq n \colon \wr c \wr_{t(i)}$ we can apply $\mathrm{BaI_a}$ to have $(t, t', n)\underset{c}{\models}\text{“}\phi\text{”}$.
- Case $\exists i \colon \wr c \wr_{t(i)} \wedge (\nexists i \geq n \colon \wr c \wr_{t(i)}) \wedge (\Pi_c(t)^\frown t', {}_c{\Downarrow}_t(n)) \models \text{“}\phi\text{”}$: Since $\exists i \colon \wr c \wr_{t(i)} \wedge (\nexists i \geq n \colon \wr c \wr_{t(i)})$ conclude $\Pi_c(t)^\frown t'({}_c{\Downarrow}_t(n)) = t'(n - \mathit{last}(c, t) - 1)\big)$. Thus, since ϕ is a basic behavior assertion and $(\Pi_c(t)^\frown t', {}_c{\Downarrow}_t(n)) \models \text{“}\phi\text{”}$ conclude $t'\big({}_c{\Downarrow}_t(n)\big) \models \text{“}\phi\text{”}$. Thus, since $\exists i \geq n \colon \wr c \wr_{t(i)}$ we can apply $\mathrm{BaI_{n1}}$ to have $(t, t', n)\underset{c}{\models}\text{“}\phi\text{”}$.
- Case $\nexists i \colon \wr c \wr_{t(i)} \wedge (t', n) \models \text{“}\phi\text{”}$: Since ϕ is a basic behavior assertion and $(t', n) \models \text{“}\phi\text{”}$ conclude $t'(n) \models \text{“}\phi\text{”}$. Thus, since $\nexists i \colon \wr c \wr_{t(i)}$ we can apply $\mathrm{BaI_{n2}}$ to have $(t, t', n)\underset{c}{\models}\text{“}\phi\text{”}$.

Case $\gamma = \text{“}\bigcirc\gamma'\text{”}$: Since $(t, t', n)\underset{c}{\models}\gamma$ conclude $\Big(\exists i \geq n \colon \wr c \wr_{t(i)} \wedge (\Pi_c(t)^\frown t', \#_c^n(t)) \models \text{“}\bigcirc\gamma'\text{”}\Big) \vee \Big(\exists i \colon \wr c \wr_{t(i)} \wedge (\nexists i \geq n \colon \wr c \wr_{t(i)}) \wedge (\Pi_c(t)^\frown t', {}_c{\Downarrow}_t(n)) \models \text{“}\bigcirc\gamma'\text{”}\Big) \vee \Big(\nexists i \colon \wr c \wr_{t(i)} \wedge (t', n) \models \text{“}\bigcirc\gamma'\text{”}\Big)$ by Def. 15.

- Case $\exists i \geq n \colon \wr c \wr_{t(i)} \wedge (\Pi_c(t)^\frown t', \#_c^n(t)) \models \text{“}\bigcirc\gamma'\text{”}$: We show $\exists \underset{.}{i} > c \xrightarrow{n} t \colon \wr c \wr_{t(i)} \implies \exists \underset{.}{n}' \geq n \colon (\exists! n \leq \underset{.}{i} < n' \colon \wr c \wr_{t(i)}) \wedge (t, t', n')\underset{c}{\models}\text{“}\gamma'\text{”}$ and $\nexists \underset{.}{i} > c \xrightarrow{n} t \colon \wr c \wr_{t(i)} \implies (t, t', c \xrightarrow{n} t + 1)\underset{c}{\models}\text{“}\gamma'\text{”}$ to conclude $(t, t', n)\underset{c}{\models}\gamma$ by rule $\mathrm{NxtI_a}$.
    - $\exists \underset{.}{i} > c \xrightarrow{n} t \colon \wr c \wr_{t(i)} \implies \exists \underset{.}{n}' \geq n \colon (\exists! n \leq \underset{.}{i} < n' \colon \wr c \wr_{t(i)}) \wedge (t, t', n')\underset{c}{\models}\text{“}\gamma'\text{”}$: Assume $\exists \underset{.}{i} > c \xrightarrow{n} t \colon \wr c \wr_{t(i)}$. Thus, since $(\Pi_c(t)^\frown t', \#_c^n(t)) \models \text{“}\bigcirc\gamma'\text{”}$ conclude $(\Pi_c(t)^\frown t', \#_c^n(t) + 1) \models \text{“}\gamma'\text{”}$. Moreover, since $\exists \underset{.}{i} > c \xrightarrow{n} t$ have $\exists \underset{.}{n}' \geq n \colon (\exists! n \leq \underset{.}{i} < n' \colon \wr c \wr_{t(i)})$". Thus, $\#_c^{n'}(t) = \#_c^n(t) + 1$ and since $(\Pi_c(t)^\frown t', \#_c^n(t) + 1) \models \text{“}\gamma'\text{”}$ conclude $(\Pi_c(t)^\frown t', \#_c^{n'}(t)) \models \text{“}\gamma'\text{”}$. Moreover, since $\exists \underset{.}{i} > c \xrightarrow{n} t$ and $\exists! n \leq \underset{.}{i} < n' \colon \wr c \wr_{t(i)}$ conclude $\exists i \geq n'$. Thus, since $(\Pi_c(t)^\frown t', \#_c^{n'}(t)) \models \text{“}\gamma'\text{”}$ conclude $(t, t', n')\underset{c}{\models}\gamma$ by Def. 15.
    - $\nexists \underset{.}{i} > c \xrightarrow{n} t \colon \wr c \wr_{t(i)} \implies (t, t', c \xrightarrow{n} t + 1)\underset{c}{\models}\text{“}\gamma'\text{”}$: Assume $\nexists \underset{.}{i} > c \xrightarrow{n} t \colon \wr c \wr_{t(i)}$. Thus, since $\exists \underset{.}{i} \geq n \colon \wr c \wr_{t(i)}$ and $(\Pi_c(t)^\frown t', \#_c^n(t)) \models \text{“}\bigcirc\gamma'\text{”}$ conclude $(\Pi_c(t)^\frown t', {}_c{\Downarrow}_t(n) + 1) \models \text{“}\gamma'\text{”}$. Thus, $(\Pi_c(t)^\frown t', c \xrightarrow{n} t + 1) \models \text{“}\gamma'\text{”}$. Moreover,

since $\nexists \underset{.}{i} > c \overset{n}{\rightarrow} t$ have $\nexists i \geq c \overset{n}{\rightarrow} t+1 \colon \lightning c \lightning_{t(i)}$. Thus, since $\exists i \geq n \colon \lightning c \lightning_{t(i)}$ and $\left(\Pi_c(t) {}^\frown t', c \overset{n}{\rightarrow} t+1\right) \models$ "γ'" conclude $(t, t', c \overset{n}{\rightarrow} t+1) \underset{c}{\models}$"$\gamma'$" by Def. 15.

- Case $\exists i \colon \lightning c \lightning_{t(i)} \wedge \left(\nexists i \geq n \colon \lightning c \lightning_{t(i)}\right) \wedge \left(\Pi_c(t) {}^\frown t', {}_c{\Downarrow}_t(n)\right) \models$ "$\bigcirc \gamma'$": Thus, $\left(\Pi_c(t) {}^\frown t', {}_c{\Downarrow}_t(n) + 1\right) \models$ "γ'" and hence $\left(\Pi_c(t) {}^\frown t', {}_c{\Downarrow}_t(n+1)\right) \models$ "γ'". Thus, since $\exists i \colon \lightning c \lightning_{t(i)} \wedge \left(\nexists i \geq n \colon \lightning c \lightning_{t(i)}\right)$ conclude $(t, t', n+1) \underset{c}{\models}$"$\gamma'$" by Def. 15. Thus, since $\nexists i \geq n \colon \lightning c \lightning_{t(i)}$ we can apply $\mathrm{NxtI_n}$ to have $(t, t', n) \underset{c}{\models}$"$\gamma$".
- Case $\nexists i \colon \lightning c \lightning_{t(i)} \wedge (t', n) \models$ "$\bigcirc \gamma'$": Thus, $(t', n+1) \models$ "γ'" and since $\nexists i \colon \lightning c \lightning_{t(i)}$ conclude $(t, t', n+1) \models$ "γ'" by Def. 15. Thus, since $\nexists i \colon \lightning c \lightning_{t(i)}$ we can apply $\mathrm{NxtI_n}$ to have $(t, t', n) \underset{c}{\models}$"$\gamma$".

Case $\gamma =$ "$\Diamond \gamma'$": Since $(t, t', n) \underset{c}{\models} \gamma$ conclude $\Big(\exists i \geq n \colon \lightning c \lightning_{t(i)} \wedge \left(\Pi_c(t) {}^\frown t', \#_c^n(t)\right) \models$ "$\Diamond \gamma'$"$\Big) \vee \Big(\exists i \colon \lightning c \lightning_{t(i)} \wedge \left(\nexists i \geq n \colon \lightning c \lightning_{t(i)}\right) \wedge \left(\Pi_c(t) {}^\frown t', {}_c{\Downarrow}_t(n)\right) \models$ "$\Diamond \gamma'$"$\Big) \vee \Big(\nexists i \colon \lightning c \lightning_{t(i)} \wedge (t', n) \models$ "$\Diamond \gamma'$"$\Big)$ by Def. 15.

- Case $\exists i \geq n \colon \lightning c \lightning_{t(i)} \wedge \left(\Pi_c(t) {}^\frown t', \#_c^n(t)\right) \models$ "$\Diamond \gamma'$": From $\left(\Pi_c(t) {}^\frown t', \#_c^n(t)\right) \models$ "$\Diamond \gamma'$" have $\exists x \geq \#_c^n(t) \colon \left(\Pi_c(t) {}^\frown t', x\right) \models$ "γ'".
  - Case $\exists n' \colon \#_c^{n'}(t) = x$: Since $x \geq \#_c^n(t)$ it follows that $\#_c^n(t) \leq \#_c^{n'}(t)$ and thus $c \overset{n}{\Leftarrow} t \leq n'$. Thus, we show $\exists \underset{.}{i} \geq n' \colon \lightning c \lightning_{t(i)} \implies \exists c \overset{n'}{\Leftarrow} t \leq \underset{.}{n}'' \leq c \overset{n'}{\rightarrow} t \colon (t, t', n'') \underset{c}{\models}$"$\gamma'$" and $\nexists \underset{.}{i} \geq n' \colon \lightning c \lightning_{t(i)} \implies (t, t', n') \underset{c}{\models}$"$\gamma'$" to conclude $(t, t', n) \underset{c}{\models} \gamma$ by rule $\mathrm{EvtI_a}$.
    - $\exists \underset{.}{i} \geq n' \colon \lightning c \lightning_{t(i)} \implies \exists c \overset{n'}{\Leftarrow} t \leq \underset{.}{n}'' \leq c \overset{n'}{\rightarrow} t \colon (t, t', n'') \underset{c}{\models}$"$\gamma'$": Assume $\exists \underset{.}{i} \geq n' \colon \lightning c \lightning_{t(i)}$. Thus, since $\left(\Pi_c(t) {}^\frown t', x\right) \models$ "γ'" and $\#_c^{n'}(t) = x$ conclude $(t, t', n') \underset{c}{\models}$"$\gamma'$" by Def. 15. Moreover, have $c \overset{n'}{\Leftarrow} t \leq n'$ and $n' \leq c \overset{n'}{\rightarrow} t$ to conclude $\exists c \overset{n'}{\Leftarrow} t \leq \underset{.}{n}'' \leq c \overset{n'}{\rightarrow} t \colon (t, t', n'') \underset{c}{\models}$"$\gamma'$".
    - $\nexists \underset{.}{i} \geq n' \colon \lightning c \lightning_{t(i)} \implies (t, t', n') \underset{c}{\models}$"$\gamma'$": Assume $\nexists \underset{.}{i} \geq n' \colon \lightning c \lightning_{t(i)}$. Hence, since $x = \#_c^{n'}(t)$ conclude $x = {}_c{\Downarrow}_t(n')$. Thus, since $\left(\Pi_c(t) {}^\frown t', x\right) \models$ "γ'" conclude $\left(\Pi_c(t) {}^\frown t', {}_c{\Downarrow}_t(n')\right) \models$ "γ'". Thus, since $\exists i \geq n \colon \lightning c \lightning_{t(i)}$ and $\nexists \underset{.}{i} \geq n' \colon \lightning c \lightning_{t(i)}$ conclude $(t, t', n') \underset{c}{\models}$"$\gamma'$" by Def. 15.
  - Case $\neg \exists n' \colon \#_c^{n'}(t) = x$: Hence $\exists n' \colon x = {}_c{\Downarrow}_t(n')$. Hence, $n' \geq \mathit{last}(c, t)$ and thus $c \overset{n}{\Leftarrow} t \leq n'$. Moreover, since $\neg \exists n' \colon \#_c^{n'}(t) = x$ conclude $\nexists \underset{.}{i} \geq n' \colon \lightning c \lightning_{t(i)}$. Thus, we show $(t, t', n') \underset{c}{\models}$"$\gamma'$" to conclude $(t, t', n) \underset{c}{\models} \gamma$ by rule $\mathrm{EvtI_a}$: Since $\left(\Pi_c(t) {}^\frown t', x\right) \models$ "γ'" and $x = {}_c{\Downarrow}_t(n')$ conclude $\left(\Pi_c(t) {}^\frown t', {}_c{\Downarrow}_t(n')\right) \models$ "γ'". Thus, since $\exists i \geq n \colon \lightning c \lightning_{t(i)}$ and $\nexists \underset{.}{i} \geq n' \colon \lightning c \lightning_{t(i)}$ conclude $(t, t', n') \underset{c}{\models}$"$\gamma'$" by Def. 15.
- Case $\exists i \colon \lightning c \lightning_{t(i)} \wedge \left(\nexists i \geq n \colon \lightning c \lightning_{t(i)}\right) \wedge \left(\Pi_c(t) {}^\frown t', {}_c{\Downarrow}_t(n)\right) \models$ "$\Diamond \gamma'$": From $\left(\Pi_c(t) {}^\frown t', {}_c{\Downarrow}_t(n)\right) \models$ "$\Diamond \gamma'$" have $\exists x \geq {}_c{\Downarrow}_t(n) \colon \left(\Pi_c(t) {}^\frown t', x\right) \models$ "γ'". Thus, $\exists n' \geq n \colon x = {}_c{\Downarrow}_t(n')$. Since $\nexists i \geq n \colon \lightning c \lightning_{t(i)}$ and $n' \geq n$, we show $(t, t', n') \underset{c}{\models}$"$\gamma'$" to conclude $(t, t', n) \underset{c}{\models}$"$\gamma$" by rule $\mathrm{EvtI_n}$: Since $\left(\Pi_c(t) {}^\frown t', x\right) \models$ "γ'" and $x = {}_c{\Downarrow}_t(n')$ conclude

$(\Pi_c(t)^\frown t', {}_c\Downarrow_t(n')) \models$ “γ'”. Thus, since $\exists i \geq n\colon \wr c\wr_{t(i)}$ and $\nexists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)}$ conclude $(t, t', n')\models_{c}$“γ'” by Def. 15.

- Case $\nexists i\colon \wr c\wr_{t(i)} \wedge (t', n) \models$ “$\Diamond\gamma'$”: Thus, $\exists n' \geq n\colon (t', n') \models$ “γ'”. Since $\nexists i \geq n\colon \wr c\wr_{t(i)}$ and $n' \geq n$, we show $(t, t', n')\models_{c}$“γ'” to conclude $(t, t', n)\models_{c}$“γ” by rule EvtI$_\mathrm{n}$: Since $\nexists i\colon \wr c\wr_{t(i)}$ and $(t', n') \models$ “γ'” conclude $(t, t', n')\models_{c}$“γ'” by Def. 15.

Case $\gamma =$ “$\Box\gamma'$”: Since $(t, t', n)\models_{c}\gamma$ conclude $\Big(\exists i \geq n\colon \wr c\wr_{t(i)} \wedge (\Pi_c(t)^\frown t', \#_c^n(t)) \models$ “$\Box\gamma'$”$\Big) \vee \Big(\exists i\colon \wr c\wr_{t(i)} \wedge (\nexists i \geq n\colon \wr c\wr_{t(i)}) \wedge (\Pi_c(t)^\frown t', {}_c\Downarrow_t(n)) \models$ “$\Box\gamma'$”$\Big) \vee \Big(\nexists i\colon \wr c\wr_{t(i)} \wedge (t', n) \models$ “$\Box\gamma'$”$\Big)$ by Def. 15.

- Case $\exists i \geq n\colon \wr c\wr_{t(i)} \wedge (\Pi_c(t)^\frown t', \#_c^n(t)) \models$ “$\Box\gamma'$”: From $(\Pi_c(t)^\frown t', \#_c^n(t)) \models$ “$\Box\gamma'$” have $\forall x \geq \#_c^n(t)\colon (\Pi_c(t)^\frown t', x) \models$ “γ'”. We show that for all n', $\exists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)} \wedge c \overset{n}{\rightarrow} t \leq n' \implies \exists c \overset{n'}{\Leftarrow} t \leq \underset{.}{n}'' \leq c \overset{n'}{\rightarrow} t\colon (t, t', n'')\models_{c}$“$\gamma'$” and $\nexists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)} \wedge c \overset{n}{\rightarrow} t \leq n' \implies (t, t', n')\models_{c}$“$\gamma'$” to conclude $(t, t', n)\models_{c}$“$\Box\gamma$” by rule GlobI$_\mathrm{a}$.
  - $\exists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)} \wedge c \overset{n}{\rightarrow} t \leq n' \implies \exists c \overset{n'}{\Leftarrow} t \leq \underset{.}{n}'' \leq c \overset{n'}{\rightarrow} t\colon (t, t', n'')\models_{c}$“$\gamma'$”: Assume $\exists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)} \wedge c \overset{n}{\rightarrow} t \leq n'$. Since $c \overset{n}{\rightarrow} t \leq n'$ conclude $\#_c^{n'}(t) \geq \#_c^n(t)$ and since $\forall x \geq \#_c^n(t)\colon (\Pi_c(t)^\frown t', x) \models$ “γ'” conclude $(\Pi_c(t)^\frown t', \#_c^{n'}(t)) \models$ “γ'”. Thus, since $\exists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)}$ conclude $(t, t', n')\models_{c}$“γ'” by Def. 15. Moreover, have $c \overset{n'}{\Leftarrow} t \leq n'$ and since $\exists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)}$ conclude $n' \leq c \overset{n'}{\rightarrow} t$.
  - $\nexists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)} \wedge c \overset{n}{\rightarrow} t \leq n' \implies (t, t', n')\models_{c}$“$\gamma'$” to conclude $(t, t', n)\models_{c}$“$\Box\gamma$”: Assume $\nexists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)} \wedge c \overset{n}{\rightarrow} t \leq n'$. Since $c \overset{n}{\rightarrow} t \leq n'$ conclude ${}_c\Downarrow_t(n') \geq \#_c^n(t)$ and since $\forall x \geq \#_c^n(t)\colon (\Pi_c(t)^\frown t', x) \models$ “γ'” conclude $(\Pi_c(t)^\frown t', {}_c\Downarrow_t(n')) \models$ “γ'”. Thus, since $\exists \underset{.}{i} \geq n\colon \wr c\wr_{t(i)}$ and $\nexists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)}$ conclude $(t, t', n')\models_{c}$“γ'” by Def. 15.
- Case $\exists i\colon \wr c\wr_{t(i)} \wedge (\nexists i \geq n\colon \wr c\wr_{t(i)}) \wedge (\Pi_c(t)^\frown t', {}_c\Downarrow_t(n)) \models$ “$\Box\gamma'$”: From $(\Pi_c(t)^\frown t', {}_c\Downarrow_t(n)) \models$ “$\Box\gamma'$” have $\forall x \geq {}_c\Downarrow_t(n)\colon (\Pi_c(t)^\frown t', x) \models$ “γ'”. Since $\nexists i \geq n\colon \wr c\wr_{t(i)}$ we show that for all n', $\nexists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)} \wedge c \overset{n}{\rightarrow} t \leq n' \implies (t, t', n')\models_{c}$“$\gamma'$” to conclude $(t, t', n)\models_{c}$“$\Box\gamma$” by rule GlobI$_\mathrm{a}$: Assume $\nexists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)} \wedge c \overset{n}{\rightarrow} t \leq n'$. Thus, ${}_c\Downarrow_t(n') \geq {}_c\Downarrow_t(n)$ and since $\forall x \geq {}_c\Downarrow_t(n)\colon (\Pi_c(t)^\frown t', x) \models$ “γ'” conclude $(\Pi_c(t)^\frown t', {}_c\Downarrow_t(n')) \models$ “γ'”. Thus, since $\exists \underset{.}{i} \geq n\colon \wr c\wr_{t(i)}$ and $\nexists \underset{.}{i} \geq n'\colon \wr c\wr_{t(i)}$ conclude $(t, t', n')\models_{c}$“γ'” by Def. 15.
- Case $\nexists i\colon \wr c\wr_{t(i)} \wedge (t', n) \models$ “$\Box\gamma'$”: From $(t', n) \models$ “$\Box\gamma'$” have $\forall n' \geq n\colon (t', n') \models$ “γ'”. Since $\nexists i\colon \wr c\wr_{t(i)}$, we show $\forall n' \geq n\colon (t, t', n')\models_{c}$“$\gamma'$” to conclude $(t, t', n)\models_{c}$“$\Box\gamma$” by rule GlobI$_\mathrm{n}$. Thus, assume $n' \geq n$ and since $\forall n' \geq n\colon (t', n') \models$ “γ'” conclude $(t', n') \models$ “γ'”. Thus, since $\nexists i\colon \wr c\wr_{t(i)}$ conclude $(t, t', n')\models_{c}$“γ'” by Def. 15.

The case for $\gamma =$ “$\gamma' \, \mathcal{U} \, \gamma''$” can be obtained by a combination of the proof of eventually and globally and is omitted here.

E Soundness of Algorithm 1

In the following, we provide an argument of why Alg. 1 preserves the semantics of a FACTUM specification. The following diagram provides an overview of our reasoning:

$$\begin{tikzcd}
\text{FACTUM Specification} \arrow[rr, "\text{Alg. 1}"] \arrow[dr, "\text{Sect. 3}", "\models"'] & & \text{Isabelle HOL Theory} \arrow[dl, "{[\text{W}^+04]}"', "\models"] \\
& AT &
\end{tikzcd}$$

We need to show that a set of architecture traces AT satisfies a FACTUM specification iff it satisfies the Isabelle/HOL theory generated from the specification by algorithm 1.

To this end, we assume the existence of a FACTUM specification $PS = (DS, CS, AS)$, consisting of an algebraic specification of datatypes DS, a specification of component types CT, and an architecture specification AS.

E.1 Case $\Longrightarrow$

We fix a set of architecture traces AT and assume that it satisfies PS. We show that AT also satisfies the Isabelle theory generated from PS by algorithm 1. To this end, we fix an architecture trace $t \in AT$ and show that t satisfies the corresponding Isabelle theory. Again, the idea of the argument is depicted by the following diagram:

$$\begin{tikzcd}
CS \arrow[rrrrr, dashed, "\text{Alg. 1 (22-24)}"] \arrow[ddd] & & & & & BAsmp \arrow[ddddll, "eval"] \\
& DS \arrow[rrr, dashed, "\text{Alg. 1 (1-3)}"] \arrow[d] & & & DS \arrow[d] & \\
& MS & AS \arrow[r, dashed, "\text{Alg. 1 (25-27)}"] \arrow[d] & AAsmp \arrow[dd] & MS & \\
CT \arrow[r] \arrow[ur] & \otimes \arrow[drr] & AS' \arrow[l] \arrow[ul] & & & \\
& & & AT \arrow[uur] & &
\end{tikzcd}$$

Since AT satisfies PS, the semantics of FACTUM (discussed in Sect. 3) requires the existence of an architecture specification AS' which satisfies AS, such that $t \in AS'$. Thus, t also satisfies the locale assumptions generated from AS by lines $25 - 27$ of algorithm 1.

Similarly, since AT satisfies PS, the semantics of FACTUM requires the existence of a behavior CT_c, for each component c, which satisfies the corresponding behavior specification CS_c. Moreover, the semantics of FACTUM also requires the existence of a behavior trace t', such that $\Pi_c(t)^\frown t' \in CT_c$. Thus, according to the definition of *eval* (discussed in Sect. 6.4.2) we have $eval(c, t, t', \gamma)$ for each component c and locale assumption γ (generated by lines $25 - 27$ of Alg. 1).

Thus, t' fulfills all locale assumptions generated by Alg. 1 and thus it satisfies the the Isabelle theory generated from PS.

E.2 Case $\Longleftarrow$

We may use a symmetric argument as the one presented in case E.1 for the reverse direction.

F Pattern Hierarchy

F.1 A Theory of Singletons

In the following, we formalize the specification of the singleton pattern as described in [Mar18b].

theory *Singleton*
imports *Dynamic-Architecture-Calculus*
begin

In the following we formalize a variant of the Singleton pattern.

locale *singleton* = *dynamic-component cmp active*
  for $\mathit{active} :: {'}\mathit{id} \Rightarrow \mathit{cnf} \Rightarrow \mathit{bool}\ (\wr\text{-}\wr_{\text{-}}\ [0{,}110]60)$
  and $\mathit{cmp} :: {'}\mathit{id} \Rightarrow \mathit{cnf} \Rightarrow {'}\mathit{cmp}\ (\sigma_{\text{-}}(\text{-})\ [0{,}110]60)$ +
assumes *alwaysActive*: $\bigwedge k.\ \exists\, \mathit{id}.\ \wr \mathit{id} \wr_{k}$
  and *unique*: $\exists\, \mathit{id}.\ \forall\, k.\ \forall\, \mathit{id}'.\ (\wr \mathit{id}' \wr_{k} \longrightarrow \mathit{id} = \mathit{id}')$
begin

F.1.1 Calculus Interpretation

baIA: $[\![\exists\, i{\geq}n.\ \wr c \wr_{t\ i};\ \varphi\ (\sigma_{c}t\ \langle c \rightarrow t\rangle_{n})]\!] \Longrightarrow \mathit{eval}\ c\ t\ t'\ n\ (\mathit{ba}\ \varphi)$

$\mathit{baIN1}$: $[\![\exists\, i.\ \wr c \wr_{t\ i};\ \neg\ (\exists\, i{\geq}n.\ \wr c \wr_{t\ i});\ \varphi\ (t'\ (n - \langle c \wedge t\rangle - 1))]\!] \Longrightarrow \mathit{eval}\ c\ t\ t'\ n\ (\mathit{ba}\ \varphi)$

$\mathit{baIN2}$: $[\![\nexists\, i.\ \wr c \wr_{t\ i};\ \varphi\ (t'\ n)]\!] \Longrightarrow \mathit{eval}\ c\ t\ t'\ n\ (\mathit{ba}\ \varphi)$

F.1.2 Architectural Guarantees

definition $\mathit{the\text{-}singleton} \equiv \mathit{THE}\ \mathit{id}.\ \forall\, k.\ \forall\, \mathit{id}'.\ \wr \mathit{id}' \wr_{k} \longrightarrow \mathit{id}' = \mathit{id}$

theorem *ts-prop*:
 fixes $k{::}\mathit{cnf}$
 shows $\bigwedge \mathit{id}.\ \wr \mathit{id} \wr_{k} \Longrightarrow \mathit{id} = \mathit{the\text{-}singleton}$
  and $\wr \mathit{the\text{-}singleton} \wr_{k}$
proof –
 { **fix** *id*
  assume *a1*: $\wr \mathit{id} \wr_{k}$
  have $(\mathit{THE}\ \mathit{id}.\ \forall\, k.\ \forall\, \mathit{id}'.\ \wr \mathit{id}' \wr_{k} \longrightarrow \mathit{id}' = \mathit{id}) = \mathit{id}$
  proof (*rule the-equality*)
   show $\forall\, k\ \mathit{id}'.\ \wr \mathit{id}' \wr_{k} \longrightarrow \mathit{id}' = \mathit{id}$
   proof
    fix *k* **show** $\forall\, \mathit{id}'.\ \wr \mathit{id}' \wr_{k} \longrightarrow \mathit{id}' = \mathit{id}$
    proof

```
          fix id' show ≀id'≀k ⟶ id' = id
          proof
            assume ≀id'≀k
            from unique have ∃ id. ∀ k. ∀ id'. (≀id'≀k ⟶ id = id') .
            then obtain i'' where ∀ k. ∀ id'. (≀id'≀k ⟶ i'' = id') by auto
            with ‹≀id'≀k› have id=i'' and id'=i'' using a1 by auto
            thus id' = id by simp
          qed
        qed
      qed
    next
      fix i'' show ∀ k id'. ≀id'≀k ⟶ id' = i'' ⟹ i'' = id using a1 by auto
    qed
    hence ≀id≀k ⟹ id = the-singleton by (simp add: the-singleton-def)
  } note g1 = this
  thus ⋀id. ≀id≀k ⟹ id = the-singleton by simp

  from alwaysActive obtain id where ≀id≀k by blast
  with g1 have id = the-singleton by simp
  with ‹≀id≀k› show ≀the-singleton≀k by simp
qed

declare ts-prop(2)[simp]

lemma lNact-active[simp]:
  fixes cid t n
  shows ⟨the-singleton ⇐ t⟩n = n
  using lNact-active ts-prop(2) by auto

lemma lNxt-active[simp]:
  fixes cid t n
  shows ⟨the-singleton → t⟩n = n
by (simp add: nxtAct-active)

lemma baI[intro]:
  fixes t n a
  assumes φ (σthe-singleton(t n))
  shows eval the-singleton t t' n (ba φ) using assms by (simp add: baIANow)

lemma baE[elim]:
  fixes t n a
  assumes eval the-singleton t t' n (ba φ)
  shows φ (σthe-singleton(t n)) using assms by (simp add: baEANow)

lemma evtE[elim]:
  fixes t id n a
  assumes eval the-singleton t t' n (evt γ)
  shows ∃ n'≥n. eval the-singleton t t' n' γ
proof –
```

```
  have ⦚the-singleton⦚_t n by simp
  with assms obtain n' where n'≥⟨the-singleton → t⟩_n and (∃ i≥n'. ⦚the-singleton⦚_t i ∧
    (∀ n''≥⟨the-singleton ⇐ t⟩_n'. n'' ≤ ⟨the-singleton → t⟩_n' ⟶ eval the-singleton t t' n'' γ)) ∨
    ¬ (∃ i≥n'. ⦚the-singleton⦚_t i) ∧ eval the-singleton t t' n' γ using evtEA[of n the-singleton t]
    by blast
  moreover have ⦚the-singleton⦚_t n' by simp
  ultimately have
    ∀ n''≥⟨the-singleton ⇐ t⟩_n'. n'' ≤ ⟨the-singleton → t⟩_n' ⟶ eval the-singleton t t' n'' γ
    by auto
  hence eval the-singleton t t' n' γ by simp
  moreover from ‹n'≥⟨the-singleton → t⟩_n› have n'≥n by (simp add: nxtAct-active)
  ultimately show ?thesis by auto
qed

lemma globE[elim]:
  fixes t id n a
  assumes eval the-singleton t t' n (glob γ)
  shows ∀ n'≥n. eval the-singleton t t' n' γ
proof
  fix n' show n ≤ n' ⟶ eval the-singleton t t' n' γ
  proof
    assume n≤n'
    hence ⟨the-singleton ⇐ t⟩_n ≤ n' by simp
    moreover have ⦚the-singleton⦚_t n by simp
    ultimately show eval the-singleton t t' n' γ
      using ‹eval the-singleton t t' n (glob γ)› globEA by blast
  qed
qed

lemma untilI[intro]:
  fixes t::nat ⇒ cnf
    and t'::nat ⇒ 'cmp
    and n::nat
    and n'::nat
  assumes n'≥n
    and eval the-singleton t t' n' γ
    and ⋀n''. ⟦n≤n''; n''<n'⟧ ⟹ eval the-singleton t t' n'' γ'
  shows eval the-singleton t t' n (γ' 𝔘_b γ)
proof −
  have ⦚the-singleton⦚_t n by simp
  moreover from ‹n'≥n› have ⟨the-singleton ⇐ t⟩_n ≤ n' by simp
  moreover have ⦚the-singleton⦚_t n' by simp
  moreover have
    ∃ n''≥⟨the-singleton ⇐ t⟩_n'. n'' ≤ ⟨the-singleton → t⟩_n' ∧ eval the-singleton t t' n'' γ ∧
    (∀ n'''≥⟨the-singleton → t⟩_n. n''' < ⟨the-singleton ⇐ t⟩_n'' ⟶
      (∃ n''''≥⟨the-singleton ⇐ t⟩_n'''. n'''' ≤ ⟨the-singleton → t⟩_n'''
      ∧ eval the-singleton t t' n'''' γ'))
  proof −
    have n'≥⟨the-singleton ⇐ t⟩_n' by simp
```

```
    moreover have n' ≤ ⟨the-singleton → t⟩n' by simp
    moreover from assms(3) have
      (∀ n''≥⟨the-singleton → t⟩n. n'' < ⟨the-singleton ⇐ t⟩n' ⟶
      (∃ n'''≥⟨the-singleton ⇐ t⟩n''. n''' ≤ ⟨the-singleton → t⟩n''
        ∧ eval the-singleton t t' n''' γ'))
      by auto
    ultimately show ?thesis using ‹eval the-singleton t t' n' γ› by auto
  qed
  ultimately show ?thesis using untilIA[of n the-singleton t n' t' γ γ'] by blast
qed

lemma untilE[elim]:
  fixes t id n γ' γ
  assumes eval the-singleton t t' n (until γ' γ)
  shows
    ∃ n'≥n. eval the-singleton t t' n' γ ∧ (∀ n''≥n. n'' < n' ⟶ eval the-singleton t t' n'' γ')
proof –
  have ≀the-singleton≀t n by simp
  with ‹eval the-singleton t t' n (until γ' γ)› obtain n' where n'≥⟨the-singleton → t⟩n and
   (∃ i≥n'. ≀the-singleton≀t i) ∧
   (∀ n''≥⟨the-singleton ⇐ t⟩n'. n'' ≤ ⟨the-singleton → t⟩n' ⟶ eval the-singleton t t' n'' γ) ∧
   (∀ n''≥⟨the-singleton ⇐ t⟩n. n'' < ⟨the-singleton ⇐ t⟩n' ⟶ eval the-singleton t t' n'' γ') ∨
   ¬ (∃ i≥n'. ≀the-singleton≀t i) ∧
   eval the-singleton t t' n' γ ∧
    (∀ n''≥⟨the-singleton ⇐ t⟩n. n'' < n' ⟶ eval the-singleton t t' n'' γ')
  using untilEA[of n the-singleton t t' γ' γ] by auto
  moreover have ≀the-singleton≀t n' by simp
  ultimately have
    (∀ n''≥⟨the-singleton ⇐ t⟩n'. n'' ≤ ⟨the-singleton → t⟩n' ⟶ eval the-singleton t t' n'' γ) ∧
    (∀ n''≥⟨the-singleton ⇐ t⟩n. n'' < ⟨the-singleton ⇐ t⟩n' ⟶ eval the-singleton t t' n'' γ')
    by auto
  hence eval the-singleton t t' n' γ and (∀ n''≥n. n'' < n' ⟶ eval the-singleton t t' n'' γ')
    by auto
  with ‹eval the-singleton t t' n' γ› ‹n'≥⟨the-singleton → t⟩n› show ?thesis by auto
qed
end

end
```

F.2 A Theory of Publisher-Subscriber Architectures

In the following, we formalize the specification of the publisher subscriber pattern as described in [Mar18b].

```
theory Publisher-Subscriber
imports Singleton
begin
```

F.2.1 Subscriptions

datatype $'evt\ subscription = sub\ 'evt \mid unsub\ 'evt$

F.2.2 Publisher-Subscriber Architectures

locale $publisher\text{-}subscriber =$
pb: $singleton\ pbactive\ pbcmp\ +$
sb: $dynamic\text{-}component\ sbcmp\ sbactive$
for $pbactive :: 'pid \Rightarrow cnf \Rightarrow bool$
and $pbcmp :: 'pid \Rightarrow cnf \Rightarrow 'PB$
and $sbactive :: 'sid \Rightarrow cnf \Rightarrow bool$
and $sbcmp :: 'sid \Rightarrow cnf \Rightarrow 'SB\ +$
fixes $pbsb :: 'PB \Rightarrow ('evt\ set)\ subscription\ set$
and $pbnt :: 'PB \Rightarrow ('evt \times 'msg)$
and $sbnt :: 'SB \Rightarrow ('evt \times 'msg)\ set$
and $sbsb :: 'SB \Rightarrow ('evt\ set)\ subscription$
assumes $conn1$: $\bigwedge k\ pid.\ pbactive\ pid\ k$
$\Longrightarrow pbsb\ (pbcmp\ pid\ k) = (\bigcup sid \in \{sid.\ sbactive\ sid\ k\}.\ \{sbsb\ (sbcmp\ sid\ k)\})$
and $conn2$: $\bigwedge t\ n\ n''\ sid\ pid\ E\ e\ m.$
$[\![t \in arch;\ pbactive\ pid\ (t\ n);\ sbactive\ sid\ (t\ n);$
$sub\ E = sbsb\ (sbcmp\ sid\ (t\ n));\ n'' \geq n;\ e \in E;$
$\nexists n'\ E'.\ n' \geq n \land n' \leq n'' \land sbactive\ sid\ (t\ n') \land$
$unsub\ E' = sbsb\ (sbcmp\ sid\ (t\ n')) \land e \in E';$
$(e,\ m) = pbnt\ (pbcmp\ pid\ (t\ n''));\ sbactive\ sid\ (t\ n'')]\!]$
$\Longrightarrow pbnt\ (pbcmp\ pid\ (t\ n'')) \in sbnt\ (sbcmp\ sid\ (t\ n''))$
begin

notation $pb.imp$ (**infixl** $\longrightarrow^p$ 10)
notation $pb.disj$ (**infixl** $\vee^p$ 15)
notation $pb.conj$ (**infixl** $\wedge^p$ 20)
notation $pb.not$ ($\neg^p$ - $[19]19$)
no-notation $pb.all$ (**binder** $\forall_b$ 10)
no-notation $pb.ex$ (**binder** $\exists_b$ 10)
notation $pb.all$ (**binder** $\forall_p$ 10)
notation $pb.ex$ (**binder** $\exists_p$ 10)

notation $sb.imp$ (**infixl** $\longrightarrow^s$ 10)
notation $sb.disj$ (**infixl** $\vee^s$ 15)
notation $sb.conj$ (**infixl** $\wedge^s$ 20)
notation $sb.not$ ($\neg^s$ - $[19]19$)
no-notation $sb.all$ (**binder** $\forall_b$ 10)
no-notation $sb.ex$ (**binder** $\exists_b$ 10)
notation $sb.all$ (**binder** $\forall_s$ 10)
notation $sb.ex$ (**binder** $\exists_s$ 10)

