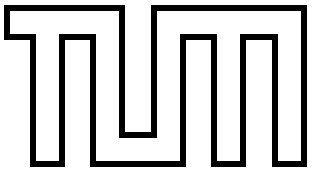
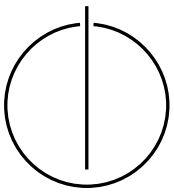


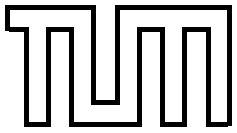
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN
Lehrstuhl für Informatik VII



CAVA – A Verified Model Checker

René Neumann



FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl für Informatik VII
Grundlagen der Softwarezuverlässigkeit und Theoretische Informatik

CAVA – A Verified Model Checker

René Neumann

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. Dr. Helmut Seidl

Prüfer der Dissertation:

1. Prof. Dr. Dr. h.c. Francisco Javier Esparza Estaun
2. Prof. Tobias Nipkow, Ph.D.

Die Dissertation wurde am 19. 01. 2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 08. 05. 2017 angenommen.

Abstract

Model checkers like SPIN provide a way to gain certainty about the behavior of programs and protocols. To guarantee that the model checker itself is correct, the CAVA project developed a verified and executable LTL model checker using Isabelle/HOL.

This thesis reports on various building blocks of CAVA by presenting the first formalized and executable Promela semantics, a framework for verifying depth-first search based algorithms, and an automata library. As a part thereof, this thesis will detail the development rationale and history of those building blocks.

Kurzzusammenfassung

Modelchecker, wie z.B. SPIN, sind eine Möglichkeit, Gewissheit über das Verhalten von Programmen und Protokollen zu erlangen. Um nun sicherzustellen, dass der Modelchecker selber korrekt ist, wurde im Projekt CAVA ein verifizierter und ausführbarer Modelchecker mit Hilfe von Isabelle/HOL entwickelt.

In dieser Dissertation stellen wir verschiedene Bausteine von CAVA vor: Die erste formalisierte und ausführbare Semantik von Promela, ein Framework um auf Tiefensuche basierende Algorithmen zu verifizieren, sowie eine Automatenbibliothek. Ein Augenmerk liegt dabei auf der Geschichte und den Hintergründen der jeweiligen Implementierung.

Acknowledgements

My thanks go first and foremost to Prof. Esparza and Prof. Nipkow for allowing me to work on this very interesting topic and trying to bridge the world between Interactive Theorem Proving and Model Checking. This work would also not have been possible without the important input of Peter Lammich, Thomas Tuerk and Alexander Schimpf.

I also want to thank Michael, Maximilian, Stefan, and André for the vital mental support and expertise throughout all the years.

This work was funded by the DFG as part of the CAVA project, for which I'm grateful.

Contents

1 Introduction

Nearly every hand-written software in the ecosystem suffers from bugs. This is almost inevitable when the software is geared towards high performance and therefore uses highly complicated algorithms and data structures which are not easily provable. In Software Engineering this is, in general, countered by various testing techniques. But testing can inherently only prove the presence of errors, not their absence.

Different approaches exist to respond to this shortcoming and prove the absence of errors and the adherence to some specification. The most thorough one is to mathematically prove correctness and soundness of a program. While this is used in different projects like C compilers [32] or the seL4 operating system kernel [22], in general the effort for a thorough proof of correctness is too high for most projects. For that matter, other approaches exist that serve as a middle-way between mathematical proof and plain testing. One such measure is model checking [55], where an implementation or protocol is checked for adherence to some specification (or, conversely, ensuring a certain condition does not hold). The downside to a manual proof is the possible non-termination, i. e., the checker may not find a result. The upside is the automatic way of using it: Given that a program representation exists which is understood by the model checker, there is no further effort needed short of formalizing the specification / error-conditions.

Applied as such, a model checker is in the role of a trust-multiplier. Hence, their verdict must not be wrong. Now the recursion begins – or as [52] puts it: "*Quis custodiet ipsos custodes?*" – "Who will watch the watchmen?". That is, how can we ensure that the software used for model checking is indeed correct, as to avoid the dangerous multiplicative effect bugs in such a system would have?

Different approaches to tackle this problem exist (an overview is given in [14]). The CAVA project[1], which we were part of, advertises a pragmatic solution to that matter: It introduced a formally verified and executable LTL model checker for finite-state systems that can serve as a *reference implementation*. That is, while not necessarily as efficient as most products on the market, those products can be tested against the reference implementation to gain confidence in their implementation. Formalized using the interactive theorem prover Isabelle/HOL, we gained both correctness in a mathematical sense but also executable code.

The finite-state LTL model checker presented in CAVA follows the well-known automata-theoretic approach [55]: Given a finite-state program P and a formula ϕ, two Büchi automata are constructed that recognize the executions of P, and all potential executions of P that violate ϕ, respectively. Then the product of the two automata is computed and tested on-the-fly, that is during the product construction, for emptiness (see Fig. 1.1). This approach is also followed by SPIN [17], which probably is the best-known model checker. Therefore, we modelled CAVA to be largely compatible to SPIN in order to facilitate using

[1] `http://cava.in.tum.de`

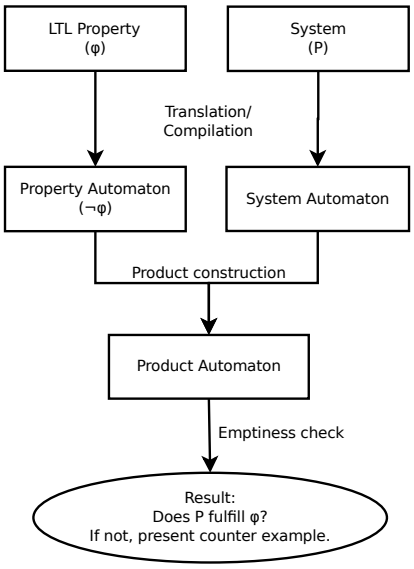


Figure 1.1: Model Checker

CAVA directly without larger conversions.

We presented a first version of that model checker in [10] with a later addition of Promela, the language for program description in SPIN, in [37]. Brunner and Lammich then extended CAVA with Partial Order Reduction in [4]. Further publications of ours as part of the CAVA project deal with the generalization of depth-first search in [35, 29]. Most of the papers are also accompanied with the Isabelle developments. Those are published in the *Archive of Formal Proofs* ([11, 36, 30]) or available for download [1].

In this thesis, we will highlight different parts of the development of CAVA in the interactive theorem prover Isabelle/HOL. Thus, we structure the thesis as follows: In Chapter 2 we start with an introduction to Isabelle/HOL and to Refinement, a proof/developing technique that will be used throughout most parts of the thesis. With Chapter 3 we lay out the automata-theoretic foundations of the model-checker. This is followed by Chapter 4, where we will introduce a generic framework for formalizing depth-first searches in Isabelle/HOL. We will show the usage of that framework for different use cases. One use case will be the application inside our model checker for testing the emptiness of the product language of system and property. Chapter 5 will describe modelling input programs, namely a proof-of-concept language for Boolean Programs, and the input language of SPIN: Promela. All of those parts are then assembled in Chapter 6 to receive a final model checker[2]. Finally, we will conclude in Chapter 7.

One particular focus of this work, especially in Chapters 3 and 4, will be to analyze competing strategies for formalization. That is, show different approaches that we have taken during the creation process of the model checker and show the different pros and cons.

[2]The translation of an input formula in LTL to the corresponding automaton is not covered in this thesis. Please refer to Schimpf [49] for details on that topic.

2 Prerequisites

2.1 Isabelle/HOL

Isabelle/HOL [39] is an interactive theorem prover, based on Higher-Order Logic (HOL). It allows to declare data structures, functions, and properties about those in a style similar to functional programming languages. More importantly, Isabelle/HOL allows to show the mathematical correctness of those properties. In contrast to automatic theorem provers, the user/developer guides Isabelle/HOL interactively to develop a final proof.

This thesis, to a large amount, consists of developments done in Isabelle/HOL. Therefore, this section shall allow a reader not familiar with the software and its syntax to read and understand the snippets in the rest of the thesis. Of course, it is not feasible to cover all possible concepts of Isabelle used in this thesis. We thus sometimes rely on the understanding from the context, or that the point made by the snippet should be comprehensible without knowing every keyword.

A good introduction into the language and the prover system can be found in the book *Concrete Semantics* by Nipkow and Klein [38].

2.1.1 Basic Notation

Similar to other functional languages, function application is written *f a b* instead of *f*(*a*,*b*). Also, lambda terms with the standard syntax $\lambda x.\ t$ (i. e., the function mapping x to t) are employed.

In general, types are not explicitly expressed but are inferred. When explicitly assigning a type τ to some term t, we write $t :: \tau$. Types can either be a type variable written *'a*, some otherwise declared type (e. g., the well-known *bool*, *int*, *nat*, ...), functions *'a* $\Rightarrow$ *'b*, or compound types. The latter ones are written in a postfix notation, thus *'a list* denotes a list of some type *'a*. Other basic compound types are sets (*'a set*) and tuples (*'a* $\times$ *'b*).

On basic datatypes we have well known syntactic sugar like + on *int* and *nat*, or $\cup$ on sets. On lists, Isabelle introduces *x#xs* to denote prepending the element x to the list *xs*. Also, it denotes appending to lists *xs* and *ys* by *xs@ys*.

2.1.2 Defining new Types

Isabelle/HOL supports various ways of defining new types. We will only show the ones used throughout the thesis.

Synonyms for existing types, i. e., syntactic sugar only, are defined with

type_synonym (*'k,'v*) *map* = *'k* $\Rightarrow$ *'v option*

Here, we define a new type (*'k,'v*) *map* that is identical to the function from *'k* to optional *'v*. The type *'v option* can either be *None* to denote an empty result, or *Some v*.

Inductive datatypes are introduced using the ***datatype*** construction, listing each possible constructor:

datatype *'a option = None | Some 'a*

The construction can be recursive:

datatype *nat = Zero | Suc nat*

A third way, which we heavily rely on, is the definition of compound structures, called ***record*** in Isabelle. Having defined a new record type *'a point* for two-dimensional coordinates of any kind as

record *'a point =*
  x :: 'a
  y :: 'a

the fields are accessed via accessor functions named similar to the fields, i. e., *x p* to get the x-coordinate of some point *p*. An instance of that record is constructed with the following syntax:

$(\!|\, x = 0, y = 0 \,|\!)$

Updates[1] are written similar:

$p(\!|\, y := 5 \,|\!)$

Records can also be extended. So, for instance, we can add a z-coordinate to the point definition above, yielding a type *'a coord*:

record *'a coord = 'a point +*
  z :: 'a

Instances of type *'a coord* are now also of type *'a point*[2].

2.1.3 Definitions

In Isabelle/HOL, there are, again, multiple ways of introducing new functions. We will show the ones used in this thesis.

The most simple variant is an abbreviation of some construct. Similar to type synonyms, they do not serve as a construct in their own right but are only used for pretty-printing:

abbreviation *f_of_x ≡ f x*

Full-fledged new definitions are given similarly:

definition *first_of_list xs ≡ xs!0*

Or more verbose:

definition *first_of_list :: 'a list ⇒ 'a* ***where***
  first_of_list xs ≡ xs!0

[1] As we are in the functional world, this is not an in-place update but returns an updated instance.
[2] Strictly speaking, they are of type *('a,'more) point_scheme*, but we will ignore this for the thesis.

In this thesis, the semantical differences of definition and abbreviation will not show. They are only used side-by-side to copy their usage in the theories and for those readers knowing Isabelle.

Similarly, in definitions we will use $\equiv$ and = interchangebly. That is, the definition above might also be written

definition *first_of_list xs = xs!0*

A third variant for definitions is also used sparingly: ***fun*** allows for recursion and pattern-matching on arguments:

fun *sum :: nat ⇒ nat ⇒ nat* ***where***
sum Zero b = b
| *sum (Suc a) b = sum a (Suc b)*

2.1.4 Lemmas and Proving Them

Lemmas are defined as follows:

lemma *zero_less_or_equal:*
$0 \leq (x::nat)$

When assumptions are needed, they can be supplied in different forms:

lemma *sum_greater:*
$x > 0 \Longrightarrow y > 0 \Longrightarrow x + y > 0$

lemma *sum_greater:*
$[\![x > 0;\ y > 0]\!] \Longrightarrow x + y > 0$

lemma *sum_greater:*
assumes *x > 0* ***and*** *y > 0*
shows *x + y > 0*

Further, there exist some synonyms for ***lemma***, for instance ***theorem*** and ***corollary***. They only serve as a semantical hint to the human reader and do not have any other effect.

This thesis will only very seldomly contain proofs of lemmas, they are to be found in the referenced Isabelle theory files instead. But sometimes they are given as an illustration. A proof can be very short, consisting of one or two calls to proof-procedures:

lemma *sum_greater:*
$x > 0 \Longrightarrow y > 0 \Longrightarrow x + y > 0$
by *simp*

lemma *foo:*
(∗ some more complicated property ∗)
by *simp blast*

But only a fraction of all properties can be shown in such an easy way. For most of them, a manual proof is needed:

```
lemma bar:
  (* something very elaborate *)
proof −
  (* here follows the manual proof *)
qed
```

We will spare the details of how such a proof looks like and will refer to Isabelle documentation [9].

2.1.5 Local Context

Quite often a collection of properties shares a common set of assumptions and/or definitions (e. g., an invariant on a data structure). It is possible to repeat them for each lemma, but this clutters the core message of that lemma. For this reason, Isabelle allows local contexts, called ***locale***, that fix those assumptions and definitions:

```
locale example =
  fixes ds :: data_structure
  assumes valid ds
begin
  lemma some_property:
    some_property ds
  proof −
    (* in the proof, the fact that ds is valid may be used freely *)
  qed
end
```

For presentation, we do not use the form above. Instead we use a semantically equivalent variant, where we separate the declaration of the locale from the definitions there-in:

```
locale example =
  fixes ds :: data_structure
  assumes valid ds

lemma (in example) some_property:
  some_property ds
```

Similar to records, locales can build a hierarchy. Sublocales hereby inherit lemmas and definitions of their parent(s). For example, we might define a locale for general graphs:

```
locale graph =
  fixes E :: ('v × 'v) set

definition (in graph) V ≡ Range E ∪ Domain E
```

On this we base finite graphs, introducing additional assumptions that allow additional properties to be shown:

```
locale fin_graph = graph E
  for E :: ('v × 'v) set +
  assumes finite E
```

lemma (***in*** *fin_graph*) *finite_V:*
finite V

This hiearchy can also be built after the fact: When using explicit inheritance like above, the assumptions from the parent locales are added to the one being defined. But when the child locale already has them given/shown otherwise, duplication would occur. Therefore one can omit the explicit inheritance and show the sublocale status afterwards. For instance automata might not be constructed in terms of graphs, but it can be shown that an automata also can be seen as one:

locale *automaton* = ...

sublocale (***in*** *automaton*) *fin_graph*
proof −
...
qed

Eventually, after having finished a locale, one can *interpret* a locale, possibly with concrete parameters. Without this, the properties and definitions of the locale would not be accessible outside of the locale. For instance

interpretation *gen!: fin_graph E* ***for*** *E*

would make all lemmas and definitions of *fin_graph* accessible with the prefix *gen* but adding a parameter *E* and the implicit assumptions of the locale. On the other hand, given some concrete graph *G* one could do:

interpretation *conc!: fin_graph G*
proof −
(∗ *show that G meets all assumptions* ∗)
qed

Now all lemmas and properties would (also) be accessible with the *conc* prefix, but having been fixed for *G*. This, albeit with slightly different syntax, is often used in proofs to make facts accessible for some concrete instance of that locale which is created in the process of the proof.

2.1.6 Deviation from the Theories

In this work, we wanted to be true to the Isabelle theories. But sometimes we had to deviate from them in order to fit something for the writing or to omit certain technicalities that would distract from the point in question. The deviation also includes slight changes to the syntax, leaving out some punctuation or rearranging arguments for better readability.

One larger change from the theories is using one locale, where the theories may employ two or more: In Isabelle it is not possible to use abbreviations or definitions in the assumptions of the locale that are only valid inside the locale. For that reason, we often had a locale holding the definitions and a second one adding the properties. This approach unfortunately bloats the inheritance hierarchy and complicates understanding. That is why we haven chosen to ignore this limitation in writing.

A second point is using explicit sets of edges E and initial nodes V_0 to represent graphs. In the theories they are often, but not always, passed around as a structure. To improve homogeneity, we have not followed this path here.

2.2 Refinement

When developing formally verified algorithms, there is a trade-off between the efficiency of the algorithm and the efficiency of the proof: For complex algorithms, a direct proof of an efficient implementation tends to get unmanageable, as proving implementation details blows up the proof and obfuscates the main ideas of the proof. A standard approach to this problem is stepwise refinement [2, 3], where this problem is solved by modularization of the correctness proof: One starts with an abstract version of the algorithm and then refines it (in possibly many steps) to the concrete, efficient version. A refinement step may reduce the non-determinism of a program and replace abstract datatypes by their implementations. For example, selecting an arbitrary element from a set may be refined to getting the head of a list. The main point is, that correctness properties can be transferred over refinements, such that correctness of the concrete program easily follows from correctness of the abstract algorithm and correctness of the refinement steps. The abstract algorithm is not cluttered with implementation details, such that its correctness proof can focus on the main algorithmic ideas. Moreover, the refinement proofs only focus on the local changes in a particular refinement step, not caring about the overall correctness property.

In Isabelle, this approach is supported by the Isabelle Refinement and Collections Frameworks [31, 28], and the Autoref tool [24]. Using ideas of refinement calculus [3], the Isabelle Refinement Framework provides a set of tools to concisely express non-deterministic programs, reason about their correctness, and refine them (in possibly many steps) towards efficient implementations. The Isabelle Collections Framework provides a library of verified efficient data structures for standard types such as sets and maps. Finally, the Autoref tool automates the refinement to efficient implementations, based on user-adjustable heuristics for selecting suitable data structures to implement the abstract types.

In the following, we describe the basics of the Isabelle Refinement Framework. Given the general result type *'a* of an algorithm, it is encapsulated to express the possible non-determinism of that algorithm in the type *'a nres*:

datatype *'a nres = RES 'a set | FAIL*

A result *RES X* expresses that the program returns one of the values of *X* non-deterministically, while *FAIL* expresses failure of an assertion.

On results, we define an ordering by lifting the subset ordering, *FAIL* being the greatest element.

$$RES\ X \leq RES\ Y \textit{ iff } X \subseteq Y \mid m \leq FAIL \mid FAIL \not\leq RES\ X$$

Note that this ordering forms a complete lattice, where $RES\ \{\}$ is the bottom, and *FAIL* is the top element. The intuitive meaning of $m \leq m'$ is that all possible values of m are also possible for m'. We say that *m refines m'*. In order to describe that all values in m satisfy a condition Φ, we write $m \leq \textbf{\textit{spec}}\ x.\ \Phi\ x$ (or shorter: $m \leq \textbf{\textit{spec}}\ \Phi$), where $\textbf{\textit{spec}}\ x.\ \Phi\ x \equiv RES\ \{x.\ \Phi\ x\}$.

Example 2.2.1
Let *cyc_checker E* V_0 be an algorithm that checks a graph over edges E and start nodes V_0 for cyclicity. Its correctness is described by the following formula, stating that it should return *true* if and only if the graph contains a cycle reachable from V_0, which is expressed by the predicate *cyclic*:

cyc_checker E $V_0 \leq$ ***spec*** *r. r = cyclic E* V_0

Now let *cyc_checker_impl* be an efficient implementation of *cyc_checker*. For refinement, we have to prove:

cyc_checker_impl E $V_0 \leq$ *cyc_checker E* V_0.

Note that, by transitivity, we also get that the implementation is correct:

cyc_checker_impl E $V_0 \leq$ ***spec*** *r. r = cyclic E* V_0

To express non-deterministic algorithms, the Isabelle Refinement Framework uses a monad [56] over non-deterministic results. It is defined by the two functions ***return*** and ***bind***:

return $x \equiv RES\ \{x\}$
bind $FAIL\ f \equiv FAIL \mid$ ***bind*** $(RES\ X)\ f \equiv RES\ (\bigcup x{\in}X.\ f\ x)$

Intuitively, ***return*** x returns the deterministic outcome x, and ***bind*** $m\ f$ is a sequential composition, which describes the result of non-deterministically choosing a value from m and applying f to it. In this thesis, we write $x \leftarrow m; f\ x$ instead of ***bind*** $m\ f$, to make program text more readable.

Another useful construct are assertions:

assert $\Phi \equiv$ ***if*** Φ ***then return*** $()$ ***else*** *FAIL*

An assertion generates an additional proof obligation when proving a program correct. However, when refining the program, the condition of the assertion can be assumed.

Example 2.2.2
The following program removes an arbitrary element from a non-empty set. It returns the element and the new set.

definition *select s* ≡ ***do*** {
  assert $s \neq \{\}$;
  $x \leftarrow$ ***spec*** *x.* $x{\in}s$;
  return $(x, s - \{x\})$}

The assertion in the first line expresses the precondition that the set is not empty. If the set is empty, the result of the program is *FAIL*. The second line non-deterministically selects an element from the set, and the last line assembles the result: A pair of the element and the new set.

Using the verification condition generator of the Isabelle Refinement Framework, it is straightforward to prove the following lemma, which states that the program refines the specification of the correct result:

lemma *select_correct:*
  $s \neq \{\} \implies$ *select* $s \leq$ ***spec*** $(x,s').\ \{x \in s \wedge s' = s - \{x\}\}$
  unfolding *select_def* ***by*** *refine_vcg auto*

Recursion is described by a least fixed point:

rec *x B* ≡ ***do*** {***assert*** (*mono B*); *lfp B x*}

Based on recursion, the Isabelle Refinement Framework provides ***while*** and ***foreach*** loops. Note that we agree on a partial correctness semantics in this thesis[3], i. e., infinite executions do not contribute to the result of a recursion.

Typically, a refinement also changes the representation of data, e. g., a set of successor nodes may be implemented by a list. Such a *data refinement* is described by a relation $\mathcal{R}$ between concrete and abstract values. We define a *concretization function* $\Downarrow\mathcal{R}$, that maps an abstract result to a concrete result:

$\Downarrow\mathcal{R}\ FAIL \equiv FAIL$
$\mid\ \Downarrow\mathcal{R}\ (RES\ X) \equiv \{c.\ \exists x{\in}X.\ (c,x) \in \mathcal{R}\}$

Intuitively, $\Downarrow\mathcal{R}\ m$ contains all concrete values with an abstract counterpart in m.

Example 2.2.3
A finite set can be implemented by a duplicate-free list of its elements. This is described by the following relation:

definition *ls_rel* ≡ {(*l*,*s*). *set l* = *s* ∧ *distinct l*}

The *select*-function from Example 2.2.2 can be implemented on lists as follows:

definition $select^i\ l \equiv$ ***do*** {
assert $l \neq []$;
x = *hd l*;
return (*x*, *tl l*)}

Again, it is straightforward to show that $select^i$ refines *select*:

lemma *select_refine:*
$(l,s) \in ls\_rel \Longrightarrow select^i\ l \leq \Downarrow(Id{\times}ls\_rel)\ (select\ s)$
unfolding $select^i$*_def select_def*
by (*refine_vcg*) (*auto simp: ls_rel_def neq_Nil_conv*)

Intuitively, this lemma states that, given the list l is an implementation of set s, the results of *select* and $select^i$ are related by $Id{\times}ls\_rel$, i. e., the first elements are equal, and the second elements are related by *ls_rel*. When refining functions, we often use a more concise syntax to express the refinement relation:

$(select^i, select) \in ls\_rel \rightarrow (Id \times ls\_rel)$

Note that the assertion in the abstract *select*-function is crucial for this proof to work: For the empty set, we have *select* {} = *FAIL*, and the statement holds trivially. Thus, we may assume that $s{\neq}\{\}$, which implies $l{\neq}[]$, which, in turn, is required to reason about *hd l* and *tl l*. The handling of assertions is automated by the verification condition generator, which inserts the assumptions and discharges the trivial goals.

[3]The Isabelle Refinement Framework supports both, partial and total correctness semantics. However, the code generator of Isabelle/HOL currently only guarantees partial correctness of the generated code.

3 Constructing the Search Space

A foundational part of a model checker is the representation of the state space of the program to check. While, in general, a Kripke structure is sufficient to represent the state space, more advanced structures like different forms of automata are chosen. This is done because the further operations of the model checker build upon other automata-theoretic approaches. Namely, representing the property as an automaton and using the product construction on property and system automaton to yield a resulting automaton. This automaton is then accepting exactly those runs in the system that fulfill the property.

Due to this automata-theoretic background, CAVA, our model checker, contains also libraries modelling this background. In this chapter, we want to give an overview about the evolution of our automata formalization in CAVA. The libraries represented were not developed as parts of this thesis (except for some work in the first library (Section 3.1) on the product construction on Büchi automata and the integration of the framework into CAVA). Instead, we show them here as they form an integral part of the model checker and also as a show-case for the different development models for a theory. Furthermore, the first library is not documented anywhere else; Section 3.1 can now be used as documentation in that regard.

3.1 The Comprehensive Library

Originally, the model checker has not been the main formalization goal. Instead, a comprehensive automata-theoretic library was envisioned and the model checker should serve as some non-trivial example usage of that library. For this reason, the library that we will describe in this section is a comprehensive one. That is, it formalizes more than is needed inside our model checker, and also does so in a more general way. In this section, we will show the formalization of this library, with a focus on the parts relevant for the model checker. In a later section (3.3), we will discuss how this approach is inferior to a specialized variant as shown in the next Section 3.2.

This library has been designed and implemented by Tuerk and Malik. The ω-automata were largely designed by Schimpf and Neumann.

3.1.1 Labeled Transition System

The library starts with the general notion of a labeled transition system (LTS), as a relation of states and label to states:

type_synonym $('q,'l)\ LTS = ('q \times 'l \times 'q)\ set$

Here, $'q$ denotes the type of states and $'l$ the type of the labels.

Furthermore we make use of the library on infinite words by Stephan Merz, which is part of [50]. Of interest for this section is the datatype for infinite words:

type_synonym *'l word = nat ⇒ 'l*

From this definition follows, that for some word w the expression $w\ i$ returns the letter at the ith position. For finite words, we use the basic datatype *'l list*, where the ith letter (given that $i <$ *length w*) is accessed with *w ! i*.