F.2.2.1 Calculus Interpretation

$pb.nxtEA1$: $[\![\exists\, i{>}pb.nxtAct\ c\ t\ n.\ pbactive\ c\ (t\ i);\ pb.eval\ c\ t\ t'\ n\ (\bigcirc_b\gamma);\ n \leq n';\ \exists!i.\ n \leq i \land pb.latestAct\text{-}cond\ c\ t\ n'\ i]\!] \Longrightarrow pb.eval\ c\ t\ t'\ n'\ \gamma$

sb.nxtEA1: $[\![\exists\, i{>}sb.nxtAct\ c\ t\ n.\ sbactive\ c\ (t\ i);\ sb.eval\ c\ t\ t'\ n\ (\bigcirc_b \gamma);\ n \leq n';\ \exists!i.\ n \leq i \wedge sb.latestAct\text{-}cond\ c\ t\ n'\ i]\!] \Longrightarrow sb.eval\ c\ t\ t'\ n'\ \gamma$

F.2.2.2 Results from Singleton

abbreviation *the-pb* :: *'pid* **where**
the-pb $\equiv$ *pb.the-singleton*

pb.ts-prop (*1*): *pbactive id k* $\Longrightarrow$ *id* = *the-pb*

pb.ts-prop (*2*): *pbactive the-pb k*

F.2.2.3 Architectural Guarantees

The following theorem ensures that a subscriber indeed receives all messages associated with an event for which he is subscribed.

theorem *msgDelivery*:
  fixes *t n n'' sid E e m*
  assumes $t \in arch$
    and *sbactive sid* (*t n*)
    and *sub E* = *sbsb* (*sbcmp sid* (*t n*))
    and $n'' \geq n$
    and $\nexists\, n'\ E'.\ n' \geq n \wedge n' \leq n''$
      $\wedge$ *sbactive sid* ($t\ n'$)
      $\wedge$ *unsub* E' = *sbsb*(*sbcmp sid* ($t\ n'$))
      $\wedge\ e \in E'$
    and $e \in E$
    and (*e*,*m*) = *pbnt* (*pbcmp the-pb* ($t\ n''$))
    and *sbactive sid* ($t\ n''$)
  shows (*e*,*m*) $\in$ *sbnt* (*sbcmp sid* ($t\ n''$))
  using *assms conn2 pb.ts-prop*(*2*) **by** *simp*

Since a publisher is actually a singleton, we can provide an alternative version of constraint *conn1*.

lemma *conn1A*:
  fixes *k*
  shows *pbsb* (*pbcmp the-pb k*) = ($\bigcup$ *sid*$\in$\{*sid. sbactive sid k*\}. \{*sbsb* (*sbcmp sid k*)\})
  using *conn1* [*OF pb.ts-prop*(*2*)] **.**
end

end

F.3 A Theory of Blackboard Architectures

In the following, we formalize the specification of the blackboard pattern as described in [Mar18b].

theory *Blackboard*

imports *Publisher-Subscriber*
begin

F.3.1 Problems and Solutions

Blackboards work with problems and solutions for them.

typedecl *PROB*
consts *sb* :: (*PROB* × *PROB*) *set*
axiomatization where *sbWF*: *wf sb*
typedecl *SOL*
consts *solve*:: *PROB* ⇒ *SOL*

F.3.2 Blackboard Architectures

In the following, we describe the locale for the blackboard pattern.

locale *blackboard* = *publisher-subscriber bbactive bbcmp ksactive kscmp bbrp bbcs kscs ksrp*
  for *bbactive* :: *'bid* ⇒ *cnf* ⇒ *bool* (⌇-⌇$_-$ [0,110]60)
    and *bbcmp* :: *'bid* ⇒ *cnf* ⇒ *'BB* (σ_-(-) [0,110]60)
    and *ksactive* :: *'kid* ⇒ *cnf* ⇒ *bool* (⌇-⌇$_-$ [0,110]60)
    and *kscmp* :: *'kid* ⇒ *cnf* ⇒ *'KS* (σ_-(-) [0,110]60)
    and *bbrp* :: *'BB* ⇒ (*PROB set*) *subscription set*
    and *bbcs* :: *'BB* ⇒ (*PROB* × *SOL*)
    and *kscs* :: *'KS* ⇒ (*PROB* × *SOL*) *set*
    and *ksrp* :: *'KS* ⇒ (*PROB set*) *subscription* +
  fixes *bbns* :: *'BB* ⇒ (*PROB* × *SOL*) *set*
    and *ksns* :: *'KS* ⇒ (*PROB* × *SOL*)
    and *bbop* :: *'BB* ⇒ *PROB*
    and *ksop* :: *'KS* ⇒ *PROB set*
    and *prob* :: *'kid* ⇒ *PROB*
  assumes
    ks1: ∀*p*. ∃*ks*. *p*=*prob ks* — Component Parameter
    — Assertions about component behavior.
    and *bhvbb1*: ⋀*t t' bId p s*. ⟦*t* ∈ *arch*⟧ ⟹ *pb.eval bId t t' 0*
      (*pb.glob* (*pb.ba* (λ*bb*. (*p*,*s*)∈*bbns bb*)
      $\longrightarrow^p$ (*pb.evt* (*pb.ba* (λ*bb*. (*p*,*s*) = *bbcs bb*)))))
    and *bhvbb2*: ⋀*t t' bId P q*. ⟦*t*∈*arch*⟧ ⟹ *pb.eval bId t t' 0*
      (*pb.glob* (*pb.ba* (λ*bb*. *sub P* ∈ *bbrp bb* ∧ *q* ∈ *P*) $\longrightarrow^p$
      (*pb.evt* (*pb.ba* (λ*bb*. *q* = *bbop bb*)))))
    and *bhvbb3*: ⋀*t t' bId p* . ⟦*t*∈*arch*⟧ ⟹ *pb.eval bId t t' 0*
      (*pb.glob* (*pb.ba* (λ*bb*. *p* = *bbop*(*bb*)) $\longrightarrow^p$
      (*pb.wuntil* (*pb.ba* (λ*bb*. *p*=*bbop*(*bb*))) (*pb.ba* (λ*bb*. (*p*,*solve*(*p*)) = *bbcs*(*bb*))))))
    and *bhvks1*: ⋀*t t' kId p P*. ⟦*t*∈*arch*; *p* = *prob kId*⟧ ⟹ *sb.eval kId t t' 0*
      (*sb.glob* ((*sb.ba* (λ*ks*. *sub P* = *ksrp ks*)) $\wedge^s$
      (*sb.all* (λ*q*. (*sb.pred* (*q*∈*P*)) $\longrightarrow^s$ (*sb.evt* (*sb.ba* (λ*ks*. (*q*,*solve*(*q*)) ∈ *kscs ks*)))))
      $\longrightarrow^s$ (*sb.evt* (*sb.ba* (λ*ks*. (*p*, *solve p*) = *ksns ks*)))))
    and *bhvks2*: ⋀*t t' kId p P q*. ⟦*t* ∈ *arch*;*p* = *prob kId*⟧ ⟹ *sb.eval kId t t' 0*
      (*sb.glob* (*sb.ba* (λ*ks*. *sub P* = *ksrp ks* ∧ *q* ∈ *P* ⟶ (*q*,*p*) ∈ *sb*)))
    and *bhvks3*: ⋀*t t' kId p*. ⟦*t*∈*arch*;*p* = *prob kId*⟧ ⟹ *sb.eval kId t t' 0*
      (*sb.glob* ((*sb.ba* (λ*ks*. *p*∈*ksop ks*)) $\longrightarrow^s$ (*sb.evt* (*sb.ba* (λ*ks*. (∃*P*. *sub P* = *ksrp ks*))))))

and $bhvks4$: $\bigwedge t\ t'\ kId\ p\ P.\ \llbracket t{\in}arch;\ p{\in}P\rrbracket \Longrightarrow sb.eval\ kId\ t\ t'\ 0$
$(sb.glob\ ((sb.ba\ (\lambda ks.\ sub\ P = ksrp\ ks)) \longrightarrow^{s}$
$(sb.wuntil\ (\neg^{s}\ (\exists_{s}\ P'.\ (sb.pred\ (p{\in}P')\ \wedge^{s}\ (sb.ba\ (\lambda ks.\ unsub\ P' = ksrp\ ks)))))$
$(sb.ba\ (\lambda ks.\ (p{,}solve\ p) \in kscs\ ks)))))$

— Assertions about component activation.
and $actks$:
$\bigwedge t\ n\ kid\ p.\ \llbracket t \in arch;\ ksactive\ kid\ (t\ n);\ p{=}prob\ kid;\ p{\in}ksop\ (kscmp\ kid\ (t\ n))\rrbracket$
$\Longrightarrow (\exists\, n'{\geq}n.\ ksactive\ kid\ (t\ n') \wedge (p,\ solve\ p) = ksns\ (kscmp\ kid\ (t\ n')) \wedge$
$(\forall\, n''{\geq}n.\ n''{<}n' \longrightarrow ksactive\ kid\ (t\ n'')))$
$\vee\ (\forall\, n'{\geq}n.\ (ksactive\ kid\ (t\ n') \wedge (\neg(p,\ solve\ p) = ksns\ (kscmp\ kid\ (t\ n')))))$

— Assertions about connections.
and $conn1$: $\bigwedge k\ bid.\ bbactive\ bid\ k$
$\Longrightarrow bbns\ (bbcmp\ bid\ k) = (\bigcup kid{\in}\{kid.\ ksactive\ kid\ k\}.\ \{ksns\ (kscmp\ kid\ k)\})$
and $conn2$: $\bigwedge k\ kid.\ ksactive\ kid\ k$
$\Longrightarrow ksop\ (kscmp\ kid\ k) = (\bigcup bid{\in}\{bid.\ bbactive\ bid\ k\}.\ \{bbop\ (bbcmp\ bid\ k)\})$
begin
notation $pb.lNAct$ $(\langle\text{-} \Leftarrow \text{-}\rangle_{\text{-}})$
notation $pb.nxtAct$ $(\langle\text{-} \rightarrow \text{-}\rangle_{\text{-}})$

F.3.2.1 Calculus Interpretation

$pb.baIA$: $\llbracket\exists\, i{\geq}n.\ \wr c\wr_{t\ i};\ \varphi\ (\sigma_{c}t\ \langle c \rightarrow t\rangle_{n})\rrbracket \Longrightarrow pb.eval\ c\ t\ t'\ n\ (pb.ba\ \varphi)$

$sb.baIA$: $\llbracket\exists\, i{\geq}n.\ \wr c\wr_{t\ i};\ \varphi\ (\sigma_{c}t\ (sb.nxtAct\ c\ t\ n))\rrbracket \Longrightarrow sb.eval\ c\ t\ t'\ n\ (sb.ba\ \varphi)$

F.3.2.2 Results from Singleton

abbreviation $the\text{-}bb \equiv the\text{-}pb$

$pb.ts\text{-}prop\ (\ 1\ )$: $\wr id\wr_{k} \Longrightarrow id = the\text{-}bb$

$pb.ts\text{-}prop\ (\ 2\ )$: $\wr the\text{-}bb\wr_{k}$

F.3.2.3 Results from Publisher Subscriber

$msgDelivery$: $\llbracket t \in arch;\ \wr sid\wr_{t\ n};\ sub\ E = ksrp\ (\sigma_{sid}t\ n);\ n \leq n'';\ \nexists\, n'\ E'.\ n \leq n' \wedge n' \leq n'' \wedge \wr sid\wr_{t\ n'} \wedge unsub\ E' = ksrp\ (\sigma_{sid}t\ n') \wedge e \in E';\ e \in E;\ (e,\ m) = bbcs\ (\sigma_{the\text{-}bb}t\ n'');\ \wr sid\wr_{t\ n''}\rrbracket \Longrightarrow (e,\ m) \in kscs\ (\sigma_{sid}t\ n'')$

lemma $conn2\text{-}bb$:
fixes $k\ kid$
assumes $ksactive\ kid\ k$
shows $bbop\ (bbcmp\ the\text{-}bb\ k){\in}ksop\ (kscmp\ kid\ k)$
proof $-$
from $assms$ **have** $ksop\ (kscmp\ kid\ k) = (\bigcup bid{\in}\{bid.\ bbactive\ bid\ k\}.\ \{bbop\ (bbcmp\ bid\ k)\})$
using $conn2$ **by** $simp$
moreover have $(\bigcup bid.\{bid.\ bbactive\ bid\ k\}){=}\{the\text{-}bb\}$ **using** $pb.ts\text{-}prop(1)$ **by** $auto$
hence $(\bigcup bid{\in}\{bid.\ bbactive\ bid\ k\}.\ \{bbop\ (bbcmp\ bid\ k)\}) = \{bbop\ (bbcmp\ the\text{-}bb\ k)\}$

```
    by auto
  ultimately show ?thesis by simp
qed
```

F.3.2.4 Knowledge Sources

In the following we introduce an abbreviation for knowledge sources which are able to solve a specific problem.

```
definition sKs:: PROB ⇒ 'kid where
  sKs p ≡ (SOME kid. p = prob kid)

lemma sks-prob:
  p = prob (sKs p)
using sKs-def someI-ex[of λkid. p = prob kid] ks1 by auto
```

F.3.3 Architectural Guarantees

The following theorem verifies that a problem is eventually solved by the pattern even if no knowledge source exist which can solve the problem on its own. It assumes, however, that for every open sub problem, a corresponding knowledge source able to solve the problem will be eventually activated.

```
lemma pSolved-Ind:
  fixes t and t'::nat ⇒'BB and p and t''::nat ⇒'KS
  assumes t∈arch and
    ∀n. (∃n'≥n. ksactive (sKs (bbop(bbcmp the-bb (t n)))) (t n'))
  shows
    ∀n. (∃P. sub P ∈ bbrp(bbcmp the-bb (t n)) ∧ p ∈ P) ⟶
      (∃m≥n. (p,solve(p)) = bbcs (bbcmp the-bb (t m)))
— The proof is by well-founded induction over the subproblem relation sb
proof (rule wf-induct[where r=sb])
  — We first show that the subproblem relation is indeed well-founded ...
  show wf sb by (simp add: sbWF)
next
  — ... then we show that a problem p is indeed solved
  — if all its sub-problems p' are eventually solved
  fix p assume
    indH: ∀p'. (p', p) ∈ sb ⟶ (∀n. (∃P. sub P ∈ bbrp (bbcmp the-bb (t n)) ∧ p'∈P)
    ⟶ (∃m≥n. (p',solve(p')) = bbcs (bbcmp the-bb (t m))))
  show ∀n. (∃P. sub P ∈ bbrp (bbcmp the-bb (t n)) ∧ p ∈ P)
    ⟶ (∃m≥n. (p,solve(p)) = bbcs (bbcmp the-bb (t m)))
  proof
    fix n0 show (∃P. sub P ∈ bbrp (bbcmp the-bb (t n0)) ∧ p ∈ P) ⟶
    (∃m≥n0. (p,solve(p)) = bbcs (bbcmp the-bb (t m)))
    proof
      assume ∃P. sub P ∈ bbrp (bbcmp the-bb (t n0)) ∧ p ∈ P
      moreover have (∃P. sub P ∈ bbrp (bbcmp the-bb (t n0)) ∧ p ∈ P) ⟶
        (∃n'≥n0. p=bbop(bbcmp the-bb (t n')))
      proof
```

```
  assume ∃ P. sub P ∈ bbrp (bbcmp the-bb (t n0)) ∧ p ∈ P
  then obtain P where sub P ∈ bbrp (bbcmp the-bb (t n0)) and p ∈ P by auto
  hence pb.eval the-bb t t' n0 (pb.ba (λbb. sub P ∈ bbrp bb ∧ p ∈ P))
    using pb.baI by simp
  moreover from pb.globE[OF bhvbb2] have
    pb.eval the-bb t t' n0 (pb.ba (λbb. sub P ∈ bbrp bb ∧ p ∈ P) ⟶^p
    ◇_b pb.ba (λbb. p = bbop bb))
    using ‹t∈arch› by simp
  ultimately have pb.eval the-bb t t' n0 (◇_b pb.ba (λbb. p = bbop bb))
    using pb.impE by blast
  then obtain n' where n'≥n0 and pb.eval the-bb t t' n' (pb.ba (λbb. p = bbop bb))
    using pb.evtE by blast
  hence p=bbop(bbcmp the-bb (t n')) using pb.baE by auto
  with ‹n'≥n0› show ∃ n'≥n0. p=bbop(bbcmp the-bb (t n')) by auto
qed
ultimately obtain n where n≥n0 and p=bbop(bbcmp the-bb (t n)) by auto

— Problem p is provided at the output of the blackboard until it is solved
— or forever...
from pb.globE[OF bhvbb3] have
  pb.eval the-bb t t' n (pb.ba (λ bb. p = bbop(bb)) ⟶^p
  (pb.wuntil (pb.ba (λ bb. p=bbop(bb)))
  (pb.ba (λbb. (p,solve(p)) = bbcs(bb)))))
  using ‹t∈arch› by auto
moreover from ‹p = bbop (bbcmp the-bb (t n))› have
  pb.eval the-bb t t' n (pb.ba (λ bb. p=bbop bb))
  using ‹t∈arch› pb.baI by simp
ultimately have pb.eval the-bb t t' n
  (pb.wuntil (pb.ba (λ bb. p=bbop(bb)))
  (pb.ba (λ bb. (p,solve(p)) = bbcs(bb))))
  using pb.impE by blast
hence pb.eval the-bb t t' n ((pb.until (pb.ba (λ bb. p=bbop bb))
  (pb.ba (λ bb. (p,solve(p)) = bbcs bb))) ∨^p (pb.glob (pb.ba (λ bb. p=bbop bb))))
  using pb.wuntil-def by simp
hence pb.eval the-bb t t' n
  (pb.until (pb.ba (λbb. p=bbop bb))
    (pb.ba (λbb. (p,solve(p)) = bbcs bb))) ∨
  (pb.eval the-bb t t' n (pb.glob (pb.ba (λ bb. p=bbop bb))))
  using pb.disjE by simp
thus ∃ m≥n0. (p,solve p) = bbcs(bbcmp the-bb (t m))
— We need to consider both cases, the case in which the problem is eventually
— solved and the case in which the problem is always provided as an output
proof
  — First we consider the case in which the problem is eventually solved:
  assume pb.eval the-bb t t' n
    (pb.until (pb.ba (λbb. p=bbop bb))
    (pb.ba (λbb. (p,solve(p)) = bbcs bb)))
  hence ∃ i≥n. (pb.eval the-bb t t' i
    (pb.ba (λbb. (p,solve(p)) = bbcs bb)) ∧
```

```
    (∀k≥n. k<i ⟶ pb.eval the-bb t t' k (pb.ba (λ bb. p = bbop bb))))
    using ‹t∈arch› pb.untilE by simp
  then obtain i where i≥n and
    pb.eval the-bb t t' i (pb.ba (λbb. (p,solve(p)) = bbcs bb)) by auto
  hence (p,solve(p)) = bbcs(bbcmp the-bb (t i))
    using ‹t∈arch› pb.baEA by auto
  moreover from ‹i≥n› ‹n≥n_0› have i≥n_0 by simp
  ultimately show ?thesis by auto
next
  — Now we consider the case in which p is always provided at the output
  — of the blackboard:
  assume pb.eval the-bb t t' n
    (pb.glob (pb.ba (λbb. p=bbop bb)))
  hence ∀n'≥n. (pb.eval the-bb t t' n' (pb.ba (λbb. p = bbop bb)))
    using ‹t∈arch› pb.globE by auto
  hence outp: ∀n'≥n. (p = bbop (bbcmp the-bb (t n')))
    using ‹t∈arch› pb.baE by blast

  — thus, by assumption there exists a KS which is able to solve p and which
  — is active at n'...
  with assms(2) have ∃n'≥n. ksactive (sKs p) (t n') by auto
  then obtain n_k where n_k≥n and ksactive (sKs p) (t n_k) by auto
  — ... and get the problem as its input.
  moreover from ‹n_k≥n› have p = bbop (bbcmp the-bb (t n_k))
    using outp by simp
 ultimately have p∈ksop(kscmp (sKs p) (t n_k)) using conn2-bb[of sKs p t n_k] by simp

  — thus the ks will either solve the problem or not solve it and
  — be activated forever
  hence (∃n'≥n_k. ksactive (sKs p) (t n') ∧
    (p, solve p) = ksns (kscmp (sKs p) (t n')) ∧
    (∀n''≥n_k. n''<n' ⟶ ksactive (sKs p) (t n''))) ∨
    (∀n'≥n_k. (ksactive (sKs p) (t n') ∧
    (¬(p, solve p) = ksns (kscmp (sKs p) (t n')))))
    using ‹ksactive (sKs p) (t n_k)› actks[of t sKs p] ‹t∈arch› sks-prob by simp
  thus ?thesis
  proof
    — if the ks solves it
    assume ∃n'≥n_k. ⦃sKs p⦄_{t n'} ∧ (p, solve p) = ksns (σ_{sKs p} t n')
      ∧ (∀n''≥n_k. n'' < n' ⟶ ⦃sKs p⦄_{t n''})
    — it is forwarded to the blackboard
    then obtain n_s where n_s≥n_k and ⦃sKs p⦄_{t n_s}
      and (p, solve p) = ksns (σ_{sKs p} t n_s) by auto
    moreover have sb.nxtAct (sKs p) t n_s = n_s
      by (simp add: ‹⦃sKs p⦄_{t n_s}› sb.nxtAct-active)
    ultimately have
      (p,solve(p)) ∈ bbns (bbcmp the-bb (t (sb.nxtAct (sKs p) t n_s)))
      using conn1[OF pb.ts-prop(2)] ‹⦃sKs p⦄_{t n_s}› by auto
```

```
  — finally, the blackboard will forward the solution which finishes the proof.
  with bhvbb1 have pb.eval the-bb t t' (sb.nxtAct (sKs p) t n_s)
    (pb.evt (pb.ba (λbb. (p, solve p) = bbcs bb)))
    using ‹t∈arch› pb.globE pb.impE[of the-bb t t'] by blast
  then obtain n_f where n_f≥sb.nxtAct (sKs p) t n_s and
    pb.eval the-bb t t' n_f (pb.ba (λbb. (p, solve p) = bbcs bb))
    using ‹t∈arch› pb.evtE[of t t' sb.nxtAct (sKs p) t n_s] by auto
  hence (p, solve p) = bbcs (bbcmp the-bb (t n_f))
    using ‹t ∈ arch› pb.baEA by auto
  moreover have n_f≥n_0
  proof −
    from ‹ksactive (sKs p) (t n_k)› have sb.nxtAct (sKs p) t n_k≥n_k
      using sb.nxtActI by blast
    with ‹sb.nxtAct (sKs p) t n_s = n_s› show ?thesis
      using ‹n_f≥sb.nxtAct (sKs p) t n_s› ‹n_s≥n_k› ‹n_k≥n› ‹n≥n_0› by arith
  qed
  ultimately show ?thesis by auto
next
  — otherwise, we derive a contradiction
  assume case-ass: ∀ n'≥n_k. ⦃sKs p⦄_t n' ∧ ¬(p, solve p) = ksns (σ_sKs p t n')

  — first, the KS will eventually register for the subproblems P it requires to solve p...
  from ‹ksactive (sKs p) (t n_k)› have ∃ i≥0. ksactive (sKs p) (t i) by auto
  moreover have sb.lNAct (sKs p) t 0 ≤ n_k by simp
  ultimately have sb.eval (sKs p) t t'' n_k
    ((sb.ba (λks. p∈ksop ks)) ⟶^s
    (sb.evt (sb.ba (λks. ∃ P. sub P = ksrp ks))))
    using sb.globEA[OF - bhvks3[of t p sKs p t'']] ‹t∈arch› sks-prob by simp
  moreover have sb.eval (sKs p) t t'' n_k (sb.ba (λks. p ∈ ksop ks))
  proof −
    from ‹ksactive (sKs p) (t n_k)› have ∃ n'≥n_k. ksactive (sKs p) (t n') by auto
    moreover have p ∈ ksop (kscmp (sKs p) (t (sb.nxtAct (sKs p) t n_k)))
    proof −
      from ‹ksactive (sKs p) (t n_k)› have sb.nxtAct (sKs p) t n_k=n_k
        using sb.nxtAct-active by blast
      with ‹p∈ksop(kscmp (sKs p) (t n_k))› show ?thesis by simp
    qed
    ultimately show ?thesis using sb.baIA[of n_k sKs p t] by blast
  qed
  ultimately have sb.eval (sKs p) t t'' n_k (sb.evt (sb.ba (λks. ∃ P. sub P = ksrp ks)))
    using sb.impE by blast
  then obtain n_r where n_r≥sb.nxtAct (sKs p) t n_k and
    ∃ i≥n_r. ksactive (sKs p) (t i) ∧
    (∀ n''≥sb.lNAct (sKs p) t n_r. n'' ≤ sb.nxtAct (sKs p) t n_r
    ⟶ sb.eval (sKs p) t t'' n'' (sb.ba (λks. ∃ P. sub P = ksrp ks))) ∨
    ¬ (∃ i≥n_r. ksactive (sKs p) (t i)) ∧
    sb.eval (sKs p) t t'' n_r (sb.ba (λks. ∃ P. sub P = ksrp ks))
    using ‹ksactive (sKs p) (t n_k)› sb.evtEA[of n_k sKs p t] by blast
  moreover from case-ass have sb.nxtAct (sKs p) t n_k≥n_k using sb.nxtActI by blast
```

with $\langle n_r{\geq}sb.nxtAct\ (sKs\ p)\ t\ n_k\rangle$ **have** $n_r{\geq}n_k$ **by** *arith*
hence $\exists\, i{\geq}n_r.\ ksactive\ (sKs\ p)\ (t\ i)$ **using** *case-ass* **by** *auto*
hence $n_r \leq sb.nxtAct\ (sKs\ p)\ t\ n_r$ **using** *sb.nxtActLe* **by** *simp*
moreover have $n_r \geq sb.lNAct\ (sKs\ p)\ t\ n_r$ **by** *simp*
ultimately have
  $sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.ba\ (\lambda ks.\ \exists\, P.\ sub\ P = ksrp\ ks))$ **by** *blast*
with $\langle\exists\, i{\geq}n_r.\ ksactive\ (sKs\ p)\ (t\ i)\rangle$ **obtain** P **where**
  $sub\ P = ksrp\ (kscmp\ (sKs\ p)\ (t\ (sb.nxtAct\ (sKs\ p)\ t\ n_r)))$
  using *sb.baEA* **by** *blast*
hence $sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.ba\ (\lambda ks.\ sub\ P = ksrp\ ks))$
  using $\langle\exists\, i{\geq}n_r.\ ksactive\ (sKs\ p)\ (t\ i)\rangle$ *sb.baIA sks-prob* **by** *blast*

— the knowledgesource will eventually get a solution for each required subproblem:
moreover have $sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.all\ (\lambda\ p'.\ sb.pred\ (p'{\in}P) \longrightarrow^s$
  $(sb.evt\ (sb.ba\ (\lambda ks.\ (p',solve\ p') \in kscs\ ks)))))$
proof –
  have $\forall\, p'.\ sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.pred\ (p'{\in}P) \longrightarrow^s$
    $(sb.evt\ (sb.ba\ (\lambda ks.\ (p',solve\ p') \in kscs\ ks))))$
  proof
    — by induction hypothesis,
    — the blackboard will eventually provide solutions for subproblems
    fix p'
    have $sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.pred\ (p'{\in}P)) \longrightarrow$
      $(sb.eval\ (sKs\ p)\ t\ t''\ n_r$
      $(sb.evt\ (sb.ba\ (\lambda ks.\ (p',solve\ p') \in kscs\ ks))))$
    proof
      assume $sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.pred\ (p'{\in}P))$
      hence $p' \in P$ **using** *sb.predE* **by** *blast*
      thus $(sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.evt\ (sb.ba\ (\lambda ks.\ (p',solve\ p') \in kscs\ ks))))$
      proof –
        have $sb.lNAct\ (sKs\ p)\ t\ 0 \leq n_r$ **by** *simp*
        moreover from $\langle ksactive\ (sKs\ p)\ (t\ n_k)\rangle$ **have** $\exists\, i{\geq}0.\ ksactive\ (sKs\ p)\ (t\ i)$
          by *auto*
        ultimately have $sb.eval\ (sKs\ p)\ t\ t''\ n_r\ ((sb.ba\ (\lambda ks.\ sub\ P = ksrp\ ks))$
          $\longrightarrow^s (sb.wuntil\ (\neg^s\ (\exists_s\ P'.\ (sb.pred\ (p'{\in}P') \wedge^s$
          $(sb.ba\ (\lambda ks.\ unsub\ P' = ksrp\ ks)))))$
          $(sb.ba\ (\lambda ks.\ (p',solve\ p') \in kscs\ ks))))$
          using $sb.globEA[OF\ \text{-}\ bhvks4[of\ t\ p'\ P\ sKs\ p\ t'']]$
          $\langle t{\in}arch\rangle\ \langle ksactive\ (sKs\ p)\ (t\ n_k)\rangle\ \langle p'{\in}P\rangle$ **by** *simp*
        with $\langle sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.ba\ (\lambda ks.\ sub\ P = ksrp\ ks))\rangle$ **have**
          $sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.wuntil\ (\neg^s\ (\exists_s\ P'.\ (sb.pred\ (p'{\in}P') \wedge^s$
          $(sb.ba\ (\lambda ks.\ unsub\ P' = ksrp\ ks)))))\ (sb.ba\ (\lambda ks.\ (p',solve\ p') \in kscs\ ks)))$
          using $sb.impE[of\ (sKs\ p)\ t\ t''\ n_r\ sb.ba\ (\lambda ks.\ sub\ P = ksrp\ ks)]$ **by** *blast*
        hence $sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.until\ (\neg^s\ (\exists_s\ P'.\ (sb.pred\ (p'{\in}P') \wedge^s$
          $(sb.ba\ (\lambda ks.\ unsub\ P' = ksrp\ ks)))))\ (sb.ba\ (\lambda ks.\ (p',solve\ p') \in kscs\ ks))) \vee$
          $sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.glob\ (\neg^s\ (\exists_s\ P'.\ (sb.pred\ (p'{\in}P') \wedge^s$
          $sb.ba\ (\lambda ks.\ unsub\ P' = ksrp\ ks)))))$ **using** *sb.wuntil-def* **by** *auto*
        thus $(sb.eval\ (sKs\ p)\ t\ t''\ n_r\ (sb.evt\ (sb.ba\ (\lambda ks.\ (p',solve\ p') \in kscs\ ks))))$
        proof

```
  let ?γ'=¬ˢ (∃ₛ P'. (sb.pred (p'∈P') ∧ˢ (sb.ba (λks. unsub P' = ksrp ks))))
  let ?γ=sb.ba (λks. (p',solve p') ∈ kscs ks)
  assume sb.eval (sKs p) t t'' n_r (sb.until ?γ' ?γ)
  with ⟨∃ i≥n_r. ≀sKs p≀_t i⟩ obtain n' where n'≥sb.nxtAct (sKs p) t n_r and
    lass: (∃ i≥n'. ≀sKs p≀_t i) ∧
    (∀ n''≥sb.lNAct (sKs p) t n'. n'' ≤ sb.nxtAct (sKs p) t n'
    ⟶ sb.eval (sKs p) t t'' n'' ?γ) ∧
    (∀ n''≥sb.lNAct (sKs p) t n_r. n'' < sb.lNAct (sKs p) t n'
    ⟶ sb.eval (sKs p) t t'' n'' ?γ') ∨
    ¬ (∃ i≥n'. ≀sKs p≀_t i) ∧ sb.eval (sKs p) t t'' n' ?γ ∧
    (∀ n''≥sb.lNAct (sKs p) t n_r. n'' < n' ⟶ sb.eval (sKs p) t t'' n'' ?γ')
    using sb.untilEA[of n_r sKs p t t''] ⟨∃ i≥n_r. ksactive (sKs p) (t i)⟩ by blast
  thus ?thesis
  proof cases
    assume ∃ i≥n'. ≀sKs p≀_t i
    with lass have ∀ n''≥sb.lNAct (sKs p) t n'. n'' ≤ sb.nxtAct (sKs p) t n'
      ⟶ sb.eval (sKs p) t t'' n'' ?γ by auto
    moreover have n'≥sb.lNAct (sKs p) t n' by simp
    moreover have n' ≤ sb.nxtAct (sKs p) t n'
      using ⟨∃ i≥n'. ≀sKs p≀_t i⟩ sb.nxtActLe by simp
    ultimately have sb.eval (sKs p) t t'' n' ?γ by simp
    moreover have sb.lNAct (sKs p) t n_r ≤ n'
      using ⟨n_r ≤ sb.nxtAct (sKs p) t n_r⟩
       ⟨sb.lNAct (sKs p) t n_r ≤ n_r⟩ ⟨sb.nxtAct (sKs p) t n_r ≤ n'⟩ by linarith
    ultimately show ?thesis using ⟨∃ i≥n_r. ≀sKs p≀_t i⟩ ⟨∃ i≥n'. ≀sKs p≀_t i⟩
      ⟨n'≥sb.lNAct (sKs p) t n'⟩ ⟨n' ≤ sb.nxtAct (sKs p) t n'⟩
      sb.evtIA[of n_r sKs p t n' t'' ?γ] by blast
  next
    assume ¬ (∃ i≥n'. ≀sKs p≀_t i)
    with lass have sb.eval (sKs p) t t'' n' ?γ ∧
      (∀ n''≥sb.lNAct (sKs p) t n_r. n'' < n' ⟶ sb.eval (sKs p) t t'' n'' ?γ')
      by auto
    moreover have sb.lNAct (sKs p) t n_r ≤ n'
      using ⟨n_r ≤ sb.nxtAct (sKs p) t n_r⟩ ⟨sb.lNAct (sKs p) t n_r ≤ n_r⟩
      ⟨sb.nxtAct (sKs p) t n_r ≤ n'⟩ by linarith
    ultimately show ?thesis using ⟨∃ i≥n_r. ≀sKs p≀_t i⟩ ⟨¬ (∃ i≥n'. ≀sKs p≀_t i)⟩
        sb.evtIA[of n_r sKs p t n' t'' ?γ] by blast
  qed
next
  assume cass: sb.eval (sKs p) t t'' n_r
    (sb.glob (¬ˢ (∃ₛ P'. (sb.pred (p'∈P') ∧ˢ sb.ba (λks. unsub P' = ksrp ks)))))

  have sub P = ksrp (kscmp (sKs p) (t (sb.nxtAct (sKs p) t n_r))) ∧
    p' ∈ P ⟶ (p', p) ∈ sb
  proof −
    have ∃ i≥0. ksactive (sKs p) (t i) using ⟨∃ i≥0. ksactive (sKs p) (t i)⟩
      by auto
    moreover have sb.lNAct (sKs p) t 0 ≤ (sb.nxtAct (sKs p) t n_r) by simp
    ultimately have sb.eval (sKs p) t t'' (sb.nxtAct (sKs p) t n_r)
```

$(sb.ba\ (\lambda ks.\ sub\ P = ksrp\ ks \wedge p' \in P \longrightarrow (p',\ p) \in sb))$
using $sb.globEA[OF - bhvks2[of\ t\ p\ sKs\ p\ t''\ P]]$ $\langle t \in arch\rangle$ *sks-prob* **by** *blast*
moreover from $\langle \exists\, i{\geq}n_r.\ ksactive\ (sKs\ p)\ (t\ i)\rangle$ **have**
$ksactive\ (sKs\ p)\ (t\ (sb.nxtAct\ (sKs\ p)\ t\ n_r))$ **using** *sb.nxtActI* **by** *blast*
ultimately show *?thesis*
using $sb.baEANow[of\ sKs\ p\ t\ t''\ sb.nxtAct\ (sKs\ p)\ t\ n_r]$ **by** *simp*
qed
with $\langle p' \in P\rangle$ **have** $(p',\ p) \in sb$
using $\langle sub\ P = ksrp\ (kscmp\ (sKs\ p)\ (t\ (sb.nxtAct\ (sKs\ p)\ t\ n_r)))\rangle$
sks-prob **by** *simp*
moreover from $\langle \exists\, i{\geq}n_r.\ ksactive\ (sKs\ p)\ (t\ i)\rangle$ **have**
$ksactive\ (sKs\ p)\ (t\ (sb.nxtAct\ (sKs\ p)\ t\ n_r))$ **using** *sb.nxtActI* **by** *blast*
with $\langle sub\ P = ksrp\ (kscmp\ (sKs\ p)\ (t\ (sb.nxtAct\ (sKs\ p)\ t\ n_r)))\rangle$
have $sub\ P \in bbrp\ (bbcmp\ the\text{-}bb\ (t\ (sb.nxtAct\ (sKs\ p)\ t\ n_r)))$
using *conn1A* **by** *auto*
with $\langle\ p' \in P\rangle$ **have**
$sub\ P \in bbrp\ (\sigma_{the\text{-}bb} t\ (sb.nxtAct\ (sKs\ p)\ t\ n_r)) \wedge p' \in P$ **by** *auto*
ultimately obtain m **where** $m{\geq}sb.nxtAct\ (sKs\ p)\ t\ n_r$ **and**
$(p',\ solve\ p') = bbcs\ (bbcmp\ the\text{-}bb\ (t\ m))$
using *indH* **by** *auto*

— and due to the publisher subscriber property,
— the knowledge source will receive them
moreover have
$\nexists\, n\ P.\ sb.nxtAct\ (sKs\ p)\ t\ n_r \leq n \wedge n \leq m \wedge ksactive\ (sKs\ p)\ (t\ n) \wedge$
$unsub\ P = ksrp\ (kscmp\ (sKs\ p)\ (t\ n)) \wedge p' \in P$
proof
assume $\exists\, n\ P'.\ sb.nxtAct\ (sKs\ p)\ t\ n_r \leq n \wedge n \leq m \wedge$
$ksactive\ (sKs\ p)\ (t\ n) \wedge$
$unsub\ P' = ksrp\ (kscmp\ (sKs\ p)\ (t\ n)) \wedge p' \in P'$
then obtain $n\ P'$ **where**
$ksactive\ (sKs\ p)\ (t\ n)$ **and** $sb.nxtAct\ (sKs\ p)\ t\ n_r \leq n$ **and** $n \leq m$ **and**
$unsub\ P' = ksrp\ (kscmp\ (sKs\ p)\ (t\ n))$ **and** $p' \in P'$ **by** *auto*
hence $sb.eval\ (sKs\ p)\ t\ t''\ n\ (\exists_s\ P'.\ sb.pred\ (p'{\in}P') \wedge^s$
$sb.ba\ (\lambda ks.\ unsub\ P' = ksrp\ ks))$ **by** *blast*
moreover have $sb.lNAct\ (sKs\ p)\ t\ n_r \leq n$
using $\langle n_r \leq sb.nxtAct\ (sKs\ p)\ t\ n_r\rangle$ $\langle sb.lNAct\ (sKs\ p)\ t\ n_r \leq n_r\rangle$
$\langle sb.nxtAct\ (sKs\ p)\ t\ n_r \leq n\rangle$ **by** *linarith*
with *cass* **have** $sb.eval\ (sKs\ p)\ t\ t''\ n\ (\neg^s\ (\exists_s\ P'.\ (sb.pred\ (p'{\in}P')$
$\wedge^s\ sb.ba\ (\lambda ks.\ unsub\ P' = ksrp\ ks))))$
using $sb.globEA[of\ n_r\ sKs\ p\ t\ t''$
$\neg^s\ (\exists_s P'.\ sb.pred\ (p' \in P') \wedge^s\ sb.ba\ (\lambda ks.\ unsub\ P' = ksrp\ ks))\ n]$
$\langle \exists\, i{\geq}n_r.\ ksactive\ (sKs\ p)\ (t\ i)\rangle$ **by** *auto*
ultimately show *False* **using** *sb.notE* **by** *auto*
qed
moreover from $\langle \exists\, i{\geq}n_r.\ ksactive\ (sKs\ p)\ (t\ i)\rangle$ **have**
$ksactive\ (sKs\ p)\ (t\ (sb.nxtAct\ (sKs\ p)\ t\ n_r))$ **using** *sb.nxtActI* **by** *blast*
moreover have $sub\ P = ksrp\ (kscmp\ (sKs\ p)\ (t\ (sb.nxtAct\ (sKs\ p)\ t\ n_r)))$
using $\langle sub\ P = ksrp\ (kscmp\ (sKs\ p)\ (t\ (sb.nxtAct\ (sKs\ p)\ t\ n_r)))\rangle$ **.**

```
         moreover from ⟨m≥sb.nxtAct (sKs p) t n_r⟩ have sb.nxtAct (sKs p) t n_r ≤ m
            by simp
          moreover from ⟨∃ i≥n_r. ksactive (sKs p) (t i)⟩
            have sb.nxtAct (sKs p) t n_r≥n_r using sb.nxtActI by blast
        hence m≥n_k using ⟨sb.nxtAct (sKs p) t n_r ≤ m⟩ ⟨sb.nxtAct (sKs p) t n_k ≤ n_r⟩
          ⟨sb.nxtAct (sKs p) t n_k ≥ n_k⟩ by simp
          with case-ass have ksactive (sKs p) (t m) by simp
          ultimately have (p', solve p') ∈ kscs (kscmp (sKs p) (t m))
            and ksactive (sKs p) (t m)
            using ⟨t ∈ arch⟩
            msgDelivery[of t sKs p sb.nxtAct (sKs p) t n_r P m p' solve p']
            ⟨p' ∈ P⟩ by auto
          hence sb.eval (sKs p) t t'' m (sb.ba (λks. (p',solve p') ∈ kscs ks))
            using sb.baIANow by simp
          moreover have m ≥ sb.lNAct (sKs p) t m by simp
          moreover from ⟨ksactive (sKs p) (t m)⟩ have m ≤ sb.nxtAct (sKs p) t m
            using sb.nxtActLe by auto
          moreover from ⟨∃ i≥n_r. ksactive (sKs p) (t i)⟩ have
            sb.lNAct (sKs p) t n_r ≤ sb.nxtAct (sKs p) t n_r by simp
          with ⟨sb.nxtAct (sKs p) t n_r ≤ m⟩ have sb.lNAct (sKs p) t n_r ≤ m by arith
          ultimately show sb.eval (sKs p) t t'' n_r
            (sb.evt (sb.ba (λks. (p',solve p') ∈ kscs ks)))
            using ⟨∃ i≥n_r. ksactive (sKs p) (t i)⟩ sb.evtIA by blast
        qed
      qed
    qed
    thus sb.eval (sKs p) t t'' n_r (sb.pred (p'∈P) ⟶^s
      (sb.evt (sb.ba (λks. (p',solve p') ∈ kscs ks))))
      using sb.impI by auto
  qed
  thus ?thesis using sb.allI by blast
qed

— Thus, the knowlege source will eventually solve the problem at hand...
ultimately have sb.eval (sKs p) t t'' n_r
  (sb.ba (λks. sub P = ksrp ks) ∧^s
  (∀_s q. (sb.pred (q ∈ P) ⟶^s sb.evt (sb.ba (λks. (q, solve q) ∈ kscs ks)))))
  using sb.conjI by simp
moreover from ⟨∃ i≥n_r. ksactive (sKs p) (t i)⟩ have ∃ i≥0. ksactive (sKs p) (t i)
  by blast
hence sb.eval (sKs p) t t'' n_r
  ((sb.ba (λks. sub P = ksrp ks) ∧^s
  (∀_s q. (sb.pred (q ∈ P) ⟶^s
  sb.evt (sb.ba (λks. (q, solve q) ∈ kscs ks))))) ⟶^s
  (sb.evt (sb.ba (λks. (p, solve p) = ksns ks)))) using ⟨t ∈ arch⟩
  sb.globEA[OF - bhvks1[of t p sKs p t'' P]] sks-prob by simp
ultimately have sb.eval (sKs p) t t'' n_r
  (sb.evt (sb.ba (λks. (p,solve(p))=ksns(ks))))
  using sb.impE[of sKs p t t'' n_r] by blast
```

```
— and forward it to the blackboard
then obtain n_s where n_s≥sb.nxtAct (sKs p) t n_r and
  (∃ i≥n_s. ksactive (sKs p) (t i) ∧
  (∀ n''≥sb.lNAct (sKs p) t n_s. n'' ≤ sb.nxtAct (sKs p) t n_s ⟶
  sb.eval (sKs p) t t'' n'' (sb.ba (λks. (p,solve(p))=ksns(ks))))) ∨
  ¬ (∃ i≥n_s. ksactive (sKs p) (t i)) ∧
  sb.eval (sKs p) t t'' n_s (sb.ba (λks. (p,solve(p))=ksns(ks)))
  using sb.evtEA[of n_r sKs p t] ‹∃ i≥n_r. ksactive (sKs p) (t i)› by blast
moreover from ‹sb.nxtAct (sKs p) t n_r ≥ n_r› ‹n_r≥n_k› ‹n_s≥sb.nxtAct (sKs p) t n_r›
  have n_s≥n_k by arith
with case-ass have ∃ i≥n_s. ksactive (sKs p) (t i) by auto
moreover have n_s≥sb.lNAct (sKs p) t n_s by simp
moreover from ‹∃ i≥n_s. ksactive (sKs p) (t i)› have n_s ≤ sb.nxtAct (sKs p) t n_s
  using sb.nxtActLe by simp
ultimately have sb.eval (sKs p) t t'' n_s (sb.ba (λks. (p,solve(p))=ksns(ks)))
  using sb.evtEA[of n_r sKs p t] ‹∃ i≥n_r. ksactive (sKs p) (t i)› by blast
with ‹∃ i≥n_s. ksactive (sKs p) (t i)› have
  (p,solve(p)) = ksns (kscmp (sKs p) (t (sb.nxtAct (sKs p) t n_s)))
  using sb.baEA[of n_s sKs p t t'' λks. (p, solve p) = ksns ks] by auto
moreover from ‹∃ i≥n_s. ksactive (sKs p) (t i)›
  have ksactive (sKs p) (t (sb.nxtAct (sKs p) t n_s)) using sb.nxtActI by simp
ultimately have (p,solve(p)) ∈ bbns (bbcmp the-bb (t (sb.nxtAct (sKs p) t n_s)))
  using conn1[OF pb.ts-prop(2)[of t (sb.nxtAct (sKs p) t n_s)]] by auto
hence pb.eval the-bb t t'
  (sb.nxtAct (sKs p) t n_s) (pb.ba (λbb. (p,solve(p)) ∈ bbns bb))
  using ‹t∈arch› pb.baI by simp

— finally, the blackboard will forward the solution which finishes the proof.
with bhvbb1 have pb.eval the-bb t t' (sb.nxtAct (sKs p) t n_s)
  (pb.evt (pb.ba (λbb. (p, solve p) = bbcs bb)))
  using ‹t∈arch› pb.globE pb.impE[of the-bb t t'] by blast
then obtain n_f where n_f≥sb.nxtAct (sKs p) t n_s and
  pb.eval the-bb t t' n_f (pb.ba (λbb. (p, solve p) = bbcs bb))
  using ‹t∈arch› pb.evtE[of t t' sb.nxtAct (sKs p) t n_s] by auto
hence (p, solve p) = bbcs (bbcmp the-bb (t n_f))
  using ‹t ∈ arch› pb.baEA by auto
moreover have n_f≥n_0
proof –
  from ‹∃ n'''≥n_s. ksactive (sKs p) (t n''')› have sb.nxtAct (sKs p) t n_s≥n_s
    using sb.nxtActLe by simp
  moreover from ‹n_k≥n› and ‹ksactive (sKs p) (t n_k)›
    have sb.nxtAct (sKs p) t n_k≥n_k
    using sb.nxtActI by blast
  ultimately show ?thesis
    using ‹n_f≥sb.nxtAct (sKs p) t n_s› ‹n_s≥sb.nxtAct (sKs p) t n_r›
    ‹sb.nxtAct (sKs p) t n_r≥n_r› ‹n_r≥sb.nxtAct (sKs p) t n_k› ‹n_k≥n› ‹n≥n_0› by arith
qed
ultimately show ?thesis by auto
```

```
         qed
        qed
       qed
      qed
     qed

     theorem pSolved:
      fixes t and t'::nat ⇒'BB and t''::nat ⇒'KS
      assumes t∈arch and
       ∀ n. (∃ n'≥n. ksactive (sKs (bbop(bbcmp the-bb (t n)))) (t n'))
      shows
       ∀ n. (∀ P. (sub P ∈ bbrp(bbcmp the-bb (t n))
        ⟶ (∀ p ∈ P. (∃ m≥n. (p,solve(p)) = bbcs (bbcmp the-bb (t m))))))
      using assms pSolved-Ind by blast

    end

    end
```

G Verification of Blockchain Architectures

G.1 Some Auxiliary Results

```
theory Auxiliary imports Main
begin

lemma disjE3: P ∨ Q ∨ R ⟹ (P ⟹ S) ⟹ (Q ⟹ S) ⟹ (R ⟹ S) ⟹ S by auto

lemma ge-induct[consumes 1, case-names step]:
  fixes i::nat and j::nat and P::nat ⇒ bool
  shows i ≤ j ⟹ (⋀n. i ≤ n ⟹ ((∀ m ≥ i.  m<n ⟶ P m) ⟹ P n)) ⟹ P j
proof −
  assume a0: i ≤ j and a1: (⋀n. i ≤ n ⟹ ((∀ m ≥ i.  m<n ⟶ P m) ⟹ P n))
  have (⋀n. ∀ m<n. i ≤ m ⟶ P m ⟹ i ≤ n ⟶ P n)
  proof
    fix n
    assume a2: ∀ m<n. i ≤ m ⟶ P m
    show i ≤ n ⟹ P n
    proof −
      assume i ≤ n
      with a1 have (∀ m ≥ i.  m<n ⟶ P m) ⟹ P n by simp
      moreover from a2 have ∀ m ≥ i.  m<n ⟶ P m by simp
      ultimately show P n by simp
    qed
  qed
  with nat-less-induct[of λj. i ≤ j ⟶ P j j] have i ≤ j ⟶ P j .
  with a0 show ?thesis by simp
qed

lemma my-induct[consumes 1, case-names base step]:
  fixes P::nat⇒bool
assumes less: i ≤ j
    and base: P j
    and step: ⋀n. i ≤ n ⟹ n < j ⟹ (∀ n'>n. n'≤j ⟶ P n') ⟹ P n
  shows P i
proof cases
  assume j=0
  thus ?thesis using less base by simp
next
  assume ¬ j=0
  have j − (j − i) ≥ i ⟶ P (j − (j − i))
  proof (rule less-induct[of λn::nat. j−n ≥ i ⟶ P (j−n) j−i])
```

```
    fix x assume asmp: ⋀y. y < x ⟹ i ≤ j − y ⟶ P (j − y)
    show i ≤ j − x ⟶ P (j − x)
    proof cases
      assume x=0
      with base show ?thesis by simp
    next
      assume ¬ x=0
      with ‹j ≠ 0› have j − x < j by simp
      show ?thesis
      proof
        assume i ≤ j − x
        moreover have ∀ n′>j−x. n′≤j ⟶ P n′
        proof
          fix n′
          show n′>j−x ⟶ n′≤j ⟶ P n′
          proof (rule HOL.impI[OF HOL.impI])
            assume j − x < n′ and n′ ≤ j
            hence j − n′ < x by simp
            moreover from ‹i ≤ j − x› ‹j − x < n′› have i ≤ n′
              using le-less-trans less-imp-le-nat by blast
            with ‹n′ ≤ j› have i ≤ j − (j − n′) by simp
            ultimately have  P (j − (j − n′)) using asmp by simp
            moreover from ‹n′ ≤ j› have j − (j − n′) = n′ by simp
            ultimately show P n′ by simp
          qed
        qed
        ultimately show P (j − x) using ‹j−x<j› step[of j−x] by simp
      qed
    qed
  qed
  moreover from less have j − (j − i) = i by simp
  ultimately show ?thesis by simp
qed

lemma Greatest-ex-le-nat: assumes ∃ k. P k ∧ (∀ k′. P k′ ⟶ k′ ≤ k) shows ¬(∃ n′>Greatest
P. P n′)
  by (metis Greatest-equality assms less-le-not-le)

lemma cardEx: assumes finite A and finite B and card A > card B shows ∃ x∈A. ¬ x∈B
proof cases
  assume A ⊆ B
  with assms have card A≤card B using card-mono by blast
  with assms have False by simp
  thus ?thesis by simp
next
  assume ¬ A ⊆ B
  thus ?thesis by auto
qed
```

```
lemma cardshift:
card {i::nat. i>n ∧ i ≤ n' ∧ p (n'' + i)} = card {i. i>(n + n'') ∧ i ≤ (n' + n'') ∧ p i}
proof –
  let ?f=λi. i+n''
  have bij-betw ?f {i::nat. i>n ∧ i ≤ n' ∧ p (n'' + i)} {i. i>(n + n'') ∧ i ≤ (n' + n'') ∧ p i}
  proof (rule bij-betwI')
    fix x y assume x ∈ {i. n < i ∧ i ≤ n' ∧ p (n'' + i)}
      and y ∈ {i. n < i ∧ i ≤ n' ∧ p (n'' + i)}
    show (x + n'' = y + n'') = (x = y) by simp
  next
    fix x::nat assume x ∈ {i. n < i ∧ i ≤ n' ∧ p (n'' + i)}
    hence n<x and x ≤ n' and p(n''+x) by auto
    moreover have n''+x=x+n'' by simp
    ultimately have n + n'' < x + n'' and x + n'' ≤ n' + n'' and p (x + n'') by auto
    thus x + n'' ∈ {i. n + n'' < i ∧ i ≤ n' + n'' ∧ p i} by auto
  next
    fix y::nat assume y ∈ {i. n + n'' < i ∧ i ≤ n' + n'' ∧ p i}
    hence n+n''<y and y≤n'+n'' and p y by auto
    then obtain x where x=y−n'' by simp
    with ‹n+n''<y› have y=x+n'' by simp
    moreover from ‹x=y−n''› ‹n+n''<y› have x>n by simp
    moreover from ‹x=y−n''› ‹y≤n'+n''› have x≤n' by simp
    moreover from ‹y=x+n''› have y=n''+x by simp
    with ‹p y› have p (n'' + x) by simp
    ultimately show ∃x∈{i. n < i ∧ i ≤ n' ∧ p (n'' + i)}. y = x + n'' by auto
  qed
  thus ?thesis using bij-betw-same-card by auto
qed

end
```

G.2 Relative Frequency LTL

```
theory RF-LTL
  imports Main HOL−Library.Sublist Auxiliary Dynamic-Architecture-Calculus
begin

type-synonym 's seq = nat ⇒ 's

abbreviation ccard n n' p ≡ card {i. i>n ∧ i ≤ n' ∧ p i}

lemma ccard-same:
  assumes ¬ p (Suc n')
  shows ccard n n' p = ccard n (Suc n') p
proof –
  have {i. i > n ∧ i ≤ Suc n' ∧ p i} = {i. i>n ∧ i ≤ n' ∧ p i}
  proof
    show {i. n < i ∧ i ≤ Suc n' ∧ p i} ⊆ {i. n < i ∧ i ≤ n' ∧ p i}
    proof
```

```
      fix x assume x ∈ {i. n < i ∧ i ≤ Suc n' ∧ p i}
      hence n<x and x≤Suc n' and p x by auto
      with assms (1) have x≠Suc n' by auto
      with ‹x≤Suc n'› have x ≤ n' by simp
      with ‹n<x› ‹p x› show x ∈ {i. n < i ∧ i ≤ n' ∧ p i} by simp
    qed
  next
    show {i. n < i ∧ i ≤ n' ∧ p i} ⊆ {i. n < i ∧ i ≤ Suc n' ∧ p i} by auto
  qed
  thus ?thesis by simp
qed

lemma ccard-zero[simp]:
  fixes n::nat
  shows ccard n n p = 0
  by auto

lemma ccard-inc:
  assumes p (Suc n')
    and n' ≥ n
  shows ccard n (Suc n') p = Suc (ccard n n' p)
proof −
  let ?A = {i. i > n ∧ i ≤ n' ∧ p i}
  have finite ?A by simp
  moreover have Suc n' ∉ ?A by simp
  ultimately have card (insert (Suc n') ?A) = Suc (card ?A)
    using card-insert-disjoint[of ?A] by simp
  moreover have insert (Suc n') ?A = {i. i>n ∧ i ≤ (Suc n') ∧ p i}
  proof
    show insert (Suc n') ?A ⊆ {i. n < i ∧ i ≤ Suc n' ∧ p i}
    proof
      fix x assume x ∈ insert (Suc n') {i. n < i ∧ i ≤ n' ∧ p i}
      hence x=Suc n' ∨ n < x ∧ x ≤ n' ∧ p x by simp
      thus x ∈ {i. n < i ∧ i ≤ Suc n' ∧ p i}
      proof
        assume x = Suc n'
        with assms (1) assms (2) show ?thesis by simp
      next
        assume n < x ∧ x ≤ n' ∧ p x
        thus ?thesis by simp
      qed
    qed
  next
    show {i. n < i ∧ i ≤ Suc n' ∧ p i} ⊆ insert (Suc n') ?A by auto
  qed
  ultimately show ?thesis by simp
qed

lemma ccard-mono:
```

```
  assumes n'≥n
  shows n''≥n' ⟹ ccard n (n''::nat) p ≥ ccard n n' p
proof (induction n'' rule: dec-induct)
  case base
  then show ?case ..
next
  case (step n'')
  then show ?case
  proof cases
    assume p (Suc n'')
    moreover from step.hyps assms have n≤n'' by simp
    ultimately have ccard n (Suc n'') p = Suc (ccard n n'' p)
      using ccard-inc[of p n'' n] by simp
    also have ... ≥ ccard n n' p using step.IH by simp
    finally show ?case .
  next
    assume ¬ p (Suc n'')
    moreover from step.hyps assms have n≤n'' by simp
    ultimately have ccard n (Suc n'') p = ccard n n'' p
      using ccard-same[of p n'' n] by simp
    also have ... ≥ ccard n n' p using step.IH by simp
    finally show ?case by simp
  qed
qed

lemma ccard-ub[simp]:
  ccard n n' p ≤ Suc n' − n
proof −
  have {i. i>n ∧ i ≤ n' ∧ p i} ⊆ {i. i≥n ∧ i ≤ n'} by auto
  hence ccard n n' p ≤ card {i. i≥n ∧ i ≤ n'} by (simp add: card-mono)
  moreover have {i. i≥n ∧ i ≤ n'} = {n..n'} by auto
  hence card {i. i≥n ∧ i ≤ n'} = Suc n' − n by simp
  ultimately show ?thesis by simp
qed

lemma ccard-sum:
  fixes n::nat
  assumes n'≥n''
    and n''≥n
  shows ccard n n' P = ccard n n'' P + ccard n'' n' P
proof −
  have ccard n n' P = card {i. i>n ∧ i ≤ n' ∧ P i} by simp
  moreover have {i. i>n ∧ i ≤ n' ∧ P i} =
    {i. i>n ∧ i ≤ n'' ∧ P i} ∪ {i. i>n'' ∧ i ≤ n' ∧ P i} (is ?LHS = ?RHS)
  proof
    show ?LHS ⊆ ?RHS by auto
  next
    show ?RHS ⊆ ?LHS
    proof
```

```
      fix x
      assume x∈?RHS
      hence x>n ∧ x ≤ n'' ∧ P x ∨ x>n'' ∧ x ≤ n' ∧ P x by auto
      thus x∈?LHS
      proof
        assume n < x ∧ x ≤ n'' ∧ P x
        with assms show ?thesis by simp
      next
        assume n'' < x ∧ x ≤ n' ∧ P x
        with assms show ?thesis by simp
      qed
    qed
  qed
  hence card ?LHS = card ?RHS by simp
  ultimately have ccard n n' P = card ?RHS by simp
  moreover have
    card ?RHS = card {i. i>n ∧ i ≤ n'' ∧ P i} + card {i. i>n'' ∧ i ≤ n' ∧ P i}
  proof (rule card-Un-disjoint)
    show finite {i. n < i ∧ i ≤ n'' ∧ P i} by simp
    show finite {i. n'' < i ∧ i ≤ n' ∧ P i} by simp
    show {i. n < i ∧ i ≤ n'' ∧ P i} ∩ {i. n'' < i ∧ i ≤ n' ∧ P i} = {} by auto
  qed
  moreover have ccard n n'' P = card {i. i>n ∧ i ≤ n'' ∧ P i} by simp
  moreover have ccard n'' n' P= card {i. i>n'' ∧ i ≤ n' ∧ P i} by simp
  ultimately show ?thesis by simp
qed

lemma ccard-ex:
  fixes n::nat
  shows c≥1 ⟹ c < ccard n n'' P ⟹ ∃n'<n''. n'>n ∧ ccard n n' P = c
proof (induction c rule: dec-induct)
  let ?l = LEAST i::nat. n < i ∧ i < n'' ∧ P i
  case base
  moreover have ccard n n'' P ≤ Suc (card {i. n < i ∧ i < n'' ∧ P i})
  proof −
    from ‹ccard n n'' P > 1› have n''>n using less-le-trans by force
    then obtain n' where Suc n' = n'' and Suc n' ≥ n by (metis lessE less-imp-le-nat)
    moreover have {i. n < i ∧ i < Suc n' ∧ P i} = {i. n < i ∧ i ≤ n' ∧ P i} by auto
    hence card {i. n < i ∧ i < Suc n' ∧ P i} = card {i. n < i ∧ i ≤ n' ∧ P i} by simp
    moreover have
      card {i. n < i ∧ i ≤ Suc n' ∧ P i} ≤ Suc (card {i. n < i ∧ i ≤ n' ∧ P i})
    proof cases
      assume P (Suc n')
      moreover from ‹n''>n› ‹Suc n'=n''› have n'≥n by simp
      ultimately show ?thesis using ccard-inc[of P n' n] by simp
    next
      assume ¬ P (Suc n')
      moreover from ‹n''>n› ‹Suc n'=n''› have n'≥n by simp
      ultimately show ?thesis using ccard-same[of P n' n] by simp
```

```
      qed
      ultimately show ?thesis by simp
    qed
    ultimately have card {i. n < i ∧ i < n″ ∧ P i} ≥ 1 by simp
    hence {i. n < i ∧ i < n″ ∧ P i} ≠ {} by fastforce
    hence ∃ i. n < i ∧ i < n″ ∧ P i by auto
    hence ?l>n and ?l<n″ and P ?l using LeastI-ex[of λi::nat. n < i ∧ i < n″ ∧ P i] by auto
    moreover have {i. n < i ∧ i ≤ ?l ∧ P i} = {?l}
    proof
      show {i. n < i ∧ i ≤ ?l ∧ P i} ⊆ {?l}
      proof
        fix i
        assume i∈{i. n < i ∧ i ≤ ?l ∧ P i}
        hence n < i and i ≤ ?l and P i by auto
        with ‹∃ i. n < i ∧ i < n″ ∧ P i› have i=?l
          using Least-le[of λi. n < i ∧ i < n″ ∧ P i] by (meson antisym le-less-trans)
        thus i∈{?l} by simp
      qed
    next
      show {?l} ⊆ {i. n < i ∧ i ≤ ?l ∧ P i}
      proof
        fix i
        assume i∈{?l}
        hence i=?l by simp
        with ‹?l>n› ‹?l<n″› ‹P ?l› show i∈{i. n < i ∧ i ≤ ?l ∧ P i} by simp
      qed
    qed
    hence ccard n ?l P = 1 by simp
    ultimately show ?case by auto
  next
    case (step c)
    moreover from step.prems have Suc c<ccard n n″ P by simp
    ultimately obtain n′ where n′<n″ and n < n′ and ccard n n′ P = c by auto
    hence ccard n n″ P = ccard n n′ P + ccard n′ n″ P using ccard-sum[of n′ n″ n] by simp
    with ‹Suc c<ccard n n″ P› ‹ccard n n′ P = c› have ccard n′ n″ P>1 by simp
    moreover have ccard n′ n″ P ≤ Suc (card {i. n′ < i ∧ i < n″ ∧ P i})
    proof –
      from ‹ccard n′ n″ P > 1› have n″>n′ using less-le-trans by force
      then obtain n‴ where Suc n‴ = n″ and Suc n‴ ≥ n′ by (metis lessE less-imp-le-nat)
      moreover have {i. n′ < i ∧ i < Suc n‴ ∧ P i} = {i. n′ < i ∧ i ≤ n‴ ∧ P i} by auto
      hence card {i. n′ < i ∧ i < Suc n‴ ∧ P i} = card {i. n′ < i ∧ i ≤ n‴ ∧ P i} by simp
      moreover have
        card {i. n′ < i ∧ i ≤ Suc n‴ ∧ P i} ≤ Suc (card {i. n′ < i ∧ i ≤ n‴ ∧ P i})
      proof cases
        assume P (Suc n‴)
        moreover from ‹n″>n′› ‹Suc n‴=n″› have n‴≥n′ by simp
        ultimately show ?thesis using ccard-inc[of P n‴ n′] by simp
      next
        assume ¬ P (Suc n‴)
```

```
      moreover from ‹n''>n'› ‹Suc n'''=n''› have n'''≥n' by simp
      ultimately show ?thesis using ccard-same[of P n''' n'] by simp
    qed
    ultimately show ?thesis by simp
  qed
  ultimately have card {i. n' < i ∧ i < n'' ∧ P i} ≥ 1 by simp
  hence {i. n' < i ∧ i < n'' ∧ P i} ≠ {} by fastforce
  hence ∃ i. n' < i ∧ i < n'' ∧ P i by auto
  let ?l = LEAST i::nat. n' < i ∧ i < n'' ∧ P i
  from ‹∃ i. n' < i ∧ i < n'' ∧ P i› have n' < ?l
    using LeastI-ex[of λi::nat. n' < i ∧ i < n'' ∧ P i] by auto
  with ‹n < n'› have ccard n ?l P = ccard n n' P + ccard n' ?l P using ccard-sum[of n' ?l n]
by simp
  moreover have {i. n' < i ∧ i ≤ ?l ∧ P i} = {?l}
  proof
    show {i. n' < i ∧ i ≤ ?l ∧ P i} ⊆ {?l}
    proof
      fix i
      assume i∈{i. n' < i ∧ i ≤ ?l ∧ P i}
      hence n' < i and i ≤ ?l and P i by auto
      with ‹∃ i. n' < i ∧ i < n'' ∧ P i› have i=?l
        using Least-le[of λi. n' < i ∧ i < n'' ∧ P i] by (meson antisym le-less-trans)
      thus i∈{?l} by simp
    qed
  next
    show {?l} ⊆ {i. n' < i ∧ i ≤ ?l ∧ P i}
    proof
      fix i
      assume i∈{?l}
      hence i=?l by simp
      moreover from ‹∃ i. n' < i ∧ i < n'' ∧ P i› have ?l<n'' and P ?l
        using LeastI-ex[of λi. n' < i ∧ i < n'' ∧ P i] by auto
      ultimately show i∈{i. n' < i ∧ i ≤ ?l ∧ P i} using ‹?l>n'› by simp
    qed
  qed
  hence ccard n' ?l P = 1 by simp
  ultimately have card {i. n < i ∧ i ≤ ?l ∧ P i} = Suc c using ‹ccard n n' P = c› by simp
  moreover from ‹∃ i. n' < i ∧ i < n'' ∧ P i› have n' < ?l and ?l < n'' and P ?l
    using LeastI-ex[of λi::nat. n' < i ∧ i < n'' ∧ P i] by auto
  with ‹n < n'› have n<?l and ?l<n'' by auto
  ultimately show ?case by auto
qed

lemma ccard-freq:
  assumes (n'::nat)≥n
    and ccard n n' P > ccard n n' Q + cnf
  shows ∃ n' n''. ccard n' n'' P > cnf ∧ ccard n' n'' Q ≤ cnf
proof cases
  assume cnf = 0
```

```
  with assms(2) have ccard n n′ P > ccard n n′ Q by simp
  hence card {i. n < i ∧ i ≤ n′ ∧ P i}>card {i. n < i ∧ i ≤ n′ ∧ Q i}
    (is card ?LHS>card ?RHS) by simp
  then obtain i where i∈?LHS and ¬ i ∈ ?RHS and i>0 using cardEx[of ?LHS ?RHS] by auto
  hence P i and ¬ Q i by auto
  with ‹i>0› obtain n″ where P (Suc n″) and ¬Q (Suc n″) using gr0-implies-Suc by auto
  hence ccard n″ (Suc n″) P = 1 using ccard-inc by auto
  with ‹cnf = 0› have ccard n″ (Suc n″) P > cnf by simp
  moreover from ‹¬Q (Suc n″)› have ccard n″ (Suc n″) Q = 0
    using ccard-same[of Q n″ n″] by auto
  with ‹cnf = 0› have ccard n″ (Suc n″) Q ≤ cnf by simp
  ultimately show ?thesis by auto
next
  assume ¬ cnf = 0
  show ?thesis
  proof (rule ccontr)
    assume ¬ (∃ n′ n″. ccard n′ n″ P > cnf ∧ ccard n′ n″ Q ≤ cnf)
    hence hyp: ∀ n′ n″. ccard n′ n″ Q ≤ cnf ⟶ ccard n′ n″ P ≤ cnf
      using leI less-imp-le-nat by blast
    show False
    proof cases
      assume ccard n n′ Q ≤ cnf
      with hyp have ccard n n′ P ≤ cnf by simp
      with assms show False by simp
    next
      let ?gcond=λn″. n″≥n ∧ n″≤n′ ∧ (∃ x≥1. ccard n n″ Q = x ∗ cnf)
      let ?g= GREATEST n″. ?gcond n″
      assume ¬ ccard n n′ Q ≤ cnf
      hence ccard n n′ Q > cnf by simp
      hence ∃ n″. ?gcond n″
      proof −
        from ‹ccard n n′ Q > cnf› ‹¬cnf=0› obtain n″
          where n″>n and n″≤n′ and ccard n n″ Q = cnf
          using ccard-ex[of cnf n n′ Q ] by auto
        moreover from ‹ccard n n″ Q = cnf› have ∃ x≥1. ccard n n″ Q = x ∗ cnf by auto
        ultimately show ?thesis using less-imp-le-nat by blast
      qed
      moreover have ∀ n″>n′. ¬ ?gcond n″ by simp
      ultimately have gex: ∃ n″. ?gcond n″ ∧ (∀ n‴. ?gcond n‴ ⟶ n‴≤n″)
        using boundedGreatest[of ?gcond - n′] by blast
      hence ∃ x≥1. ccard n ?g Q = x ∗ cnf and ?g ≥ n
        using GreatestI-ex-nat[of ?gcond] by auto
      moreover {fix n″
      have n″≥n ⟹ ∃ x≥1. ccard n n″ Q = x ∗ cnf ⟹ ccard n n″ P ≤ ccard n n″ Q
      proof (induction n″ rule: ge-induct)
        case (step n′)
        from step.prems obtain x where x≥1 and cas: ccard n n′ Q = x ∗ cnf by auto
        then show ?case
```

```
  proof cases
    assume x=1
    with cas have ccard n n' Q = cnf by simp
    with hyp have ccard n n' P ≤ cnf by simp
    with ‹ccard n n' Q = cnf› show ?thesis by simp
  next
    assume ¬x=1
    with ‹x≥1› have x>1 by simp
    hence x−1 ≥ 1 by simp
    moreover from ‹cnf≠0› ‹x−1 ≥ 1›
      have (x−1) ∗ cnf < x ∗ cnf ∧ (x − 1) ∗ cnf ≠ 0 by auto
    with ‹x−1 ≥ 1› ‹cnf≠0›‹ccard n n' Q = x ∗ cnf› obtain n''
      where n''>n and n''<n' and ccard n n'' Q = (x−1) ∗ cnf
      using ccard-ex[of (x−1)∗cnf n n' Q ] by auto
    ultimately have ∃x≥1. ccard n n'' Q = x ∗ cnf and n''≥n by auto
    with ‹n''≥n› ‹n''<n'› have ccard n n'' P ≤ ccard n n'' Q using step.IH by simp
    moreover have ccard n'' n' Q = cnf
    proof −
      from ‹x−1 ≥ 1› have x∗cnf = ((x−1) ∗ cnf) + cnf
        using semiring-normalization-rules(2)[of (x − 1) cnf] by simp
      with ‹ccard n n'' Q = (x−1) ∗ cnf› ‹ccard n n' Q = x ∗ cnf›
      have ccard n n' Q = ccard n n'' Q + cnf by simp
     moreover from ‹n''≥n› ‹n''<n'› have ccard n n' Q = ccard n n'' Q + ccard n'' n' Q
        using ccard-sum[of n'' n' n] by simp
      ultimately show ?thesis by simp
    qed
    moreover from ‹ccard n'' n' Q = cnf› have ccard n'' n' P ≤ cnf using hyp by simp
    ultimately show ?thesis using ‹n''≥n› ‹n''<n'› ccard-sum[of n'' n' n] by simp
  qed
qed } note geq = this
ultimately have ccard n ?g P ≤ ccard n ?g Q by simp
moreover have ccard ?g n' P ≤ cnf
proof (rule ccontr)
  assume ¬ ccard ?g n' P ≤ cnf
  hence ccard ?g n' P > cnf by simp
  have ccard ?g n' Q > cnf
  proof (rule ccontr)
    assume ¬ccard ?g n' Q > cnf
    hence ccard ?g n' Q ≤ cnf by simp
    with ‹ccard ?g n' P > cnf› show False
      using ‹¬ (∃n' n''. ccard n' n'' P > cnf ∧ ccard n' n'' Q ≤ cnf)› by simp
  qed
  with ‹¬ cnf=0› obtain n'' where n''>?g and n''<n' and ccard ?g n'' Q = cnf
    using ccard-ex[of cnf ?g n' Q] by auto
  moreover have ∃x≥1. ccard n n'' Q = x ∗ cnf
  proof −
    from ‹∃x≥1. ccard n ?g Q = x ∗ cnf› obtain x
      where x≥1 and ccard n ?g Q = x ∗ cnf by auto
    from ‹n''>?g› ‹?g≥n› have ccard n n'' Q = ccard n ?g Q + ccard ?g n'' Q
```

```
            using ccard-sum[of ?g n'' n Q] by simp
          with ‹ccard n ?g Q = x ∗ cnf› have ccard n n'' Q = x ∗ cnf + ccard ?g n'' Q by simp
          with ‹ccard ?g n'' Q = cnf› have ccard n n'' Q = Suc x ∗ cnf by simp
          thus ?thesis by auto
        qed
        moreover from ‹n''>?g› ‹?g≥n› have n''≥n by simp
        ultimately have ∃ n''>?g. ?gcond n'' by auto
        moreover from gex have ∀ n'''. ?gcond n''' ⟶ n'''≤?g
          using Greatest-le-nat[of ?gcond] by auto
        ultimately show False by auto
      qed
      moreover from gex have n'≥?g
        using GreatestI-ex-nat[of ?gcond] by auto
      ultimately have ccard n n' P≤ccard n n' Q + cnf
        using ccard-sum[of ?g n' n] using ‹?g ≥ n› by simp
      with assms show False by simp
    qed
  qed
qed

locale trusted =
  fixes bc:: ('a list) seq
    and n::nat
  assumes growth: n'≠0 ⟹ n'≤n ⟹ bc n' = bc (n'−1) ∨ (∃ b. bc n' = bc (n' − 1) @ b)
begin
end

locale untrusted =
  fixes bc:: ('a list) seq
    and mining::bool seq
  assumes growth:
    ⋀n::nat. prefix (bc (Suc n)) (bc n) ∨ (∃ b::'a. bc (Suc n) = bc n @ [b]) ∧ mining (Suc n)
begin

lemma prefix-save:
  assumes prefix sbc (bc n')
    and ∀ n'''>n'. n'''≤n'' ⟶ length (bc n''') ≥ length sbc
  shows n''≥n' ⟹ prefix sbc (bc n'')
proof (induction n'' rule: dec-induct)
  case base
  with assms(1) show ?case by simp
next
  case (step n)
  from growth[of n] show ?case
  proof
    assume prefix (bc (Suc n)) (bc n)
    moreover from step.hyps have length (bc (Suc n)) ≥ length sbc using assms(2) by simp
    ultimately show ?thesis using step.IH using prefix-length-prefix by auto
  next
```

```
    assume (∃ b. bc (Suc n) = bc n @ [b]) ∧ mining (Suc n)
    with step.IH show ?thesis by auto
  qed
qed

theorem prefix-length:
  assumes prefix sbc (bc n') and ¬ prefix sbc (bc n'') and n'≤n''
  shows ∃ n'''>n'. n'''≤n'' ∧ length (bc n''') < length sbc
proof (rule ccontr)
  assume ¬ (∃ n'''>n'. n'''≤n'' ∧ length (bc n''') < length sbc)
  hence ∀ n'''>n'. n'''≤n'' ⟶ length (bc n''') ≥ length sbc by auto
  with assms have prefix sbc (bc n'') using prefix-save[of sbc n' n''] by simp
  with assms (2) show False by simp
qed

theorem grow-mining:
  assumes length (bc n) < length (bc (Suc n))
  shows mining (Suc n)
  using assms growth leD prefix-length-le by blast

lemma length-suc-length:
  length (bc (Suc n)) ≤ Suc (length (bc n))
  by (metis eq-iff growth le-SucI length-append-singleton prefix-length-le)

end

locale untrusted-growth =
  fixes bc:: nat seq
    and mining:: nat ⇒ bool
  assumes as1: ⋀n::nat. bc (Suc n) ≤ Suc (bc n)
    and as2: ⋀n::nat. bc (Suc n) > bc n ⟹ mining (Suc n)
begin

end

sublocale untrusted ⊆ untrusted-growth λn. length (bc n) using grow-mining length-suc-length
by unfold-locales auto

context untrusted-growth
begin
  theorem ccard-diff-lgth:
    n'≥n ⟹ ccard n n' (λn. mining n) ≥ (bc n' − bc n)
  proof (induction n' rule: dec-induct)
    case base
    then show ?case by simp
    next
    case (step n')
    from as1 have bc (Suc n') < Suc (bc n') ∨ bc (Suc n') = Suc (bc n')
      using le-neq-implies-less by blast
```

```
    then show ?case
    proof
      assume bc (Suc n′) < Suc (bc n′)
      hence bc (Suc n′) − bc n ≤ bc n′ − bc n by simp
      moreover from step.hyps have
        ccard n (Suc n′) (λn. mining n) ≥ ccard n n′ (λn. mining n)
        using ccard-mono[of n n′ Suc n′] by simp
      ultimately show ?thesis using step.IH by simp
    next
      assume bc (Suc n′) = Suc (bc n′)
      hence bc (Suc n′) − bc n ≤ Suc (bc n′ − bc n) by simp
      moreover from ‹bc (Suc n′) = Suc (bc n′)› have mining (Suc n′) using as2 by simp
      with step.hyps have ccard n (Suc n′) (λn. mining n) ≥ Suc (ccard n n′ (λn. mining n))
        using ccard-inc by simp
      ultimately show ?thesis using step.IH by simp
    qed
  qed
end

locale trusted-growth =
  fixes bc:: nat seq
    and mining:: nat ⇒ bool
    and init:: nat
  assumes as1: ⋀n::nat. bc (Suc n) ≥ bc n
    and as2: ⋀n::nat. mining (Suc n) ⟹ bc (Suc n) > bc n
begin
  lemma grow-mono: n′≥n⟹bc n′≥bc n
  proof (induction n′ rule: dec-induct)
    case base
    then show ?case by simp
  next
    case (step n′)
    then show ?case using as1[of n′] by simp
  qed

  theorem ccard-diff-lgth:
    shows n′≥n ⟹ bc n′ − bc n ≥ ccard n n′ (λn. mining n)
  proof (induction n′ rule: dec-induct)
    case base
    then show ?case by simp
  next
    case (step n′)
    then show ?case
    proof cases
      assume mining (Suc n′)
      with as2 have bc (Suc n′) > bc n′ by simp
      moreover from step.hyps have bc n′≥bc n using grow-mono by simp
      ultimately have bc (Suc n′) − bc n > bc n′ − bc n by simp
      moreover from as1 have bc (Suc n′) − bc n ≥ bc n′ − bc n by (simp add: diff-le-mono)
```

```
      moreover from ‹mining (Suc n')› step.hyps
        have ccard n (Suc n') (λn. mining n) ≤ Suc (ccard n n' (λn. mining n))
        using ccard-inc by simp
      ultimately show ?thesis using step.IH by simp
    next
      assume ¬ mining (Suc n')
      hence ccard n (Suc n') (λn. mining n) ≤ (ccard n n' (λn. mining n))
        using ccard-same by simp
      moreover from as1 have bc (Suc n') − bc n ≥ bc n' − bc n by (simp add: diff-le-mono)
      ultimately show ?thesis using step.IH by simp
    qed
  qed
end

locale bounded-growth = tg: trusted-growth tbc tmining + ug: untrusted-growth ubc umining
    for tbc:: nat seq
    and ubc:: nat seq
    and tmining:: nat ⇒ bool
    and umining:: nat ⇒ bool
    and sbc::nat
    and cnf::nat +
  assumes fair: ⋀n n'. ccard n n' (λn. umining n) > cnf ⟹ ccard n n' (λn. tmining n) > cnf
    and a2: tbc 0 ≥ sbc+cnf
    and a3: ubc 0 < sbc
begin

theorem tr-upper-bound: shows ubc n < tbc n
proof (rule ccontr)
  assume ¬ ubc n < tbc n
  hence ubc n ≥ tbc n by simp
  moreover from a2 a3 have tbc 0 > ubc 0 + cnf by simp
  moreover have tbc n≥tbc 0 using tg.grow-mono by simp
  ultimately have ubc n − ubc 0 > tbc n − tbc 0 + cnf by simp
  moreover have ccard 0 n (λn. tmining n) ≤ tbc n − tbc 0 using tg.ccard-diff-lgth by simp
  moreover have ubc n − ubc 0 ≤ ccard 0 n (λn. umining n) using ug.ccard-diff-lgth by simp
  ultimately have ccard 0 n (λn. umining n) > ccard 0 n (λn. tmining n) + cnf by simp
  hence ∃n' n''. ccard n' n'' (λn. umining n) > cnf ∧ ccard n' n'' (λn. tmining n) ≤ cnf
    using ccard-freq by blast
  with fair show False using leD by blast
qed

end

end
```