General terms, like finite and infinite runs, or reachability, are defined on our base structure of labeled transition systems (the LTS is denoted by Δ in the following):

definition *LTS_is_fin_run :: ('q,'l) LTS ⇒ 'l list ⇒ 'q list ⇒ bool* ***where***
LTS_is_fin_run Δ w r ⟷
length r = Suc (length w) ∧
∀i < length w. (r ! i, w ! i, r ! (Suc i)) ∈ Δ

definition *LTS_is_inf_run :: ('q,'l) LTS ⇒ 'l word ⇒ 'q word ⇒ bool* ***where***
LTS_is_inf_run Δ w r ⟷ ∀i. (r i, w i, r (Suc i)) ∈ Δ

fun *LTS_is_reachable :: ('q, 'l) LTS ⇒ 'q ⇒ 'l list ⇒ 'q ⇒ bool* ***where***
LTS_is_reachable Δ q [] q' ⟷ q = q'
| *LTS_is_reachable Δ q (σ # w) q' ⟷*
∃q''. (q, σ, q'') ∈ Δ ∧ LTS_is_reachable Δ q'' w q'

Of course, this includes proving properties about those definitions, like showing that each suffix of an infinite run is again an infinite run

lemma *LTS_is_inf_run___suffix :*
LTS_is_inf_run Δ w r
⟹ LTS_is_inf_run Δ (suffix k w) (suffix k r)

or linking reachability to finite runs:

lemma *LTS_is_reachable_alt_def :*
LTS_is_reachable Δ q w q' ⟷
∃r. LTS_is_fin_run Δ w r ∧ hd r = q ∧ last r = q'

Further concepts introduced at this level are determinism and completeness:

definition *LTS_is_deterministic :: ('q, 'l) LTS ⇒ bool* ***where***
LTS_is_deterministic Δ ⟷
∀q σ q1' q2'. ((q, σ, q1') ∈ Δ ∧ (q, σ, q2') ∈ Δ) ⟶ q1' = q2'

definition *LTS_is_complete :: 'q set ⇒ 'l set ⇒ ('q, 'l) LTS ⇒ bool* ***where***
LTS_is_complete Q Σ Δ ⟷ ∀q∈Q. ∀σ∈Σ. ∃q'∈Q. (q, σ, q') ∈ Δ

As deterministic transition systems occur often as a basis for automata, they are often used if the determinism is part of the type (i. e., not explicitly added via an additional assumption). Thus, an additional type *DLTS* is introduced, together with conversion functions. It is to note that here a transition function is used instead of a relation, as this spares the requirement for an additional invariant of determinism. By chosing two different types, instead of one plus an invariant, implementations may later be optimized by allowing different implementations depending on the type:

type_synonym *('q,'l) DLTS = 'q × 'l ⇒ 'q option*

definition *DLTS_to_LTS :: ('q, 'l) DLTS ⇒ ('q, 'l) LTS*
where *DLTS_to_LTS δ ≡ {(q, σ, q') | q σ q'. δ (q, σ) = Some q'}*

definition *LTS_to_DLTS :: ('q, 'l) LTS ⇒ ('q, 'l) DLTS **where***
*LTS_to_DLTS Δ ≡ λ(q,σ). **if** (∃q'. (q,σ,q') ∈ Δ) **then** Some (ε q'. (q, σ, q') ∈ Δ) **else** None*

Here, $\varepsilon\ x.\ P\ x$ is the choice operator, returning some x, such that $P\ x$ holds.

It is shown that, when a *DLTS* is converted into the regular LTS, it is a determinstic one:

lemma *DLTS_to_LTS___LTS_is_deterministic:*
LTS_is_deterministic (DLTS_to_LTS δ)

Similarly, types for complete LTS (*CLTS*) and deterministic and complete LTS (*CDLTS*) are introduced. But we will omit them here.

Finally, multiple operations are defined for such transition systems. The one of most importance for this work is the definition of a product:

definition *LTS_product :: ('q₁,'l) LTS ⇒ ('q₂,'l) LTS ⇒ ('q₁ × 'q₂,'l) LTS **where***
LTS_product Δ₁ Δ₂ = {((q₁,q₂), σ, (q₁',q₂')) | q₁ q₁' σ q₂ q₂'.
(q₁, σ, q₁') ∈ Δ₁ ∧ (q₂, σ, q₂') ∈ Δ₂}

Again, this definition is accompanied by useful properties like showing that determinism is preserved:

lemma *LTS_product_LTS_is_deterministic :*
⟦ LTS_is_deterministic Δ₁; LTS_is_deterministic Δ₂ ⟧ ⟹
LTS_is_deterministic (LTS_product Δ₁ Δ₂)

3.1.2 Semi-Automaton

The next step is to define the notion of a *Semi-Automaton*, which is the common denominator for finite automata and ω-automata. Compared to the bare LTS, we now add an explicit set of states and an alphabet. Furthermore, a set of states is declared *initial*:

record *('q,'l) SemiAutomaton =*
𝒬 :: 'q set (* set of states *)
Σ :: 'l set (* set of labels *)
Δ :: ('q,'l) LTS (* transition relation *)
ℐ :: 'q set (* set of initial states *)

Please note that no set of final states or similar is given in this definition. This is because different kinds of automata define different means to encode acceptance, which therefore cannot be modelled in a general and useful way.

Besides the data represented by the record, a semi-automaton needs to adhere to additional wellformedness constraints (Δ is defined in terms of $\mathcal{Q}$ and Σ, $\mathcal{I}$ is a subset of $\mathcal{Q}$). Isabelle/HOL allows to define types with an inherent invariant. But for practical reasons (extensibility, constructing intermediate invalid structures) this is not used. Instead, the invariants are encapsulated into their own locale. Recall from Section 2.1 that this mechanism is used to create environments that fix certain assumptions and variables:

locale *SemiAutomaton* =
 fixes $\mathcal{A}$::('q,'l) *SemiAutomaton*
 assumes $(q,\sigma,q') \in \Delta\ \mathcal{A} \Longrightarrow (q \in \mathcal{Q}\ \mathcal{A}) \wedge (\sigma \in \Sigma\ \mathcal{A}) \wedge (q' \in \mathcal{Q}\ \mathcal{A})$
  and $\mathcal{I}\ \mathcal{A} \subseteq \mathcal{Q}\ \mathcal{A}$

Later on, it is sufficient to show that some instance $\mathcal{A}$ of the record above fulfills the assumptions of the locale to use all the properties specified therein:

 show *SemiAutomaton* $\mathcal{A}$

This is then extended to also provide for finite semi-automata:

locale *FinSemiAutomaton* = *SemiAutomaton* $\mathcal{A}$
 for $\mathcal{A}$::('q,'l) *SemiAutomaton* +
 assumes *finite* $(\mathcal{Q}\ \mathcal{A})$
  and *finite* $(\Delta\ \mathcal{A})$

Basic terminologies like finite and infinite runs are lifted from the their LTS-counterparts, adding the requirements of semi-automata, that is, requiring that a run starts in an initial node:

definition *SemiAutomaton_is_fin_run* ***where***
 SemiAutomaton_is_fin_run $\mathcal{A}$ *w r* $\longleftrightarrow$
 hd r $\in \mathcal{I}\ \mathcal{A} \wedge$ *LTS_is_fin_run* $(\Delta\ \mathcal{A})$ *w r*

definition *SemiAutomaton_is_inf_run* ***where***
 SemiAutomaton_is_inf_run $\mathcal{A}$ *w r* $\longleftrightarrow$
 r 0 $\in \mathcal{I}\ \mathcal{A} \wedge$ *LTS_is_inf_run* $(\Delta\ \mathcal{A})$ *w r*

It is then continued with showing properties of (finite) semi-automata like

lemma (***in*** *FinSemiAutomaton*) *finite_*$\mathcal{I}$:
 finite $(\mathcal{I}\ \mathcal{A})$

lemma (***in*** *SemiAutomaton*) *SemiAutomaton_*Δ*_cons___is_inf_run* :
  assumes *SemiAutomaton_is_inf_run* $\mathcal{A}$ *w r*
  shows *w i* $\in (\Sigma\ \mathcal{A})$ ***and*** *r i* $\in (\mathcal{Q}\ \mathcal{A})$

The purpose of this semi-automaton formalization, as laid out before, is to give a foundation for various automaton variants. For this reason, the formalization now defines a type for deterministic semi-automata and shows certain constructions and special properties (like $\mathcal{Q}\ \mathcal{A} \neq \{\}$) for it. We will omit that part here.

We will further omit certain formalized operations like removing unreachable states, or constructing a semi-automaton from some specified list representation. Both are for example used in Tuerk's formalization of Hopcroft's Algorithm [31] (the latter for benchmarking), but are not important for model checking.

The next interesting operation then is the product construction on semi-automata, which is defined using the product operation on LTS:

definition *product_SemiAutomaton* ::
 ('q_1, 'l) *SemiAutomaton* $\Rightarrow$
 ('q_2, 'l) *SemiAutomaton* $\Rightarrow$ ('$q_1 \times$ 'q_2, 'l) *SemiAutomaton* ***where***

product_SemiAutomaton $\mathcal{A}_1$ $\mathcal{A}_2 \equiv (\!|$
$\mathcal{Q} = \mathcal{Q}\ \mathcal{A}_1 \times \mathcal{Q}\ \mathcal{A}_2$,
$\Sigma = \Sigma\ \mathcal{A}_1 \cap \Sigma\ \mathcal{A}_2$,
$\Delta =$ *LTS_product* $(\Delta\ \mathcal{A}_1)\ (\Delta\ \mathcal{A}_2)$,
$\mathcal{I} = \mathcal{I}\ \mathcal{A}_1 \times \mathcal{I}\ \mathcal{A}_2 |\!)$

This, of course, is combined with appropriate lemmas, like:

lemma *product_SemiAutomaton___is_well_formed :*
$[\![$*SemiAutomaton* $\mathcal{A}_1$; *SemiAutomaton* $\mathcal{A}_2]\!]$
$\Longrightarrow$ *SemiAutomaton* (*product_SemiAutomaton* $\mathcal{A}_1$ $\mathcal{A}_2$)

lemma *product_SemiAutomaton_is_fin_run :*
SemiAutomaton_is_fin_run (*product_SemiAutomaton* $\mathcal{A}_1$ $\mathcal{A}_2$) w $r \longleftrightarrow$
SemiAutomaton_is_fin_run $\mathcal{A}_1$ w (*map fst r*) $\wedge$
SemiAutomaton_is_fin_run $\mathcal{A}_2$ w (*map snd r*)

Again, the formalization here specifies some more operations like the powerset construction, which we will omit here.

3.1.3 NFA and DFA

Based upon the (final) semi-automaton, both deterministic finite automata (DFA) and non-deterministic finite automata (NFA) are defined. An NFA is a semi-automaton with an additional set of final states, so far missing from the semi-automaton:

record *('q,'l) NFA = ('q, 'l) SemiAutomaton +*
$\mathcal{F}$ *:: 'q set* (∗ *set of final states* ∗)

locale *NFA = FinSemiAutomaton* $\mathcal{A}$
for $\mathcal{A}$*::('q,'l) NFA +*
assumes $\mathcal{F}\ \mathcal{A} \subseteq \mathcal{Q}\ \mathcal{A}$
and *finite* $(\Sigma\ \mathcal{A})$

Based on this definition of final states, the regular acceptance definition is specified for NFAs:

definition *NFA_accept* $\mathcal{A}$ $w \equiv (\exists\ q{\in}(\mathcal{I}\ \mathcal{A}).\ \exists\ q'{\in}(\mathcal{F}\ \mathcal{A}).$ *LTS_is_reachable* $(\Delta\ \mathcal{A})$ q w $q')$

This can then be used to define the language of such an automaton:

definition $\mathcal{L}\ \mathcal{A} \equiv \{w.$ *NFA_accept* $\mathcal{A}$ $w\}$

Furthermore, the now well-known operations on semi-automata are also lifted to the world of the NFA; we will omit this here as they are very straightforward. Of course, this also includes a product operation (*NFA_product*). Having defined the language in a former step, we can also show that the language of the product is as expected:

lemma *NFA_product_accept :*
$[\![$ *NFA* $\mathcal{A}_1$; *NFA* $\mathcal{A}_2$ $]\!]$
$\Longrightarrow$ *NFA_accept* (*NFA_product* $\mathcal{A}_1$ $\mathcal{A}_2$) w
$\longleftrightarrow$ *NFA_accept* $\mathcal{A}_1$ $w \wedge$ *NFA_accept* $\mathcal{A}_2$ w

lemma *NFA_product_$\mathcal{L}$:*
$[\![NFA\ \mathcal{A}_1; NFA\ \mathcal{A}_2]\!] \Longrightarrow \mathcal{L}\ (NFA\_product\ \mathcal{A}_1\ \mathcal{A}_2) = \mathcal{L}\ \mathcal{A}_1 \cap \mathcal{L}\ \mathcal{A}_2$

The NFA is then extended into a DFA by mixing in the deterministic semi-automaton, that was mentioned before:

locale *DFA = NFA $\mathcal{A}$ + DetFinSemiAutomaton $\mathcal{A}$*
for *$\mathcal{A}$::('q,'l) NFA*

It is to note, that here a DFA is not a new structure, but simply the NFA enriched with the determinism invariants. Then determinisation operations from NFA to DFA are introduced, together with operations that are possible on DFAs only, like complementing and minimizing. The latter is described in detail in Tuerk's paper on the Hopcroft-Algorithm [31]. Whenever an algorithm is more efficient on DFAs, its implementation will branch, depending on whether the automaton is deterministic.

We will not cover those details, but instead show how the formalization is refined into executable code.

3.1.4 Implementation

The goal of the original automata library was to have an executable library, that could be used by other programs, if needed.

Therefore the formalizations need to be refined to executable code. This, in effect, only needs to cover the NFA, as a semi-automaton is not going to be used by the user. Since the main component of an automaton, as described so far, is the underlying transition system, the LTS is refined on its own and then used by the NFA as an existing component.

The library, due to its age, uses the original version of the Isabelle Collections Framework [28] directly and not the Isabelle Refinement Framework [24] as an intermediary. In this original framework, the first step (when refining data structures, not necessarily plain algorithms), is to give a specification of said structure. This is not the formalization, but a set of functions that an implementation must have (i. e., an *interface* in Software Engineering terms).

The most important function of such a specification is the abstraction function (regularly denoted by the suffix _α) that converts an object of the implementation world into the datatype as used in the formalization. In case of the labeled transition system, this is the following function-type, where $'L^i$ is the implementation type:

type_synonym *('q,'l,$'L^i$) lts_α = $'L^i \Rightarrow$ ('q,'l) LTS*

Together with a possible invariant on the implementation this results in a locale representing some implementation of LTS:

locale *lts =*
fixes *α :: ('q,'l,$'L^i$) lts_α*
fixes *invar :: $'L^i \Rightarrow$ bool*

Further extensions like finiteness or determinism are gained by extending said locale, where the aforementioned additional invariant is assumed:

locale *finite_lts = lts +*
  assumes *invar l ⟹ finite (α l)*

locale *dlts = lts +*
  assumes *invar l ⟹ LTS_is_deterministic (α l)*

Operations on the structure are added piece by piece: for each operation a corresponding function on the implementation is defined, which is then encapsulated in a specific locale with corresponding assumptions[1]. For instance, the successor function is added like this:

type_synonym *('q,'l,'L^i) lts_succ = 'L^i ⇒ 'q ⇒ 'l ⇒ 'q option*
locale *lts_succ = lts +*
  fixes *succ :: ('q,'l,'L^i) lts_succ*
  assumes
    invar l ⟹ succ l v w = None ⟹ ∀v'. (v, w, v') ∉ (α l)
    invar l ⟹ succ l v w = Some v' ⟹ (v, w, v') ∈ (α l)

Here *lts_succ* is a function type, that when passed an instance of the LTS implementation, yields a successor function. In the same-named locale, an instance of the *lts_succ* function is fixed (i. e., theoretically, an implementation may have multiple variants of successor definitions).

In the same façon other operations like membership-testing, emptiness check, insertion etc. are defined.

For an easier usage, common operations are combined into one record-type, so that an LTS-implemention can be expressed by an instance of such type:

record *('q,'l,'L^i) lts_ops =*
  lts_op_α :: ('q,'l,'L^i) lts_α
  lts_op_invar :: 'L^i ⇒ bool
  lts_op_empty :: ('q,'l,'L^i) lts_empty
  lts_op_memb :: ('q,'l,'L^i) lts_memb
  lts_op_succ :: ('q,'l,'L^i) lts_succ
  ...

The record itself only contains the operations. This has two drawbacks: the correctness properties defined inside the appropriate locales are not contained, and using any implementation needs the implementation-record as a parameter, making the code cumbersome to read and write. For example, given that *lts* is an LTS and *L* is the operation, one would have to write *lts_op_succ L lts q a* to get the successors for node *q* and label *a*.

Therefore, an additional locale[2] is introduced: It takes the implementation as a parameter and then defines abbreviations on them, allowing direct usage of the functions. It also connects those functions to the appropriate correctness-locale:

locale *StdLTS =*
    finite_lts α invar +
    lts_empty α invar empty +

[1] This approach allows an implementation to only provide a subset of operations.
[2] For implementation reasons, in Isabelle proper, this is defined as two locales, as inheritance cannot use the later-defined abbreviations.

```
    lts_memb α invar memb +
    lts_succ α invar succ +
    ...
  fixes ops :: ('q,'l,'L^i) lts_ops
begin
  abbreviation α where α ≡ lts_op_α ops
  abbreviation invar where invar ≡ lts_op_invar ops
  abbreviation empty where empty ≡ lts_op_empty ops
  abbreviation memb where memb ≡ lts_op_memb ops
  abbreviation succ where succ ≡ lts_op_succ ops
  ...
end
```

A similar locale is created for deterministic LTS.

Tuerk provides multiple implementations of LTS, all of which are based on what he calls *TripleSet*s, a map of map of sets (those TripleSets are also introduced by Tuerk, but any details will be omitted here). The difference between those implementations is the order (starting node $\times$ label to set of resulting nodes; label $\times$ starting node to set of resulting nodes; starting node $\times$ resulting node to set of labels between them), as the use case might make one variant perform better than another. All the implementations are based on the same principle:

1. Define a new locale fixing possible parameters and sub-implementations. In the case presented this is the actual implementation of TripleSets:

   ```
   locale ltsbm_QAQ_defs =
    ts: triple_set ts_α ts_invar
    for ts_α::'ts ⇒ ('Q × 'A × 'Q) set
    and ts_invar
   ```

2. Define the basic constructs: An abstraction function and the general invariant on the concrete data structure:

   ```
   abbreviation (in ltsbm_QAQ_defs) ltsbm_α ≡ ts_α
   abbreviation (in ltsbm_QAQ_defs) ltsbm_invar ≡ ts_invar
   ```

   As the LTS implementation is a very shallow layer on top of the TripleSets, both the abstraction and the invariant do not define anything on their own, but are renames of the counter-parts in the TripleSet. So they are, in this instance, only defined out of convenience, because by convention every implementation is expected to provide both *_invar* and *_α*.

3. Define the basic operations as needed by *lts_ops* and show that they fulfill the necessary properties as defined in the respective locales. For the LTS implementations, Tuerk does not define the operations for they are identical to the operations on the underlying TripleSet. Thus he only proves that they are correct for the application of an LTS. In theory, additional abbreviations (like for *ltsbm_α*) could have been introduced. But in practice the defintions inside the locale are seldomly used and therefore not necessary.

```
lemma ltsbm_memb_correct:
  triple_set_memb ts_α ts_invar memb ⟹
    lts_memb ltsbm_α ltsbm_invar memb
  unfolding lts_memb_def triple_set_memb_def
  by simp

lemma ltsbm_add_correct:
  triple_set_add ts_α ts_invar add ⟹
    lts_add ltsbm_α ltsbm_invar add
 unfolding lts_add_def triple_set_add_def
 by simp
```

4. Usually, the final step is the definition of a particular instance of the ops record (*lts_ops*) and showing that the assumptions of the according locale (*StdLTS*) are matched. Tuerk omits this from the theories for the LTS implementations and instead only defines them at the stage prior to code generation when all decisions for the underlying data structures have been made. This is probably due to the number of parameters that can be passed to the implementations of TripleSets (two for the maps, one for the final result set). To follow the example of Tuerk, when using Red-Black-Trees for all those underlying data structures, the final step looks like the following[3]:

```
definition rs_lts_ops :: ('V,'E,('V,'E) rs_lts) lts_ops where
  rs_lts_ops ≡ (|
   lts_op_α = rs_lts_α,
   lts_op_invar = rs_lts_invar,
   lts_op_empty = rs_lts_empty,
   lts_op_memb = rs_lts_memb,
   ... |)

lemma rs_lts_impl: StdLTS rs_lts_ops
```

The implementation of NFAs (represented as a tuple) is then defined in terms of those LTS implementations:

```
type_synonym ('q_set, 'l_set, 'd) NFA_impl =
  'q_set × 'l_set × 'd × 'q_set × 'q_set

locale nfa_by_lts_defs =
 s!: StdSet s_ops (* Set operations on states *) +
 l!: StdSet l_ops (* Set operations on labels *) +
 d!: StdLTS d_ops (* An LTS *)
 for s_ops::('q,'q_set,_) set_ops
 and l_ops::('l,'l_set,_) set_ops
 and d_ops::('q,'l,'d,_) lts_ops
```

[3] The *rs_lts_* functions are explicit definitions for an instance of *ltsbm_QAQ_defs* with Red-Black-Trees. The old Collections Framework unfortunately needed a lot of boilerplate code and technical definitions being lifted from locales.

Due to the tuple structure, the composing parts of the automaton need additional extraction functions to remain readable:

definition (***in*** *nfa_by_lts_defs*) *nfa_states A* ≡ *fst A*
definition (***in*** *nfa_by_lts_defs*) *nfa_labels A* ≡ *fst* (*snd A*)
definition (***in*** *nfa_by_lts_defs*) *nfa_trans A* ≡ *fst* (*snd* (*snd A*))
...

which can then be used to define the straightforward abstraction function:

definition (***in*** *nfa_by_lts_defs*) *nfa_α* :: (*'q_set, 'l_set, 'd*) *NFA_impl* ⇒ (*'q, 'l*) *NFA* ***where***
nfa_α A =
(‖ *Q* = *s.α* (*nfa_states A*),
Σ = *l.α* (*nfa_labels A*),
Δ = *d.α* (*nfa_trans A*),
I = *s.α* (*nfa_initial A*),
F = *s.α* (*nfa_accepting A*) ‖)

While the LTS implementation itself was very short and in general only mapped to the underlying TripleSet, the NFA implementation is more involved (88 lines vs 3500 lines). This is due to the multitude of operations that are defined on the abstract NFA definition and now are replaced by efficient implementations. So, while for the LTS we had more or less a chaining of the underlying map/set structures, the implementations for the NFA are often very different from their abstract counterpart. Therefore the proofs are more complicated.

We will not go in any more detail for the NFA implementation. While, as mentioned, being more complicated, the general idea is like the one given for the LTS.

Eventually, Tuerk defines an instance of the NFA implementation using all Red-Black-Trees. This is then used for code generation and providing an accessor layer around the generated structures to make it possible to be used in raw SML or OCaml code as a mathematically correct library. In [31], Tuerk and Lammich go in more detail for one specific algorithm (Hopcroft's algorithm for automata minimisation [20]) and present benchmarks for the generated code in comparison with other, unchecked, implementations.

3.1.5 ω-Automaton

Similar to how NFA and DFA were defined, general ω-automata are defined in terms of semi-automata. The only addition to semi-automata is a general set $\mathcal{F}$ to be a *'f set*, for some type *'f*:

record (*'q, 'l, 'f*) *OmegaAutomaton* =
(*'q, 'l*) *SemiAutomaton* +
F :: *'f set*

locale *OmegaAutomaton* = *SemiAutomaton* A
for A::(*'q, 'l, 'f*) *OmegaAutomaton*

$\mathcal{F}$ is deliberately not of type *'q set* like for NFAs, because different types of ω-automata have different ways of encoding acceptance. Thus while for Büchi automata (BA) *'q set* is a

right choice, for Generalized Büchi Automata (GBA) a set of sets of nodes (*'q set set*) has to be used.

Mixing in finite semi-automata via the locale *FinSemiAutomaton*, finite ω-automata are defined:

locale *FinOmegaAutomaton = OmegaAutomaton* $\mathcal{A}$ *+ FinSemiAutomaton* $\mathcal{A}$
 for $\mathcal{A}$*::('q, 'l, 'f) OmegaAutomaton*

A general acceptance condition on ω-automata is defined in terms of an acceptance function *a_fun* (*OmegaAutomaton_is_inf_run* is a typed abbreviation of *SemiAutomaton_is_inf_run*):

abbreviation *ExOmegaAutomaton_accept a_fun* $\mathcal{A}$ *w*
 $\equiv \exists r$*::'q word. OmegaAutomaton_is_inf_run* $\mathcal{A}$ *w r* $\wedge$ *a_fun* $\mathcal{A}$ *w r*

That is, a word w is accepted by $\mathcal{A}$, iff there exists a run r such that some acceptance condition *a_fun* holds on it. For Büchi automata, *a_fun* would for example encode that "some state of $\mathcal{F}$ would occur infinitly often in r".

Furthermore some general properties on ω-automata are shown. For example, it is proven that for any infinite run r, the *limit*, i. e., the set of states that occur infinitly often in the run r, are all part of $\mathcal{Q}$ and also that it is always non-empty:

lemma (***in*** *FinOmegaAutomaton*) *is_inf_run__limit_sub_*$\mathcal{Q}$*:*
 assumes *OmegaAutomaton_is_inf_run* $\mathcal{A}$ *w r*
  shows *limit r* $\subseteq \mathcal{Q}(\mathcal{A})$

lemma (***in*** *FinOmegaAutomaton*) *is_inf_run__limit_nempty:*
 assumes *OmegaAutomaton_is_inf_run* $\mathcal{A}$ *w r*
  shows *limit r* $\cap \mathcal{Q}(\mathcal{A}) \neq \{\}$

Using this as a basis, different kinds of Büchi automata are defined, which we will describe in detail in the following.

Generalized Büchi Automata

Generalized Büchi Automata are defined in terms of *FinOmegaAutomaton*, with *'f* being instantiated with *'q set* thus modelling sets of sets of states[4]:

type_synonym *('q,'l) GBArel = ('q, 'l, 'q set) OmegaAutomaton*

locale *GBArel = FinOmegaAutomaton* $\mathcal{A}$
 for $\mathcal{A}$*::('q,'l) GBArel +*
 assumes $\mathcal{F}$*_consistent:* $\bigcup\mathcal{F}(\mathcal{A}) \subseteq \mathcal{Q}(\mathcal{A})$

The acceptance condition is modelled accordingly, thus accepting a word if the run hits a state of each of the acceptance sets:

definition *GBArel_accept* $\equiv$
 ExOmegaAutomaton_accept $(\lambda\mathcal{A}\ w\ r.\ (\forall S{\in}\mathcal{F}(\mathcal{A}).\ \textit{limit}\ r \cap S \neq \{\}))$

[4]The name *GBArel* stems from them being defined in terms of an underlying relation (the LTS), opposed to the elementary ω-automata of Section 3.1.6.