G.3 A Theory of Blockchain Architectures

```
theory Blockchain imports Auxiliary Dynamic-Architecture-Calculus RF-LTL
begin
```

G.3.1 Blockchains

A blockchain itself is modeled as a simple list.

```
type-synonym 'a BC = 'a list

abbreviation max-cond:: ('a BC) set ⇒ 'a BC ⇒ bool
  where max-cond B b ≡ b ∈ B ∧ (∀ b'∈B. length b' ≤ length b)

no-syntax
  -MAX1     :: pttrns ⇒ 'b ⇒ 'b           ((3MAX -./ -) [0, 10] 10)
  -MAX      :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b  ((3MAX -:-./ -) [0, 0, 10] 10)
  -MAX1     :: pttrns ⇒ 'b ⇒ 'b           ((3MAX -./ -) [0, 10] 10)
  -MAX      :: pttrn ⇒ 'a set ⇒ 'b ⇒ 'b  ((3MAX -∈-./ -) [0, 0, 10] 10)

definition MAX:: ('a BC) set ⇒ 'a BC
  where MAX B = (SOME b. max-cond B b)

lemma max-ex:
  fixes XS::('a BC) set
  assumes XS ≠ {}
    and finite XS
  shows ∃ xs∈XS. (∀ ys∈XS. length ys ≤ length xs)
proof (rule Finite-Set.finite-ne-induct)
  show finite XS using assms by simp
next
  from assms show XS ≠ {} by simp
next
  fix x::'a BC
  show ∃ xs∈{x}. ∀ ys∈{x}. length ys ≤ length xs by simp
next
  fix zs::'a BC and F::('a BC) set
  assume finite F and F ≠ {} and zs ∉ F and ∃ xs∈F. ∀ ys∈F. length ys ≤ length xs
  then obtain xs where xs∈F and ∀ ys∈F. length ys ≤ length xs by auto
  show ∃ xs∈insert zs F. ∀ ys∈insert zs F. length ys ≤ length xs
  proof (cases)
    assume length zs ≥ length xs
    with ‹∀ ys∈F. length ys ≤ length xs› show ?thesis by auto
  next
    assume ¬ length zs ≥ length xs
    hence length zs ≤ length xs by simp
    with ‹xs ∈ F› show ?thesis using ‹∀ ys∈F. length ys ≤ length xs› by auto
  qed
qed

lemma max-prop:
  fixes XS::('a BC) set
  assumes XS ≠ {}
    and finite XS
  shows MAX XS ∈ XS
```

```
    and ∀ b'∈XS. length b' ≤ length (MAX XS)
proof –
  from assms have ∃ xs∈XS. ∀ ys∈XS. length ys ≤ length xs using max-ex[of XS] by auto
  with MAX-def[of XS] show MAX XS ∈ XS and ∀ b'∈XS. length b' ≤ length (MAX XS)
    using someI-ex[of λb. b ∈ XS ∧ (∀ b'∈XS. length b' ≤ length b)] by auto
qed

lemma max-less:
  fixes b::'a BC and b'::'a BC and B::('a BC) set
  assumes b∈B
    and finite B
    and length b > length b'
  shows length (MAX B) > length b'
proof –
  from assms have ∃ xs∈B. ∀ ys∈B. length ys ≤ length xs using max-ex[of B] by auto
  with MAX-def[of B] have ∀ b'∈B. length b' ≤ length (MAX B)
    using someI-ex[of λb. b ∈ B ∧ (∀ b'∈B. length b' ≤ length b)] by auto
  with ‹b∈B› have length b ≤ length (MAX B) by simp
  with ‹length b > length b'› show ?thesis by simp
qed
```

G.3.2 Blockchain Architectures

In the following we describe the locale for blockchain architectures.

```
locale Blockchain = dynamic-component cmp active
  for active :: 'nid ⇒ cnf ⇒ bool (⌇-⌇_ [0,110]60)
    and cmp :: 'nid ⇒ cnf ⇒ 'ND (σ_(-) [0,110]60) +
  fixes pin :: 'ND ⇒ ('nid BC) set
    and pout :: 'ND ⇒ 'nid BC
    and bc :: 'ND ⇒ 'nid BC
    and mining :: 'ND ⇒ bool
    and trusted :: 'nid ⇒ bool
    and actTr :: cnf ⇒ 'nid set
    and actUt :: cnf ⇒ 'nid set
    and PoW:: trace ⇒ nat ⇒ nat
    and tmining:: trace ⇒ nat ⇒ bool
    and umining:: trace ⇒ nat ⇒ bool
    and cb:: nat
  defines actTr k ≡ {nid. ⌇nid⌇_k ∧ trusted nid}
    and actUt k ≡ {nid. ⌇nid⌇_k ∧ ¬ trusted nid}
    and PoW t n ≡ (LEAST x. ∀ nid∈actTr (t n). length (bc (σ_nid(t n))) ≤ x)
    and tmining t ≡ (λn. ∃ nid∈actTr (t n). mining (σ_nid(t n)))
    and umining t ≡ (λn. ∃ nid∈actUt (t n). mining (σ_nid(t n)))
  assumes consensus: ⋀nid t t' bc'::('nid BC). ⟦trusted nid⟧ ⟹ eval nid t t' 0
    (□_b(ba (λnd. bc' =
      (if (∃ b∈pin nd. length b > length (bc nd)) then (MAX (pin nd)) else (bc nd)))
      ⟶^b ○_b(ba (λnd.(¬ mining nd ∧ bc nd = bc' ∨ mining nd ∧ (∃ b. bc nd = bc' @ [b]))))))
    and attacker: ⋀nid t t' bc'. ⟦¬ trusted nid⟧ ⟹ eval nid t t' 0
    (□_b(ba (λnd. bc' = (SOME b. b ∈ (pin nd ∪ {bc nd})))) ⟶^b
```

$\bigcirc_b$ $(ba\ (\lambda nd.(\neg\ mining\ nd \wedge prefix\ (bc\ nd)\ bc' \vee mining\ nd \wedge (\exists\, b.\ bc\ nd = bc'\ @\ [b]))))))$
and *forward*: $\bigwedge nid\ t\ t'.\ eval\ nid\ t\ t'\ 0\ (\Box_b(ba\ (\lambda nd.\ pout\ nd = bc\ nd)))$
— At each time point a node will forward its blockchain to the network
and *init*: $\bigwedge nid\ t\ t'.\ eval\ nid\ t\ t'\ 0\ (ba\ (\lambda nd.\ bc\ nd{=}[]))$
and *conn*: $\bigwedge k\ nid.\ [\![active\ nid\ k;\ trusted\ nid]\!]$
$\Longrightarrow pin\ (cmp\ nid\ k) = (\bigcup nid'{\in}actTr\ k.\ \{pout\ (cmp\ nid'\ k)\})$
and *act*: $\bigwedge t\ n{::}nat.\ finite\ \{nid{::}'nid.\ \wr nid\wr_{t\ n}\}$
and *actTr*: $\bigwedge t\ n{::}nat.\ \exists\, nid.\ trusted\ nid \wedge \wr nid\wr_{t\ n} \wedge \wr nid\wr_{t\ (Suc\ n)}$
and *fair*: $\bigwedge n\ n'.\ ccard\ n\ n'\ (umining\ t) > cb \Longrightarrow ccard\ n\ n'\ (tmining\ t) > cb$
and *closed*: $\bigwedge t\ nid\ b\ n{::}nat.\ [\![\wr nid\wr_{t\ n};\ b \in pin\ (\sigma_{nid}(t\ n))]\!] \Longrightarrow$
$\exists\, nid'.\ \wr nid'\wr_{t\ n} \wedge bc\ (\sigma_{nid'}(t\ n)) = b$
and *mine*: $\bigwedge t\ nid\ n{::}nat.\ [\![trusted\ nid;\ \wr nid\wr_{t\ (Suc\ n)};\ mining\ (\sigma_{nid}(t\ (Suc\ n)))]\!] \Longrightarrow \wr nid\wr_{t\ n}$
begin

lemma *init-model*:
assumes $\neg\ (\exists\, n'.\ latestAct\text{-}cond\ nid\ t\ n\ n')$
and $\wr nid\wr_{t\ n}$
shows $bc\ (\sigma_{nid}t\ n) = []$
proof –
from *assms(2)* **have** $\exists\, i{\geq}0.\ \wr nid\wr_{t\ i}$ **by** *auto*
with *init* **have** $bc\ (\sigma_{nid}t\ \langle nid \rightarrow t\rangle_0) = []$ **using** *baEA[of 0 nid t]* **by** *blast*
moreover from *assms* **have** $n{=}\langle nid \rightarrow t\rangle_0$ **using** *nxtAct-eq* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *fwd-bc*:
fixes *nid* **and** $t{::}nat \Rightarrow cnf$ **and** $t'{::}nat \Rightarrow 'ND$
assumes $\wr nid\wr_{t\ n}$
shows $pout\ (\sigma_{nid}t\ n) = bc\ (\sigma_{nid}t\ n)$
using *assms forward globEANow[THEN baEANow[of nid t t′ n]]* **by** *blast*

lemma *finite-input*:
fixes *t n nid*
assumes $\wr nid\wr_{t\ n}$
defines $dep\ nid' \equiv pout\ (\sigma_{nid'}(t\ n))$
shows $finite\ (pin\ (cmp\ nid\ (t\ n)))$
proof –
have $finite\ \{nid'.\ \wr nid'\wr_{t\ n}\}$ **using** *act* **by** *auto*
moreover have $pin\ (cmp\ nid\ (t\ n)) \subseteq dep\ `\ \{nid'.\ \wr nid'\wr_{t\ n}\}$
proof
fix x **assume** $x \in pin\ (cmp\ nid\ (t\ n))$
show $x \in dep\ `\ \{nid'.\ \wr nid'\wr_{t\ n}\}$
proof –
from *assms* **obtain** nid' **where** $\wr nid'\wr_{t\ n}$ **and** $bc\ (\sigma_{nid'}(t\ n)) = x$
using *closed* ‹$x \in pin\ (cmp\ nid\ (t\ n))$› **by** *blast*
hence $pout\ (\sigma_{nid'}(t\ n)) = x$ **using** *fwd-bc* **by** *auto*
hence $x{=}dep\ nid'$ **using** *dep-def* **by** *simp*
moreover from ‹$\wr nid'\wr_{t\ n}$› **have** $nid' \in \{nid'.\ \wr nid'\wr_{t\ n}\}$ **by** *simp*
ultimately show *?thesis* **using** *image-eqI* **by** *simp*

qed
qed
ultimately show *?thesis* **using** *finite-surj* **by** *metis*
qed

lemma *nempty-input*:
fixes $t\ n\ nid$
assumes $\wr nid \wr_{t\ n}$
and *trusted nid*
shows $pin\ (cmp\ nid\ (t\ n)) \neq \{\}$ **using** *conn*[*of nid t n*] *act assms actTr-def* **by** *auto*

lemma *onlyone*:
assumes $\exists\, n' \geq n.\ \wr tid \wr_{t\ n'}$
and $\exists\, n' < n.\ \wr tid \wr_{t\ n'}$
shows $\exists!i.\ \langle tid \leftarrow t\rangle_n \leq i \wedge i < \langle tid \rightarrow t\rangle_n \wedge \wr tid \wr_{t\ i}$
proof
show $\langle tid \leftarrow t\rangle_n \leq \langle tid \leftarrow t\rangle_n \wedge \langle tid \leftarrow t\rangle_n < \langle tid \rightarrow t\rangle_n \wedge \wr tid \wr_{t\ \langle tid \leftarrow t\rangle_n}$
by (*metis assms dynamic-component.nxtActI latestAct-prop*(*1*) *latestAct-prop*(*2*) *less-le-trans order-refl*)
next
fix i
show $\langle tid \leftarrow t\rangle_n \leq i \wedge i < \langle tid \rightarrow t\rangle_n \wedge \wr tid \wr_{t\ i} \Longrightarrow i = \langle tid \leftarrow t\rangle_n$
by (*metis latestActless*(*1*) *leI le-less-Suc-eq le-less-trans nxtActI order-refl*)
qed

G.3.2.1 Component Behavior

lemma *bhv-tr-ex*:
fixes t **and** $t'{::}nat \Rightarrow {'}ND$ **and** tid
assumes *trusted tid*
and $\exists\, n' \geq n.\ \wr tid \wr_{t\ n'}$
and $\exists\, n' < n.\ \wr tid \wr_{t\ n'}$
and $\exists\, b \in pin\ (\sigma_{tid} t\ \langle tid \leftarrow t\rangle_n).\ length\ b > length\ (bc\ (\sigma_{tid} t\ \langle tid \leftarrow t\rangle_n))$
shows $\neg\ mining\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) \wedge bc\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) =$
$Blockchain.MAX\ (pin\ (\sigma_{tid} t\ \langle tid \leftarrow t\rangle_n)) \vee mining\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) \wedge$
$(\exists\, b.\ bc\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) = Blockchain.MAX\ (pin\ (\sigma_{tid} t\ \langle tid \leftarrow t\rangle_n))\ @\ [b])$
proof −
let $?cond = \lambda nd.\ MAX\ (pin\ (\sigma_{tid} t\ \langle tid \leftarrow t\rangle_n)) =$
$(if\ (\exists\, b \in pin\ nd.\ length\ b > length\ (bc\ nd))\ then\ (MAX\ (pin\ nd))\ else\ (bc\ nd))$
let $?check = \lambda nd.\ \neg\ mining\ nd \wedge bc\ nd = MAX\ (pin\ (\sigma_{tid} t\ \langle tid \leftarrow t\rangle_n)) \vee mining\ nd \wedge$
$(\exists\, b.\ bc\ nd = MAX\ (pin\ (\sigma_{tid} t\ \langle tid \leftarrow t\rangle_n))\ @\ [b])$
from ‹*trusted tid*› **have** $eval\ tid\ t\ t'\ 0\ ((\Box_b((ba\ ?cond) \longrightarrow^b \bigcirc_b\ (ba\ ?check))))$
using *consensus*[*of tid - - MAX* $(pin\ (\sigma_{tid} t\ \langle tid \leftarrow t\rangle_n))$] **by** *simp*
moreover from *assms* **have** $\exists\, i \geq 0.\ \wr tid \wr_{t\ i}$ **by** *auto*
moreover have $\langle tid \Leftarrow t\rangle_0 \leq \langle tid \leftarrow t\rangle_n$ **by** *simp*
ultimately have $eval\ tid\ t\ t'\ \langle tid \leftarrow t\rangle_n\ (ba\ (?cond) \longrightarrow^b \bigcirc_b\ (ba\ ?check))$
using *globEA*[*of 0 tid t t'* $((ba\ ?cond) \longrightarrow^b \bigcirc_b\ (ba\ ?check))\ \langle tid \leftarrow t\rangle_n$] **by** *fastforce*
moreover have $eval\ tid\ t\ t'\ \langle tid \leftarrow t\rangle_n\ (ba\ (?cond))$
proof (*rule baIA*)

```
    from ‹∃ n'<n. ⌇tid⌇_t n'› show ∃ i≥⟨tid ← t⟩_n. ⌇tid⌇_t i using latestAct-prop(1) by blast
    from assms(3) assms(4) show ?cond (σ_tid t ⟨tid → t⟩_⟨tid ← t⟩_n)
      using latestActNxt by simp
  qed
  ultimately have eval tid t t' ⟨tid ← t⟩_n (○_b (ba ?check))
    using impE[of tid t t' - ba (?cond) ○_b (ba ?check)] by simp
  moreover have ∃ i>⟨tid → t⟩_⟨tid ← t⟩_n. ⌇tid⌇_t i
  proof –
    from assms have ⟨tid → t⟩_n>⟨tid ← t⟩_n using latestActNxtAct by simp
    with assms(3) have ⟨tid → t⟩_n>⟨tid → t⟩_⟨tid ← t⟩_n using latestActNxt by simp
    moreover from ‹∃ n'≥n. ⌇tid⌇_t n'› have ⌇tid⌇_t ⟨tid → t⟩_n using nxtActI by simp
    ultimately show ?thesis by auto
  qed
  moreover from assms have ⟨tid ← t⟩_n ≤ ⟨tid → t⟩_n
    using latestActNxtAct by (simp add: order.strict-implies-order)
  moreover from assms have ∃!i. ⟨tid ← t⟩_n ≤ i ∧ i < ⟨tid → t⟩_n ∧ ⌇tid⌇_t i
    using onlyone by simp
  ultimately have eval tid t t' ⟨tid → t⟩_n (ba ?check)
    using nxtEA1[of tid t ⟨tid ← t⟩_n t' ba ?check ⟨tid → t⟩_n] by simp
  moreover from ‹∃ n'≥n. ⌇tid⌇_t n'› have ⌇tid⌇_t ⟨tid → t⟩_n using nxtActI by simp
  ultimately show ?thesis using baEANow[of tid t t' ⟨tid → t⟩_n ?check] by simp
qed

lemma bhv-tr-in:
  fixes t and t'::nat ⇒ 'ND and tid
  assumes trusted tid
    and ∃ n'≥n. ⌇tid⌇_t n'
    and ∃ n'<n. ⌇tid⌇_t n'
    and ¬ (∃ b∈pin (σ_tid t ⟨tid ← t⟩_n). length b > length (bc (σ_tid t ⟨tid ← t⟩_n)))
  shows ¬ mining (σ_tid t ⟨tid → t⟩_n) ∧ bc (σ_tid t ⟨tid → t⟩_n) = bc (σ_tid t ⟨tid ← t⟩_n) ∨
    mining (σ_tid t ⟨tid → t⟩_n) ∧
    (∃ b. bc (σ_tid t ⟨tid → t⟩_n) = bc (σ_tid t ⟨tid ← t⟩_n) @ [b])
proof –
  let ?cond = λnd. bc (σ_tid t ⟨tid ← t⟩_n) =
    (if (∃ b∈pin nd. length b > length (bc nd)) then (MAX (pin nd)) else (bc nd))
  let ?check = λnd. ¬ mining nd ∧ bc nd = bc (σ_tid t ⟨tid ← t⟩_n) ∨
    mining nd ∧ (∃ b. bc nd = bc (σ_tid t ⟨tid ← t⟩_n) @ [b])
  from ‹trusted tid› have eval tid t t' 0 ((□_b((ba ?cond) ⟶^b ○_b (ba ?check))))
    using consensus[of tid - - bc (σ_tid t ⟨tid ← t⟩_n)] by simp
  moreover from assms have ∃ i≥0. ⌇tid⌇_t i by auto
  moreover have ⟨tid ⇐ t⟩_0 ≤ ⟨tid ← t⟩_n by simp
  ultimately have eval tid t t' ⟨tid ← t⟩_n (ba (?cond) ⟶^b ○_b (ba ?check))
    using globEA[of 0 tid t t' (ba ?cond) ⟶^b ○_b (ba ?check) ⟨tid ← t⟩_n] by fastforce
  moreover have eval tid t t' ⟨tid ← t⟩_n (ba (?cond))
  proof (rule baIA)
    from ‹∃ n'<n. ⌇tid⌇_t n'› show ∃ i≥⟨tid ← t⟩_n. ⌇tid⌇_t i using latestAct-prop(1) by blast
    from assms(3) assms(4) show ?cond (σ_tid t ⟨tid → t⟩_⟨tid ← t⟩_n)
      using latestActNxt by simp
```

```
  qed
  ultimately have eval tid t t' ⟨tid ← t⟩_n (○_b (ba ?check))
    using impE[of tid t t' - ba (?cond) ○_b (ba ?check)] by simp
  moreover have ∃ i>⟨tid → t⟩_⟨tid ← t⟩_n. ≀tid≀_t i
  proof –
    from assms have ⟨tid → t⟩_n>⟨tid ← t⟩_n using latestActNxtAct by simp
    with assms(3) have ⟨tid → t⟩_n>⟨tid → t⟩_⟨tid ← t⟩_n using latestActNxt by simp
    moreover from ‹∃ n'≥n. ≀tid≀_t n'› have ≀tid≀_t ⟨tid → t⟩_n using nxtActI by simp
    ultimately show ?thesis by auto
  qed
  moreover from assms have ⟨tid ← t⟩_n ≤ ⟨tid → t⟩_n
    using latestActNxtAct by (simp add: order.strict-implies-order)
  moreover from assms have ∃!i. ⟨tid ← t⟩_n ≤ i ∧ i < ⟨tid → t⟩_n ∧ ≀tid≀_t i
    using onlyone by simp
  ultimately have eval tid t t' ⟨tid → t⟩_n (ba ?check)
    using nxtEA1[of tid t ⟨tid ← t⟩_n t' ba ?check ⟨tid → t⟩_n] by simp
  moreover from ‹∃ n'≥n. ≀tid≀_t n'› have ≀tid≀_t ⟨tid → t⟩_n using nxtActI by simp
  ultimately show ?thesis using baEANow[of tid t t' ⟨tid → t⟩_n ?check] by simp
qed

lemma bhv-tr-context:
  assumes trusted tid
      and ≀tid≀_t n
      and ∃ n'<n. ≀tid≀_t n'
    shows ∃ nid'. ≀nid'≀_t ⟨tid ← t⟩_n ∧
      (mining (σ_tid t n) ∧ (∃ b. bc (σ_tid t n) = bc (σ_nid' t ⟨tid ← t⟩_n) @ [b]) ∨
      ¬ mining (σ_tid t n) ∧ bc (σ_tid t n) = bc (σ_nid' t ⟨tid ← t⟩_n))
proof cases
  assume casmp: ∃ b∈pin (σ_tid t ⟨tid ← t⟩_n). length b > length (bc (σ_tid t ⟨tid ← t⟩_n))
  moreover from assms(2) have ∃ n'≥n. ≀tid≀_t n' by auto
  moreover from assms(3) have ∃ n'<n. ≀tid≀_t n' by auto
  ultimately have ¬ mining (σ_tid t ⟨tid → t⟩_n) ∧
    bc (σ_tid t ⟨tid → t⟩_n) = Blockchain.MAX (pin (σ_tid t ⟨tid ← t⟩_n)) ∨
    mining (σ_tid t ⟨tid → t⟩_n) ∧
    (∃ b. bc (σ_tid t ⟨tid → t⟩_n) = Blockchain.MAX (pin (σ_tid t ⟨tid ← t⟩_n)) @ [b])
    using assms(1) bhv-tr-ex by auto
  moreover from assms(2) have ⟨tid → t⟩_n = n using nxtAct-active by simp
  ultimately have
    ¬ mining (σ_tid t ⟨tid → t⟩_n) ∧ bc (σ_tid t n) =
      Blockchain.MAX (pin (σ_tid t ⟨tid ← t⟩_n)) ∨
    mining (σ_tid t ⟨tid → t⟩_n) ∧ (∃ b. bc (σ_tid t n) =
      Blockchain.MAX (pin (σ_tid t ⟨tid ← t⟩_n)) @ [b])
    by simp
  moreover from assms(2) have ⟨tid → t⟩_n = n using nxtAct-active by simp
  ultimately have ¬ mining (σ_tid t n) ∧
    bc (σ_tid t n) = Blockchain.MAX (pin (σ_tid t ⟨tid ← t⟩_n)) ∨
    mining (σ_tid t n) ∧ (∃ b. bc (σ_tid t n) = Blockchain.MAX (pin (σ_tid t ⟨tid ← t⟩_n)) @ [b])
      by simp
```

```
    moreover have Blockchain.MAX (pin (σ_{tid}t ⟨tid ← t⟩_{n})) ∈ pin (σ_{tid}t ⟨tid ← t⟩_{n})
    proof –
      from ‹∃ n'<n. ⦃tid⦄_{t n'}› have ⦃tid⦄_{t ⟨tid ← t⟩_{n}} using latestAct-prop(1) by simp
      hence finite (pin (σ_{tid}(t ⟨tid ← t⟩_{n}))) using finite-input[of tid t ⟨tid ← t⟩_{n}] by simp
      moreover from casmp obtain b where
        b ∈ pin (σ_{tid}t ⟨tid ← t⟩_{n}) and length b > length (bc (σ_{tid}t ⟨tid ← t⟩_{n})) by auto
      ultimately show ?thesis using max-prop(1) by auto
    qed
    with ‹∃ n'<n. ⦃tid⦄_{t n'}› obtain nid where ⦃nid⦄_{t ⟨tid ← t⟩_{n}}
      and bc (σ_{nid}t ⟨tid ← t⟩_{n}) = Blockchain.MAX (pin (σ_{tid}t ⟨tid ← t⟩_{n})) using
      closed[of tid t ⟨tid ← t⟩_{n} MAX (pin (σ_{tid}t ⟨tid ← t⟩_{n}))] latestAct-prop(1) by auto
    ultimately show ?thesis by auto
  next
    assume ¬ (∃ b∈pin (σ_{tid}t ⟨tid ← t⟩_{n}). length b > length (bc (σ_{tid}t ⟨tid ← t⟩_{n})))
    moreover from assms(2) have ∃ n'≥n. ⦃tid⦄_{t n'} by auto
    moreover from assms(3) have ∃ n'<n. ⦃tid⦄_{t n'} by auto
    ultimately have ¬ mining (σ_{tid}t ⟨tid → t⟩_{n}) ∧ bc (σ_{tid}t ⟨tid → t⟩_{n}) = bc (σ_{tid}t ⟨tid ← t⟩_{n})
      ∨ mining (σ_{tid}t ⟨tid → t⟩_{n}) ∧ (∃ b. bc (σ_{tid}t ⟨tid → t⟩_{n}) = bc (σ_{tid}t ⟨tid ← t⟩_{n}) @ [b])
      using assms(1) bhv-tr-in[of tid n t] by auto
    moreover from assms(2) have ⟨tid → t⟩_{n} = n using nxtAct-active by simp
    ultimately have ¬ mining (σ_{tid}t n) ∧ bc (σ_{tid}t n) = bc (σ_{tid}t ⟨tid ← t⟩_{n}) ∨
      mining (σ_{tid}t n) ∧ (∃ b. bc (σ_{tid}t n) = bc (σ_{tid}t ⟨tid ← t⟩_{n}) @ [b]) by simp
    moreover from ‹∃ n'. latestAct-cond tid t n n'› have ⦃tid⦄_{t ⟨tid ← t⟩_{n}}
      using latestAct-prop(1) by simp
    ultimately show ?thesis by auto
  qed

  lemma bhv-ut:
    fixes t and t'::nat ⇒ 'ND and uid
    assumes ¬ trusted uid
      and ∃ n'≥n. ⦃uid⦄_{t n'}
      and ∃ n'<n. ⦃uid⦄_{t n'}
    shows ¬ mining (σ_{uid}t ⟨uid → t⟩_{n}) ∧
            prefix (bc (σ_{uid}t ⟨uid → t⟩_{n}))
            (SOME b. b ∈ pin (σ_{uid}t ⟨uid ← t⟩_{n}) ∪ {bc (σ_{uid}t ⟨uid ← t⟩_{n})})
           ∨ mining (σ_{uid}t ⟨uid → t⟩_{n}) ∧
            (∃ b. bc (σ_{uid}t ⟨uid → t⟩_{n}) =
          (SOME b. b ∈ pin (σ_{uid}t ⟨uid ← t⟩_{n}) ∪ {bc (σ_{uid}t ⟨uid ← t⟩_{n})}) @ [b])
  proof –
    let ?cond = λnd. (SOME b. b ∈ (pin (σ_{uid}t ⟨uid ← t⟩_{n}) ∪
      {bc (σ_{uid}t ⟨uid ← t⟩_{n})})) = (SOME b. b ∈ pin nd ∪ {bc nd})
    let ?check = λnd. ¬ mining nd ∧ prefix (bc nd)
      (SOME b. b ∈ pin (σ_{uid}t ⟨uid ← t⟩_{n}) ∪ {bc (σ_{uid}t ⟨uid ← t⟩_{n})})
      ∨ mining nd ∧ (∃ b. bc nd = (SOME b. b ∈ pin (σ_{uid}t ⟨uid ← t⟩_{n}) ∪
      {bc (σ_{uid}t ⟨uid ← t⟩_{n})}) @ [b])
    from ‹¬ trusted uid› have eval uid t t' 0 ((□_{b}((ba ?cond) ⟶^{b} ○_{b} (ba ?check))))
      using attacker[of uid - - (SOME b. b ∈ pin (σ_{uid}t ⟨uid ← t⟩_{n}) ∪ {bc (σ_{uid}t ⟨uid ← t⟩_{n})})]
      by simp
    moreover from assms have ∃ i≥0. ⦃uid⦄_{t i} by auto
```

```
  moreover have ⟨uid ⇐ t⟩_0 ≤ ⟨uid ← t⟩_n by simp
  ultimately have eval uid t t' ⟨uid ← t⟩_n (ba (?cond) ⟶^b ○_b(ba ?check))
    using globEA[of 0 uid t t' ((ba ?cond) ⟶^b ○_b(ba ?check)) ⟨uid ← t⟩_n] by fastforce
  moreover have eval uid t t' ⟨uid ← t⟩_n (ba (?cond))
  proof (rule baIA)
    from ‹∃ n'<n. ≀uid≀_t n'› show ∃ i≥⟨uid ← t⟩_n. ≀uid≀_t i using latestAct-prop(1) by blast
    with assms(3) show ?cond (σ_uid t ⟨uid → t⟩_⟨uid ← t⟩_n) using latestActNxt by simp
  qed
  ultimately have eval uid t t' ⟨uid ← t⟩_n (○_b (ba ?check))
    using impE[of uid t t' - ba (?cond) ○_b (ba ?check)] by simp
  moreover have ∃ i>⟨uid → t⟩_⟨uid ← t⟩_n. ≀uid≀_t i
  proof –
    from assms have ⟨uid → t⟩_n>⟨uid ← t⟩_n using latestActNxtAct by simp
    with assms(3) have ⟨uid → t⟩_n>⟨uid → t⟩_⟨uid ← t⟩_n using latestActNxt by simp
    moreover from ‹∃ n'≥n. ≀uid≀_t n'› have ≀uid≀_t ⟨uid → t⟩_n using nxtActI by simp
    ultimately show ?thesis by auto
  qed
  moreover from assms have ⟨uid ← t⟩_n ≤ ⟨uid → t⟩_n
    using latestActNxtAct by (simp add: order.strict-implies-order)
  moreover from assms have ∃!i. ⟨uid ← t⟩_n ≤ i ∧ i < ⟨uid → t⟩_n ∧ ≀uid≀_t i
    using onlyone by simp
  ultimately have eval uid t t' ⟨uid → t⟩_n (ba ?check)
    using nxtEA1[of uid t ⟨uid ← t⟩_n t' ba ?check ⟨uid → t⟩_n] by simp
  moreover from ‹∃ n'≥n. ≀uid≀_t n'› have ≀uid≀_t ⟨uid → t⟩_n using nxtActI by simp
  ultimately show ?thesis using baEANow[of uid t t' ⟨uid → t⟩_n ?check] by simp
qed

lemma bhv-ut-context:
  assumes ¬ trusted uid
      and ≀uid≀_t n
      and ∃ n'<n. ≀uid≀_t n'
  shows ∃ nid'. ≀nid'≀_t ⟨uid ← t⟩_n ∧ (mining (σ_uid t n) ∧
  (∃ b. prefix (bc (σ_uid t n)) (bc (σ_nid' t ⟨uid ← t⟩_n) @ [b]))
  ∨ ¬ mining (σ_uid t n) ∧ prefix (bc (σ_uid t n)) (bc (σ_nid' t ⟨uid ← t⟩_n)))
proof –
  let ?bc=SOME b. b ∈ pin (σ_uid t ⟨uid ← t⟩_n) ∪ {bc (σ_uid t ⟨uid ← t⟩_n)}
  have bc-ex: ?bc ∈ pin (σ_uid t ⟨uid ← t⟩_n) ∨ ?bc ∈ {bc (σ_uid t ⟨uid ← t⟩_n)}
  proof –
    have ∃ b. b∈ pin (σ_uid t ⟨uid ← t⟩_n) ∪ {bc (σ_uid t ⟨uid ← t⟩_n)} by auto
    hence ?bc ∈ pin (σ_uid t ⟨uid ← t⟩_n) ∪ {bc (σ_uid t ⟨uid ← t⟩_n)} using someI-ex by simp
    thus ?thesis by auto
  qed

  from assms(2) have ∃ n'≥n. ≀uid≀_t n' by auto
  moreover from assms(3) have ∃ n'<n. ≀uid≀_t n' by auto
  ultimately have ¬ mining (σ_uid t ⟨uid → t⟩_n) ∧ prefix (bc (σ_uid t ⟨uid → t⟩_n)) ?bc ∨
    mining (σ_uid t ⟨uid → t⟩_n) ∧ (∃ b. bc (σ_uid t ⟨uid → t⟩_n) = ?bc @ [b])
    using bhv-ut[of uid n t] assms(1) by simp
```

moreover from *assms(2)* **have** $\langle uid \rightarrow t\rangle_{n} = n$ **using** *nxtAct-active* **by** *simp*
ultimately have *casmp*: $\neg\ mining\ (\sigma_{uid}t\ n) \wedge prefix\ (bc\ (\sigma_{uid}t\ n))\ ?bc\ \vee$
$mining\ (\sigma_{uid}t\ n) \wedge (\exists\, b.\ bc\ (\sigma_{uid}t\ n) = ?bc\ @\ [b])$ **by** *simp*

from *bc-ex* **have** $?bc \in pin\ (\sigma_{uid}t\ \langle uid \leftarrow t\rangle_{n}) \vee ?bc \in \{bc\ (\sigma_{uid}t\ \langle uid \leftarrow t\rangle_{n})\}$ **.**
thus *?thesis*
proof
assume $?bc \in pin\ (\sigma_{uid}t\ \langle uid \leftarrow t\rangle_{n})$
moreover from ‹$\exists\, n'{<}n.\ \wr uid\wr_{t\ n'}$› **have** $\wr uid\wr_{t\ \langle uid \leftarrow t\rangle_{n}}$
using *latestAct-prop(1)* **by** *simp*
ultimately obtain *nid* **where** $\wr nid\wr_{t\ \langle uid \leftarrow t\rangle_{n}}$ **and** $bc\ (\sigma_{nid}t\ \langle uid \leftarrow t\rangle_{n}) = ?bc$
using *closed* **by** *blast*
with *casmp* **have** $\neg\ mining\ (\sigma_{uid}t\ n) \wedge prefix\ (bc\ (\sigma_{uid}t\ n))\ (bc\ (\sigma_{nid}t\ \langle uid \leftarrow t\rangle_{n})) \vee$
$mining\ (\sigma_{uid}t\ n) \wedge (\exists\, b.\ bc\ (\sigma_{uid}t\ n) = (bc\ (\sigma_{nid}t\ \langle uid \leftarrow t\rangle_{n}))\ @\ [b])$ **by** *simp*
with ‹$\wr nid\wr_{t\ \langle uid \leftarrow t\rangle_{n}}$› **show** *?thesis* **by** *auto*
next
assume $?bc \in \{bc\ (\sigma_{uid}t\ \langle uid \leftarrow t\rangle_{n})\}$
hence $?bc = bc\ (\sigma_{uid}t\ \langle uid \leftarrow t\rangle_{n})$ **by** *simp*
moreover from ‹$\exists\, n'.\ latestAct\text{-}cond\ uid\ t\ n\ n'$› **have** $\wr uid\wr_{t\ \langle uid \leftarrow t\rangle_{n}}$
using *latestAct-prop(1)* **by** *simp*
ultimately show *?thesis* **using** *casmp* **by** *auto*
qed
qed