This leads to the obvious definition of its language:

definition $\mathcal{L}$*_GBArel* $\mathcal{A} = \{w.$ *GBArel_accept* $\mathcal{A}$ $w\}$

The most important operation for our model checking application is the product construction, which is then defined as a function *GBArel_product*, such that the following is true (we omit the definition itself for its technicalities):

lemma *GBArel_product_simps:*
shows $\mathcal{Q}$(*GBArel_product* $\mathcal{A}_1$ $\mathcal{A}_2$) = $\mathcal{Q}$ $\mathcal{A}_1 \times \mathcal{Q}$ $\mathcal{A}_2$
and Σ (*GBArel_product* $\mathcal{A}_1$ $\mathcal{A}_2$) = Σ $\mathcal{A}_1 \cap \Sigma$ $\mathcal{A}_2$
and (*q, a, q'*) $\in \Delta$(*GBArel_product* $\mathcal{A}_1$ $\mathcal{A}_2$) $\longleftrightarrow$
(*fst q, a, fst q'*) $\in \Delta$ $\mathcal{A}_1 \wedge$ (*snd q, a, snd q'*) $\in \Delta$ $\mathcal{A}_2$
and $\mathcal{I}$ (*GBArel_product* $\mathcal{A}_1$ $\mathcal{A}_2$) = $\mathcal{I}$ $\mathcal{A}_1 \times \mathcal{I}$ $\mathcal{A}_2$
and $\mathcal{F}$ (*GBArel_product* $\mathcal{A}_1$ $\mathcal{A}_2$) = ($\lambda F.$ $F \times \mathcal{Q}$ $\mathcal{A}_2$) `` $\mathcal{F}$ $\mathcal{A}_1 \cup$ *op* $\times$ ($\mathcal{Q}$ $\mathcal{A}_1$) `` $\mathcal{F}$ $\mathcal{A}_2$

The definition of the acceptance sets expresses[5]: The cartesian product of each acceptance set of $\mathcal{A}_1$ with all states of $\mathcal{A}_2$ union the cartesian product of all states of $\mathcal{A}_1$ with each acceptance set of $\mathcal{A}_2$.

It can be shown that this construction is indeed correct:

lemma *GBArel_product_is_GBArel:*
assumes *GBArel* $\mathcal{A}_1$ ***and*** *GBArel* $\mathcal{A}_2$
shows *GBArel* (*GBArel_product* $\mathcal{A}_1$ $\mathcal{A}_2$)

lemma *GBArel_product__language_eq:*
assumes *GBArel* $\mathcal{A}_1$ ***and*** *GBArel* $\mathcal{A}_2$
shows $\mathcal{L}$*_GBArel* (*GBArel_product* $\mathcal{A}_1$ $\mathcal{A}_2$) = $\mathcal{L}$*_GBArel* $\mathcal{A}_1 \cap \mathcal{L}$*_GBArel* $\mathcal{A}_2$

Büchi Automata

Büchi automata are defined similarly to GBAs, but with *'f* being instantiated with *'q*, thus defining $\mathcal{F}$ $\mathcal{A}$ to be a set of states:

type_synonym (*'q,'l*) *BArel* = (*'q, 'l, 'q*) *OmegaAutomaton*

locale *BArel* = *FinOmegaAutomaton* $\mathcal{A}$
for $\mathcal{A}$::(*'q,'l*) *BArel* +
assumes $\mathcal{F}$ $\mathcal{A} \subseteq \mathcal{Q}$ $\mathcal{A}$

definition *BArel_accept* $\equiv$ *ExOmegaAutomaton_accept* ($\lambda\mathcal{A}$ *w r.* (*limit r* $\cap$ $\mathcal{F}$ $\mathcal{A} \neq \{\}$))

definition $\mathcal{L}$*_BArel* $\mathcal{A} \equiv \{w.$ *BArel_accept* $\mathcal{A}$ $w\}$

Having a Büchi automaton with a non-empty language, we can find a run that is part of the language, i. e., accepted by the automaton. Such a run is represented by a lasso: For acceptance in a Büchi automaton, accepting states have to be visited infinitly often, thus ending the run in a circle. Combined with the (possibly empty) part from an initial

[5] f `` A denotes the image of f on A.

node to the first node of the circle, the run forms a lasso. We need to show that this lasso property holds, i. e., if the language of a Büchi automaton is non-empty, there exists a lasso. And also the counter-side: If a lasso exists, we can construct a word that is accepted by this automaton:

lemma *BArel_accept_lasso:*
 assumes $\mathcal{L}\_BArel\ \mathcal{A} \neq \{\}$
 shows $\exists q_i\ q_f\ r_1\ r_2.\ q_i \in \mathcal{I}\ \mathcal{A} \wedge q_f \in \mathcal{F}\ \mathcal{A}$
  $\wedge\ (\exists w.\ LTS\_is\_fin\_run\ (\Delta\ \mathcal{A})\ w\ r_1) \wedge hd\ r_1 = q_i \wedge last\ r_1 = q_f$
  $\wedge\ (\exists w.\ LTS\_is\_fin\_run\ (\Delta\ \mathcal{A})\ w\ r_2) \wedge hd\ r_2 = q_f \wedge last\ r_2 = q_f \wedge length\ r_2 > 1$

lemma *lasso_in_$\mathcal{L}$_BArel:*
 assumes $q_i \in \mathcal{I}\ \mathcal{A}$
 and $q_f \in \mathcal{F}\ \mathcal{A}$
 and $LTS\_is\_fin\_run\ (\Delta\ \mathcal{A})\ w_1\ r_1 \wedge hd\ r_1 = q_i \wedge last\ r_1 = q_f$
 and $LTS\_is\_fin\_run\ (\Delta\ \mathcal{A})\ w_2\ r_2 \wedge hd\ r_2 = q_f \wedge last\ r_2 = q_f \wedge length\ r_2 > 1$
 shows $w_1 \frown {w_2}^\omega \in \mathcal{L}\_BArel\ \mathcal{A}$
 and $OmegaAutomaton\_is\_inf\_run\ \mathcal{A}\ (w_1 \frown {w_2}^\omega)\ ((butlast\ r_1) \frown (butlast\ r_2)^\omega)$

Here, $w_1 \frown w_2$ concatenates the two words w_1 and w_2, and w^ω denotes the infinite word $w \frown w \frown w \frown ...$

Thereafter, the relation between Büchi automata and their generalized counterpart is established. First the simple direction: A GBA is constructed from a BA. This is expressed by setting the accepting set of the BA as the one and only accepting set of the GBA. As a further optimization, when $\mathcal{F}\ \mathcal{A} = \mathcal{Q}\ \mathcal{A}$, i. e., each state is accepting, the set of accepting sets for the GBA is set to the empty set: This is formally equivalent but gives a good runtime advantage in a later implementation. For technical reasons we omit the full definition and only show the important part for the initialization of $\mathcal{F}$:

definition *BArel_to_GBArel :: ('q, 'l) BArel ⇒ ('q,'l) GBArel*
where *BArel_to_GBArel $\mathcal{A} \equiv \ldots$ (**if** $\mathcal{F}\ \mathcal{A} = \mathcal{Q}\ \mathcal{A}$ **then** $\{\}$ **else** $\{\mathcal{F}\ \mathcal{A}\}$)*

This construction is shown to have all expected properties:

lemma *BArel_to_GBArel__is_GBArel:*
 assumes *BArel $\mathcal{A}$*
  shows *GBArel (BArel_to_GBArel $\mathcal{A}$)*

lemma *OmegaAutomaton_is_inf_run__BArel_to_GBArel_eq:*
 OmegaAutomaton_is_inf_run (BArel_to_GBArel $\mathcal{A}$) w r
  $\longleftrightarrow$ OmegaAutomaton_is_inf_run $\mathcal{A}$ w r

theorem *(**in** BArel) BArel_to_GBArel__language_eq:*
  shows *$\mathcal{L}$_BArel $\mathcal{A}$ = $\mathcal{L}$_GBArel (BArel_to_GBArel $\mathcal{A}$)*

Showing the other direction is more involved for the (possible) different accepting sets need to be encoded as a single set. In this formalization different counter constructions are used – further details are given by Schimpf [49].

System Automata

After having established this relation, it can be shown that the product construction on GBAs can be lifted to BAs – given that one of the BAs is a *System Automaton*. With this term we denote a BA where all states are final. The name stems from the fact that in model checking the system, or model, (as yielded by the construction in Chapter 5) is represented by such an automaton. We define such a specialization

locale *SArel* = *BArel* $\mathcal{A}$
 for $\mathcal{A}$ +
 assumes $\mathcal{F}\ \mathcal{A} = \mathcal{Q}\ \mathcal{A}$

lemma (***in*** *SArel*) *system_accept:*
 BArel_accept $\mathcal{A}\ w \longleftrightarrow (\exists r.$ *OmegaAutomaton_is_inf_run* $\mathcal{A}\ w\ r)$

and show that the native definition of the product is equal to the product at the GBA-level ($\mathcal{A}_S$ denotes the system, $\mathcal{A}_B$ the "normal" Büchi automaton, i. e., the property in model checking):

definition *SArel_BArel_product* $\mathcal{A}_S\ \mathcal{A}_B \equiv$
 $\ldots\ \{(q_1,q_2).\ q_1 \in \mathcal{Q}\ \mathcal{A}_S \wedge q_2 \in \mathcal{Q}\ \mathcal{A}_B \wedge q_2 \in \mathcal{F}\ \mathcal{A}_B\}$

definition *SArel_BArel_product_gba* $\mathcal{A}_S\ \mathcal{A}_B$
 $\equiv$ *GBArel_to_BArel* (*GBArel_product* (*BArel_to_GBArel* $\mathcal{A}_S$) (*BArel_to_GBArel* $\mathcal{A}_B$))

lemma *SArel_BArel_product__SArel_BArel_product_gba__eq:*
 assumes *SArel* $\mathcal{A}_S$ ***and*** *BArel* $\mathcal{A}_B$
 shows *SArel_BArel_product* $\mathcal{A}_S\ \mathcal{A}_B =$ *SArel_BArel_product_gba* $\mathcal{A}_S\ \mathcal{A}_B$

Labelled Büchi Automata

The algorithm of Gerth [15] uses a different representation of automata, where the labels are not part of the transition, but part of the state. As the CAVA project uses this algorithm to convert LTL properties into Büchi automata, they are also modelled here.

We will not go into detail here, and refer to Schimpf instead [49]. It is just to be noted that those structures are placed in the same hierarchy of automata types, by defining them as GBAs where the alphabet consists of the single token () and then adding a labelling function on states.

3.1.6 Elementary ω-Automaton

The Isabelle Collections Framework from the time the original automata framework was developed was unable to leave out parts of the data structure on refining to an implementation. This is similar to the early version of the Refinement Framework, which had the same restriction (cf. Example 4.8.1)[6]. As a consequence, redundant (or better:

[6] While by no means a restriction of the theory itself, i. e., one could write the refinement by hand, this would in essence rule out the usage of the Isabelle Collections Framework and any automations it provided.

effectively redundant) data would have been necessary in an implementation. In case of the automata, this especially means the state space: While states are already encoded as part of the underlying LTS (Δ), they are also encoded explicitly ($\mathcal{Q}$)[7]. For smaller automata, like the property automata generated from the LTL formula, this might be a nuissance, but can be easily worked with.

The problem surfaces when trying to encode the system automata: One of the large obstacles in model checking is representing the state space of the original system, or (depending on the property) the resulting product automaton. That is, in all other parts of the model checker a design goal has been to only evaluate the system lazily by providing a successor function. Hence, having to encode the state space as part of the automaton is a violation of this goal and could render the whole approach pointless.

One possible train of thought was omitting the set of states from the implementation and provide them at time of abstraction by *UNIV*, i. e., the set of all instances of the type *'q*. But unfortunately, this poses some problems:

1. If the type variable *'q* is instantiated by an infinite type, the set of states of the automaton would be infinite. This could render certain assumptions moot which rely on a finite state space.

2. If the type variable *'q* is instantiated by a finite type, the number of created states must not exceed the cardinality of the type's universe. This is hard to show for a lazy state space generation, where the number of states is not known before-hand.

3. Even if both those problems could be solved, the abstraction of the concretization of any automaton might yield a different automaton. This hits the same single-value restriction as the original problem.

For this reason, an additional refinement layer was created by means of another type of Büchi automata. This type, named *elementary Büchi automaton* in early stages, only consisted of those parts that were required for the operation as part of the model checker:

record *('q, 'l) BA =*
BA_Δ :: 'q ⇒ 'l ⇒ 'q set
BA_$\mathcal{I}$ :: 'q set
BA_$\mathcal{F}$ :: 'q ⇒ bool

Those parts describe a transition function *BA_Δ* (not a relation, so to better fit the use of a successor function in the other parts), a set of initial states *BA_$\mathcal{I}$*, and a characteristic function for the final states *BA_$\mathcal{F}$*. The latter is encoded as a function instead of a set, as the states are in general not known and therefore the final states are neither. But the characteristic function is known in advance.

The missing parts are defined in terms of the existing ones:

inductive_set *BA_$\mathcal{Q}$ :: ('q, 'l) BA ⇒ 'q set*
for *$\mathcal{A}$ :: ('q, 'l) BA* ***where***
$q \in BA\_\mathcal{I}\ \mathcal{A} \Longrightarrow q \in BA\_\mathcal{Q}\ \mathcal{A}$
| *$[\![\, q \in BA\_\mathcal{Q}\ \mathcal{A};\ q' \in BA\_\Delta\ \mathcal{A}\ q\ a \,]\!] \Longrightarrow q' \in BA\_\mathcal{Q}\ \mathcal{A}$*

[7] It is of course possible to add states in $\mathcal{Q}$ that are not encoded in Δ. As such states would not be reachable, they are uninteresting for the use case model checker. Therefore, we used *effectively redundant*.

definition *BA_LTS :: ('q,'l) BA ⇒ ('q,'l) LTS* ***where***
BA_LTS 𝒜 = {(q, a, q'). q ∈ BA_𝒬 𝒜 ∧ q' ∈ BA_Δ 𝒜 q a}

definition *BA_Σ :: ('q,'l) BA ⇒ 'l set* ***where***
BA_Σ 𝒜 = LTS_labels (BA_LTS 𝒜)

They can then be combined into a mapping function from the elementary Büchi automaton into the original *BArel*:

definition *BA_to_BArel 𝒜*
≡ (| 𝒬 = BA_𝒬 𝒜,
Σ = BA_Σ 𝒜,
Δ = BA_LTS 𝒜,
ℐ = BA_ℐ 𝒜,
ℱ = {q∈BA_𝒬 𝒜. BA_ℱ 𝒜 q} |)

Given the following straightforward definition of acceptance and the language of the automaton,

definition *BA_accept 𝒜 w*
≡ ∃r. r 0 ∈ BA_ℐ 𝒜
∧ (∀i. r (Suc i) ∈ BA_Δ 𝒜 (r i) (w i))
∧ (∃q∈limit r. BA_ℱ 𝒜 q)

definition *ℒ_BA 𝒜 ≡ {w. BA_accept 𝒜 w}*

it can be shown that *BA_to_BArel* preserves the language, i. e., that the two variants of a Büchi automaton are language-equivalent:

lemma *BA_to_BArel__language_eq:*
ℒ_BArel (BA_to_BArel 𝒜) = ℒ_BA 𝒜

As, for the usage of model checking, we are only interested in the language of the automaton, this equivalence is sufficient: Given an elementary automaton created as the representation of the system for the sake of implementation, we can lift it in the correctness proofs to a language equivalent relational automaton.

Similar to the more complex relational version of the Büchi automata, we define a specialization for system automata, where all states are set to be final (definition omitted). The original product construction on the relational versions cannot be simply used in the implementation, because it used the set of states of the system automaton, which is, as detailed earlier, not usefully accessible. Therefore, we define a new, implementable product construction and show its language-equivalence for the abstract definition:

definition *SA_BA_product* ***where***
SA_BA_product 𝒜_S 𝒜_B
= (| BA_Δ = λ(qs,qb) l. BA_Δ 𝒜_S qs l × BA_Δ 𝒜_B qb l,
BA_ℐ = BA_ℐ 𝒜_S × BA_ℐ 𝒜_B,
BA_ℱ = λ(qs,qb). BA_ℱ 𝒜_B qb |)

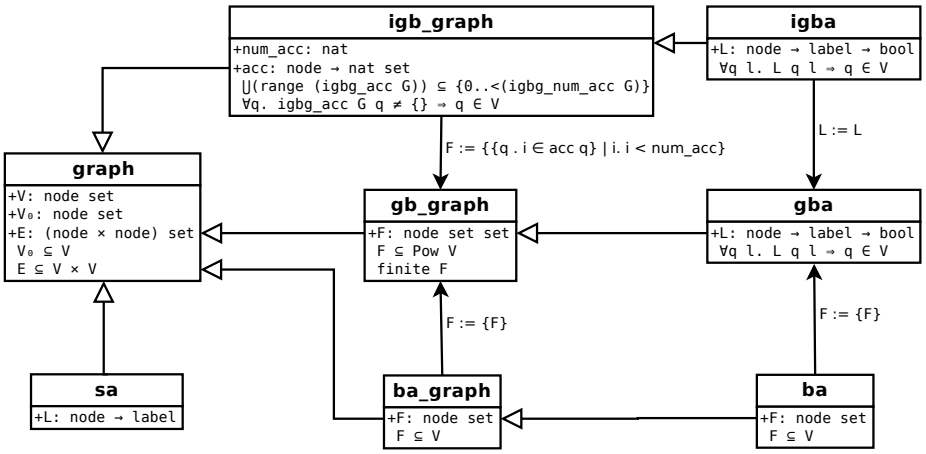


Figure 3.1: Structure of the CAVA automata library (based on [26])

lemma *product_elem_language_eq_product_language:*
$\mathcal{L}$*_BA* (*SA_BA_product* $\mathcal{A}_S$ $\mathcal{A}_B$)
= $\mathcal{L}$*_BArel* (*SArel_BArel_product* (*BA_to_BArel* $\mathcal{A}_S$) (*BA_to_BArel* $\mathcal{A}_B$))

While the system implementation can easily provide an elementary Büchi automaton, the LTL-to-BA conversion by Schimpf relied on the relational definition of automata. Due to the feasible size of the resulting state space, they can be (and are) also mapped directly to their implementation (omitted here). To make use of the product construction above, Schimpf therefore provided a mapping from labelled GBAs as used in Gerth's Algorithm to our elementary Büchi automata. The mapping is slightly more involved (it needs to move the labels from the states to the transitions, and also de-generalize). As the LTL part is not presented in this thesis, we will omit it at this point.

3.2 Current Formalization

In 2014, Lammich re-implemented the automata formalization as used in CAVA [26, 25], with the goal of gaining a unified framework that fits all the use cases in CAVA and avoids having multiple versions of the same thing (see the multiple definitions of Büchi automata from the previous section). In difference to Tuerk, he does not start off with labelled transition systems, but with generic graphs. Also, he chose to use a node-labelled representation as it is better suited for the use in CAVA.

We are not going to go into too much detail on the internal workings of the new library; the paper [26] gives an excellent (albeit slightly outdated) overview and describes design rationale.

Similar to the original work by Tuerk detailed in the previous section, Lammich chose a class-based[8] approach. The general structure of the classes of graphs/automata provided are best reflected in Fig. 3.1.

The basis of the object hierarchy is given by a simple definition of a digraph, only consisting of a set of edges:

[8] In the object-oriented sense, not the same-named Isabelle type-classes.

type_synonym *'v digraph = ('v×'v) set*
locale *digraph =*
  fixes *E :: 'v digraph*

This basis is then used to introduce basic definitions like paths. They are defined as a list of nodes, where the last node is not included in the list.

inductive *path :: 'v digraph ⇒ 'v ⇒ 'v list ⇒ 'v ⇒ bool* ***for*** *E* ***where***
  path E u [] u
| *⟦ (u,v)∈E; path E v l w ⟧ ⟹ path E u (u#l) w*

This definition allows for easy splitting and concatenating of paths: For example, concatenation can now be achieved by a simple append of the two paths:

lemma *path_conc:*
  assumes *path E u* p_1 *v*
  assumes *path E v* p_2 *w*
  shows *path E u* $(p_1@p_2)$ *w*

Moreover, the predicate *ipath* is introduced, defining when a *word* is an actual infinite path in a given graph:

definition *ipath E r ≡ ∀i. (r i, r (Suc i))∈E*

On (infinite) paths as well as other constructs like SCCs, different general properties are shown, which will not be covered here.

A more complex digraph, now also consisting of a set of nodes and a set of inital nodes, is then defined:

record *'v graph =*
  V :: 'v set
  E :: 'v digraph
  V_0 *:: 'v set*

Properties about graphs in general are shown inside a specific locale that fixes an instance of the graph together with general assumptions. The assumptions here are well-formedness spelled out. To ease reading, we will follow the theories and leave out the graph as an argument when the context is clear, i. e., use *V* instead of *V G* when *G* is fixed and clear to be a graph:

locale *graph =*
  fixes *G :: ('v) graph*
  assumes $V_0 \subseteq V$
  assumes $E \subseteq V \times V$

Here, general concepts like reachability or runs are also defined and basic properties shown:

definition *(**in** graph) reachable ≡* E^* `` V_0
definition *(**in** graph) is_run r ≡ r 0 ∈* V_0 *∧ ipath E r*
lemma *(**in** graph) reachable_V:*
  reachable ⊆ V

Lammich then introduces the Generalized Büchi Automaton as the first automaton structure, showing the original intent of the library to serve as the automata library for the CAVA project. Moreover, as can be seen later, the only introduced operations on automata will be product and renaming (the latter will not be covered here). That is, while the general structure of the library allows to introduce any operation needed, its intent is not to do so until the need arises. This is in contrast to the automata library from the previous chapter, where the intent originally was to provide as many automata-related operations as possible (this is not as strong in the later parts of the library, namely ω-automata).

The Generalized Büchi Automaton is modelled first without labels, which are then added in a second step. The formalization without labels is here called *gb_graph*:

record *'q gb_graph = 'q graph +*
F :: 'q set set

locale *gb_graph = graph G*
for *G :: ('q) gb_graph +*
assumes *finite F*
assumes $F \subseteq 2^V$

As labels are not yet defined, the only additional field is the set of the acceptance classes F. It is to note, that the class-oriented architecture of the library is reflected here by both the structure definition and the property-environment (***locale***) inheriting from the already defined *graph*.

For such a *gb_graph*, accepting runs are defined in the obvious way (with $\exists_\infty i.\ P\ i$ denoting "there exist inifinitly many i, such that $P\ i$"):

definition *(**in** gb_graph) is_acc_run r ≡*
is_run r ∧ (∀A∈F. ∃∞i. r i ∈ A)

When adding labels, which are per node, not per transition, one obtains the complete automaton structure:

record *('q,'l) gba = 'q gb_graph +*
L :: 'q ⇒ 'l ⇒ bool

locale *gba = gb_graph G*
for *G :: ('q,'l) gba +*
assumes *L q l ⟹ q ∈ V*

Now it is also possible to define the language of such an automaton:

definition *(**in** gba) accept w ≡ ∃r. is_acc_run r ∧ (∀i. L (r i) (w i))*
definition *(**in** gba) lang ≡ {w. accept w}*

A similar construction is done to introduce Büchi automata; the structures are named *ba_graph* and *ba*. We will omit them here.

Additionally, for BAs it is shown that one can construct GBAs

definition *(**in** ba) to_gba ≡ (|*
V = V,
E = E,

$V_0 = V_0$,
$F = \textbf{if } F = UNIV \textbf{ then } \{\} \textbf{ else } \{F\}$,
$L = L$
⟫

and that this construction yields valid GBAs:

sublocale in (ba) *gba!: gba to_gba*

Furthermore, Lammich adds a second version of GBAs, indexed GBAs, which he also proves to be equivalent (proof omitted here). Instead of having a set of acceptance classes that can be queried for a particular state like *gba/gb_graph* above, a function is added that allows to query a node for the classes it is in. Similar to the $\mathcal{F}$-function for *BA*s in the previous section, it allows to define the acceptance lazily. Additionally, the number of acceptance classes is needed, such that acceptance is decidable.

record *'q igb_graph = 'q graph +*
num_acc :: nat
acc :: 'q ⇒ nat set

locale *igb_graph = graph G*
for *G :: 'q igb_graph +*
assumes $\bigcup(range\ acc) \subseteq \{0..<num\_acc\}$
assumes $acc\ q \neq \{\} \Longrightarrow q \in V$

Accepting runs are then defined as expected:

definition $(\textbf{in } igb\_graph)$ $is\_acc\_run\ r \equiv is\_run\ r \wedge (\forall n{<}num\_acc.\ \exists_\infty i.\ n \in acc\ (r\ i))$

Dual to GBAs, labels are added yielding record and locale *igba*, that defines the language equally:

definition $(\textbf{in } igba)$ $accept\ w \equiv \exists r.\ is\_acc\_run\ r \wedge (\forall i.\ L\ (r\ i)\ (w\ i))$
definition $(\textbf{in } igba)$ $lang \equiv \{w.\ accept\ w\}$

While the conversions of *ba* to *gba* and *igba* to *gba* were straightforward, the other directions are not. Lammich also provides constructions for those, but we will omit them here.