G.3.2.2 Maximal Trusted Blockchains

abbreviation *mbc-cond*:: $trace \Rightarrow nat \Rightarrow {'}nid \Rightarrow bool$
where $mbc\text{-}cond\ t\ n\ nid \equiv nid{\in}actTr\ (t\ n) \wedge (\forall\, nid'{\in}actTr\ (t\ n).\ length\ (bc\ (\sigma_{nid'}(t\ n))) \leq length\ (bc\ (\sigma_{nid}(t\ n))))$

lemma *mbc-ex*:
fixes *t n*
shows $\exists\, x.\ mbc\text{-}cond\ t\ n\ x$
proof –
let $?ALL{=}\{b.\ \exists\, nid{\in}actTr\ (t\ n).\ b = bc\ (\sigma_{nid}(t\ n))\}$
have $MAX\ ?ALL \in\ ?ALL$
proof (*rule max-prop*)
from *actTr* **have** $actTr\ (t\ n) \neq \{\}$ **using** *actTr-def* **by** *blast*
thus $?ALL{\neq}\{\}$ **by** *auto*
from *act* **have** $finite\ (actTr\ (t\ n))$ **using** *actTr-def* **by** *simp*
thus $finite\ ?ALL$ **by** *simp*
qed
then obtain *nid* **where** $nid \in actTr\ (t\ n) \wedge bc\ (\sigma_{nid}(t\ n)) = MAX\ ?ALL$ **by** *auto*
moreover have $\forall\, nid'{\in}actTr\ (t\ n).\ length\ (bc\ (\sigma_{nid'}(t\ n))) \leq length\ (MAX\ ?ALL)$
proof
fix *nid*
assume $nid \in actTr\ (t\ n)$
hence $bc\ (\sigma_{nid}(t\ n)) \in\ ?ALL$ **by** *auto*
moreover have $\forall\, b'{\in}?ALL.\ length\ b' \leq length\ (MAX\ ?ALL)$

```
    proof (rule max-prop)
      from ‹bc (σ_nid(t n)) ∈ ?ALL› show ?ALL≠{} by auto
      from act have finite (actTr (t n)) using actTr-def by simp
      thus finite ?ALL by simp
    qed
    ultimately show
      length (bc (σ_nid t n)) ≤ length (Blockchain.MAX {b. ∃nid∈actTr (t n). b = bc (σ_nid t n)})
by simp
  qed
  ultimately show ?thesis by auto
qed

definition MBC:: trace ⇒ nat ⇒ 'nid
  where MBC t n = (SOME b. mbc-cond t n b)

lemma mbc-prop[simp]:
  shows mbc-cond t n (MBC t n)
  using someI-ex[OF mbc-ex] MBC-def by simp
```

G.3.2.3 Trusted Proof of Work

An important construction is the maximal proof of work available in the trusted community. The construction was already introduces in the locale itself since it was used to express some of the locale assumptions.

```
abbreviation pow-cond:: trace ⇒ nat ⇒ nat ⇒ bool
  where pow-cond t n n' ≡ ∀nid∈actTr (t n). length (bc (σ_nid(t n))) ≤ n'

lemma pow-ex:
  fixes t n
  shows pow-cond t n (length (bc (σ_MBC t n(t n))))
    and ∀x'. pow-cond t n x' ⟶ x'≥length (bc (σ_MBC t n(t n)))
  using mbc-prop by auto

lemma pow-prop:
  pow-cond t n (PoW t n)
proof -
  from pow-ex have pow-cond t n (LEAST x. pow-cond t n x)
    using LeastI-ex[of pow-cond t n] by blast
  thus ?thesis using PoW-def by simp
qed

lemma pow-eq:
  fixes n
  assumes ∃tid∈actTr (t n). length (bc (σ_tid(t n))) = x
    and ∀tid∈actTr (t n). length (bc (σ_tid(t n))) ≤ x
  shows PoW t n = x
proof -
  have (LEAST x. pow-cond t n x) = x
  proof (rule Least-equality)
```

```
    from assms(2) show ∀ nid∈actTr (t n). length (bc (σ_{nid}t n)) ≤ x by simp
  next
    fix y
    assume ∀ nid∈actTr (t n). length (bc (σ_{nid}t n)) ≤ y
    thus x ≤ y using assms(1) by auto
  qed
  with PoW-def show ?thesis by simp
qed

lemma pow-mbc:
  shows length (bc (σ_{MBC t n}t n)) = PoW t n
  by (metis mbc-prop pow-eq)

lemma pow-less:
  fixes t n nid
  assumes pow-cond t n x
  shows PoW t n ≤ x
proof –
  from pow-ex assms have (LEAST x. pow-cond t n x) ≤ x using Least-le[of pow-cond t n] by
blast
  thus ?thesis using PoW-def by simp
qed

lemma pow-le-max:
  assumes trusted tid
    and ⦃tid⦄_{t n}
  shows PoW t n ≤ length (MAX (pin (σ_{tid}t n)))
proof –
  from mbc-prop have trusted (MBC t n) and ⦃MBC t n⦄_{t n} using actTr-def by auto
  hence pout (σ_{MBC t n}t n) = bc (σ_{MBC t n}t n)
    using forward globEANow[THEN baEANow[of MBC t n t t' n λnd. pout nd = bc nd]]
    by auto
  with assms ‹⦃MBC t n⦄_{t n}› ‹trusted (MBC t n)› have bc (σ_{MBC t n}t n) ∈ pin (σ_{tid}t n)
    using conn actTr-def by auto
  moreover from assms (2) have finite (pin (σ_{tid}t n)) using finite-input[of tid t n] by simp
  ultimately have length (bc (σ_{MBC t n}t n)) ≤ length (MAX (pin (σ_{tid}t n)))
    using max-prop(2) by auto
  with pow-mbc show ?thesis by simp
qed

lemma pow-ge-lgth:
  assumes trusted tid
    and ⦃tid⦄_{t n}
  shows length (bc (σ_{tid}t n)) ≤ PoW t n
proof –
  from assms have tid ∈ actTr (t n) using actTr-def by simp
  thus ?thesis using pow-prop by simp
qed
```

lemma *pow-le-lgth*:
assumes *trusted tid*
and $\wr tid \wr_{t\ n}$
and $\neg(\exists\, b{\in}pin\ (\sigma_{tid} t\ n).\ length\ b > length\ (bc\ (\sigma_{tid} t\ n)))$
shows $length\ (bc\ (\sigma_{tid} t\ n)) \geq PoW\ t\ n$
proof –
from *assms* (*3*) **have** $\forall\, b{\in}pin\ (\sigma_{tid} t\ n).\ length\ b \leq length\ (bc\ (\sigma_{tid} t\ n))$ **by** *auto*
moreover from *assms nempty-input*[*of tid t n*] *finite-input*[*of tid t n*]
have $MAX\ (pin\ (\sigma_{tid} t\ n)) \in pin\ (\sigma_{tid} t\ n)$ **using** *max-prop*(*1*)[*of pin* $(\sigma_{tid} t\ n)$] **by** *simp*
ultimately have $length\ (MAX\ (pin\ (\sigma_{tid} t\ n))) \leq length\ (bc\ (\sigma_{tid} t\ n))$ **by** *simp*
moreover from *assms* **have** $PoW\ t\ n \leq length\ (MAX\ (pin\ (\sigma_{tid} t\ n)))$
using *pow-le-max* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *pow-mono*:
shows $n'{\geq}n \Longrightarrow PoW\ t\ n' \geq PoW\ t\ n$
proof (*induction* n' *rule*: *dec-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step* n')
hence $PoW\ t\ n \leq PoW\ t\ n'$ **by** *simp*
moreover have $PoW\ t\ (Suc\ n') \geq PoW\ t\ n'$
proof –
from *actTr* **obtain** *tid* **where** *trusted tid* **and** $\wr tid \wr_{t\ n'}$ **and** $\wr tid \wr_{t\ (Suc\ n')}$ **by** *auto*
show *?thesis*
proof *cases*
assume $\exists\, b{\in}pin\ (\sigma_{tid} t\ n').\ length\ b > length\ (bc\ (\sigma_{tid} t\ n'))$
moreover from $\langle \wr tid \wr_{t\ (Suc\ n')} \rangle$ **have** $\langle tid \rightarrow t \rangle_{Suc\ n'} = Suc\ n'$
using *nxtAct-active* **by** *simp*
moreover from $\langle \wr tid \wr_{t\ n'} \rangle$ **have** $\langle tid \leftarrow t \rangle_{Suc\ n'} = n'$
using *latestAct-prop*(*2*) *latestActless le-less-Suc-eq* **by** *blast*
moreover from $\langle \wr tid \wr_{t\ n'} \rangle$ **have** $\exists\, n''{<}Suc\ n'.\ \wr tid \wr_{t\ n''}$ **by** *blast*
moreover from $\langle \wr tid \wr_{t\ (Suc\ n')} \rangle$ **have** $\exists\, n''{\geq}Suc\ n'.\ \wr tid \wr_{t\ n''}$ **by** *auto*
ultimately have $bc\ (\sigma_{tid} t\ (Suc\ n')) = Blockchain.MAX\ (pin\ (\sigma_{tid} t\ n')) \vee$
$(\exists\, b.\ bc\ (\sigma_{tid} t\ (Suc\ n')) = Blockchain.MAX\ (pin\ (\sigma_{tid} t\ n'))\ @\ b)$
using ‹*trusted tid*› *bhv-tr-ex*[*of tid Suc* n' *t*] **by** *auto*
hence $length\ (bc\ (\sigma_{tid} t\ (Suc\ n'))) \geq length\ (Blockchain.MAX\ (pin\ (\sigma_{tid} t\ n')))$ **by** *auto*
moreover from ‹*trusted tid*› $\langle \wr tid \wr_{t\ n'} \rangle$
have $length\ (Blockchain.MAX\ (pin\ (\sigma_{tid} t\ n'))) \geq PoW\ t\ n'$ **using** *pow-le-max* **by** *simp*
ultimately have $PoW\ t\ n' \leq length\ (bc\ (\sigma_{tid} t\ (Suc\ n')))$ **by** *simp*
moreover from ‹*trusted tid*› $\langle \wr tid \wr_{t\ (Suc\ n')} \rangle$
have $length\ (bc\ (\sigma_{tid} t\ (Suc\ n'))) \leq PoW\ t\ (Suc\ n')$ **using** *pow-ge-lgth* **by** *simp*
ultimately show *?thesis* **by** *simp*
next
assume *asmp*: $\neg(\exists\, b{\in}pin\ (\sigma_{tid} t\ n').\ length\ b > length\ (bc\ (\sigma_{tid} t\ n')))$
moreover from $\langle \wr tid \wr_{t\ (Suc\ n')} \rangle$ **have** $\langle tid \rightarrow t \rangle_{Suc\ n'} = Suc\ n'$

using *nxtAct-active* **by** *simp*
moreover from ‹$\{\!\!\{tid\}\!\!\}_{t\ n'}$› **have** $\langle tid \leftarrow t\rangle_{Suc\ n'} = n'$
using *latestAct-prop(2) latestActless le-less-Suc-eq* **by** *blast*
moreover from ‹$\{\!\!\{tid\}\!\!\}_{t\ n'}$› **have** $\exists\, n''{<}Suc\ n'.\ \{\!\!\{tid\}\!\!\}_{t\ n''}$ **by** *blast*
moreover from ‹$\{\!\!\{tid\}\!\!\}_{t\ (Suc\ n')}$› **have** $\exists\, n''{\geq}Suc\ n'.\ \{\!\!\{tid\}\!\!\}_{t\ n''}$ **by** *auto*
ultimately have $bc\ (\sigma_{tid} t\ (Suc\ n')) = bc\ (\sigma_{tid} t\ n') \vee$
$(\exists\, b.\ bc\ (\sigma_{tid} t\ (Suc\ n')) = bc\ (\sigma_{tid} t\ n')\ @\ b)$
using ‹*trusted tid*› *bhv-tr-in*[*of tid Suc n′ t*] **by** *auto*
hence $length\ (bc\ (\sigma_{tid} t\ (Suc\ n'))) \geq length\ (bc\ (\sigma_{tid} t\ n'))$ **by** *auto*
moreover from ‹*trusted tid*› ‹$\{\!\!\{tid\}\!\!\}_{t\ n'}$› *asmp* **have** $length\ (bc\ (\sigma_{tid} t\ n')) \geq PoW\ t\ n'$
using *pow-le-lgth* **by** *simp*
moreover from ‹*trusted tid*› ‹$\{\!\!\{tid\}\!\!\}_{t\ (Suc\ n')}$›
have $length\ (bc\ (\sigma_{tid} t\ (Suc\ n'))) \leq PoW\ t\ (Suc\ n')$ **using** *pow-ge-lgth* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
qed
ultimately show *?case* **by** *auto*
qed

lemma *pow-equals*:
assumes $PoW\ t\ n = PoW\ t\ n'$
and $n'{\geq}n$
and $n''{\geq}n$
and $n''{\leq}n'$
shows $PoW\ t\ n = PoW\ t\ n''$ **by** (*metis pow-mono assms*(*1*) *assms*(*3*) *assms*(*4*) *eq-iff*)

lemma *pow-mining-suc*:
assumes *tmining t* (*Suc n*)
shows $PoW\ t\ n < PoW\ t\ (Suc\ n)$
proof −
from *assms* **obtain** *nid* **where** $nid{\in}actTr\ (t\ (Suc\ n))$ **and** $mining\ (\sigma_{nid}(t\ (Suc\ n)))$
using *tmining-def* **by** *auto*
show *?thesis*
proof *cases*
assume *asmp*: $(\exists\, b{\in}pin\ (\sigma_{nid} t\ \langle nid \leftarrow t\rangle_{Suc\ n}).$
$length\ b > length\ (bc\ (\sigma_{nid} t\ \langle nid \leftarrow t\rangle_{Suc\ n})))$
moreover from ‹$nid{\in}actTr\ (t\ (Suc\ n))$› **have** *trusted nid* **and** $\{\!\!\{nid\}\!\!\}_{t\ (Suc\ n)}$
using *actTr-def* **by** *auto*
moreover from ‹*trusted nid*› ‹$mining\ (\sigma_{nid}(t\ (Suc\ n)))$› ‹$\{\!\!\{nid\}\!\!\}_{t\ (Suc\ n)}$› **have** $\{\!\!\{nid\}\!\!\}_{t\ n}$
using *mine* **by** *simp*
hence $\exists\, n'.\ latestAct\text{-}cond\ nid\ t\ (Suc\ n)\ n'$ **by** *auto*
ultimately have $\neg\ mining\ (\sigma_{nid} t\ \langle nid \rightarrow t\rangle_{Suc\ n}) \wedge$
$bc\ (\sigma_{nid} t\ \langle nid \rightarrow t\rangle_{Suc\ n}) = MAX\ (pin\ (\sigma_{nid} t\ \langle nid \leftarrow t\rangle_{Suc\ n})) \vee$
$mining\ (\sigma_{nid} t\ \langle nid \rightarrow t\rangle_{Suc\ n}) \wedge$
$(\exists\, b.\ bc\ (\sigma_{nid} t\ \langle nid \rightarrow t\rangle_{Suc\ n}) = MAX\ (pin\ (\sigma_{nid} t\ \langle nid \leftarrow t\rangle_{Suc\ n}))\ @\ [b])$
using *bhv-tr-ex*[*of nid Suc n*] **by** *auto*
moreover from ‹$\{\!\!\{nid\}\!\!\}_{t\ (Suc\ n)}$› **have** $\langle nid \rightarrow t\rangle_{Suc\ n} = Suc\ n$ **using** *nxtAct-active* **by** *simp*
moreover have $\langle nid \leftarrow t\rangle_{Suc\ n} = n$

```
    proof (rule latestActEq)
      from ‹⦚nid⦚_(t n)› show ⦚nid⦚_(t n) by simp
      show ¬ (∃ n''>n. n'' < Suc n ∧ ⦚nid⦚_(t n)) by simp
      show n < Suc n by simp
    qed
    hence ⟨nid ← t⟩_(Suc n) = n using latestAct-def by simp
    ultimately have ¬ mining (σ_nid t (Suc n)) ∧ bc (σ_nid t (Suc n)) = MAX (pin (σ_nid t n)) ∨
    mining (σ_nid t (Suc n)) ∧ (∃ b. bc (σ_nid t (Suc n)) = MAX (pin (σ_nid t n)) @ [b]) by simp
    with ‹mining (σ_nid(t (Suc n)))›
      have ∃ b. bc (σ_nid t (Suc n)) = MAX (pin (σ_nid t n)) @ [b] by auto
    moreover from ‹trusted nid› ‹⦚nid⦚_(t (Suc n))›
      have length (bc (σ_nid t (Suc n))) ≤ PoW t (Suc n)
      using pow-ge-lgth[of nid t Suc n] by simp
    ultimately have length (MAX (pin (σ_nid t n))) < PoW t (Suc n) by auto
    moreover from ‹trusted nid› ‹⦚nid⦚_(t n)› have length (MAX (pin (σ_nid t n))) ≥ PoW t n
      using pow-le-max by simp
    ultimately show ?thesis by simp
  next
    assume asmp:
      ¬ (∃ b∈pin (σ_nid t ⟨nid ← t⟩_(Suc n)). length b > length (bc (σ_nid t ⟨nid ← t⟩_(Suc n))))
    moreover from ‹nid∈actTr (t (Suc n))› have trusted nid and ⦚nid⦚_(t (Suc n))
      using actTr-def by auto
    moreover from ‹trusted nid› ‹mining (σ_nid(t (Suc n)))› ‹⦚nid⦚_(t (Suc n))› have ⦚nid⦚_(t n)
      using mine by simp
    hence ∃ n'. latestAct-cond nid t (Suc n) n' by auto
    ultimately have ¬ mining (σ_nid t ⟨nid → t⟩_(Suc n)) ∧
      bc (σ_nid t ⟨nid → t⟩_(Suc n)) = bc (σ_nid t ⟨nid ← t⟩_(Suc n)) ∨
      mining (σ_nid t ⟨nid → t⟩_(Suc n)) ∧
      (∃ b. bc (σ_nid t ⟨nid → t⟩_(Suc n)) = bc (σ_nid t ⟨nid ← t⟩_(Suc n)) @ [b])
      using bhv-tr-in[of nid Suc n] by auto
    moreover from ‹⦚nid⦚_(t (Suc n))› have ⟨nid → t⟩_(Suc n) = Suc n using nxtAct-active by simp
    moreover have ⟨nid ← t⟩_(Suc n) = n
    proof (rule latestActEq)
      from ‹⦚nid⦚_(t n)› show ⦚nid⦚_(t n) by simp
      show ¬ (∃ n''>n. n'' < Suc n ∧ ⦚nid⦚_(t n)) by simp
      show n < Suc n by simp
    qed
    hence ⟨nid ← t⟩_(Suc n) = n using latestAct-def by simp
    ultimately have ¬ mining (σ_nid t (Suc n)) ∧ bc (σ_nid t (Suc n)) = bc (σ_nid t n) ∨
    mining (σ_nid t (Suc n)) ∧ (∃ b. bc (σ_nid t (Suc n)) = bc (σ_nid t n) @ [b]) by simp
    with ‹mining (σ_nid(t (Suc n)))› have ∃ b. bc (σ_nid t (Suc n)) = bc (σ_nid t n) @ [b] by simp
    moreover from ‹⟨nid ← t⟩_(Suc n) = n›
      have ¬ (∃ b∈pin (σ_nid t n). length (bc (σ_nid t n)) < length b)
      using asmp by simp
    with ‹trusted nid› ‹⦚nid⦚_(t n)› have length (bc (σ_nid t n)) ≥ PoW t n
      using pow-le-lgth[of nid t n] by simp
    moreover from ‹trusted nid› ‹⦚nid⦚_(t (Suc n))› have
      length (bc (σ_nid t (Suc n))) ≤ PoW t (Suc n)
```

using *pow-ge-lgth*[*of nid t Suc n*] **by** *simp*
ultimately show *?thesis* **by** *auto*
qed
qed

G.3.2.4 History

In the following we introduce an operator which extracts the development of a blockchain up to a time point n.

abbreviation $\mathit{his\text{-}prop}\ t\ n\ nid\ n'\ nid'\ x \equiv$
$(\exists\, n.\ \mathit{latestAct\text{-}cond}\ nid'\ t\ n'\ n) \land \wr \mathit{snd}\ x \wr_{t\ (\mathit{fst}\ x)} \land \mathit{fst}\ x = \langle nid' \leftarrow t\rangle_{n'} \land$
$(\mathit{prefix}\ (bc\ (\sigma_{nid'}(t\ n')))\ (bc\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x)))) \lor$
$(\exists\, b.\ bc\ (\sigma_{nid'}(t\ n')) = (bc\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x)))) @ [b] \land \mathit{mining}\ (\sigma_{nid'}(t\ n'))))$

inductive-set
$\mathit{his}{::}\ \mathit{trace} \Rightarrow \mathit{nat} \Rightarrow {'}\mathit{nid} \Rightarrow (\mathit{nat} \times {'}\mathit{nid})\ \mathit{set}$
for $t{::}\mathit{trace}$ **and** $n{::}\mathit{nat}$ **and** $\mathit{nid}{::}{'}\mathit{nid}$
where $[\![\wr \mathit{nid} \wr_{t\ n}]\!] \Longrightarrow (n,\mathit{nid}) \in \mathit{his}\ t\ n\ \mathit{nid}$
$\mid [\![(n',\mathit{nid}') \in \mathit{his}\ t\ n\ \mathit{nid};\ \exists\, x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n'\ \mathit{nid}'\ x]\!] \Longrightarrow$
$(\mathit{SOME}\ x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n'\ \mathit{nid}'\ x) \in \mathit{his}\ t\ n\ \mathit{nid}$

lemma *his-act*:
assumes $(n',\mathit{nid}') \in \mathit{his}\ t\ n\ \mathit{nid}$
shows $\wr \mathit{nid}' \wr_{t\ n'}$
using *assms*
proof (*rule his.cases*)
assume $(n',\ \mathit{nid}') = (n,\ \mathit{nid})$ **and** $\wr \mathit{nid} \wr_{t\ n}$
thus $\wr \mathit{nid}' \wr_{t\ n'}$ **by** *simp*
next
fix $n''\ \mathit{nid}''$ **assume** asmp: $(n',\ \mathit{nid}') = (\mathit{SOME}\ x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n''\ \mathit{nid}''\ x)$
and $(n'',\ \mathit{nid}'') \in \mathit{his}\ t\ n\ \mathit{nid}$ **and** $\exists\, x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n''\ \mathit{nid}''\ x$
hence $\mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n''\ \mathit{nid}''\ (\mathit{SOME}\ x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n''\ \mathit{nid}''\ x)$
using $\mathit{someI\text{-}ex}[\mathit{of}\ \lambda x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n''\ \mathit{nid}''\ x]$ **by** *auto*
hence $\wr \mathit{snd}\ (\mathit{SOME}\ x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n''\ \mathit{nid}''\ x) \wr_{t\ (\mathit{fst}\ (\mathit{SOME}\ x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n''\ \mathit{nid}''\ x))}$
by *blast*
moreover from *asmp* **have** $\mathit{fst}\ (\mathit{SOME}\ x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n''\ \mathit{nid}''\ x) = \mathit{fst}\ (n',\ \mathit{nid}')$
by *simp*
moreover from *asmp* **have** $\mathit{snd}\ (\mathit{SOME}\ x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n''\ \mathit{nid}''\ x) = \mathit{snd}\ (n',\ \mathit{nid}')$
by *simp*
ultimately show *?thesis* **by** *simp*
qed

In addition we also introduce an operator to obtain the predecessor of a blockchains development.

definition *hisPred*
where $\mathit{hisPred}\ t\ n\ \mathit{nid}\ n' \equiv (\mathit{GREATEST}\ n''.\ \exists\, \mathit{nid}'.\ (n'',\mathit{nid}') \in \mathit{his}\ t\ n\ \mathit{nid} \land n'' < n')$

lemma *hisPrev-prop*:

```
  assumes ∃ n''<n'. ∃ nid'. (n'',nid')∈ his t n nid
  shows hisPred t n nid n' < n' and ∃ nid'. (hisPred t n nid n',nid')∈ his t n nid
proof −
  from assms obtain n'' where ∃ nid'. (n'',nid')∈ his t n nid ∧ n''<n' by auto
  moreover from ‹∃ nid'. (n'',nid')∈ his t n nid ∧ n''<n'›
    have ∃ i'≤n'. (∃ nid'. (i', nid') ∈ his t n nid ∧ i' < n') ∧
    (∀ n'a. (∃ nid'. (n'a, nid') ∈ his t n nid ∧ n'a < n') ⟶ n'a ≤ i')
    using boundedGreatest[of λn''. ∃ nid'. (n'',nid')∈ his t n nid ∧ n'' < n' n'' n'] by simp
  then obtain i' where ∀ n'a. (∃ nid'. (n'a, nid') ∈ his t n nid ∧ n'a < n') ⟶ n'a ≤ i'
    by auto
  ultimately show hisPred t n nid n' < n' and ∃ nid'. (hisPred t n nid n',nid')∈ his t n nid
    using GreatestI-nat[of λn''. ∃ nid'. (n'',nid')∈ his t n nid ∧ n'' < n' n'' i'] hisPred-def
    by auto
qed

lemma hisPrev-nex-less:
  assumes ∃ n''<n'. ∃ nid'. (n'',nid')∈ his t n nid
  shows ¬(∃ x∈his t n nid. fst x < n' ∧ fst x>hisPred t n nid n')
proof (rule ccontr)
  assume ¬¬(∃ x∈his t n nid. fst x < n' ∧ fst x>hisPred t n nid n')
  then obtain n'' nid'' where (n'',nid'')∈his t n nid and n''< n'
    and n''>hisPred t n nid n' by auto
  moreover have n''≤hisPred t n nid n'
  proof −
    from ‹(n'',nid'')∈his t n nid› ‹n''< n'› have ∃ nid'. (n'',nid')∈ his t n nid ∧ n''<n' by auto
    moreover from ‹∃ nid'. (n'',nid')∈ his t n nid ∧ n''<n'› have
      ∃ i'≤n'. (∃ nid'. (i', nid') ∈ his t n nid ∧ i' < n') ∧
      (∀ n'a. (∃ nid'. (n'a, nid') ∈ his t n nid ∧ n'a < n') ⟶ n'a ≤ i')
      using boundedGreatest[of λn''. ∃ nid'. (n'',nid')∈ his t n nid ∧ n'' < n' n'' n'] by simp
    then obtain i' where ∀ n'a. (∃ nid'. (n'a, nid') ∈ his t n nid ∧ n'a < n') ⟶ n'a ≤ i'
      by auto
    ultimately show ?thesis using
      Greatest-le-nat[of λn''. ∃ nid'. (n'',nid')∈ his t n nid ∧ n'' < n' n'' i'] hisPred-def by simp
  qed
  ultimately show False by simp
qed

lemma his-le:
  assumes x ∈ his t n nid
  shows fst x≤n
using assms
proof (induction rule: his.induct)
  case 1
  then show ?case by simp
next
  case (2 n' nid')
  moreover have fst (SOME x. his-prop t n nid n' nid' x) ≤ n'
  proof −
    from 2.hyps have ∃ x. his-prop t n nid n' nid' x by simp
```

```
    hence his-prop t n nid n' nid' (SOME x. his-prop t n nid n' nid' x)
      using someI-ex[of λx. his-prop t n nid n' nid' x] by auto
    hence fst (SOME x. his-prop t n nid n' nid' x) = ⟨nid' ← t⟩_n' by force
    moreover from ‹his-prop t n nid n' nid' (SOME x. his-prop t n nid n' nid' x)›
    have ∃ n. latestAct-cond nid' t n' n by simp
    ultimately show ?thesis using latestAct-prop(2)[of n' nid' t] by simp
  qed
  ultimately show ?case by simp
qed

lemma his-determ-base:
  shows (n, nid') ∈ his t n nid ⟹ nid'=nid
proof (rule his.cases)
  assume (n, nid') = (n, nid)
  thus ?thesis by simp
next
  fix n' nid'a
  assume (n, nid') ∈ his t n nid and (n, nid') = (SOME x. his-prop t n nid n' nid'a x)
    and (n', nid'a) ∈ his t n nid and ∃ x. his-prop t n nid n' nid'a x
  hence his-prop t n nid n' nid'a (SOME x. his-prop t n nid n' nid'a x)
    using someI-ex[of λx. his-prop t n nid n' nid'a x] by auto
  hence fst (SOME x. his-prop t n nid n' nid'a x) = ⟨nid'a ← t⟩_n' by force
  moreover from ‹his-prop t n nid n' nid'a (SOME x. his-prop t n nid n' nid'a x)›
    have ∃ n. latestAct-cond nid'a t n' n by simp
  ultimately have fst (SOME x. his-prop t n nid n' nid'a x) < n'
    using latestAct-prop(2)[of n' nid'a t] by simp
  with ‹(n, nid') = (SOME x. his-prop t n nid n' nid'a x)› have fst (n, nid')<n' by simp
  hence n<n' by simp
  moreover from ‹(n', nid'a) ∈ his t n nid› have n'≤n using his-le by auto
  ultimately show nid' = nid by simp
qed

lemma hisPrev-same:
  assumes ∃ n'<n''. ∃ nid'. (n',nid')∈ his t n nid
  and ∃ n''<n'. ∃ nid'. (n'',nid')∈ his t n nid
  and (n',nid')∈ his t n nid
  and (n'',nid'')∈ his t n nid
  and hisPred t n nid n'=hisPred t n nid n''
  shows n'=n''
proof (rule ccontr)
  assume ¬ n'=n''
  hence n'>n'' ∨ n'<n'' by auto
  thus False
  proof
    assume n'<n''
    hence fst (n',nid')<n'' by simp
    moreover from assms(2) have hisPred t n nid n'<n' using hisPrev-prop(1) by simp
    with assms have hisPred t n nid n''<n' by simp
    hence hisPred t n nid n''<fst (n',nid') by simp
```

```
    ultimately show False using hisPrev-nex-less[of n'' t n nid] assms by auto
  next
    assume n'>n''
    hence fst (n'',nid')<n' by simp
    moreover from assms(1) have hisPred t n nid n''<n'' using hisPrev-prop(1) by simp
    with assms have hisPred t n nid n'<n'' by simp
    hence hisPred t n nid n'<fst (n'',nid') by simp
    ultimately show False using hisPrev-nex-less[of n' t n nid] assms by auto
  qed
qed

lemma his-determ-ext:
  shows n'≤n ⟹ (∃nid'. (n',nid')∈his t n nid) ⟹ (∃!nid'. (n',nid')∈his t n nid) ∧
    ((∃n''<n'. ∃nid'. (n'',nid')∈ his t n nid) ⟶
      (∃x. his-prop t n nid n' (THE nid'. (n',nid')∈his t n nid) x) ∧
    (hisPred t n nid n', (SOME nid'. (hisPred t n nid n', nid') ∈ his t n nid)) =
      (SOME x. his-prop t n nid n' (THE nid'. (n',nid')∈his t n nid) x))
proof (induction n' rule: my-induct)
  case base
  then obtain nid' where (n, nid') ∈ his t n nid by auto
  hence ∃!nid'. (n, nid') ∈ his t n nid
  proof
    fix nid'' assume (n, nid'') ∈ his t n nid
    with his-determ-base have nid''=nid by simp
    moreover from ‹(n, nid') ∈ his t n nid› have nid'=nid using his-determ-base by simp
    ultimately show nid'' = nid' by simp
  qed
  moreover have (∃n''<n. ∃nid'. (n'',nid')∈ his t n nid) ⟶
    (∃x. his-prop t n nid n (THE nid'. (n,nid')∈his t n nid) x) ∧
    (hisPred t n nid n, (SOME nid'. (hisPred t n nid n, nid') ∈ his t n nid)) =
      (SOME x. his-prop t n nid n (THE nid'. (n,nid')∈his t n nid) x)
  proof
    assume ∃n''<n. ∃nid'. (n'',nid')∈ his t n nid
    hence ∃nid'. (hisPred t n nid n, nid')∈ his t n nid using hisPrev-prop(2) by simp
    hence (hisPred t n nid n, (SOME nid'. (hisPred t n nid n, nid') ∈ his t n nid)) ∈ his t n nid
      using someI-ex[of λnid'. (hisPred t n nid n, nid') ∈ his t n nid] by simp
    thus (∃x. his-prop t n nid n (THE nid'. (n,nid')∈his t n nid) x) ∧
      (hisPred t n nid n, (SOME nid'. (hisPred t n nid n, nid') ∈ his t n nid)) =
        (SOME x. his-prop t n nid n (THE nid'. (n,nid')∈his t n nid) x)
    proof (rule his.cases)
      assume (hisPred t n nid n, SOME nid'. (hisPred t n nid n, nid') ∈ his t n nid) = (n, nid)
      hence hisPred t n nid n=n by simp
      with ‹∃n''<n. ∃nid'. (n'',nid')∈ his t n nid› show ?thesis
        using hisPrev-prop(1)[of n t n nid] by force
    next
      fix n'' nid'' assume asmp:
        (hisPred t n nid n, SOME nid'. (hisPred t n nid n, nid') ∈ his t n nid) =
        (SOME x. his-prop t n nid n'' nid'' x)
      and (n'', nid'') ∈ his t n nid and ∃x. his-prop t n nid n'' nid'' x
```

```
      moreover have n″=n
      proof (rule antisym)
        show n″≥n
        proof (rule ccontr)
          assume (¬n″≥n)
          hence n″<n by simp
          moreover have n″>hisPred t n nid n
          proof –
            let ?x=λx. his-prop t n nid n″ nid″ x
            from ‹∃x. his-prop t n nid n″ nid″ x› have his-prop t n nid n″ nid″ (SOME x. ?x x)
              using someI-ex[of ?x] by auto
            hence n″>fst (SOME x. ?x x) using latestAct-prop(2)[of n″ nid″ t] by force
            moreover from asmp have
              fst (hisPred t n nid n, SOME nid′. (hisPred t n nid n, nid′) ∈ his t n nid) =
              fst (SOME x. ?x x) by simp
            ultimately show ?thesis by simp
          qed
          moreover from ‹∃n″<n. ∃nid′. (n″,nid′)∈ his t n nid›
            have ¬(∃x∈his t n nid. fst x < n ∧ fst x > hisPred t n nid n)
            using hisPrev-nex-less by simp
          ultimately show False using ‹(n″, nid″) ∈ his t n nid› by auto
        qed
      next
        from ‹(n″, nid″) ∈ his t n nid› show n″ ≤ n using his-le by auto
      qed
      ultimately have (hisPred t n nid n, SOME nid′. (hisPred t n nid n, nid′) ∈ his t n nid) =
        (SOME x. his-prop t n nid n nid″ x) by simp
      moreover from ‹n″=n› ‹(n″, nid″) ∈ his t n nid› have (n, nid″) ∈ his t n nid by simp
      with ‹∃!nid′. (n,nid′) ∈ his t n nid› have nid″=(THE nid′. (n,nid′)∈his t n nid)
        using the1-equality[of λnid′. (n, nid′) ∈ his t n nid] by simp
      moreover from ‹∃x. his-prop t n nid n″ nid″ x› ‹n″=n›
       ‹nid″=(THE nid′. (n,nid′)∈his t n nid)›
        have ∃x. his-prop t n nid n (THE nid′. (n,nid′)∈his t n nid) x by simp
      ultimately show ?thesis by simp
    qed
  qed
  ultimately show ?case by simp
next
  case (step n′)
  then obtain nid′ where (n′, nid′) ∈ his t n nid by auto
  hence ∃!nid′. (n′, nid′) ∈ his t n nid
  proof (rule his.cases)
    assume (n′, nid′) = (n, nid)
    hence n′=n by simp
    with step.hyps show ?thesis by simp
  next
    fix n′′′′ nid′′′′
    assume (n′′′′, nid′′′′) ∈ his t n nid
      and n′nid′: (n′, nid′) = (SOME x. his-prop t n nid n′′′′ nid′′′′ x)
```

```
  and (n'''', nid'''') ∈ his t n nid and ∃x. his-prop t n nid n'''' nid'''' x
from ‹(n', nid') ∈ his t n nid› show ?thesis
proof
  fix nid'' assume (n', nid'') ∈ his t n nid
  thus nid'' = nid'
  proof (rule his.cases)
    assume (n', nid'') = (n, nid)
    hence n'=n by simp
    with step.hyps show ?thesis by simp
  next
    fix n''' nid'''
    assume (n''', nid''') ∈ his t n nid
      and n'nid'': (n', nid'') = (SOME x. his-prop t n nid n''' nid''' x)
      and (n''', nid''') ∈ his t n nid and ∃x. his-prop t n nid n''' nid''' x
    moreover have n'''=n''''
    proof –
      have hisPred t n nid n''' = n'
      proof –
        from n'nid'' ‹∃x. his-prop t n nid n''' nid''' x›
          have his-prop t n nid n''' nid''' (n',nid'')
          using someI-ex[of λx. his-prop t n nid n''' nid''' x] by auto
        hence n'''>n' using latestAct-prop(2) by simp
        moreover from ‹(n''', nid''') ∈ his t n nid› have n'''≤ n using his-le by auto
        moreover from ‹(n''', nid''') ∈ his t n nid›
          have ∃nid'. (n''', nid') ∈ his t n nid by auto
        ultimately have (∃n'<n'''. ∃nid'. (n',nid')∈ his t n nid) ⟶
          (∃!nid'. (n''',nid') ∈ his t n nid) ∧
          (hisPred t n nid n''', (SOME nid'. (hisPred t n nid n''', nid') ∈ his t n nid)) =
            (SOME x. his-prop t n nid n''' (THE nid'. (n''',nid')∈his t n nid) x)
          using step.IH by auto
        with ‹n'''>n'› ‹(n', nid') ∈ his t n nid› have ∃!nid'. (n''',nid') ∈ his t n nid and
          (hisPred t n nid n''', (SOME nid'. (hisPred t n nid n''', nid') ∈ his t n nid)) =
            (SOME x. his-prop t n nid n''' (THE nid'. (n''',nid')∈his t n nid) x) by auto
        moreover from ‹∃!nid'. (n''',nid') ∈ his t n nid› ‹(n''', nid''') ∈ his t n nid› have
          nid'''=(THE nid'. (n''',nid')∈his t n nid)
          using the1-equality[of λnid'. (n''', nid') ∈ his t n nid] by simp
        ultimately have
          (hisPred t n nid n''', (SOME nid'. (hisPred t n nid n''', nid') ∈ his t n nid)) =
            (SOME x. his-prop t n nid n''' nid''' x) by simp
        with n'nid'' have (n', nid'') =
         (hisPred t n nid n''', (SOME nid'. (hisPred t n nid n''', nid') ∈ his t n nid)) by simp
        thus ?thesis by simp
      qed
      moreover have hisPred t n nid n'''' = n'
      proof –
        from n'nid' ‹∃x. his-prop t n nid n'''' nid'''' x›
          have his-prop t n nid n'''' nid'''' (n',nid')
          using someI-ex[of λx. his-prop t n nid n'''' nid'''' x] by auto
        hence n''''>n' using latestAct-prop(2) by simp
```

```
    moreover from ‹(n'''', nid'''') ∈ his t n nid› have n''''≤ n using his-le by auto
    moreover from ‹(n'''', nid'''') ∈ his t n nid›
      have ∃ nid'. (n'''', nid') ∈ his t n nid by auto
      ultimately have (∃ n'<n''''. ∃ nid'. (n',nid')∈ his t n nid) ⟶
      (∃!nid'. (n'''',nid') ∈ his t n nid) ∧
      (hisPred t n nid n'''', (SOME nid'. (hisPred t n nid n'''', nid') ∈ his t n nid)) =
      (SOME x. his-prop t n nid n'''' (THE nid'. (n'''',nid')∈his t n nid) x)
        using step.IH by auto
      with ‹n''''>n'› ‹(n', nid') ∈ his t n nid› have ∃!nid'. (n'''',nid') ∈ his t n nid and
        (hisPred t n nid n'''', (SOME nid'. (hisPred t n nid n'''', nid') ∈ his t n nid)) =
        (SOME x. his-prop t n nid n'''' (THE nid'. (n'''',nid')∈his t n nid) x) by auto
    moreover from ‹∃!nid'. (n'''',nid') ∈ his t n nid› ‹(n'''', nid'''') ∈ his t n nid›
      have nid''''=(THE nid'. (n'''',nid')∈his t n nid)
      using the1-equality[of λnid'. (n'''', nid') ∈ his t n nid] by simp
    ultimately have
      (hisPred t n nid n'''', (SOME nid'. (hisPred t n nid n'''', nid') ∈ his t n nid))
      = (SOME x. his-prop t n nid n'''' nid'''' x) by simp
    with n'nid' have (n', nid') =
      (hisPred t n nid n'''', (SOME nid'. (hisPred t n nid n'''', nid') ∈ his t n nid))
      by simp
    thus ?thesis by simp
  qed
  ultimately have hisPred t n nid n'''=hisPred t n nid n'''' ..
  moreover have ∃ n'<n'''. ∃ nid'. (n',nid')∈ his t n nid
  proof –
    from n'nid'' ‹∃ x. his-prop t n nid n''' nid''' x›
      have his-prop t n nid n''' nid''' (n',nid'')
      using someI-ex[of λx. his-prop t n nid n''' nid''' x] by auto
    hence n'''>n' using latestAct-prop(2) by simp
    with ‹(n', nid') ∈ his t n nid› show ?thesis by auto
  qed
  moreover have ∃ n'<n''''. ∃ nid'. (n',nid')∈ his t n nid
  proof –
    from n'nid' ‹∃ x. his-prop t n nid n'''' nid'''' x›
      have his-prop t n nid n'''' nid'''' (n',nid')
      using someI-ex[of λx. his-prop t n nid n'''' nid'''' x] by auto
    hence n''''>n' using latestAct-prop(2) by simp
    with ‹(n', nid') ∈ his t n nid› show ?thesis by auto
  qed
  ultimately show ?thesis
    using hisPrev-same ‹(n''', nid''') ∈ his t n nid› ‹(n'''', nid'''') ∈ his t n nid›
    by blast
qed
moreover have nid'''=nid''''
proof –
  from n'nid'' ‹∃ x. his-prop t n nid n''' nid''' x›
    have his-prop t n nid n''' nid''' (n',nid'')
    using someI-ex[of λx. his-prop t n nid n''' nid''' x] by auto
  hence n'''>n' using latestAct-prop(2) by simp
```

```
        moreover from ‹(n''', nid''') ∈ his t n nid› have n'''≤ n using his-le by auto
        moreover from ‹(n''', nid''') ∈ his t n nid›
          have ∃nid'. (n''', nid') ∈ his t n nid by auto
        ultimately have ∃!nid'. (n''', nid') ∈ his t n nid using step.IH by auto
        with ‹(n''', nid''') ∈ his t n nid› ‹(n'''', nid'''') ∈ his t n nid› ‹n'''=n''''›
          show ?thesis by auto
      qed
      ultimately have (n', nid') = (n', nid'') using n'nid' by simp
      thus nid'' = nid' by simp
    qed
  qed
qed
moreover have (∃n''<n'. ∃nid'. (n'',nid')∈ his t n nid) ⟶
  (∃x. his-prop t n nid n' (THE nid'. (n',nid')∈his t n nid) x) ∧
  (hisPred t n nid n', (SOME nid'. (hisPred t n nid n', nid') ∈ his t n nid)) =
  (SOME x. his-prop t n nid n' (THE nid'. (n',nid')∈his t n nid) x)
proof
  assume ∃n''<n'. ∃nid'. (n'',nid')∈ his t n nid
  hence ∃nid'. (hisPred t n nid n', nid')∈ his t n nid using hisPrev-prop(2) by simp
  hence (hisPred t n nid n',
    (SOME nid'. (hisPred t n nid n', nid') ∈ his t n nid)) ∈ his t n nid
    using someI-ex[of λnid'. (hisPred t n nid n', nid') ∈ his t n nid] by simp
  thus (∃x. his-prop t n nid n' (THE nid'. (n',nid')∈his t n nid) x) ∧
    (hisPred t n nid n', (SOME nid'. (hisPred t n nid n', nid') ∈ his t n nid)) =
    (SOME x. his-prop t n nid n' (THE nid'. (n',nid')∈his t n nid) x)
  proof (rule his.cases)
    assume (hisPred t n nid n',
    SOME nid'. (hisPred t n nid n', nid') ∈ his t n nid) = (n, nid)
    hence hisPred t n nid n'=n by simp
    moreover from ‹∃n''<n'. ∃nid'. (n'',nid')∈ his t n nid› have hisPred t n nid n'<n'
      using hisPrev-prop(1)[of n'] by force
    ultimately show ?thesis using step.hyps by simp
  next
    fix n'' nid'' assume asmp:
      (hisPred t n nid n', SOME nid'. (hisPred t n nid n', nid') ∈ his t n nid) =
      (SOME x. his-prop t n nid n'' nid'' x)
    and (n'', nid'') ∈ his t n nid and ∃x. his-prop t n nid n'' nid'' x
    moreover have n''=n'
    proof (rule antisym)
      show n''≥n'
      proof (rule ccontr)
        assume (¬n''≥n')
        hence n''<n' by simp
        moreover have n''>hisPred t n nid n'
        proof -
          let ?x=λx. his-prop t n nid n'' nid'' x
          from ‹∃x. his-prop t n nid n'' nid'' x› have his-prop t n nid n'' nid'' (SOME x. ?x x)
            using someI-ex[of ?x] by auto
          hence n''>fst (SOME x. ?x x) using latestAct-prop(2)[of n'' nid'' t] by force
```

```
      moreover from asmp have
        fst (hisPred t n nid n', SOME nid'. (hisPred t n nid n', nid') ∈ his t n nid)
         = fst (SOME x. ?x x) by simp
      ultimately show ?thesis by simp
    qed
    moreover from ‹∃n''<n'. ∃nid'. (n'',nid')∈ his t n nid›
      have ¬(∃x∈his t n nid. fst x < n' ∧ fst x > hisPred t n nid n')
      using hisPrev-nex-less by simp
    ultimately show False using ‹(n'', nid'') ∈ his t n nid› by auto
  qed
next
  show n'≥n''
  proof (rule ccontr)
    assume (¬n'≥n'')
    hence n'<n'' by simp
    moreover from ‹(n'', nid'') ∈ his t n nid› have n''≤ n using his-le by auto
    moreover from ‹(n'', nid'') ∈ his t n nid› have ∃nid'. (n'', nid') ∈ his t n nid
      by auto
    ultimately have (∃n'<n''. ∃nid'. (n',nid')∈ his t n nid)
      ⟶ (∃!nid'. (n'',nid') ∈ his t n nid) ∧
        (hisPred t n nid n'', (SOME nid'. (hisPred t n nid n'', nid') ∈ his t n nid)) =
        (SOME x. his-prop t n nid n'' (THE nid'. (n'',nid')∈his t n nid) x)
      using step.IH by auto
    with ‹n'<n''› ‹(n', nid') ∈ his t n nid› have ∃!nid'. (n'',nid') ∈ his t n nid and
      (hisPred t n nid n'', (SOME nid'. (hisPred t n nid n'', nid') ∈ his t n nid)) =
      (SOME x. his-prop t n nid n'' (THE nid'. (n'',nid')∈his t n nid) x) by auto
    moreover from ‹∃!nid'. (n'',nid') ∈ his t n nid› ‹(n'', nid'') ∈ his t n nid›
      have nid'' = (THE nid'. (n'',nid')∈his t n nid)
      using the1-equality[of λnid'. (n'', nid') ∈ his t n nid] by simp
    ultimately have (hisPred t n nid n'',
      (SOME nid'. (hisPred t n nid n'', nid') ∈ his t n nid))
      = (SOME x. his-prop t n nid n'' nid'' x) by simp
    with asmp have (hisPred t n nid n',
      SOME nid'. (hisPred t n nid n', nid') ∈ his t n nid) =
      (hisPred t n nid n'', SOME nid'. (hisPred t n nid n'', nid') ∈ his t n nid) by simp
    hence hisPred t n nid n' = hisPred t n nid n'' by simp
    with ‹∃n''<n'. ∃nid'. (n'', nid') ∈ his t n nid› ‹n'<n''› ‹(n', nid') ∈ his t n nid›
      ‹(n'', nid'') ∈ his t n nid› ‹(n', nid') ∈ his t n nid› have n'=n''
      using hisPrev-same by blast
    with ‹n'<n''› show False by simp
  qed
qed
ultimately have (hisPred t n nid n',
  SOME nid'. (hisPred t n nid n', nid') ∈ his t n nid) =
  (SOME x. his-prop t n nid n' nid'' x) by simp
moreover from ‹(n'', nid'') ∈ his t n nid› ‹n''=n'› have (n', nid'') ∈ his t n nid by simp
with ‹∃!nid'. (n',nid') ∈ his t n nid› have nid''=(THE nid'. (n',nid')∈his t n nid)
  using the1-equality[of λnid'. (n', nid') ∈ his t n nid] by simp
moreover from ‹∃x. his-prop t n nid n'' nid'' x› ‹n''=n'›
```

```
    ‹nid''=(THE nid'. (n',nid')∈his t n nid)›
    have ∃x. his-prop t n nid n' (THE nid'. (n',nid')∈his t n nid) x by simp
   ultimately show ?thesis by simp
  qed
 qed
 ultimately show ?case by simp
qed

corollary his-determ-ex:
 assumes (n',nid')∈his t n nid
 shows ∃!nid'. (n',nid')∈his t n nid
 using assms his-le his-determ-ext[of n' n t nid] by force

corollary his-determ:
 assumes (n',nid')∈his t n nid
  and (n',nid'')∈his t n nid
 shows nid'=nid'' using assms his-le his-determ-ext[of n' n t nid] by force

corollary his-determ-the:
 assumes (n',nid')∈his t n nid
 shows (THE nid'. (n', nid')∈his t n nid) = nid'
 using assms his-determ theI'[of λnid'. (n', nid')∈his t n nid] his-determ-ex by simp
```