Finally, system automata are defined, similar to the original automata library. In a system automaton, all states are final. In Lammich's implementation, they do not inherit from (General) Büchi Automaton but from graphs and therefore there is no need to have any information about final states at all. Instead, it is encoded implicitly in the language of such an automaton:

record $('q,'l)$ *sa = 'q graph +*
L :: 'q ⇒ 'l

locale *sa = g: graph G*
for *G :: ('q, 'l) sa*

definition $(\textbf{in } sa)$ $accept\ w \equiv \exists r.\ is\_run\ r \wedge w = L \circ r$
definition $(\textbf{in } sa)$ $lang \equiv \{w.\ accept\ w\}$

Using those, the first (and only) binary operation on automata is defined: The product between *igba* and *sa*. Following the same strategy as the Isabelle Collection Framework (cf. Section 3.1.4), the operation is defined inside its own locale that allows to fix the assumptions at one place. For the use inside a model checker, where the product is simply searched for an accepting cycle, the labels of that product are of no interest, hence the result of the operation is an *igb_graph*:

locale *igba_sys_prod_precond = igba!: igba G + sa!: sa S*
 for *G :: ('q,'l) igba*
 and *S :: ('s,'l) sa*

definition *(**in** igba_sys_prod_precond)*
prod ≡ ⦇
  V = igba.V × sa.V,
  E = { ((q,s),(q',s')).
    igba.L q (sa.L s) ∧ (q,q') ∈ igba.E ∧ (s,s') ∈ sa.E },
  V_0 = igba.V_0 × sa.V_0,
  num_acc = igba.num_acc,
  *acc = (λ(q,s). **if** s ∈ sa.V **then** igba.acc q **else** {})*
⦈

Lammich of course also shows this operation to be correct. As a run *r* on the product automaton consists of a sequence of pairs of states, one needs to project onto the state of the system automaton to get the corresponding run in the original system. This is achieved by *snd* ∘ *r*:

lemma *gsp_correct1:*
  assumes *prod.is_acc_run r*
  shows *sa.is_run (snd ∘ r) ∧ (sa.L ∘ snd ∘ r ∈ igba.lang)*

lemma *gsp_correct2:*
  assumes *sa.is_run r **and** sa.L ∘ r ∈ igba.lang*
  shows *∃r'. r = snd ∘ r' ∧ prod.is_acc_run r'*

After those basic definitions of automata, Lammich further adds a notion of a lasso in automata. He also adds the notion of stutter extension on automata, i. e., adding a self-loop for any sink-state in the automaton. This allows to find infinite paths in the automata where before there were only finite ones.

Both concepts, lassos and the stutter extension, will not be discussed here further.

3.3 Comparison and Concluding Remarks

We have seen two different approaches to modelling automata in Isabelle/HOL. One difference is the availability of new technologies: The second library by Lammich can make use of Lammich's own new or improved frameworks, especially the Refinement Framework. The latter makes it easier to bridge the multiple phases from abstract definition to exectuable code – a more complicated and tenious task for the first framework by Tuerk.

But such technical foundations are not the main difference, the most important one is the different directions of purpose in those frameworks. While Tuerks framework, laid out as a comprehensive automaton library, covers a lot more theoretical fields, Lammich's framework is very specialized, tailored for usage as a part of our model checker CAVA. This becomes visible, naturally, when trying to make them part of that model checker: We have shown in Section 3.1.5 that the formalization of usable ω-automata was not an easy task and we needed several versions of Büchi automaton for different use cases. In Lammich's framework, we also have different versions of (Generalized) Büchi automata, but they only differ slightly.

Also, Lammich makes graphs the foundation of his framework, which is of advantage because it enables the model checker to have one encompassing graph formalization instead of rolling different versions for different parts that need to be converted when passing them from one part to another. This lacks in Tuerks version, where the foundation is the LTS. Similarly, Lammich offers *graph* versions of the automata, which remove the labels. This also allows for an easier bridge to other parts that do not need the labels for their working. In Tuerk's part they are, due to the LTS, built in at a very deep level and only removable by workarounds (see for instance Section 3.1.5). To conclude, the aspiration to be a comprehensive library can stand in the way of the goal to be a simple-to-use part of a larger product, for the reasons mentioned above: The general versions might not be ideal for the requirements of the product; and developing the wrapping layer(s) can require the same amount of work as a from-scratch development.

Of course, one has to take the time-line into consideration: Lammich's framework was built when the intention and needed interfaces were mostly laid out, while Tuerk framework already was developed mostly when it was tried to incorporate into the model checker. For that reason, the learning effect visible in Tuerk's framework is already taken into account by Lammich.

4 Checking

Constructing the search space is only one half of the work needed for checking the model. While we now have obtained a Büchi automaton representing the intersection of the system with the property, it has to be checked whether there exists an accepting run in this automaton (which is equivalent to checking the language of the automaton for emptiness). If there exists such a run, it is possible for the system to encounter a valid state where the property holds. This approach is the general idea behind model checking and already detailed by Vardi and Wolper [55].

There are plenty of algorithms for checking the emptiness of such a language. We also require the algorithm to work on-the-fly, that is, build the search space only on demand to avoid a memory blowup. This again is not new and algorithms exist in many shapes and forms (see Zhao et al. [58] for an overview). An intention of the project is therefore to support not only one, but multiple of such algorithms. This, for instance, allows to compare the efficiency of different algorithms.

It is, of course, advisable to try to reduce any duplication of the proof effort. Hence, the current chapter is going to detail the generic framework for formalizing different checking algorithms and providing insight on the existing formalizations. Furthermore, as in the previous chapter, we will describe the evolution of the framework and how, due to the lessons learned throughout this evolution, things are done the way they are done.

There are two different main tendencies on how to write an algorithm for checking the emptiness of a Büchi automaton: Use nested depth-first search [6] to find a cycle containing a accepting node, or generate the set of (non-trivial) strongly connected components and find one that contains an accepting node [7]. Both tendencies have in common their basing on depth-first search. Hence, instead of creating a framework explicitly tailored for emptiness checks, a framework for algorithms based on depth-first search has been developed (an overview over this framework is presented in our paper [29] and available in the Archive of Formal Proofs [30]).

4.1 Depth-First Search

In its most well-known formulation, depth-first search is a very simple algorithm as laid out in Alg. 4.1: For each node v_0 from a given set V_0 of start nodes, we invoke the function *DFS*. This function, if it has not seen the node yet, recursively invokes itself for each successor of the node.

With this simple algorithm, it is already possible to introduce certain notions. While those notions are, in general, well-known, we will shortly explain them here, as they are going to be used throughout this section:

search stack The search stack are the nodes which are currently "under examination", keeping the history, i. e., a node at position i of the stack is always the successor in

Algorithm 4.1 General DFS

```
1: discovered ← {}
2: for all v_0 ∈ V_0 do
3:     DFS v_0

4: procedure DFS(u)
5:     if u ∉ discovered then
6:         discovered ← discovered ∪ {u}
7:         for all v ∈ SUCCESSORS u do
8:             DFS v
```

the graph of the node at position $i - 1$. While sometimes implicit, for example when it is only given by the call-stack in our recursive specification in Alg. 4.1, a stack is always involved in a depth-first search.

discovered A node is discovered if it has been visited by the search so far.

finished A node is finished if it has been discovered and all of its successors are finished. For most uses, this is identical to the node being discovered but not on the search stack anymore.

pending An edge (u,v) is pending, when u is part of the stack and the edge has not been walked yet.

search tree The search tree is a subgraph of the graph we are currently working on and marks those edges taken through the search which lead to the discovery of new nodes.

forward edge A forward edge is an edge of the search tree.

cross edge A cross edge is an edge, that, when discovered, links the top of the stack with an already finished node. When seen in relation to the search tree, it crosses the tree linking to subtrees.

back edge A back edge is an edge, that, when discovered, links the top of the stack with another node deeper in the stack. Self-loops are always back edges. When seen in relation to the search tree, it introduces an edge going back up the tree, introducing a cycle.

But of course, just traversing the graph is not our sole goal. Instead, we notice that depth-first search is a basic building block for a variety of algorithms, for example Cyclicity Checking (Alg. 4.2), Nested Depth-First Search (Alg. 4.3), and Tarjan's algorithm for constructing the set of strongly-connected components of a graph (Alg. 4.4). Thus, we want to extract how the basic search can be extended to yield such algorithms.

We start with the small example of a cyclicity checker, as described in Alg. 4.2. In this example, which will also serve as the running example throughout the rest of this chapter, we test whether we have encountered a back edge, i. e., the target node is already on the stack, as can be seen at line 13. This is a correct implementation, as the stack forms a path

Algorithm 4.2 Cycle Detection

```
1: discovered ← {}
2: stack ← []

3: procedure CYCLE
4:     for all v_0 ∈ V_0 do
5:         DFS v_0

6: procedure DFS(u)
7:     if u ∉ discovered then
8:         stack ← PUSH u stack
9:         discovered ← discovered ∪ {u}
10:        for all v ∈ SUCCESSORS u do
11:            DFS v
12:        stack ← POP stack
13:    else if u ∈ stack then
14:        report cycle
```

in the graph, and u is the successor of the top of the stack, hence we have encountered a cycle if and only if we see a back edge.

This implementation of the cyclicity checker already details the approach for extending the search algorithm:

1. We need an explicit stack. Hence we must add a new variable and make sure it is updated accordingly throughout the search.

2. We need our purpose (i. e., checking for cycles) to be part of the search. Thus we extend the depth-first search by injecting some actions into the case of encountering a back edge.

The approach can also be seen in the Nested DFS algorithm of Alg. 4.3: This algorithm is also checking for cycles, but only those which contain at least one accepting node. As the name already implies, we now have two instances of depth-first search: the *blue* and the *red* search. We will not go into detail here, as this algorithm is discussed in length in Section 4.6.

Algorithm 4.3 Nested DFS by Courcoubetis et al. [6]

```
1: discovered_b ← {}
2: discovered_r ← {}
3: seed ← ⊥

4: procedure DFS-BLUE(u)
5:     if u ∉ discovered_b then
6:         discovered_b ← discovered_b ∪ {u}
7:         for all v ∈ SUCCESSORS u do
8:             DFS-BLUE t
9:         if ACCEPTING u then
10:            seed ← u
11:            DFS-RED u

12: procedure DFS-RED(seed, u)
13:    if u ∉ discovered_r then
14:        discovered_r ← discovered_r ∪ {u}
15:        for all t ∈ SUCCESSORS s do
16:            if t = seed then
17:                report cycle
18:            DFS-RED t

19: procedure NESTED-DFS
20:    for all v_0 ∈ V_0 do
21:        DFS-BLUE v_0
```

But we see that each instance of the search is equipped with its own set of discovered nodes, and also we are adding an auxiliary variable *seed* (line 3). Further, we again need our purpose to be injected into the search. This time, we extend, in the blue version, the case of finishing a node (lines 9 to 11) to start the red search. In the red version, we extend the case of discovering a node by comparing it to *seed* and, if this is true, reporting a cycle (line 16).

As a third, and last, example, we want to show an implementation of Tarjan's algorithm for computing the set of strongly connected components (SCCs) in Alg. 4.4. Without going into too much detail on the general working of this algorithm, we have added multiple additional state variables like the set of found SCCs *sccs* or the Tarjan stack $stack_{tj}$. Also, we enhanced *discovered* to track the discovery time, and, with a slight abuse of notation, write *discovered* x to get the discovery time of x. We extend our basic search algorithm by some additional actions in case a node is finished (line 17 and following). Also, we have to extend the case of encountering an already discovered node (line 25 and following), without differentiating between a back edge and a cross edge.

More details on this algorithm and how it can be implemented inside the DFS framework is shown later on in Section 4.7.

With those three algorithms serving as examples, we have shown how the basic depth-first search can, in principle, be extended to become more advanced. We have also demonstrated that, on the other hand, depth-first search is a basic building block of various algorithms. Thus, we can also separate the basic search from the extension. This then can be used to specify properties about the basic search independent from the extensions, and use those properties later on in reasoning about the extension. This approach yields two benefits:

1. Properties about the search itself only have to be proven once and can then be used in multiple algorithms.

2. Separating the concerns often helps understanding the proofs. That is, by not bloating the correctness proof of an algorithm, for example Tarjan's algorithm, by general properties, e. g., about the discovery times of nodes in a depth-first search, the proofs can focus on the inherent properties of the algorithm itself. To keep with the example, this would for example include the relation between the lowlink and the Tarjan stack.

For simple algorithms, like the Cyclicity Checker of Alg. 4.2, this can drastically reduce the proof of correctness. For our example, the proof of correctness is reduced to show one property about the algorithm, provable in three steps. This is shown in detail in Example 4.4.3.

4.2 A Generic (Depth-First) Search

In the simple form presented in the previous section in Alg. 4.1, the algorithm can only be used to create the set of reachable nodes, i. e., *discovered*. But as we have also shown, other algorithms are based on DFS, and thus we need to develop another view of the algorithm: We want to provide a skeleton DFS algorithm, which is parameterized by

Algorithm 4.4 Tarjan's Algorithm

```
 1: discovered ← {}
 2: stack ← []
 3: stack_tj ← []
 4: sccs ← {}
 5: lowlink ← {}
 6: time ← 0

 7: procedure DFS(u)
 8:     if u ∉ discovered then
 9:         stack ← PUSH u stack
10:         stack_tj ← PUSH u stack_tj
11:         discovered ← discovered ∪ {(u, time)}
12:         lowlink ← lowlink ∪ {(u, time)}
13:         time ← time + 1
14:         for all v ∈ SUCCESSORS u do
15:             DFS v
16:         stack ← POP stack

17:         if stack ≠ [] then
18:             let x = TOP stack
19:             let t' = MIN (lowlink x)(lowlink u)
20:             lowlink ← lowlink ∪ {(x, t')}
21:         if lowlink u = discovered u then                    ▷ Root of SCC
22:             let (tj, scc') = COLLECT AND POP EVERYTHING UNTIL u stack_tj
23:             stack_tj ← tj
24:             sccs ← sccs ∪ {scc'}
25:     else
26:         let x = TOP stack
27:         if discovered u < discovered x ∧ u ∈ stack_tj then
28:             let t' = MIN (lowlink x)(discovered u)
29:             lowlink ← lowlink ∪ {(x, t')}

30: procedure TARJAN
31:     for all v0 ∈ V0 do
32:         DFS v0
33:     return sccs
```

hook functions that are called from well-defined *extension points* (i. e., actions injected by the extension for the different cases of the search), and modify an opaque *extension state* (i. e., a structure holding additional data needed by the extension). Moreover, we add an additional break condition, which allows to interrupt the search prematurely, before all reachable nodes have been explored. Hence, an extension of the search is then represented as a parameterization over the skeleton search algorithm.