G.3.2.5 Blockchain Development

```
definition devBC::trace ⇒ nat ⇒ 'nid ⇒ nat ⇒ 'nid option
 where devBC t n nid n' ≡
  (if (∃nid'. (n', nid')∈ his t n nid) then (Some (THE nid'. (n', nid')∈his t n nid))
  else Option.None)

lemma devBC-some[simp]: assumes ⌇nid⌇_{t n} shows devBC t n nid n = Some nid
proof –
 from assms have (n, nid)∈ his t n nid using his.intros(1) by simp
 hence devBC t n nid n = (Some (THE nid'. (n, nid')∈his t n nid)) using devBC-def by auto
 moreover have (THE nid'. (n, nid')∈his t n nid) = nid
 proof
  from ‹(n, nid)∈ his t n nid› show (n, nid)∈ his t n nid .
 next
  fix nid' assume (n, nid') ∈ his t n nid
  thus nid' = nid using his-determ-base by simp
 qed
 ultimately show ?thesis by simp
qed

lemma devBC-act: assumes ¬ Option.is-none (devBC t n nid n')
 shows ⌇the (devBC t n nid n')⌇_{t n'}
proof –
 from assms have ¬ devBC t n nid n'=Option.None by (metis is-none-simps(1))
 then obtain nid' where (n', nid')∈ his t n nid and
```

```
    devBC t n nid n' = (Some (THE nid'. (n', nid')∈his t n nid))
    using devBC-def[of t n nid] by metis
  hence nid'= (THE nid'. (n', nid')∈his t n nid) using his-determ-the by simp
  with ‹devBC t n nid n' = (Some (THE nid'. (n', nid')∈his t n nid))›
    have the (devBC t n nid n') = nid' by simp
  with ‹(n', nid')∈ his t n nid› show ?thesis using his-act by simp
qed

lemma his-ex:
  assumes ¬Option.is-none (devBC t n nid n')
  shows ∃nid'. (n',nid')∈his t n nid
proof (rule ccontr)
  assume ¬(∃nid'. (n',nid')∈his t n nid)
  with devBC-def have Option.is-none (devBC t n nid n') by simp
  with assms show False by simp
qed

lemma devExt-nopt-leq:
  assumes ¬Option.is-none (devBC t n nid n')
  shows n'≤n
proof -
  from assms have ∃nid'. (n',nid')∈his t n nid using his-ex by simp
  then obtain nid' where (n',nid')∈his t n nid by auto
  with his-le[of (n',nid')] show ?thesis by simp
qed
```

An extended version of the development in which deactivations are filled with the last value.

```
function devExt::trace ⇒ nat ⇒ 'nid ⇒ nat ⇒ nat ⇒ 'nid BC
  where ⟦∃n'<n_s. ¬Option.is-none (devBC t n nid n'); Option.is-none (devBC t n nid n_s)⟧ ⟹
      devExt t n nid n_s 0 = bc
      (σ_the (devBC t n nid (GREATEST n'. n'<n_s ∧ ¬Option.is-none (devBC t n nid n')))
      (t (GREATEST n'. n'<n_s ∧ ¬Option.is-none (devBC t n nid n'))))
  | ⟦¬ (∃n'<n_s. ¬Option.is-none (devBC t n nid n')); Option.is-none (devBC t n nid n_s)⟧ ⟹
    devExt t n nid n_s 0 = []
  | ¬ Option.is-none (devBC t n nid n_s) ⟹
    devExt t n nid n_s 0 = bc (σ_the (devBC t n nid n_s)^(t n_s))
  | ¬ Option.is-none (devBC t n nid (n_s + Suc n')) ⟹
    devExt t n nid n_s (Suc n') = bc (σ_the (devBC t n nid (n_s + Suc n'))^(t (n_s + Suc n')))
  | Option.is-none (devBC t n nid (n_s + Suc n')) ⟹
    devExt t n nid n_s (Suc n') = devExt t n nid n_s n'
proof -
  show ⋀n_s t n nid n_s' ta na nida.
      ∃n'<n_s. ¬ Option.is-none (devBC t n nid n') ⟹
      Option.is-none (devBC t n nid n_s) ⟹
      ∃n'<n_s'. ¬ Option.is-none (devBC ta na nida n') ⟹
      Option.is-none (devBC ta na nida n_s') ⟹
      (t, n, nid, n_s, 0) = (ta, na, nida, n_s', 0) ⟹
      bc (σ_the (devBC t n nid (GREATEST n'. n' < n_s ∧ ¬ Option.is-none (devBC t n nid n')))
```

t (*GREATEST* n'. $n' < n_s \land \neg$ *Option.is-none* (*devBC t n nid* n'))) =
bc ($\sigma_{the\ (devBC\ ta\ na\ nida\ (GREATEST\ n'.\ n' < n_s' \land \neg\ Option.is\text{-}none\ (devBC\ ta\ na\ nida\ n')))}$
ta (*GREATEST* n'. $n' < n_s' \land \neg$ *Option.is-none* (*devBC ta na nida* n'))) **by** *auto*

show $\bigwedge n_s$ *t n nid* n_s' *ta na nida*.
$\exists\, n' < n_s$. ¬ *Option.is-none* (*devBC t n nid* n') ⟹
Option.is-none (*devBC t n nid* n_s) ⟹
¬ ($\exists\, n' < n_s'$. ¬ *Option.is-none* (*devBC ta na nida* n')) ⟹
Option.is-none (*devBC ta na nida* n_s') ⟹
(*t*, *n*, *nid*, n_s, *0*) = (*ta*, *na*, *nida*, n_s', *0*) ⟹
bc ($\sigma_{the\ (devBC\ t\ n\ nid\ (GREATEST\ n'.\ n' < n_s \land \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')))}{}^{t}$
(*GREATEST* n'. $n' < n_s \land \neg$ *Option.is-none* (*devBC t n nid* n'))) = [] **by** *auto*

show $\bigwedge n_s$ *t n nid ta na nida* n_s'.
$\exists\, n' < n_s$. ¬ *Option.is-none* (*devBC t n nid* n') ⟹
Option.is-none (*devBC t n nid* n_s) ⟹
¬ *Option.is-none* (*devBC ta na nida* n_s') ⟹
(*t*, *n*, *nid*, n_s, *0*) = (*ta*, *na*, *nida*, n_s', *0*) ⟹
bc ($\sigma_{the\ (devBC\ t\ n\ nid\ (GREATEST\ n'.\ n' < n_s \land \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')))}{}^{t}$
(*GREATEST* n'. $n' < n_s \land \neg$ *Option.is-none* (*devBC t n nid* n'))) =
bc ($\sigma_{the\ (devBC\ ta\ na\ nida\ n_s')}{}^{ta\ n_s'}$) **by** *auto*

show $\bigwedge n_s$ *t n nid ta na nida* n_s' n'.
$\exists\, n' < n_s$. ¬ *Option.is-none* (*devBC t n nid* n') ⟹
Option.is-none (*devBC t n nid* n_s) ⟹
¬ *Option.is-none* (*devBC ta na nida* (n_s' + *Suc* n')) ⟹
(*t*, *n*, *nid*, n_s, *0*) = (*ta*, *na*, *nida*, n_s', *Suc* n') ⟹
bc ($\sigma_{the\ (devBC\ t\ n\ nid\ (GREATEST\ n'.\ n' < n_s \land \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')))}{}^{t}$
(*GREATEST* n'. $n' < n_s \land \neg$ *Option.is-none* (*devBC t n nid* n'))) =
bc ($\sigma_{the\ (devBC\ ta\ na\ nida\ (n_s' + Suc\ n'))}{}^{ta}$ (n_s' + *Suc* n')) **by** *auto*

show $\bigwedge n_s$ *t n nid ta na nida* n_s' n'.
$\exists\, n' < n_s$. ¬ *Option.is-none* (*devBC t n nid* n') ⟹
Option.is-none (*devBC t n nid* n_s) ⟹
Option.is-none (*devBC ta na nida* (n_s' + *Suc* n')) ⟹
(*t*, *n*, *nid*, n_s, *0*) = (*ta*, *na*, *nida*, n_s', *Suc* n') ⟹
bc ($\sigma_{the\ (devBC\ t\ n\ nid\ (GREATEST\ n'.\ n' < n_s \land \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')))}{}^{t}$
(*GREATEST* n'. $n' < n_s \land \neg$ *Option.is-none* (*devBC t n nid* n'))) =
devExt-sumC (*ta*, *na*, *nida*, n_s', n') **by** *auto*

show$\bigwedge n_s$ *t n nid* n_s' *ta na nida*.
¬ ($\exists\, n' < n_s$. ¬ *Option.is-none* (*devBC t n nid* n')) ⟹
Option.is-none (*devBC t n nid* n_s) ⟹
¬ ($\exists\, n' < n_s'$. ¬ *Option.is-none* (*devBC ta na nida* n')) ⟹
Option.is-none (*devBC ta na nida* n_s') ⟹
(*t*, *n*, *nid*, n_s, *0*) = (*ta*, *na*, *nida*, n_s', *0*) ⟹ [] = [] **by** *auto*

show $\bigwedge n_s$ *t n nid ta na nida* n_s'.
¬ ($\exists\, n' < n_s$. ¬ *Option.is-none* (*devBC t n nid* n')) ⟹
Option.is-none (*devBC t n nid* n_s) ⟹
¬ *Option.is-none* (*devBC ta na nida* n_s') ⟹
(*t*, *n*, *nid*, n_s, *0*) = (*ta*, *na*, *nida*, n_s', *0*) ⟹
[] = *bc* ($\sigma_{the\ (devBC\ ta\ na\ nida\ n_s')}{}^{ta\ n_s'}$) **by** *auto*

show $\bigwedge n_s$ *t n nid ta na nida* n_s' n'.

$\neg\ (\exists\, n'{<}n_s.\ \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')) \Longrightarrow$
$Option.is\text{-}none\ (devBC\ t\ n\ nid\ n_s) \Longrightarrow$
$\neg\ Option.is\text{-}none\ (devBC\ ta\ na\ nida\ (n_s' + Suc\ n')) \Longrightarrow$
$(t,\ n,\ nid,\ n_s,\ 0) = (ta,\ na,\ nida,\ n_s',\ Suc\ n') \Longrightarrow$
$[] = bc\ (\sigma_{the\ (devBC\ ta\ na\ nida\ (n_s' + Suc\ n'))}^{ta}\ (n_s' + Suc\ n'))$ **by** *auto*
show $\bigwedge n_s\ t\ n\ nid\ ta\ na\ nida\ n_s'\ n'.$
$\neg\ (\exists\, n'{<}n_s.\ \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')) \Longrightarrow$
$Option.is\text{-}none\ (devBC\ t\ n\ nid\ n_s) \Longrightarrow$
$Option.is\text{-}none\ (devBC\ ta\ na\ nida\ (n_s' + Suc\ n')) \Longrightarrow$
$(t,\ n,\ nid,\ n_s,\ 0) = (ta,\ na,\ nida,\ n_s',\ Suc\ n') \Longrightarrow$
$[] = devExt\text{-}sumC\ (ta,\ na,\ nida,\ n_s',\ n')$ **by** *auto*
show $\bigwedge t\ n\ nid\ n_s\ ta\ na\ nida\ n_s'.$
$\neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n_s) \Longrightarrow$
$\neg\ Option.is\text{-}none\ (devBC\ ta\ na\ nida\ n_s') \Longrightarrow$
$(t,\ n,\ nid,\ n_s,\ 0) = (ta,\ na,\ nida,\ n_s',\ 0) \Longrightarrow$
$bc\ (\sigma_{the\ (devBC\ t\ n\ nid\ n_s)}^{t}\ n_s) = bc\ (\sigma_{the\ (devBC\ ta\ na\ nida\ n_s')}^{ta}\ n_s')$ **by** *auto*
show $\bigwedge t\ n\ nid\ n_s\ ta\ na\ nida\ n_s'\ n'.$
$\neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n_s) \Longrightarrow$
$\neg\ Option.is\text{-}none\ (devBC\ ta\ na\ nida\ (n_s' + Suc\ n')) \Longrightarrow$
$(t,\ n,\ nid,\ n_s,\ 0) = (ta,\ na,\ nida,\ n_s',\ Suc\ n') \Longrightarrow$
$bc\ (\sigma_{the\ (devBC\ t\ n\ nid\ n_s)}^{t}\ n_s) = bc$
$(\sigma_{the\ (devBC\ ta\ na\ nida\ (n_s' + Suc\ n'))}^{ta}\ (n_s' + Suc\ n'))$ **by** *auto*
show $\bigwedge t\ n\ nid\ n_s\ ta\ na\ nida\ n_s'\ n'.$
$\neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n_s) \Longrightarrow$
$Option.is\text{-}none\ (devBC\ ta\ na\ nida\ (n_s' + Suc\ n')) \Longrightarrow$
$(t,\ n,\ nid,\ n_s,\ 0) = (ta,\ na,\ nida,\ n_s',\ Suc\ n') \Longrightarrow$
$bc\ (\sigma_{the\ (devBC\ t\ n\ nid\ n_s)}^{t}\ n_s) = devExt\text{-}sumC\ (ta,\ na,\ nida,\ n_s',\ n')$ **by** *auto*
show $\bigwedge t\ n\ nid\ n_s\ n'\ ta\ na\ nida\ n_s'\ n'a.$
$\neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ (n_s + Suc\ n')) \Longrightarrow$
$\neg\ Option.is\text{-}none\ (devBC\ ta\ na\ nida\ (n_s' + Suc\ n'a)) \Longrightarrow$
$(t,\ n,\ nid,\ n_s,\ Suc\ n') = (ta,\ na,\ nida,\ n_s',\ Suc\ n'a) \Longrightarrow$
$bc\ (\sigma_{the\ (devBC\ t\ n\ nid\ (n_s + Suc\ n'))}^{t}\ (n_s + Suc\ n')) =$
$bc\ (\sigma_{the\ (devBC\ ta\ na\ nida\ (n_s' + Suc\ n'a))}^{ta}\ (n_s' + Suc\ n'a))$ **by** *auto*
show $\bigwedge t\ n\ nid\ n_s\ n'\ ta\ na\ nida\ n_s'\ n'a.$
$\neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ (n_s + Suc\ n')) \Longrightarrow$
$Option.is\text{-}none\ (devBC\ ta\ na\ nida\ (n_s' + Suc\ n'a)) \Longrightarrow$
$(t,\ n,\ nid,\ n_s,\ Suc\ n') = (ta,\ na,\ nida,\ n_s',\ Suc\ n'a) \Longrightarrow$
$bc\ (\sigma_{the\ (devBC\ t\ n\ nid\ (n_s + Suc\ n'))}$
$t\ (n_s + Suc\ n')) = devExt\text{-}sumC\ (ta,\ na,\ nida,\ n_s',\ n'a)$**by** *auto*
show $\bigwedge t\ n\ nid\ n_s\ n'\ ta\ na\ nida\ n_s'\ n'a.$
$Option.is\text{-}none\ (devBC\ t\ n\ nid\ (n_s + Suc\ n')) \Longrightarrow$
$Option.is\text{-}none\ (devBC\ ta\ na\ nida\ (n_s' + Suc\ n'a)) \Longrightarrow$
$(t,\ n,\ nid,\ n_s,\ Suc\ n') = (ta,\ na,\ nida,\ n_s',\ Suc\ n'a) \Longrightarrow$
$devExt\text{-}sumC\ (t,\ n,\ nid,\ n_s,\ n') = devExt\text{-}sumC\ (ta,\ na,\ nida,\ n_s',\ n'a)$ **by** *auto*
show $\bigwedge P\ x.\ (\bigwedge n_s\ t\ n\ nid.\ \exists\, n'{<}n_s.\ \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n') \Longrightarrow$
$Option.is\text{-}none\ (devBC\ t\ n\ nid\ n_s) \Longrightarrow x = (t,\ n,\ nid,\ n_s,\ 0) \Longrightarrow P) \Longrightarrow$
$(\bigwedge n_s\ t\ n\ nid.\ \neg\ (\exists\, n'{<}n_s.\ \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')) \Longrightarrow$
$Option.is\text{-}none\ (devBC\ t\ n\ nid\ n_s) \Longrightarrow x = (t,\ n,\ nid,\ n_s,\ 0) \Longrightarrow P) \Longrightarrow$

```
    (⋀t n nid n_s. ¬ Option.is-none (devBC t n nid n_s) ⟹
     x = (t, n, nid, n_s, 0) ⟹ P) ⟹
    (⋀t n nid n_s n'. ¬ Option.is-none (devBC t n nid (n_s + Suc n')) ⟹
     x = (t, n, nid, n_s, Suc n') ⟹ P) ⟹
    (⋀t n nid n_s n'. Option.is-none (devBC t n nid (n_s + Suc n')) ⟹
     x = (t, n, nid, n_s, Suc n') ⟹ P) ⟹ P
proof -
  fix P::bool and x::trace ×nat×'nid×nat×nat
  assume a1:(⋀n_s t n nid. ∃n'<n_s. ¬ Option.is-none (devBC t n nid n') ⟹
          Option.is-none (devBC t n nid n_s) ⟹ x = (t, n, nid, n_s, 0) ⟹ P) and
        a2:(⋀n_s t n nid. ¬ (∃n'<n_s. ¬ Option.is-none (devBC t n nid n')) ⟹
          Option.is-none (devBC t n nid n_s) ⟹ x = (t, n, nid, n_s, 0) ⟹ P) and
        a3:(⋀t n nid n_s. ¬ Option.is-none (devBC t n nid n_s) ⟹
          x = (t, n, nid, n_s, 0) ⟹ P) and
        a4:(⋀t n nid n_s n'. ¬ Option.is-none (devBC t n nid (n_s + Suc n')) ⟹
          x = (t, n, nid, n_s, Suc n') ⟹ P) and
        a5:(⋀t n nid n_s n'. Option.is-none (devBC t n nid (n_s + Suc n')) ⟹
          x = (t, n, nid, n_s, Suc n') ⟹ P)
  show P
  proof (cases x)
    case (fields t n nid n_s n')
    then show ?thesis
    proof (cases n')
      case 0
      then show ?thesis
      proof cases
        assume Option.is-none (devBC t n nid n_s)
        thus ?thesis
        proof cases
          assume ∃n'<n_s. ¬ Option.is-none (devBC t n nid n')
          with ‹x = (t, n , nid, n_s, n')› ‹Option.is-none (devBC t n nid n_s)› ‹n'=0›
          show ?thesis using a1 by simp
        next
          assume ¬ (∃n'<n_s. ¬ Option.is-none (devBC t n nid n'))
          with ‹x = (t, n , nid, n_s, n')› ‹Option.is-none (devBC t n nid n_s)› ‹n'=0›
          show ?thesis using a2 by simp
        qed
      next
        assume ¬ Option.is-none (devBC t n nid n_s)
        with ‹x = (t, n , nid, n_s, n')› ‹n'=0› show ?thesis using a3 by simp
      qed
    next
      case (Suc n'')
      then show ?thesis
      proof cases
        assume Option.is-none (devBC t n nid (n_s + Suc n''))
        with ‹x = (t, n , nid, n_s, n')› ‹n'=Suc n''›
        show ?thesis using a5[of t n nid n_s n''] by simp
      next
```

```
          assume ¬ Option.is-none (devBC t n nid (n_s + Suc n''))
          with ‹x = (t, n , nid, n_s, n')› ‹n'=Suc n''›
          show ?thesis using a4[of t n nid n_s n''] by simp
        qed
      qed
    qed
  qed
qed
termination by lexicographic-order

lemma devExt-same:
  assumes ∀ n'''>n'. n'''≤n'' ⟶ Option.is-none (devBC t n nid n''')
    and n'≥n_s
    and n'''≤n''
  shows n'''≥n'⟹devExt t n nid n_s (n'''−n_s) = devExt t n nid n_s (n'−n_s)
proof (induction n''' rule: dec-induct)
  case base
  then show ?case by simp
next
  case (step n'''')
  hence Suc n''''>n' by simp
  moreover from step.hyps assms(3) have Suc n''''≤n'' by simp
  ultimately have Option.is-none (devBC t n nid (Suc n'''')) using assms(1) by simp
  moreover from assms(2) step.hyps have n''''≥n_s by simp
  hence Suc n'''' = n_s + Suc (n''''−n_s) by simp
  ultimately have Option.is-none (devBC t n nid (n_s + Suc (n''''−n_s))) by metis
  hence devExt t n nid n_s (Suc (n''''−n_s)) = devExt t n nid n_s (n''''−n_s) by simp
  moreover from ‹n''''≥n_s› have Suc (n''''−n_s) = Suc n''''−n_s by simp
  ultimately have devExt t n nid n_s (Suc n''''−n_s) = devExt t n nid n_s (n''''−n_s) by simp
  with step.IH show ?case by simp
qed

lemma devExt-bc[simp]:
  assumes ¬ Option.is-none (devBC t n nid (n'+n''))
  shows devExt t n nid n' n'' = bc (σ_the (devBC t n nid (n'+n''))(t (n'+n'')))
proof (cases n'')
  case 0
  with assms show ?thesis by simp
next
  case (Suc nat)
  with assms show ?thesis by simp
qed

lemma devExt-greatest:
  assumes ∃ n'''<n'+n''. ¬ Option.is-none (devBC t n nid n''')
    and Option.is-none (devBC t n nid (n'+n'')) and ¬ n''=0
  shows devExt t n nid n' n'' =
    bc (σ_the (devBC t n nid (GREATEST n'''. n'''<(n'+n'') ∧ ¬Option.is-none (devBC t n nid n''')))
    (t (GREATEST n'''. n'''<(n'+n'') ∧ ¬Option.is-none (devBC t n nid n'''))))
```

```
proof −
  let ?P=λn'''. n'''<(n'+n'') ∧ ¬Option.is-none (devBC t n nid n''')
  let ?G=GREATEST n'''. ?P n'''
  have ∀n'''>n'+n''. ¬ ?P n''' by simp
  with ‹∃n'''<n'+n''. ¬ Option.is-none (devBC t n nid n''')› have
    ∃n'''. ?P n''' ∧ (∀n''''. ?P n'''' ⟶ n''''≤n''') using boundedGreatest[of ?P] by blast
  hence ?P ?G using GreatestI-ex-nat[of ?P] by auto
  hence ¬Option.is-none (devBC t n nid ?G) by simp
  show ?thesis
  proof cases
    assume ?G>n'
    hence ?G−n'+n' = ?G by simp
    with ‹¬Option.is-none (devBC t n nid ?G)›
      have ¬Option.is-none (devBC t n nid (?G−n'+n')) by simp
    moreover from ‹?G>n'› have ?G−n'≠0 by auto
    hence ∃nat. Suc nat = ?G − n' by presburger
    then obtain nat where Suc nat = ?G−n' by auto
    ultimately have ¬Option.is-none (devBC t n nid (n'+Suc nat)) by simp
   hence devExt t n nid n' (Suc nat) = bc (σthe (devBC t n nid (n' + Suc nat)) t (n' + Suc nat))
      by simp
    with ‹Suc nat = ?G − n'› have
      devExt t n nid n' (?G − n') = bc (σthe (devBC t n nid (?G−n'+n'))(t (?G−n'+n')))
      by simp
    with ‹?G−n'+n' = ?G› have
      devExt t n nid n' (?G − n') = bc (σthe (devBC t n nid ?G)(t ?G)) by simp
    moreover have devExt t n nid n' (n' + n'' − n') = devExt t n nid n' (?G − n')
    proof −
      from ‹∃n'''. ?P n''' ∧ (∀n''''. ?P n'''' ⟶ n''''≤n''')› have ∀n'''. ?P n''' ⟶ n'''≤?G
        using Greatest-le-nat[of ?P] by blast
      hence ∀n'''>?G. n'''<n'+n'' ⟶ Option.is-none (devBC t n nid n''') by auto
      with ‹Option.is-none (devBC t n nid (n'+n''))›
        have ∀n'''>?G. n'''≤n'+n'' ⟶ Option.is-none (devBC t n nid n''') by auto
      moreover from ‹?P ?G› have ?G≤n'+n'' by simp
      moreover from ‹?G>n'› have ?G≥n' by simp
      ultimately show ?thesis using ‹?G>n'› devExt-same[of ?G n'+n'' t n nid n' n'+n'']
        by blast
    qed
    ultimately show ?thesis by simp
  next
    assume ¬?G>n'
    thus ?thesis
    proof cases
      assume ?G=n'
      with ‹¬Option.is-none (devBC t n nid ?G)› have ¬ Option.is-none (devBC t n nid n')
        by simp
      with ‹¬Option.is-none (devBC t n nid ?G)› have
        devExt t n nid n' 0 = bc (σthe (devBC t n nid n')(t n')) by simp
      moreover have devExt t n nid n' n'' = devExt t n nid n' 0
      proof −
```

```
    from ⟨∃n'''. ?P n''' ∧ (∀n''''. ?P n'''' ⟶ n''''≤n''')⟩
      have ∀n'''>?G. ?P n''' ⟶ n'''≤?G
      using Greatest-le-nat[of ?P] by blast
    with ⟨?G=n'⟩
      have ∀n'''>n'. n''' < n' + n'' ⟶ Option.is-none (devBC t n nid n''') by simp
    with ⟨Option.is-none (devBC t n nid (n'+n''))⟩
      have ∀n'''>n'. n'''≤n'+n'' ⟶ Option.is-none (devBC t n nid n''') by auto
    moreover from ⟨¬ n''=0⟩ have n'<n'+n'' by simp
    ultimately show ?thesis using devExt-same[of n' n'+n'' t n nid n' n'+n''] by simp
  qed
  ultimately show ?thesis using ⟨?G=n'⟩ by simp
next
  assume ¬?G=n'
  with ⟨¬?G>n'⟩ have ?G<n' by simp
  hence devExt t n nid n' n'' = devExt t n nid n' 0
  proof –
    from ⟨∃n'''. ?P n''' ∧ (∀n''''. ?P n'''' ⟶ n''''≤n''')⟩
      have ∀n'''>?G. ?P n''' ⟶ n'''≤?G
      using Greatest-le-nat[of ?P] by blast
    with ⟨¬?G>n'⟩ have ∀n'''>n'. n'''<n'+n'' ⟶ Option.is-none (devBC t n nid n''')
      by auto
    with ⟨Option.is-none (devBC t n nid (n'+n''))⟩
      have ∀n'''>n'. n'''≤n'+n'' ⟶ Option.is-none (devBC t n nid n''') by auto
    moreover from ⟨?P ?G⟩ have ?G<n'+n'' by simp
    moreover from ⟨¬ n''=0⟩ have n'<n'+n'' by simp
    ultimately show ?thesis using devExt-same[of n' n'+n'' t n nid n' n'+n''] by simp
  qed
  moreover have devExt t n nid n' 0 =
    bc (σ_the (devBC t n nid (GREATEST n'''. n'''<n' ∧ ¬Option.is-none (devBC t n nid n''')))
    (t (GREATEST n'''. n'''<n' ∧ ¬Option.is-none (devBC t n nid n'''))))
  proof –
    from ⟨¬ n''=0⟩ have n'<n'+n'' by simp
    moreover from ⟨∃n'''. ?P n''' ∧ (∀n''''. ?P n'''' ⟶ n''''≤n''')⟩
      have ∀n'''>?G. ?P n''' ⟶ n'''≤?G using Greatest-le-nat[of ?P] by blast
    ultimately have Option.is-none (devBC t n nid n') using ⟨?G<n'⟩ by simp
    moreover from ⟨∀n'''>?G. ?P n''' ⟶ n'''≤?G⟩ ⟨?G<n'⟩ ⟨n'<n'+n''⟩
      have ∀n'''≥n'. n'''<n'+n'' ⟶ Option.is-none (devBC t n nid n''') by auto
    have ∃n'''<n'. ¬ Option.is-none (devBC t n nid n''')
    proof –
      from ⟨∃n'''<n'+n''. ¬ Option.is-none (devBC t n nid n''')⟩ obtain n'''
        where n'''<n'+n'' and ¬ Option.is-none (devBC t n nid n''') by auto
      moreover have n'''<n'
      proof (rule ccontr)
        assume ¬n'''<n'
        hence n'''≥n' by simp
        with ⟨∀n'''≥n'. n'''<n'+n'' ⟶ Option.is-none (devBC t n nid n''')⟩ ⟨n'''<n'+n''⟩
          ⟨¬ Option.is-none (devBC t n nid n''')⟩ show False by simp
      qed
      ultimately show ?thesis by auto
```

```
        qed
        ultimately show ?thesis by simp
      qed
      moreover have (GREATEST n'''. n'''<n' ∧ ¬Option.is-none (devBC t n nid n''')) = ?G
      proof(rule Greatest-equality)
        from ‹?P ?G› have ?G < n'+n'' and ¬Option.is-none (devBC t n nid ?G) by auto
        with ‹?G<n'› show ?G < n' ∧ ¬ Option.is-none (devBC t n nid ?G) by simp
      next
        fix y assume y < n' ∧ ¬ Option.is-none (devBC t n nid y)
        moreover from ‹∃ n'''. ?P n''' ∧ (∀ n''''. ?P n'''' ⟶ n''''≤n''')›
          have ∀ n'''. ?P n''' ⟶ n'''≤?G using Greatest-le-nat[of ?P] by blast
        ultimately show y ≤ ?G by simp
      qed
      ultimately show ?thesis by simp
    qed
  qed
qed

lemma devExt-shift: devExt t n nid (n'+n'') 0 = devExt t n nid n' n''
proof (cases)
  assume n''=0
  thus ?thesis by simp
next
  assume ¬ (n''=0)
  thus ?thesis
  proof (cases)
    assume Option.is-none (devBC t n nid (n'+n''))
    thus ?thesis
    proof cases
      assume ∃ n'''<n'+n''. ¬ Option.is-none (devBC t n nid n''')
      with ‹Option.is-none (devBC t n nid (n'+n''))› have
        devExt t n nid (n'+n'') 0 =
      bc (σ_the (devBC t n nid (GREATEST n'''. n'''<(n'+n'') ∧ ¬Option.is-none (devBC t n nid n''')))
        (t (GREATEST n'''. n'''<(n'+n'') ∧ ¬Option.is-none (devBC t n nid n''')))) by simp
      moreover from ‹¬ (n''=0)› ‹Option.is-none (devBC t n nid (n'+n''))›
        ‹∃ n'''<n'+n''. ¬ Option.is-none (devBC t n nid n''')›
      have devExt t n nid n' n'' =
      bc (σ_the (devBC t n nid (GREATEST n'''. n'''<(n'+n'') ∧ ¬Option.is-none (devBC t n nid n''')))
        (t (GREATEST n'''. n'''<(n'+n'') ∧ ¬Option.is-none (devBC t n nid n'''))))
        using devExt-greatest by simp
      ultimately show ?thesis by simp
    next
      assume ¬ (∃ n'''<n'+n''. ¬ Option.is-none (devBC t n nid n'''))
     with ‹Option.is-none (devBC t n nid (n'+n''))› have devExt t n nid (n'+n'') 0=[] by simp
      moreover have devExt t n nid n' n''=[]
      proof −
        from ‹¬ (∃ n'''<n'+n''. ¬ Option.is-none (devBC t n nid n'''))› ‹n''≠0›
          have Option.is-none (devBC t n nid n') by simp
        moreover from ‹¬ (∃ n'''<n'+n''. ¬ Option.is-none (devBC t n nid n'''))›
```

```
          have ¬ (∃n'''<n'. ¬ Option.is-none (devBC t n nid n''')) by simp
        ultimately have devExt t n nid n' 0=[] by simp
        moreover have devExt t n nid n' n''=devExt t n nid n' 0
        proof -
          from ‹¬ (∃n'''<n'+n''. ¬ Option.is-none (devBC t n nid n'''))›
            have ∀n'''>n'. n''' < n' + n'' ⟶ Option.is-none (devBC t n nid n''') by simp
          with ‹Option.is-none (devBC t n nid (n'+n''))›
            have ∀n'''>n'. n'''≤n'+n'' ⟶ Option.is-none (devBC t n nid n''') by auto
          moreover from ‹¬ n''=0› have n'<n'+n'' by simp
          ultimately show ?thesis using devExt-same[of n' n'+n'' t n nid n' n'+n''] by simp
        qed
        ultimately show ?thesis by simp
      qed
      ultimately show ?thesis by simp
    qed
  next
    assume ¬ Option.is-none (devBC t n nid (n'+n''))
    hence devExt t n nid (n'+n'') 0 = bc (σ_{the (devBC t n nid (n'+n''))}(t (n'+n''))) by simp
    moreover from ‹¬ Option.is-none (devBC t n nid (n'+n''))›
      have devExt t n nid n' n'' = bc (σ_{the (devBC t n nid (n'+n''))}(t (n'+n''))) by simp
    ultimately show ?thesis by simp
  qed
qed

lemma devExt-bc-geq:
  assumes ¬ Option.is-none (devBC t n nid n') and n'≥n_s
  shows devExt t n nid n_s (n'−n_s) = bc (σ_{the (devBC t n nid n')}(t n')) (is ?LHS = ?RHS)
proof -
  have devExt t n nid n_s (n'−n_s) = devExt t n nid (n_s+(n'−n_s)) 0 using devExt-shift by auto
  moreover from assms(2) have n_s+(n'−n_s) = n' by simp
  ultimately have devExt t n nid n_s (n'−n_s) = devExt t n nid n' 0 by simp
  with assms(1) show ?thesis by simp
qed

lemma his-bc-empty:
  assumes (n',nid')∈ his t n nid and ¬(∃n''<n'. ∃nid''. (n'',nid'')∈ his t n nid)
  shows bc (σ_{nid'}(t n')) = []
proof -
  have ¬ (∃x. his-prop t n nid n' nid' x)
  proof (rule ccontr)
    assume ¬ ¬ (∃x. his-prop t n nid n' nid' x)
    hence ∃x. his-prop t n nid n' nid' x by simp
    with ‹(n',nid')∈ his t n nid› have (SOME x. his-prop t n nid n' nid' x) ∈ his t n nid
      using his.intros by simp
    moreover from ‹∃x. his-prop t n nid n' nid' x›
      have his-prop t n nid n' nid' (SOME x. his-prop t n nid n' nid' x)
      using someI-ex[of λx. his-prop t n nid n' nid' x] by auto
    hence (∃n. latestAct-cond nid' t n' n) ∧
      fst (SOME x. his-prop t n nid n' nid' x) = ⟨nid' ← t⟩_{n'}
```

by *force*
hence $\mathit{fst}\ (\mathit{SOME}\ x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n'\ \mathit{nid}'\ x) < n'$
using $\mathit{latestAct\text{-}prop}(2)[\mathit{of}\ n'\ \mathit{nid}'\ t]$ **by** *force*
ultimately have $\mathit{fst}\ (\mathit{SOME}\ x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n'\ \mathit{nid}'\ x){<}n' \wedge$
$(\mathit{fst}\ (\mathit{SOME}\ x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n'\ \mathit{nid}'\ x),$
$\mathit{snd}\ (\mathit{SOME}\ x.\ \mathit{his\text{-}prop}\ t\ n\ \mathit{nid}\ n'\ \mathit{nid}'\ x)){\in}\ \mathit{his}\ t\ n\ \mathit{nid}$ **by** *simp*
thus *False* **using** $\mathit{assms}(2)$ **by** *blast*
qed
hence $\forall x.\ \neg\ (\exists n.\ \mathit{latestAct\text{-}cond}\ \mathit{nid}'\ t\ n'\ n) \vee \neg\ \wr\mathit{snd}\ x\wr_{t\ (\mathit{fst}\ x)} \vee$
$\neg\ \mathit{fst}\ x = \langle \mathit{nid}' \leftarrow t\rangle_{n'} \vee \neg\ (\mathit{prefix}\ (\mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')))\ (\mathit{bc}\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x)))) \vee$
$(\exists b.\ \mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')) = (\mathit{bc}\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x))))\ @\ [b] \wedge \mathit{mining}\ (\sigma_{\mathit{nid}'}(t\ n'))))$
by *auto*
hence $\neg\ (\exists n.\ \mathit{latestAct\text{-}cond}\ \mathit{nid}'\ t\ n'\ n) \vee (\exists n.\ \mathit{latestAct\text{-}cond}\ \mathit{nid}'\ t\ n'\ n) \wedge$
$(\forall x.\ \neg\ \wr\mathit{snd}\ x\wr_{t\ (\mathit{fst}\ x)} \vee \neg\ \mathit{fst}\ x = \langle \mathit{nid}' \leftarrow t\rangle_{n'} \vee$
$\neg\ (\mathit{prefix}\ (\mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')))\ (\mathit{bc}\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x)))) \vee$
$(\exists b.\ \mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')) = (\mathit{bc}\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x))))\ @\ [b] \wedge \mathit{mining}\ (\sigma_{\mathit{nid}'}(t\ n')))))$
by *auto*
thus *?thesis*
proof
assume $\neg\ (\exists n.\ \mathit{latestAct\text{-}cond}\ \mathit{nid}'\ t\ n'\ n)$
moreover from $\mathit{assms}(1)$ **have** $\wr\mathit{nid}'\wr_{t\ n'}$ **using** *his-act* **by** *simp*
ultimately show *?thesis* **using** *init-model* **by** *simp*
next
assume $(\exists n.\ \mathit{latestAct\text{-}cond}\ \mathit{nid}'\ t\ n'\ n) \wedge (\forall x.\ \neg\ \wr\mathit{snd}\ x\wr_{t\ (\mathit{fst}\ x)} \vee$
$\neg\ \mathit{fst}\ x = \langle \mathit{nid}' \leftarrow t\rangle_{n'} \vee \neg\ (\mathit{prefix}\ (\mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')))\ (\mathit{bc}\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x)))) \vee$
$(\exists b.\ \mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')) = (\mathit{bc}\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x))))\ @\ [b] \wedge \mathit{mining}\ (\sigma_{\mathit{nid}'}(t\ n')))))$
hence $\exists n.\ \mathit{latestAct\text{-}cond}\ \mathit{nid}'\ t\ n'\ n$ **and**
$\forall x.\ \neg\ \wr\mathit{snd}\ x\wr_{t\ (\mathit{fst}\ x)} \vee \neg\ \mathit{fst}\ x = \langle \mathit{nid}' \leftarrow t\rangle_{n'} \vee$
$\neg\ (\mathit{prefix}\ (\mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')))\ (\mathit{bc}\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x)))) \vee$
$(\exists b.\ \mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')) = (\mathit{bc}\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x))))\ @\ [b] \wedge \mathit{mining}\ (\sigma_{\mathit{nid}'}(t\ n'))))$
by *auto*
hence *asmp*: $\forall x.\ \wr\mathit{snd}\ x\wr_{t\ (\mathit{fst}\ x)} \longrightarrow \mathit{fst}\ x = \langle \mathit{nid}' \leftarrow t\rangle_{n'} \longrightarrow$
$\neg\ (\mathit{prefix}\ (\mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')))\ (\mathit{bc}\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x)))) \vee$
$(\exists b.\ \mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')) = (\mathit{bc}\ (\sigma_{\mathit{snd}\ x}(t\ (\mathit{fst}\ x))))\ @\ [b] \wedge \mathit{mining}\ (\sigma_{\mathit{nid}'}(t\ n'))))$
by *auto*
show *?thesis*
proof *cases*
assume $\mathit{trusted}\ \mathit{nid}'$
moreover from $\mathit{assms}(1)$ **have** $\wr\mathit{nid}'\wr_{t\ n'}$ **using** *his-act* **by** *simp*
ultimately obtain nid'' **where** $\wr\mathit{nid}''\wr_{t\ \langle \mathit{nid}' \leftarrow t\rangle_{n'}}$ **and** $\mathit{mining}\ (\sigma_{\mathit{nid}'}t\ n') \wedge$
$(\exists b.\ \mathit{bc}\ (\sigma_{\mathit{nid}'}t\ n') = \mathit{bc}\ (\sigma_{\mathit{nid}''}t\ \langle \mathit{nid}' \leftarrow t\rangle_{n'})\ @\ [b]) \vee \neg\ \mathit{mining}\ (\sigma_{\mathit{nid}'}t\ n') \wedge$
$\mathit{bc}\ (\sigma_{\mathit{nid}'}t\ n') = \mathit{bc}\ (\sigma_{\mathit{nid}''}t\ \langle \mathit{nid}' \leftarrow t\rangle_{n'})$ **using** ‹$\exists n.\ \mathit{latestAct\text{-}cond}\ \mathit{nid}'\ t\ n'\ n$›
$\mathit{bhv\text{-}tr\text{-}context}[\mathit{of}\ \mathit{nid}'\ t\ n']$ **by** *auto*
moreover from ‹$\wr\mathit{nid}''\wr_{t\ \langle \mathit{nid}' \leftarrow t\rangle_{n'}}$› **have**
$\neg\ (\mathit{prefix}\ (\mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')))\ (\mathit{bc}\ (\sigma_{\mathit{nid}''}(t\ (\langle \mathit{nid}' \leftarrow t\rangle_{n'})))) \vee$
$(\exists b.\ \mathit{bc}\ (\sigma_{\mathit{nid}'}(t\ n')) = (\mathit{bc}\ (\sigma_{\mathit{nid}''}(t\ (\langle \mathit{nid}' \leftarrow t\rangle_{n'}))))\ @\ [b] \wedge$
$\mathit{mining}\ (\sigma_{\mathit{nid}'}(t\ n'))))$ **using** *asmp* **by** *auto*
ultimately have *False* **by** *auto*

```
      thus ?thesis ..
    next
      assume ¬ trusted nid'
      moreover from assms(1) have ⦃nid'⦄_{t n'} using his-act by simp
      ultimately obtain nid'' where ⦃nid''⦄_{t ⟨nid' ← t⟩_{n'}} and (mining (σ_{nid'}t n') ∧
        (∃ b. prefix (bc (σ_{nid'}t n')) (bc (σ_{nid''}t ⟨nid' ← t⟩_{n'}) @ [b])) ∨
        ¬ mining (σ_{nid'}t n') ∧ prefix (bc (σ_{nid'}t n')) (bc (σ_{nid''}t ⟨nid' ← t⟩_{n'})))
        using ‹∃ n. latestAct-cond nid' t n' n› bhv-ut-context[of nid' t n'] by auto
      moreover from ‹⦃nid''⦄_{t ⟨nid' ← t⟩_{n'}}› have
        ¬ (prefix (bc (σ_{nid'}(t n'))) (bc (σ_{nid''}(t (⟨nid' ← t⟩_{n'})))) ∨
        (∃ b. bc (σ_{nid'}(t n')) = (bc (σ_{nid''}(t (⟨nid' ← t⟩_{n'})))) @ [b] ∧
        mining (σ_{nid'}(t n')))) using asmp by auto
      ultimately have False by auto
      thus ?thesis ..
    qed
  qed
qed

lemma devExt-devop:
  prefix (devExt t n nid n_s (Suc n')) (devExt t n nid n_s n') ∨
  (∃ b. devExt t n nid n_s (Suc n') = devExt t n nid n_s n' @ [b]) ∧
  ¬ Option.is-none (devBC t n nid (n_s + Suc n')) ∧
  ⦃the (devBC t n nid (n_s + Suc n'))⦄_{t (n_s + Suc n')} ∧
  n_s + Suc n' ≤ n ∧ mining (σ_{the (devBC t n nid (n_s + Suc n'))}(t (n_s + Suc n')))
proof cases
  assume n_s + Suc n' > n
  hence ¬(∃ nid'. (n_s + Suc n', nid') ∈ his t n nid) using his-le by fastforce
  hence Option.is-none (devBC t n nid (n_s + Suc n')) using devBC-def by simp
  hence devExt t n nid n_s (Suc n') = devExt t n nid n_s n' by simp
  thus ?thesis by simp
next
  assume ¬n_s + Suc n' > n
  hence n_s + Suc n' ≤ n by simp
  show ?thesis
  proof cases
    assume Option.is-none (devBC t n nid (n_s + Suc n'))
    hence devExt t n nid n_s (Suc n') = devExt t n nid n_s n' by simp
    thus ?thesis by simp
  next
    assume ¬ Option.is-none (devBC t n nid (n_s + Suc n'))
    hence
      devExt t n nid n_s (Suc n') = bc (σ_{the (devBC t n nid (n_s + Suc n'))}(t (n_s + Suc n')))
      by simp
    moreover have prefix (bc (σ_{the (devBC t n nid (n_s + Suc n'))}(t (n_s + Suc n'))))
      (devExt t n nid n_s n') ∨
      (∃ b. bc (σ_{the (devBC t n nid (n_s + Suc n'))}(t (n_s + Suc n'))) = devExt t n nid n_s n' @ [b]
      ∧ ¬ Option.is-none (devBC t n nid (n_s + Suc n')) ∧
      ⦃the (devBC t n nid (n_s + Suc n'))⦄_{t (n_s + Suc n')} ∧
```

```
 n_s + Suc n' ≤ n ∧ mining (σ_{the (devBC t n nid (n_s + Suc n'))}^{(t (n_s + Suc n'))}))
proof cases
 assume ∃ n''<n_s + Suc n'. ∃ nid'. (n'',nid')∈ his t n nid
 let ?nid=(THE nid'. (n_s + Suc n',nid')∈his t n nid)
 let ?x=SOME x. his-prop t n nid (n_s + Suc n') ?nid x
 from ‹¬ Option.is-none (devBC t n nid (n_s + Suc n'))›
  have n_s + Suc n'≤n using devExt-nopt-leq by simp
 moreover from ‹¬ Option.is-none (devBC t n nid (n_s + Suc n'))›
  have ∃ nid'. (n_s + Suc n',nid')∈his t n nid using his-ex by simp
 ultimately have ∃ x. his-prop t n nid (n_s + Suc n')
  (THE nid'. ((n_s + Suc n'),nid')∈his t n nid) x
  and (hisPred t n nid (n_s + Suc n'),
   (SOME nid'. (hisPred t n nid (n_s + Suc n'), nid') ∈ his t n nid)) = ?x
  using ‹∃ n''<n_s + Suc n'. ∃ nid'. (n'',nid')∈ his t n nid›
  his-determ-ext[of n_s + Suc n' n t nid] by auto
 moreover have bc (σ_{(SOME nid'. (hisPred t n nid (n_s + Suc n'), nid') ∈ his t n nid)}^{(t
  (hisPred t n nid (n_s + Suc n'))))} = devExt t n nid n_s n'
 proof cases
  assume Option.is-none (devBC t n nid (n_s+n'))
  have devExt t n nid n_s n' =
  bc (σ_{the (devBC t n nid (GREATEST n''. n''<n_s+n' ∧ ¬Option.is-none (devBC t n nid n'')))}
   ^{(t (GREATEST n''. n''<n_s+n' ∧ ¬Option.is-none (devBC t n nid n''))))}
  proof cases
   assume n'=0
   moreover have ∃ n''<n_s+n'. ¬Option.is-none (devBC t n nid n'')
   proof -
    from ‹∃ n''<n_s + Suc n'. ∃ nid'. (n'',nid')∈ his t n nid› obtain n''
     where n''<Suc n_s + n' and ∃ nid'. (n'',nid')∈ his t n nid by auto
    hence ¬ Option.is-none (devBC t n nid n'') using devBC-def by simp
    moreover from ‹¬ Option.is-none (devBC t n nid n'')›
     ‹Option.is-none (devBC t n nid (n_s+n'))› have ¬ n''=n_s+n' by auto
    with ‹n''<Suc n_s+n'› have n''<n_s+n' by simp
    ultimately show ?thesis by auto
   qed
   ultimately show ?thesis using ‹Option.is-none (devBC t n nid (n_s+n'))› by simp
  next
   assume ¬ n'=0
   moreover have ∃ n''<n_s + n'. ¬ Option.is-none (devBC t n nid n'')
   proof -
    from ‹∃ n''<n_s + Suc n'. ∃ nid'. (n'',nid')∈ his t n nid› obtain n''
     where n''<Suc n_s + n' and ∃ nid'. (n'',nid')∈ his t n nid by auto
    hence ¬ Option.is-none (devBC t n nid n'') using devBC-def by simp
    moreover from ‹¬ Option.is-none (devBC t n nid n'')›
     ‹Option.is-none (devBC t n nid (n_s+n'))›
     have ¬ n''=n_s+n' by auto
    with ‹n''<Suc n_s+n'› have n''<n_s+n' by simp
    ultimately show ?thesis by auto
   qed
   with ‹¬ (n'=0)› ‹Option.is-none (devBC t n nid (n_s+n'))› show ?thesis
```

```
    using devExt-greatest[of n_s n' t n nid] by simp
qed
moreover have (GREATEST n''. n''<n_s+n' ∧ ¬Option.is-none (devBC t n nid n''))=
  hisPred t n nid (n_s + Suc n')
proof −
  have (λn''. n''<n_s+n' ∧ ¬Option.is-none (devBC t n nid n'')) =
    (λn''. ∃nid'. (n'',nid')∈ his t n nid ∧ n'' < n_s + Suc n')
  proof
    fix n''
    show (n'' < n_s + n' ∧ ¬ Option.is-none (devBC t n nid n'')) =
      (∃nid'. (n'', nid') ∈ his t n nid ∧ n'' < n_s + Suc n')
    proof
      assume n'' < n_s + n' ∧ ¬ Option.is-none (devBC t n nid n'')
      thus (∃nid'. (n'', nid') ∈ his t n nid ∧ n'' < n_s + Suc n')
        using his-ex by simp
    next
      assume (∃nid'. (n'', nid') ∈ his t n nid ∧ n'' < n_s + Suc n')
      hence ∃nid'. (n'', nid') ∈ his t n nid and n'' < n_s + Suc n' by auto
      hence ¬ Option.is-none (devBC t n nid n'') using devBC-def by simp
      moreover from ‹¬ Option.is-none (devBC t n nid n'')›
        ‹Option.is-none (devBC t n nid (n_s+n'))›
      have n''≠n_s+n' by auto
      with ‹n'' < n_s + Suc n'› have n'' < n_s + n' by simp
      ultimately show n'' < n_s + n' ∧ ¬ Option.is-none (devBC t n nid n'') by simp
    qed
  qed
  hence (GREATEST n''. n''<n_s+n' ∧
    ¬Option.is-none (devBC t n nid n'')) =
    (GREATEST n''. ∃nid'. (n'',nid')∈ his t n nid ∧ n'' < n_s + Suc n')
    using arg-cong[of λn''. n''<n_s+n' ∧ ¬Option.is-none (devBC t n nid n'')
    (λn''. ∃nid'. (n'',nid')∈ his t n nid ∧ n'' < n_s + Suc n')] by simp
  with hisPred-def show ?thesis by simp
qed
moreover have the (devBC t n nid (hisPred t n nid (n_s + Suc n'))) =
  (SOME nid'. (hisPred t n nid (n_s + Suc n'), nid') ∈ his t n nid)
proof −
  from ‹∃n''<n_s + Suc n'. ∃nid'. (n'',nid')∈ his t n nid›
    have ∃nid'. (hisPred t n nid (n_s + Suc n'), nid')∈ his t n nid
    using hisPrev-prop(2) by simp
  hence the (devBC t n nid (hisPred t n nid (n_s + Suc n'))) =
    (THE nid'. (hisPred t n nid (n_s + Suc n'), nid')∈his t n nid)
    using devBC-def by simp
  moreover from ‹∃nid'. (hisPred t n nid (n_s + Suc n'), nid')∈ his t n nid›
    have (hisPred t n nid (n_s + Suc n'),
    SOME nid'. (hisPred t n nid (n_s + Suc n'), nid') ∈ his t n nid) ∈ his t n nid
    using someI-ex[of λnid'. (hisPred t n nid (n_s + Suc n'), nid')∈his t n nid] by simp
  hence (THE nid'. (hisPred t n nid (n_s + Suc n'), nid')∈his t n nid) =
    (SOME nid'. (hisPred t n nid (n_s + Suc n'), nid') ∈ his t n nid)
    using his-determ-the by simp
```

```
      ultimately show ?thesis by simp
    qed
    ultimately show ?thesis by simp
  next
    assume ¬ Option.is-none (devBC t n nid (n_s+n'))
    hence devExt t n nid n_s n' = bc (σ_{the (devBC t n nid (n_s+n'))}(t (n_s+n')))
    proof cases
      assume n'=0
      with ⟨¬ Option.is-none (devBC t n nid (n_s+n'))⟩ show ?thesis by simp
    next
      assume ¬ n'=0
      hence ∃nat. n' = Suc nat by presburger
      then obtain nat where n' = Suc nat by auto
      with ⟨¬ Option.is-none (devBC t n nid (n_s+n'))⟩ have
        devExt t n nid n_s (Suc nat) =
        bc (σ_{the (devBC t n nid (n_s + Suc nat))}(t (n_s + Suc nat))) by simp
      with ⟨n' = Suc nat⟩ show ?thesis by simp
    qed
    moreover have hisPred t n nid (n_s + Suc n') = n_s+n'
    proof −
      have (GREATEST n''. ∃nid'. (n'',nid')∈ his t n nid ∧ n'' < (n_s + Suc n')) = n_s+n'
      proof (rule Greatest-equality)
        from ⟨¬ Option.is-none (devBC t n nid (n_s+n'))⟩
          have ∃nid'. (n_s + n', nid') ∈ his t n nid
          using his-ex by simp
        thus ∃nid'. (n_s + n', nid') ∈ his t n nid ∧ n_s + n' < n_s + Suc n' by simp
      next
        fix y assume ∃nid'. (y, nid') ∈ his t n nid ∧ y < n_s + Suc n'
        thus y ≤ n_s + n' by simp
      qed
      thus ?thesis using hisPred-def by simp
    qed
    moreover have the (devBC t n nid (hisPred t n nid (n_s + Suc n'))) =
      (SOME nid'. (hisPred t n nid (n_s + Suc n'), nid') ∈ his t n nid)
    proof −
      from ⟨∃n''<n_s + Suc n'. ∃nid'. (n'',nid')∈ his t n nid⟩
        have ∃nid'. (hisPred t n nid (n_s + Suc n'), nid')∈ his t n nid
        using hisPrev-prop(2) by simp
      hence the (devBC t n nid (hisPred t n nid (n_s + Suc n'))) =
        (THE nid'. (hisPred t n nid (n_s + Suc n'), nid')∈his t n nid)
        using devBC-def by simp
      moreover from ⟨∃nid'. (hisPred t n nid (n_s + Suc n'), nid')∈ his t n nid⟩
        have (hisPred t n nid (n_s + Suc n'),
        SOME nid'. (hisPred t n nid (n_s + Suc n'), nid') ∈ his t n nid) ∈ his t n nid
        using someI-ex[of λnid'. (hisPred t n nid (n_s + Suc n'), nid')∈his t n nid]
          by simp
      hence (THE nid'. (hisPred t n nid (n_s + Suc n'), nid')∈his t n nid) =
        (SOME nid'. (hisPred t n nid (n_s + Suc n'), nid') ∈ his t n nid)
        using his-determ-the by simp
```

```
      ultimately show ?thesis by simp
    qed
    ultimately show ?thesis by simp
  qed
  ultimately have bc (σ_(snd ?x)(t (fst ?x))) = devExt t n nid n_s n'
    using fst-conv[of hisPred t n nid (n_s + Suc n')
    (SOME nid'. (hisPred t n nid (n_s + Suc n'), nid') ∈ his t n nid)]
    snd-conv[of hisPred t n nid (n_s + Suc n')
    (SOME nid'. (hisPred t n nid (n_s + Suc n'), nid') ∈ his t n nid)] by simp
  moreover from ‹∃x. his-prop t n nid (n_s + Suc n') ?nid x›
    have his-prop t n nid (n_s + Suc n') ?nid ?x
    using someI-ex[of λx. his-prop t n nid (n_s + Suc n') ?nid x] by blast
  hence prefix (bc (σ_?nid(t (n_s + Suc n')))) (bc (σ_(snd ?x)(t (fst ?x)))) ∨
    (∃b. bc (σ_?nid(t (n_s + Suc n'))) = (bc (σ_(snd ?x)(t (fst ?x)))) @ [b] ∧
    mining (σ_?nid(t (n_s + Suc n')))) by blast
  ultimately have prefix (bc (σ_?nid(t (n_s + Suc n')))) (devExt t n nid n_s n') ∨
    (∃b. bc (σ_?nid(t (n_s + Suc n'))) = (devExt t n nid n_s n') @ [b] ∧
    mining (σ_?nid(t (n_s + Suc n')))) by simp
  moreover from ‹∃nid'. (n_s + Suc n',nid')∈ his t n nid›
    have ?nid=the (devBC t n nid (n_s + Suc n')) using devBC-def by simp
  moreover have ⌇the (devBC t n nid (n_s + Suc n'))⌇_(t (n_s + Suc n'))
  proof -
    from ‹∃nid'. (n_s + Suc n',nid')∈his t n nid› obtain nid'
      where (n_s + Suc n',nid')∈his t n nid by auto
    with his-determ-the have nid' = (THE nid'. (n_s + Suc n', nid') ∈ his t n nid) by simp
    with ‹?nid=the (devBC t n nid (n_s + Suc n'))›
      have the (devBC t n nid (n_s + Suc n')) = nid' by simp
    with ‹(n_s + Suc n',nid')∈his t n nid› show ?thesis using his-act by simp
  qed
  ultimately show ?thesis
    using ‹¬ Option.is-none (devBC t n nid (n_s+Suc n'))› ‹n_s + Suc n' ≤ n› by simp
next
  assume ¬ (∃n''<n_s + Suc n'. ∃nid'. (n'',nid')∈ his t n nid)
  moreover have (n_s + Suc n', the (devBC t n nid (n_s + Suc n'))) ∈ his t n nid
  proof -
    from ‹¬ Option.is-none (devBC t n nid (n_s + Suc n'))›
      have ∃nid'. (n_s + Suc n',nid')∈his t n nid using his-ex by simp
      hence the (devBC t n nid (n_s + Suc n')) =
      (THE nid'. (n_s + Suc n', nid') ∈ his t n nid)
      using devBC-def by simp
    moreover from ‹∃nid'. (n_s + Suc n',nid')∈his t n nid› obtain nid'
      where (n_s + Suc n',nid')∈his t n nid by auto
    with his-determ-the have nid' = (THE nid'. (n_s + Suc n', nid') ∈ his t n nid) by simp
    ultimately have the (devBC t n nid (n_s + Suc n')) = nid' by simp
    with ‹(n_s + Suc n',nid')∈his t n nid› show ?thesis by simp
  qed
  ultimately have bc (σ_(the (devBC t n nid (n_s + Suc n')))(t (n_s + Suc n'))) = []
    using his-bc-empty by simp
  thus ?thesis by simp
```

```
    qed
    ultimately show ?thesis by simp
  qed
qed

abbreviation devLgthBC where devLgthBC t n nid n_s ≡ (λn'. length (devExt t n nid n_s n'))

theorem blockchain-save:
  fixes t::nat⇒cnf and n_s and sbc and n
  assumes ∀nid. trusted nid ⟶ prefix sbc (bc (σ_nid(t (⟨nid → t⟩_{n_s}))))
    and ∀nid∈actUt (t n_s). length (bc (σ_nid(t n_s))) < length sbc
    and PoW t n_s≥length sbc + cb
    and ∀n'<n_s. ∀nid. ⦃nid⦄_{t n'} ⟶ length (bc (σ_nid t n')) < length sbc ∨
      prefix sbc (bc (σ_nid(t n')))
    and n≥n_s
  shows ∀nid ∈ actTr (t n). prefix sbc (bc (σ_nid(t n)))
proof (cases)
  assume sbc=[]
  thus ?thesis by simp
next
  assume ¬ sbc=[]
  have n≥n_s ⟹ ∀nid ∈ actTr (t n). prefix sbc (bc (σ_nid(t n)))
  proof (induction n rule: ge-induct)
    case (step n)
    show ?case
    proof
      fix nid assume nid ∈ actTr (t n)
      hence ⦃nid⦄_{t n} and trusted nid using actTr-def by auto
      show prefix sbc (bc (σ_nid t n))
      proof cases
        assume lAct: ∃n' < n. n' ≥ n_s ∧ ⦃nid⦄_{t n'}
        show ?thesis
        proof cases
          assume ∃b∈pin (σ_nid t ⟨nid ← t⟩_n). length b > length (bc (σ_nid t ⟨nid ← t⟩_n))
          moreover from ‹⦃nid⦄_{t n}› have ∃n'≥n. ⦃nid⦄_{t n'} by auto
          moreover from lAct have ∃n'. latestAct-cond nid t n n' by auto
          ultimately have
            ¬ mining (σ_nid t ⟨nid → t⟩_n) ∧ bc (σ_nid t ⟨nid → t⟩_n) =
              MAX (pin (σ_nid t ⟨nid ← t⟩_n)) ∨
            mining (σ_nid t ⟨nid → t⟩_n) ∧ (∃b. bc (σ_nid t ⟨nid → t⟩_n) =
              MAX (pin (σ_nid t ⟨nid ← t⟩_n)) @ [b])
            using ‹trusted nid› bhv-tr-ex[of nid n t] by simp
          moreover have prefix sbc (MAX (pin (σ_nid t ⟨nid ← t⟩_n)))
          proof −
            from ‹∃n'. latestAct-cond nid t n n'› have ⦃nid⦄_{t ⟨nid ← t⟩_n}
              using latestAct-prop(1) by simp
            hence pin (σ_nid t ⟨nid ← t⟩_n) ≠ {} and finite (pin (σ_nid t ⟨nid ← t⟩_n))
              using nempty-input[of nid t ⟨nid ← t⟩_n] finite-input[of nid t ⟨nid ← t⟩_n]
              ‹trusted nid› by auto
```

```
hence MAX (pin (σ_nid t ⟨nid ← t⟩_n)) ∈ pin (σ_nid t ⟨nid ← t⟩_n)
  using max-prop(1) by auto
with ‹⦃nid⦄_t ⟨nid ← t⟩_n› obtain nid' where ⦃nid'⦄_t ⟨nid ← t⟩_n
  and bc (σ_nid'(t ⟨nid ← t⟩_n)) = MAX (pin (σ_nid t ⟨nid ← t⟩_n))
  using closed[where b=MAX (pin (σ_nid t ⟨nid ← t⟩_n))] by blast
moreover have prefix sbc (bc (σ_nid'(t ⟨nid ← t⟩_n)))
proof cases
  assume trusted nid'
  with ‹⦃nid'⦄_t ⟨nid ← t⟩_n› have nid' ∈ actTr (t ⟨nid ← t⟩_n)
    using actTr-def by simp
  moreover from ‹∃ n'. latestAct-cond nid t n n'› have ⟨nid ← t⟩_n < n
    using latestAct-prop(2) by simp
  moreover from lAct have ⟨nid ← t⟩_n ≥ n_s using latestActless by blast
  ultimately show ?thesis using ‹⦃nid'⦄_t ⟨nid ← t⟩_n› step.IH by simp
next
  assume ¬ trusted nid'
  show ?thesis
  proof (rule ccontr)
    assume ¬ prefix sbc (bc (σ_nid'(t ⟨nid ← t⟩_n)))
    moreover have
      ∃ n'≤⟨nid ← t⟩_n. n'≥n_s ∧ length (devExt t ⟨nid ← t⟩_n nid' n' 0) < length sbc ∧
      (∀ n''>n'. n''≤⟨nid ← t⟩_n ∧ ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n'') ⟶
       ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n'')))
    proof cases
      assume ∃ n'≤⟨nid ← t⟩_n. n'≥n_s ∧
        ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n') ∧
        trusted (the (devBC t ⟨nid ← t⟩_n nid' n'))
      hence ∃ n'≤⟨nid ← t⟩_n. n'≥n_s ∧
        ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n') ∧
        trusted (the (devBC t ⟨nid ← t⟩_n nid' n')) ∧ (∀ n''>n'. n''≤⟨nid ← t⟩_n ∧
        ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n'') ⟶
        ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n'')))
      proof -
        let ?P=λn'. n'≤⟨nid ← t⟩_n ∧ n'≥n_s  ∧
          ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n') ∧
          trusted (the (devBC t ⟨nid ← t⟩_n nid' n'))
        from ‹∃ n'≤⟨nid ← t⟩_n. n'≥n_s  ∧
          ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n') ∧
          trusted (the (devBC t ⟨nid ← t⟩_n nid' n'))› have ∃ n'. ?P n' by simp
        moreover have ∀ n'>⟨nid ← t⟩_n. ¬ ?P n' by simp
        ultimately obtain n' where ?P n' and ∀ n''. ?P n'' ⟶ n''≤n'
          using boundedGreatest[of ?P - ⟨nid ← t⟩_n] by auto
        hence ∀ n''>n'. n''≤⟨nid ← t⟩_n ∧
          ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n'')
          ⟶ ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n'')) by auto
        thus ?thesis using ‹?P n'› by auto
      qed
      then obtain n' where n'≤⟨nid ← t⟩_n and
        ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n')
```

```
  and n'≥n_s and trusted (the (devBC t ⟨nid ← t⟩_n nid' n'))
  and ∀ n''>n'. n''≤⟨nid ← t⟩_n ∧
    ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n'')
    ⟶ ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n'')) by auto
hence n'≥n_s and untrusted: ∀ n''>n'. n''≤⟨nid ← t⟩_n ∧
  ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n'') ⟶
  ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n'')) by auto
moreover have ⟨nid ← t⟩_n<n
  using ‹∃ n'. latestAct-cond nid t n n'› latestAct-prop(2) by blast
with ‹n'≤⟨nid ← t⟩_n› have n'<n by simp
moreover from ‹¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n')›
  have ⌇the (devBC t ⟨nid ← t⟩_n nid' n')⌇_t n' using devBC-act by simp
with ‹trusted (the (devBC t ⟨nid ← t⟩_n nid' n'))›
 have the (devBC t ⟨nid ← t⟩_n nid' n') ∈actTr (t n') using actTr-def by simp
ultimately have prefix sbc (bc (σ_the (devBC t ⟨nid ← t⟩_n nid' n') t n'))
  using step.IH by simp

interpret ut: untrusted devExt t ⟨nid ← t⟩_n nid' n' λn. umining t (n' + n)
proof
  fix n''
  from devExt-devop[of t ⟨nid ← t⟩_n nid' n']
    have prefix (devExt t ⟨nid ← t⟩_n nid' n' (Suc n''))
    (devExt t ⟨nid ← t⟩_n nid' n' n'') ∨
    (∃ b. devExt t ⟨nid ← t⟩_n nid' n' (Suc n'') =
    devExt t ⟨nid ← t⟩_n nid' n' n'' @ [b]) ∧
    ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n'')) ∧
    ⌇the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))⌇_t (n' + Suc n'') ∧
    n' + Suc n'' ≤ ⟨nid ← t⟩_n ∧
    mining (σ_the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n'')) t (n' + Suc n'')) .
    thus prefix (devExt t ⟨nid ← t⟩_n nid' n' (Suc n''))
    (devExt t ⟨nid ← t⟩_n nid' n' n'') ∨
    (∃ b. devExt t ⟨nid ← t⟩_n nid' n' (Suc n'') =
    devExt t ⟨nid ← t⟩_n nid' n' n'' @ [b]) ∧ umining t (n' + Suc n'')
  proof
    assume prefix (devExt t ⟨nid ← t⟩_n nid' n' (Suc n''))
      (devExt t ⟨nid ← t⟩_n nid' n' n'')
    thus ?thesis by simp
  next
    assume (∃ b. devExt t ⟨nid ← t⟩_n nid' n' (Suc n'') =
        devExt t ⟨nid ← t⟩_n nid' n' n'' @ [b]) ∧
      ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n'')) ∧
      ⌇the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))⌇_t (n' + Suc n'') ∧
      n' + Suc n'' ≤ ⟨nid ← t⟩_n ∧
      mining (σ_the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n'')) t (n' + Suc n''))
    hence ∃ b. devExt t ⟨nid ← t⟩_n nid' n' (Suc n'') =
        devExt t ⟨nid ← t⟩_n nid' n' n'' @ [b]
      and ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))
      and ⌇the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))⌇_t (n' + Suc n'')
```

```
        and n' + Suc n'' ≤ ⟨nid ← t⟩_n and
          mining (σ_the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))^t
          (n' + Suc n'')) by auto
      moreover from ‹n' + Suc n'' ≤ ⟨nid ← t⟩_n›
        ‹¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))›
        have ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n'')))
        using untrusted by simp
      with ‹⦃the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))⦄_t (n' + Suc n'')›
      have the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))∈actUt (t (n' + Suc n''))
        using actUt-def by simp
      ultimately show ?thesis using umining-def by auto
    qed
  qed
  from ‹¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n')› have
    bc (σ_the (devBC t ⟨nid ← t⟩_n nid' n')^t n') = devExt t ⟨nid ← t⟩_n nid' n' 0
    using devExt-bc-geq[of t ⟨nid ← t⟩_n nid' n'] by simp
  moreover from ‹n'≤⟨nid ← t⟩_n› ‹⦃nid⦄_t ⟨nid ← t⟩_n› have
    bc (σ_nid'^t ⟨nid ← t⟩_n) = devExt t ⟨nid ← t⟩_n nid' n' (⟨nid ← t⟩_n−n')
    using devExt-bc-geq by simp
  with ‹¬ prefix sbc (bc (σ_nid'(t ⟨nid ← t⟩_n)))› have
    ¬ prefix sbc (devExt t ⟨nid ← t⟩_n nid' n' (⟨nid ← t⟩_n−n')) by simp
  ultimately have ∃ n'''. n''' ≤ ⟨nid ← t⟩_n−n' ∧
    length (devExt t ⟨nid ← t⟩_n nid' n' n''') < length sbc
    using ‹prefix sbc (bc (σ_the (devBC t ⟨nid ← t⟩_n nid' n')(t n')))›
    ut.prefix-length[of sbc 0 ⟨nid ← t⟩_n−n'] by auto
  then obtain n_p where n_p ≤ ⟨nid ← t⟩_n−n'
    and length (devExt t ⟨nid ← t⟩_n nid' n' n_p) < length sbc by auto
  hence length (devExt t ⟨nid ← t⟩_n nid' (n' + n_p) 0) < length sbc
    using devExt-shift[of t ⟨nid ← t⟩_n nid' n' n_p] by simp
  moreover from ‹⟨nid ← t⟩_n≥n'› ‹n_p ≤ ⟨nid ← t⟩_n−n'›
    have (n' + n_p) ≤ ⟨nid ← t⟩_n by simp
  ultimately show ?thesis using ‹n'≥n_s› untrusted by auto
next
  assume ¬(∃ n'≤⟨nid ← t⟩_n. n'≥n_s ∧
    ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n') ∧
    trusted (the (devBC t ⟨nid ← t⟩_n nid' n')))
  hence cas: ∀ n'≤⟨nid ← t⟩_n. n'≥n_s ∧
    ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n')
    ⟶ ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n')) by auto
  show ?thesis
  proof cases
    assume Option.is-none (devBC t ⟨nid ← t⟩_n nid' n_s)
    thus ?thesis
    proof cases
      assume ∀ n'<n_s. Option.is-none (devBC t ⟨nid ← t⟩_n nid' n')
      with ‹Option.is-none (devBC t ⟨nid ← t⟩_n nid' n_s)›
        have devExt t ⟨nid ← t⟩_n nid' n_s 0 = [] by simp
      with ‹¬ sbc=[]› have
```

```
    length (devExt t ⟨nid ← t⟩_n nid' n_s 0) < length sbc by simp
  moreover from lAct have ⟨nid ← t⟩_n≥n_s using latestActless by blast
  moreover from cas have
  ∀ n''>n_s. n''≤⟨nid ← t⟩_n ∧ ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n'')
    ⟶ ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n'')) by simp
  ultimately show ?thesis by auto
next
  let ?P=λn'. n'<n_s ∧ ¬Option.is-none (devBC t ⟨nid ← t⟩_n nid' n')
  let ?n'=GREATEST n'. ?P n'
  assume ¬ (∀ n'<n_s. Option.is-none (devBC t ⟨nid ← t⟩_n nid' n'))
  moreover have ∀ n'>n_s. ¬ ?P n' by simp
  ultimately have exists: ∃ n'. ?P n' ∧ (∀ n''. ?P n''⟶ n''≤n')
    using boundedGreatest[of ?P] by blast
  hence ?P ?n' using GreatestI-ex-nat[of ?P] by auto
  moreover from ‹?P ?n'› have ⌇the (devBC t ⟨nid ← t⟩_n nid' ?n')⌇_t ?n'
    using devBC-act by simp
  ultimately have
    length (bc (σ_the (devBC t ⟨nid ← t⟩_n nid' ?n')^t ?n')) < length sbc ∨
    prefix sbc (bc (σ_the (devBC t ⟨nid ← t⟩_n nid' ?n')(t ?n')))
    using assms(4) by simp
  thus ?thesis
  proof
    assume length (bc (σ_the (devBC t ⟨nid ← t⟩_n nid' ?n')^t ?n')) < length sbc
    moreover from exists have ¬(∃ n'>?n'. ?P n')
      using Greatest-ex-le-nat[of ?P] by simp
    moreover from ‹?P ?n'› have
      ∃ n'<n_s. ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n') by blast
    with ‹Option.is-none (devBC t ⟨nid ← t⟩_n nid' n_s)›
      have devExt t ⟨nid ← t⟩_n nid' n_s 0 =
      bc (σ_the (devBC t ⟨nid ← t⟩_n nid' ?n')(t ?n')) by simp
    ultimately have length (devExt t ⟨nid ← t⟩_n nid' n_s 0) < length sbc
      by simp
    moreover from lAct have ⟨nid ← t⟩_n≥n_s using latestActless by blast
    moreover from cas have ∀ n''>n_s. n''≤⟨nid ← t⟩_n ∧
      ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n'') ⟶
      ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n'')) by simp
    ultimately show ?thesis by auto
  next
  interpret ut: untrusted devExt t ⟨nid ← t⟩_n nid' n_s λn. umining t (n_s + n)
    proof
      fix n''
      from devExt-devop[of t ⟨nid ← t⟩_n nid' n_s]
        have prefix (devExt t ⟨nid ← t⟩_n nid' n_s (Suc n''))
        (devExt t ⟨nid ← t⟩_n nid' n_s n'') ∨
        (∃ b. devExt t ⟨nid ← t⟩_n nid' n_s (Suc n'') =
          devExt t ⟨nid ← t⟩_n nid' n_s n'' @ [b]) ∧
        ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' (n_s + Suc n'')) ∧
        ⌇the (devBC t ⟨nid ← t⟩_n nid' (n_s + Suc n''))⌇_t (n_s + Suc n'') ∧
```

$n_s + \mathit{Suc}\ n'' \leq \langle \mathit{nid} \leftarrow t\rangle_n \wedge$
$\mathit{mining}\ (\sigma_{\mathit{the}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ (n_s + \mathit{Suc}\ n''))}{}^{t\ (n_s + \mathit{Suc}\ n'')})$ **.**
thus $\mathit{prefix}\ (\mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ (\mathit{Suc}\ n''))$
$(\mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ n'') \vee$
$(\exists\, b.\ \mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ (\mathit{Suc}\ n'') =$
$\mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ n''\ @\ [b]) \wedge \mathit{umining}\ t\ (n_s + \mathit{Suc}\ n'')$
proof
assume $\mathit{prefix}\ (\mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ (\mathit{Suc}\ n''))$
$(\mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ n'')$
thus *?thesis* **by** *simp*
next
assume $(\exists\, b.\ \mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ (\mathit{Suc}\ n'') =$
$\mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ n''\ @\ [b]) \wedge$
$\neg\ \mathit{Option.is\text{-}none}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ (n_s + \mathit{Suc}\ n'')) \wedge$
$\wr \mathit{the}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ (n_s + \mathit{Suc}\ n''))\wr_{t\ (n_s + \mathit{Suc}\ n'')} \wedge$
$n_s + \mathit{Suc}\ n'' \leq \langle \mathit{nid} \leftarrow t\rangle_n \wedge$
$\mathit{mining}\ (\sigma_{\mathit{the}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ (n_s + \mathit{Suc}\ n''))}{}^{t\ (n_s + \mathit{Suc}\ n'')})$
hence $\exists\, b.\ \mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ (\mathit{Suc}\ n'') =$
$\mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ n''\ @\ [b]$
and $\neg\ \mathit{Option.is\text{-}none}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ (n_s + \mathit{Suc}\ n''))$
and $\wr \mathit{the}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ (n_s + \mathit{Suc}\ n''))\wr_{t\ (n_s + \mathit{Suc}\ n'')}$
and $n_s + \mathit{Suc}\ n'' \leq \langle \mathit{nid} \leftarrow t\rangle_n$
and $\mathit{mining}\ (\sigma_{\mathit{the}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ (n_s + \mathit{Suc}\ n''))}{}^{t}$
$(n_s + \mathit{Suc}\ n''))$ **by** *auto*
moreover from $\langle n_s + \mathit{Suc}\ n'' \leq \langle \mathit{nid} \leftarrow t\rangle_n\rangle$
$\langle\neg\ \mathit{Option.is\text{-}none}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ (n_s + \mathit{Suc}\ n''))\rangle$
have $\neg\ \mathit{trusted}\ (\mathit{the}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ (n_s + \mathit{Suc}\ n'')))$
using *cas* **by** *simp*
with $\langle\wr \mathit{the}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ (n_s + \mathit{Suc}\ n''))\wr_{t\ (n_s + \mathit{Suc}\ n'')}\rangle$
have $\mathit{the}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ (n_s + \mathit{Suc}\ n'')) \in$
$\mathit{actUt}\ (t\ (n_s + \mathit{Suc}\ n''))$
using *actUt-def* **by** *simp*
ultimately show *?thesis* **using** *umining-def* **by** *auto*
qed
qed

assume $\mathit{prefix}\ \mathit{sbc}\ (\mathit{bc}\ (\sigma_{\mathit{the}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ ?n')}{}^{(t\ ?n')}))$
moreover from *exists* **have** $\neg(\exists\, n'{>}?n'.\ ?P\ n')$
using *Greatest-ex-le-nat*[*of ?P*] **by** *simp*
moreover from $\langle ?P\ ?n'\rangle$ **have**
$\exists\, n'{<}n_s.\ \neg\ \mathit{Option.is\text{-}none}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n')$ **by** *blast*
with $\langle \mathit{Option.is\text{-}none}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s)\rangle$ **have**
$\mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ 0 =$
$\mathit{bc}\ (\sigma_{\mathit{the}\ (\mathit{devBC}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ ?n')}{}^{(t\ ?n')})$
by *simp*
ultimately have $\mathit{prefix}\ \mathit{sbc}\ (\mathit{devExt}\ t\ \langle \mathit{nid} \leftarrow t\rangle_n\ \mathit{nid}'\ n_s\ 0)$ **by** *simp*
moreover from *lAct* **have** $\langle \mathit{nid} \leftarrow t\rangle_n \geq n_s$ **using** *latestActless* **by** *blast*
with $\langle\wr \mathit{nid} \wr_{t\ \langle \mathit{nid} \leftarrow t\rangle_n}\rangle$ **have**

```
          bc (σ_{the (devBC t ⟨nid ← t⟩_n nid' ⟨nid ← t⟩_n)}^{t ⟨nid ← t⟩_n}) =
            devExt t ⟨nid ← t⟩_n nid' n_s (⟨nid ← t⟩_n−n_s)
          using devExt-bc-geq by simp
        with ‹¬ prefix sbc (bc (σ_{nid'}(t ⟨nid ← t⟩_n)))› ‹⦃nid'⦄_{t ⟨nid ← t⟩_n}›
          have ¬ prefix sbc (devExt t ⟨nid ← t⟩_n nid' n_s (⟨nid ← t⟩_n−n_s))
          by simp
        ultimately have ∃n'''>0. n''' ≤ ⟨nid ← t⟩_n−n_s ∧
          length (devExt t ⟨nid ← t⟩_n nid' n_s n''') < length sbc
          using ut.prefix-length[of sbc 0 ⟨nid ← t⟩_n−n_s] by simp
        then obtain n_p where n_p>0 and n_p ≤ ⟨nid ← t⟩_n−n_s and
          length (devExt t ⟨nid ← t⟩_n nid' n_s n_p) < length sbc by auto
        hence length (devExt t ⟨nid ← t⟩_n nid' (n_s + n_p) 0) < length sbc
          using devExt-shift by simp
        moreover from lAct have ⟨nid ← t⟩_n≥n_s using latestActless by blast
        with ‹n_p ≤ ⟨nid ← t⟩_n−n_s› have (n_s + n_p) ≤ ⟨nid ← t⟩_n by simp
        moreover from ‹n_p ≤ ⟨nid ← t⟩_n−n_s› have n_p ≤ ⟨nid ← t⟩_n by simp
        moreover have ∀n''>n_s + n_p. n'' ≤ ⟨nid ← t⟩_n ∧
          ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n'') ⟶
          ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n'')) using cas by simp
        ultimately show ?thesis by auto
      qed
    qed
  next
    assume asmp: ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n_s)
    moreover from lAct have n_s≤⟨nid ← t⟩_n using latestActless by blast
    ultimately have ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n_s))
      using cas by simp
    moreover from asmp have ⦃the (devBC t ⟨nid ← t⟩_n nid' n_s)⦄_{t n_s}
      using devBC-act by simp
    ultimately have the (devBC t ⟨nid ← t⟩_n nid' n_s)∈actUt (t n_s)
      using actUt-def by simp
    hence length (bc (σ_{the (devBC t ⟨nid ← t⟩_n nid' n_s)}^{(t n_s)})) < length sbc
      using assms(2) by simp
    moreover from asmp have
      devExt t ⟨nid ← t⟩_n nid' n_s 0 =
        bc (σ_{the (devBC t ⟨nid ← t⟩_n nid' n_s)}^{(t n_s)})
      by simp
    ultimately have length (devExt t ⟨nid ← t⟩_n nid' n_s 0) < length sbc by simp
    moreover from lAct have ⟨nid ← t⟩_n≥n_s using latestActless by blast
    moreover from cas have ∀n''>n_s. n''≤⟨nid ← t⟩_n ∧
      ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n'') ⟶
      ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n'')) by simp
    ultimately show ?thesis by auto
  qed
qed
then obtain n' where ⟨nid ← t⟩_n≥n' and n'≥n_s
  and length (devExt t ⟨nid ← t⟩_n nid' n' 0) < length sbc
  and untrusted: ∀n''>n'. n''≤⟨nid ← t⟩_n ∧
```

```
      ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' n'') ⟶
      ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' n'')) by auto
  interpret ut: untrusted devExt t ⟨nid ← t⟩_n nid' n' λn. umining t (n' + n)
  proof
    fix n''
    from devExt-devop[of t ⟨nid ← t⟩_n nid' n']
    have prefix (devExt t ⟨nid ← t⟩_n nid' n' (Suc n''))
        (devExt t ⟨nid ← t⟩_n nid' n' n'') ∨
      (∃ b. devExt t ⟨nid ← t⟩_n nid' n' (Suc n'') =
        devExt t ⟨nid ← t⟩_n nid' n' n'' @ [b]) ∧
      ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n'')) ∧
      ⌇the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))⌇_t (n' + Suc n'') ∧
      n' + Suc n'' ≤ ⟨nid ← t⟩_n ∧
      mining (σ_the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n'')) t (n' + Suc n'')) .
    thus prefix (devExt t ⟨nid ← t⟩_n nid' n' (Suc n''))
      (devExt t ⟨nid ← t⟩_n nid' n' n'')
      ∨ (∃ b. devExt t ⟨nid ← t⟩_n nid' n' (Suc n'') =
      devExt t ⟨nid ← t⟩_n nid' n' n'' @ [b]) ∧ umining t (n' + Suc n'')
    proof
      assume prefix (devExt t ⟨nid ← t⟩_n nid' n' (Suc n''))
        (devExt t ⟨nid ← t⟩_n nid' n' n'')
      thus ?thesis by simp
    next
      assume (∃ b. devExt t ⟨nid ← t⟩_n nid' n' (Suc n'') =
        devExt t ⟨nid ← t⟩_n nid' n' n'' @ [b]) ∧
        ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n'')) ∧
        ⌇the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))⌇_t (n' + Suc n'') ∧
        n' + Suc n'' ≤ ⟨nid ← t⟩_n ∧
        mining (σ_the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n'')) t (n' + Suc n''))
      hence ∃ b. devExt t ⟨nid ← t⟩_n nid' n' (Suc n'') =
          devExt t ⟨nid ← t⟩_n nid' n' n'' @ [b]
        and ¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))
        and ⌇the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))⌇_t (n' + Suc n'')
        and n' + Suc n'' ≤ ⟨nid ← t⟩_n
        and mining (σ_the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n'')) t (n' + Suc n''))
        by auto
      moreover from ‹n' + Suc n'' ≤ ⟨nid ← t⟩_n›
        ‹¬ Option.is-none (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))›
        have ¬ trusted (the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n'')))
        using untrusted by simp
      with ‹⌇the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))⌇_t (n' + Suc n'')›
        have the (devBC t ⟨nid ← t⟩_n nid' (n' + Suc n''))∈actUt (t (n' + Suc n''))
        using actUt-def by simp
      ultimately show ?thesis using umining-def by auto
    qed
  qed
interpret untrusted-growth devLgthBC t ⟨nid ← t⟩_n nid' n' λn. umining t (n' + n)
    by unfold-locales
```

```
interpret trusted-growth λn. PoW t (n' + n) λn. tmining t (n' + n)
proof
  show ⋀n. PoW t (n' + n) ≤ PoW t (n' + Suc n) using pow-mono by simp
  show ⋀n. tmining t (n' + Suc n) ⟹ PoW t (n' + n) < PoW t (n' + Suc n)
    using pow-mining-suc by simp
qed
interpret bg: bounded-growth
  length sbc
  λn. PoW t (n' + n)
  devLgthBC t ⟨nid ← t⟩_n nid' n'
  λn. tmining t (n' + n)
  λn. umining t (n' + n)
  length sbc cb
proof
  from assms(3) ‹n'≥n_s› show length sbc + cb ≤ PoW t (n' + 0)
    using pow-mono[of n_s n' t] by simp
next
  from ‹length (devExt t ⟨nid ← t⟩_n nid' n' 0) < length sbc›
    show length (devExt t ⟨nid ← t⟩_n nid' n' 0) < length sbc .
next
  fix n'' n'''
  assume cb < card {i. n'' < i ∧ i ≤ n''' ∧ umining t (n' + i)}
  hence cb < card {i. n'' + n' < i ∧ i ≤ n''' + n' ∧ umining t i}
    using cardshift[of n'' n''' umining t n'] by simp
  with fair[of n'' + n' n''' + n' t]
  have cb < card {i. n'' + n' < i ∧ i ≤ n''' + n' ∧ tmining t i} by simp
  thus cb < card {i. n'' < i ∧ i ≤ n''' ∧ tmining t (n' + i)}
    using cardshift[of n'' n''' tmining t n'] by simp
qed
from ‹⟨nid ← t⟩_n≥n'› have
  length (devExt t ⟨nid ← t⟩_n nid' n' (⟨nid ← t⟩_n−n')) < PoW t ⟨nid ← t⟩_n
  using bg.tr-upper-bound[of ⟨nid ← t⟩_n−n'] by simp
moreover from ‹⦃nid'⦄_t ⟨nid ← t⟩_n› ‹⟨nid ← t⟩_n≥n'›
have bc (σ_the (devBC t ⟨nid ← t⟩_n nid' ⟨nid ← t⟩_n)^t ⟨nid ← t⟩_n) =
  devExt t ⟨nid ← t⟩_n nid' n' (⟨nid ← t⟩_n−n')
  using devExt-bc-geq[of t ⟨nid ← t⟩_n nid' ⟨nid ← t⟩_n n'] by simp
ultimately have length (bc (σ_nid'(t ⟨nid ← t⟩_n))) < PoW t ⟨nid ← t⟩_n
  using ‹⦃nid'⦄_t ⟨nid ← t⟩_n› by simp
moreover have
  PoW t ⟨nid ← t⟩_n ≤ length (bc (σ_nid'(t ⟨nid ← t⟩_n))) (is ?lhs ≤ ?rhs)
proof −
  from ‹trusted nid› ‹⦃nid⦄_t ⟨nid ← t⟩_n›
  have ?lhs ≤ length (MAX (pin (σ_nid t ⟨nid ← t⟩_n))) using pow-le-max by simp
  also from ‹bc (σ_nid'(t ⟨nid ← t⟩_n)) = MAX (pin (σ_nid t ⟨nid ← t⟩_n))›
    have ... = length (bc (σ_nid'(t ⟨nid ← t⟩_n))) by simp
  finally show ?thesis .
qed
ultimately show False by simp
qed
```

```
            qed
            moreover from ‹⦃nid⦄_{t n}› have ⟨nid → t⟩_n=n using nxtAct-active by simp
            ultimately show ?thesis by auto
          qed
          moreover from ‹⦃nid⦄_{t n}› have ⟨nid → t⟩_n=n using nxtAct-active by simp
          ultimately show ?thesis by auto
        next
          assume ¬ (∃b∈pin (σ_nid t ⟨nid ← t⟩_n). length b > length (bc (σ_nid t ⟨nid ← t⟩_n)))
          moreover from ‹⦃nid⦄_{t n}› have ∃n'≥n. ⦃nid⦄_{t n'} by auto
          moreover from lAct have ∃n'. latestAct-cond nid t n n' by auto
          ultimately have ¬ mining (σ_nid t ⟨nid → t⟩_n) ∧
            bc (σ_nid t ⟨nid → t⟩_n) = bc (σ_nid t ⟨nid ← t⟩_n) ∨
            mining (σ_nid t ⟨nid → t⟩_n) ∧
            (∃b. bc (σ_nid t ⟨nid → t⟩_n) = bc (σ_nid t ⟨nid ← t⟩_n) @ [b])
            using ‹trusted nid› bhv-tr-in[of nid n t] by simp
          moreover have prefix sbc (bc (σ_nid t ⟨nid ← t⟩_n))
          proof -
            from ‹∃n'. latestAct-cond nid t n n'› have ⟨nid ← t⟩_n < n
              using latestAct-prop(2) by simp
            moreover from lAct have ⟨nid ← t⟩_n ≥ n_s using latestActless by blast
            moreover from ‹∃n'. latestAct-cond nid t n n'› have ⦃nid⦄_{t ⟨nid ← t⟩_n}
              using latestAct-prop(1) by simp
            with ‹trusted nid› have nid ∈ actTr (t ⟨nid ← t⟩_n) using actTr-def by simp
            ultimately show ?thesis using step.IH by auto
          qed
          moreover from ‹⦃nid⦄_{t n}› have ⟨nid → t⟩_n=n using nxtAct-active by simp
          ultimately show ?thesis by auto
        qed
      next
        assume nAct: ¬ (∃n' < n. n' ≥ n_s ∧ ⦃nid⦄_{t n'})
        moreover from step.hyps have n_s ≤ n by simp
        ultimately have ⟨nid → t⟩_{n_s} = n using ‹⦃nid⦄_{t n}› nxtAct-eq[of n_s n nid t] by simp
        with ‹trusted nid› show ?thesis using assms(1) by auto
      qed
    qed
  qed
  with assms(5) show ?thesis by simp
qed

end

end
```

Bibliography

[ADG98] Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer Berlin Heidelberg, 1998.

[All97] Robert J Allen. A formal approach to software architecture. Technical report, DTIC Document, 1997.

[AM02a] Nazareno Aguirre and Tom Maibaum. Reasoning about reconfigurable object-based systems in a temporal logic setting. In *Proceedings of IDPT*, 2002.

[AM02b] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In *Automated Software Engineering*, pages 271–274. IEEE, 2002.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science*, 14(03):329–366, 2004.

[Bal04] Clemens Ballarin. Locales and locale expressions in isabelle/isar. *Lecture notes in computer science*, 3085:34–50, 2004.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media, 2013.

[BCK07] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Longman Publishing Co., Inc., 2007.

[Ber96] Klaus Bergner. *Spezifikation großer Objektgeflechte mit Komponentendiagrammen.* PhD thesis, Technische Universität München, 1996.

[BFGea93] Manfred Broy, Christian Facchi, Radu Grosu, and et al. The requirement and design specification language spectrum – an informal introduction. Technical report, Technische Universität München, 1993.

[BGM16] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, volume 9604 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2016.

[BHL+14] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co) datatypes for isabelle/hol. In *International Conference on Interactive Theorem Proving*, pages 93–110. Springer, 2014.

[BK86] Jan A Bergstra and Jan Willem Klop. Algebra of communicating processes. *CWI Monograph series*, 3:89–138, 1986.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley West Sussex, England, 1996.

[Bro96] Manfred Broy. Algebraic specification of reactive systems. In *Algebraic Methodology and Software Technology*, pages 487–503. Springer, Springer Berlin Heidelberg, 1996.

[Bro10] Manfred Broy. A logical basis for component-oriented software and systems engineering. *The Computer Journal*, 53(10):1758–1782, February 2010.

[Bro14] Manfred Broy. A model of dynamic systems. In Saddek Bensalem, Yassine Lakhneck, and Axel Legay, editors, *From Programs to Systems. The Systems Perspective in Computing*, volume 8415 of *Lecture Notes in Computer Science*, pages 39–53. Springer Berlin Heidelberg, 2014.

[BS01] Manfred Broy and Ketil Stolen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Springer Science & Business Media, 2001.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Science of computer programming*, 61(2):75–113, 2006.

[CAPM10] Pablo F Castro, Nazareno M Aguirre, Carlos Gustavo López Pombo, and Thomas SE Maibaum. Towards managing dynamic reconfiguration of software systems in a categorical setting. In *Lecture Notes in Computer Science*, pages 306–321. Springer, 2010.

[CCGR00] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CCS12] Carlos Canal, Javier Cámara, and Gwen Salaün. Structural reconfiguration of systems under behavioral adaptation. *Science of Computer Programming*, 78(1):46 – 64, 2012. Special Section: Formal Aspects of Component Software (FACS'09).

[Cha89] K Mani Chandy. *Parallel program design*. Springer, 1989.

[DVdHT01] Eric M Dashofy, André Van der Hoek, and Richard N Taylor. A highly-extensible, xml-based architecture description language. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 103–112. IEEE, 2001.

[FLV06] Peter H Feiler, Bruce A Lewis, and Steve Vestal. The sae architecture analysis & design language (aadl) a standard for engineering performance critical systems. In *Computer Aided Control System Design, Control Applications, Intelligent Control*, pages 1206–1211. IEEE, 2006.

[FM97] JoséLuiz Fiadeiro and Tom Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28(2-3):111–138, 1997.

[FS97] D. Fensel and A. Schnogge. Using kiv to specify and verify architectures of knowledge-based systems. In *Automated Software Engineering*, pages 71–80, November 1997.

[Gar03] David Garlan. Formal modeling and analysis of software architecture: Components, connectors, and events. In *Formal Methods for Software Architectures*, pages 1–24. Springer, 2003.

[GH05] Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundam. Inform.*, 66:353–366, 2005.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software, 1994.

[GJS17] Thomas Göthel, Nils Jähnig, and Simon Seif. Refinement-based modelling and verification of design patterns for self-adaptive systems. In *International Conference on Formal Engineering Methods*, pages 157–173. Springer, 2017.

[GM18] Habtom Kahsay Gidey and Diego Marmsoler. FACTUM Studio. `https://habtom.github.io/factum/`, 2018.

[GMW00] David Garlan, Robert T Monroe, and David Wile. Acme: Architectural description of component-based systems. *Foundations of component-based systems*, 68:47–68, 2000.

[GR91] Michael M. Gorlick and Rami R. Razouk. Using weaves for software construction and analysis. In Les Belady, David R. Barstow, and Koji Torii, editors, *Proceedings of the 13th International Conference on Software Engineering, Austin, TX, USA, May 13-17, 1991.*, pages 23–34. IEEE Computer Society / ACM Press, 1991.

[GRABR14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and Andrew W Roscoe. Fdr3—a modern refinement checker for csp. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201. Springer, 2014.

[HF10] Florian Hölzl and Martin Feilkas. Autofocus 3: A scientific tool prototype for model-based development of component-based, reactive, distributed systems. In *Proceedings of the 2007 International Dagstuhl Conference on Model-based Engineering of Embedded Real-time Systems*, MBEERTS'07, pages 317–322, Berlin, Heidelberg, 2010. Springer-Verlag.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Jac02] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

[KG06] Jung Soo Kim and David Garlan. Analyzing architectural styles with alloy. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 70–80. ACM, 2006.

[KK99] Mark Klein and Rick Kazman. Attribute-based architectural styles. Technical Report CMU/SEI-99-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1999.

[KMLA11] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Modeling dynamic reconfigurations in reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23 – 36, 2011. Selected papers from the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures.

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[LKA$^+$95] David C Luckham, John J Kenney, Larry M Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *Software Engineering, IEEE Transactions on*, 21(4):336–354, 1995.

[LMP10] Francois Laroussinie, Antoine Meyer, and Eudes Petonnet. Counting LTL. In *2010 17th International Symposium on Temporal Representation and Reasoning*. IEEE, sep 2010.

[Loc10] Andreas Lochbihler. Coinduction. *The Archive of Formal Proof s. http://afp.sourceforge.net/entries/Coinductive.shtml*, 2010.

[LS13] Yi Li and Meng Sun. Modeling and analysis of component connectors in coq. In José Luiz Fiadeiro, Zhiming Liu, and Jinyun Xue, editors, *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*, volume 8348 of *Lecture Notes in Computer Science*, pages 273–290. Springer, 2013.

[Mar10] Diego Marmsoler. Applying the scientific method in the definition and analysis of a new architectural style. Master's thesis, Free University of Bolzano-Bozen, 2010.

[Mar14] Diego Marmsoler. Towards a theory of architectural styles. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 823–825. ACM, ACM Press, 2014.

[Mar17a] Diego Marmsoler. Dynamic architectures. *Archive of Formal Proofs*, July 2017. `http://isa-afp.org/entries/DynamicArchitectures.html`, Formal proof development.

[Mar17b] Diego Marmsoler. On the semantics of temporal specifications of component-behavior for dynamic architectures. In *Eleventh International Symposium on Theoretical Aspects of Software Engineering*. Springer, 2017.

[Mar17c] Diego Marmsoler. Towards a calculus for dynamic architectures. In Dang Van Hung and Deepak Kapur, editors, *Theoretical Aspects of Computing - ICTAC 2017 - 14th International Colloquium, Hanoi, Vietnam, October 23-27, 2017, Proceedings*, volume 10580 of *Lecture Notes in Computer Science*, pages 79–99. Springer, 2017.

[Mar18a] Diego Marmsoler. A framework for interactive verification of architectural design patterns in Isabelle/HOL. In *The 20th International Conference on Formal Engineering Methods, ICFEM 2018, Proceedings*, 2018.

[Mar18b] Diego Marmsoler. Hierarchical specication and verication of architecture design patterns. In *Fundamental Approaches to Software Engineering - 21th International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, 2018.

[Mar18c] Diego Marmsoler. A theory of architectural design patterns. *Archive of Formal Proofs*, March 2018. `http://isa-afp.org/entries/Architectural_Design_Patterns.html`, Formal proof development.

[Mar19] Diego Marmsoler. A calculus of component behavior for dynamic architectures. *Science of Computer Programming*, 2019. Under review.

[MCL04] Jeffrey KH Mak, Clifford ST Choy, and Daniel PK Lun. Precise modeling of design patterns in uml. In *Software Engineering*, pages 252–261. IEEE, 2004.

[MD17] Diego Marmsoler and Silvio Degenhardt. Verifying patterns of dynamic architectures using model checking. In *Proceedings International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA@ETAPS 2017, Uppsala, Sweden, 22nd April 2017.*, pages 16–30, 2017.

[MG16a] D. Marmsoler and M. Gleirscher. On activation, connection, and behavior in dynamic architectures. *Scientific Annals of Computer Science*, 26(2):187–248, 2016.

[MG16b] Diego Marmsoler and Mario Gleirscher. Specifying properties of dynamic architectures using configuration traces. In *International Colloquium on Theoretical Aspects of Computing*, pages 235–254. Springer, 2016.

[MG18] Diego Marmsoler and Habtom Kahsay Gidey. FACTUM Studio: A tool for the axiomatic specification and verification of architectural design patterns. In *Formal Aspects of Component Software - FACS 2018 - 15th International Conference, Proceedings*, 2018.

[MG19] Diego Marmsoler and Habtom Kahsay Gidey. Interactive verification of architectural design patterns in FACTUM. *Formal Aspects of Computing*, 2019. Under review.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge university press, 1999.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In David Garlan, editor, *SIGSOFT '96, Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, California, USA, October 16-18, 1996*, pages 3–14. ACM, 1996.

[MME15] Diego Marmsoler, Alexander Malkis, and Jonas Eckhardt. A model of layered architectures. In Bara Buhnova, Lucia Happe, and Jan Kofron, editors, *Proceedings 12th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2015, London, United Kingdom, April 12th, 2015.*, volume 178 of *EPTCS*, pages 47–61, 2015.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer New York, 1992.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`, 2008.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[Oqu04] Flavio Oquendo. π-adl: An architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, May 2004.

[otSEC$^+$00] Architecture Working Group of the Software Engineering Committee et al. Recommended practice for architectural description of software intensive systems. *IEEE Standards Department*, 2000.

[PW92] Dewayne E Perry and Alexander L Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[Rau01] Andreas Rausch. *Componentware.* Dissertation, Technische Universität München, München, 2001.

[Rei95] Wolfgang Reif. The kiv-approach to software verification. *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, pages 339–368, 1995.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the.* Pearson Higher Education, 2004.

[SBR12] Alejandro Sanchez, Luís Soares. Barbosa, and Daniel Riesco. Bigraphical modelling of architectural patterns. In Farhad Arbab and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software*, pages 313–330, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[SH04] Neelam Soundarajan and Jason O Hallstrom. Responsibilities and rewards: Specifying design patterns. In *Software Engineering*, pages 666–675. IEEE, 2004.

[SMB15] Alejandro Sanchez, Alexandre Madeira, and Luís S Barbosa. On the verification of architectural reconfigurations. *Computer Languages, Systems & Structures*, 44:218–237, 2015.

[Spi07] Maria Spichkova. *Specification and seamless verification of embedded real-time systems: FOCUS on Isabelle*. PhD thesis, Technical University Munich, Germany, 2007.

[TMD09] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[vOvdLKM00] Rob C. van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.

[W^{+}04] Makarius Wenzel et al. The isabelle/isar reference manual, 2004.

[Wen07] Makarius Wenzel. Isabelle/isar – a generic framework for human-readable proof documents. *From Insight to Proof – Festschrift in Honour of Andrzej Trybulec*, 10(23):277–298, 2007.

[WF02] Michel Wermelinger and José Luiz Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133 – 155, 2002. Special Issue on Applications of Graph Transformations (GRATRA 2000).

[Wir90] Martin Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B)*, pages 675–788. MIT Press, Cambridge, MA, USA, 1990.

[WLF01] Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. A graph based architectural reconfiguration language. *Software Engineering Notes*, 26(5):21–32, 2001.

[WSWS08] Stephen Wong, Jing Sun, Ian Warren, and Jun Sun. A scalable approach to multi-style architectural modeling and verification. In *Engineering of Complex Computer Systems*, pages 25–34. IEEE, 2008.

[ZA05] Uwe Zdun and Paris Avgeriou. Modeling architectural patterns using architectural primitives. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 133–146. ACM, 2005.

[ZLS+12] Jiexin Zhang, Yang Liu, Jing Sun, Jin Song Dong, and Jun Sun. Model checking software architecture design. In *High-Assurance Systems Engineering*, pages 193–200. IEEE, 2012.

Glossary

architectural constraint constraints about different aspects of an architecture. 6

architectural design constraint constraints about different aspects of an architecture. 4

architectural design problem an architectural design problem and a set of architectural design constraints solving the problem. 4

architectural guarantee a property about an architecture. 6

architecture assertion logic formula with interface ports as free variables and predicates to denote component activation and connections between ports. 44

architecture snapshot a set of active components, connections between their ports, and valuations of the active component's ports. 23, 24

architecture specification set of architecture traces which does not restrict behavior. 27

architecture trace stream of architecture snapshots. 25

behavior assertion logic formula with ports as free variables. 41, 335

behavior projection operator to extract the behavior of a certain component c out of a architecture trace t. 27

behavior trace stream of port valuations over a set of ports P. 19, 41

behavior trace assertion temporal logic formula over behavior assertions to specify behavior traces. 41

Blackboard pattern used for collaborative problem solving. 5

component activation number of activations of a component c within a certain architecture trace t up to time point n. 64

component port a port used by a component. 22

component port valuation port valuation for component ports. 22

component type a component interface with a set of total execution traces for a component. 19, 20

interface set of input and output ports. 18

message primitive entity which can be used and exchanged by components. 17

parametrized component type a component type with a valuated parameter port. 20, 21

port means by which components can exchange messages. 18

port valuation assignment of a set of messages to a set of ports P. 18, 335

Publisher-Subscriber pattern to support flexible communication between components of an architecture. 4

Singleton pattern used to restrict the number of active components in an architecture. 4