Moreover, the specification in Alg. 4.1 was given in a recursive form. For a correctness proof, we need to establish invariants for the two foreach-loops, and a pair of pre- and postconditions for the recursive call. This quite complex proof structure hampers the design of our framework (we will discuss a variant of this framework based on a recursive definition in Section 4.8.2). Thus, we use an iterative formulation of DFS that only consists of a single loop. Correctness proofs are done via a single loop invariant for that loop.

Taking these two points into consideration, we create the definition of a depth-first search given as in Alg. 4.5.

Algorithm 4.5 Generic DFS definition

```
definition step s ≡
 if is_empty_stack s then do {
  v0 ← spec v0. v0∈V0 ∧ ¬is_discovered v0 s;
  do_new_root v0 s
 } else do {
    ((u, nxt), s') ← get_pending s;
    case nxt of
      None ⇒ do_finish u s'
    | Some v ⇒ do {
     if is_discovered v s' then (
      if is_finished v s' then
       do_cross_edge u v s'
      else
       do_back_edge u v s'
     ) else
      do_discover u v s'
    }
  }

definition cond s
 ≡ (V0 ⊆ {v. is_discovered v s} ⟶ ¬is_empty_stack s)
  ∧ ¬break s

definition dfs ≡
 s0 ← do_init;
 while cond step s0
```

In this specification, we can identify five cases:

new root If the stack is empty, we choose a start node that has not yet been discovered (the condition guarantees that there is one). This is a special case of disovery, but allows for a better proof structure, because it explicitly distinguishes between those two cases.

discover If the stack is non-empty, the *get_pending*-function tries to select a pending edge starting at the node u. If a pending edge (u,v) exists and v has not been discovered so far, we discover it. The edge (u,v) then is a forward edge.

back edge If a pending edge (u,v) exists and v has already been discovered but is not finished yet, we have a back edge.

finish If there are no such edges left (i. e., *nxt* is *None*), u is finished.

cross edge Lastly, if v is also finished, (u,v) is a cross edge.

We want to use each of these cases as extension points. Adding initialization and the ability to abort the search, we can define the parameterization as:

```
record ('v,'s,'es) gen_parameterization =
  on_init :: 'es nres
  on_new_root :: 'v ⇒ 's ⇒ 'es nres
  on_discover :: 'v ⇒ 'v ⇒ 's ⇒ 'es nres
  on_finish :: 'v ⇒ 's ⇒ 'es nres
  on_back_edge :: 'v ⇒ 'v ⇒ 's ⇒ 'es nres
  on_cross_edge :: 'v ⇒ 'v ⇒ 's ⇒ 'es nres
  is_break :: 's ⇒ bool
```

This record used the type variables *'v* for the type of nodes, *'s* for the search state, and *'es* for the extension part, i. e., a data structure of its own chosing. Recall from Section 2.2 that *'es nres* is the type of all possible results of type *'es*, allowing for non-deterministic specifications. This also implies the type of the hook functions: From a node or an edge, and the current search state, they are expected to return an (updated) extension state.

This definition might be puzzling, because there is no current extension state as input for any of those hooks, e. g., one could expect $'s \times 'es$ instead of plain *'s*. This stems from the fact that states (both search and extension) are modeled as records, where the extension is defined by extending the search state, i. e., *'es* is implicitly given as a component of *'s*. Unfortunately, this cannot be expressed as a type restriction in Isabelle/HOL. As a consequence, the extension state is extracted from the search state with the *more* selector, and each field of the extension state can be directly queried by name.

For documentation, we will use the following definition when defining hooks without any functionality:

```
abbreviation NOOP s ≡ return (more s)
```

Moreover, we will use a shortcut notation when the extension state consists of multiple fields and only a subset is going to be updated (here: $field_1$ and $field_2$):

$(\!| field_1 = value_1, field_2 = value_2, \ldots |\!)$

Example 4.2.1 (Cyclicity Checker)
As already established in the previous section, a simple application of DFS is a cyclicity check, based on the fact that there is a back edge if and only if there is a reachable cycle. We will now represent the implementation of Alg. 4.2 as a parameterization of ours.

The extension state solely consists of a single boolean flag representing whether a back edge has been encountered. Thus, expecting some search state of type *'v state*, our extended state would be:

```
record 'v cyc_state = 'v state +
  cyc :: bool
```

With this, we can then define a parameterization for the cyclicity checker with

```
definition cyc_checker where
 cyc_checker = (|
   on_init ≡ return (| cyc = False |),  (* initially no cycle has been found *),
   on_new_root ≡ λu. NOOP,
   on_discover ≡ λu v. NOOP,
   on_finish ≡ λu. NOOP,
   on_back_edge ≡ λu v s. return (| cyc = True |) (* cycle! *),
   on_cross_edge ≡ λu v. NOOP,
   is_break ≡ λs. cyc s (* break iff cycle has been found *)
 |).
```

This covers the exact behavior of a cyclicity checker.

Example 4.2.2 (Edge Classificator)
Another small example to show how the parameterization works, is a DFS extension, where the encountered edges are classified into forward, back and cross edges.

The extension state now consists of the three different sets of edges. Thus, for some search state of type *'v state*, our extended state would be:

```
record 'v ec_state = 'v state +
  forward :: 'v rel
  back :: 'v rel
  cross :: 'v rel
```

With this, we can then define the parameterization as

```
definition edge_classifier where
 edge_classifier = (|
   on_init ≡ return (| forward = {}, back = {}, cross = {} |),
   on_new_root ≡ λu. NOOP,
   on_discover ≡ λu v s. return (|forward := insert (u,v) (forward s), ... |)
   on_finish ≡ λu. NOOP,
   on_back_edge ≡ λu v s. return (|back := insert (u,v) (back s), ... |),
   on_cross_edge ≡ λu v s. return (|cross := insert (u,v) (cross s), ... |),
   is_break ≡ λs. False (* we want to explore the whole graph *)
 |).
```

After a successfull run of *dfs* the sets *forward*, *back*, and *cross* then contain their appropriate sets (which of course needs to be shown separately). It should be noted here, that in reality such an extension is unnecessary, as this classification will already be done by the default state implementation given in Section 4.3. Therefore, we will also not cover this example any further in the rest of this chapter.

Using our definition of parameterization, we are still missing the link to the DFS specification given initially in Alg. 4.5. To be able to express this link, we first have to take a look at the other part of our formalization: The proper search. In our specification, we use functions like *is_empty_stack* or *on_discovered*, but they are not yet defined. As a consequence the algorithm does not refer to any data structures like a stack or a set of discovered nodes directly. Thus the formalization is independent of the actual representation of the search state. But this also entails that we cannot state directly what the search should do for the different cases. Instead, we want to define the search only in an abstract manner, in the same way we defined the parameterization abstractly:

record *('v,'s,'es) gen_basic_dfs_struct =*
gbs_init :: 'es ⇒ 's nres
gbs_is_empty_stack :: 's ⇒ bool
gbs_is_discovered :: 'v ⇒ 's ⇒ bool
gbs_is_finished :: 'v ⇒ 's ⇒ bool
gbs_get_pending :: 's ⇒ ('v × 'v option × 's) nres
gbs_new_root :: 'v ⇒ 's ⇒ 's nres
gbs_finish :: 'v ⇒ 's ⇒ 's nres
(∗ some fields omitted ∗)

Identically to the parameterization, the type variables *'v*, *'s*, and *'es* denote the type of nodes, search state, and extension part, respectively.

Only when we combine the specification of the search and of the parameterization, we get a final algorithm. This is expressed by introducing a locale taking three parameters: an instance of *gen_basic_dfs_struct* to represent the search, an instance of *gen_parameterization* to represent the extension, and additionally the set of initial nodes. The set of edges is not required explicitly, as *get_pending* acts as an abstraction.

locale *gen_param_dfs =*
fixes *gbs :: ('v,'s,'es) gen_basic_dfs_struct*
fixes *param :: ('v,'s,'es) gen_parameterization*
fixes V_0 *:: 'v set*

Inside this locale, we place our specification of a parameterized DFS as presented earlier. We also define the functions therein by chaining the operations of the search with the operations of the parameterization:

definition *(**in** gen_param_dfs) do_init ≡ **do** {*
e ← on_init param;
gbs_init gbs e
}

```
definition (in gen_param_dfs) do_new_root v₀ s ≡ do {
  s' ← gbs_new_root gbs v₀ s;
  e ← on_new_root param v₀ s';
  return s'(|more := e|)
}

(* same for the remaining operations *)

definition (in gen_param_dfs) get_pending ≡ gbs_get_pending gbs
definition (in gen_param_dfs) is_discovered ≡ gbs_is_discovered gbs
definition (in gen_param_dfs) is_finished ≡ gbs_is_finished gbs
definition (in gen_param_dfs) is_empty_stack ≡ gbs_is_empty_stack gbs

definition (in gen_param_dfs) break ≡ is_break param
```

As can be seen, some of the functions used are only depending on one of the two parts. For instance, breaking is just defined by the parameterization, because the search itself has no use case in aborting. On the other hand dealing with pending edges or discovered nodes should not be part of the parameterization, but is inherently part of the search.

4.2.1 Why so generic?

We have now presented how we define the generic search algorithm, and how we link the parameterization and the search itself. But in this process we have kept the search definition very generic, leaving out any details of the state. The functions *gbs_get_pending* or *gbs_is_empty_stack* serve as good illustrations of this approach. What has not been done so far, is to explain why this is useful, why we cannot define the search right from scratch, including the contents of the search state.

The main advantage of this strategy is that it is very unspecific, that is, there are no additional obligations the algorithm has to fulfill. Or, in other words, any assertions to be made are completely defined by the structure of the algorithm: For example it can be asserted that $\neg$ *is_discovered gds v s* holds on invocation of *on_discover u v s*. As a consequence, it allows to refine this algorithm into other forms very easily, i. e., it does not require any obligations of its own, thus granting more freedom to the specifications of the refinements.

The main idea for the refinement is: Any formalization in our framework is (indirectly) an instantiation of the generic search. If it can be shown that some other algorithm, depending on the same set of parameters, is a refinement of the generic search, its instantiation is also a refinement of our formalization.

The need for such easy refinement arises mainly from the possibility of structural refinement, that is the ability to replace the skeleton search algorithm by something more suited for the final use case, i. e., a performance optimization. Structural refinement will be explained later on in Section 4.5.2. There, it will also be clear why we have introduced the additional abstraction of *get_pending*, instead of querying the set of edges directly: A refinement may have additional requirements on returning the next pending edge. As an additional benefit, when implementing *get_pending* differently, our framework for

depth-first search could be extended to cover other searches like breadth-first search or priority-based search. But this has not been pursued, yet.

Example 4.2.3 (Search Refinement)
We now want to give an example on how such refinement of the search itself can look like. We therefore assume some hypothetical formalization of DFS, which is more efficient than the one defined by us, but has the restriction that initial nodes may not have any incoming edges.

We start by defining a separate locale for this optimized search, which will inherit from our *gen_param_dfs*, but adds the aforementioned restriction:

```
locale optimized_dfs = gen_param_dfs gbs param V₀
 for gbs param V₀ +
 assumes pending s ≤ spec ((u,nxt), s). case nxt of
     Some v ⇒ v ∉ V₀
   | None  ⇒ True
```

Inside this locale, we specify the optimized search and also prove that this search is a refinement of the original *dfs*:

```
definition (in optimized_dfs) opt_dfs ≡
 (* some definition making use of the parameters given by gbs *)

lemma (in optimized_dfs) opt_dfs_refine:
 opt_dfs ≤ dfs
```

Now we have an optimized search algorithm. Before we can put it to use, we need an instantiation of the generic search:

```
definition dfs_instance ≡ (|
 gbs_init = ...,
 gbs_is_empty_stack = ...,
 (* and so on *)
|)
```

We then show that this is indeed a generic parameterized search for any parameterization and set of initial nodes, and moreover prove some properties about the resulting search:

```
interpretation our_dfs: gen_param_dfs dfs_instance param V₀ for param V₀ .

lemma dfs_instance_prop:
 our_dfs.dfs ≤ spec Φ
```

Let us assume that the definition of *dfs_instance* fulfills the requirement needed by *optimized_dfs*. Then we can show that our *dfs_instance* is also an instance of *optimized_dfs*, even though this proof might now be more involved due to the additional assumptions:

```
interpretation our_dfs: optimized_dfs dfs_instance param V₀ for param V₀
 (* some proof done here *)
```

Due to the refinement shown earlier on the properties about the original generic search, due to transitivity, carry over to the instantiation of the *optimized_dfs*. Thus, we can finally use *opt_dfs* for code generation while still having our original properties:

lemma *opt_dfs_instance_prop:*
our_dfs.opt_dfs $\leq$ ***spec*** Φ
by (*rule order_trans*[*OF our_dfs.opt_dfs_refine dfs_instance_prop*])

export_code *our_dfs.opt_dfs* ***in*** *SML*

It should be noted that in the Isabelle theories, there exists another, even more generic layer on top of *gen_param_dfs*: This additional level, called *gen_dfs* does not know anything about parameterization. As this layer is not needed for the rest of the chapter, it is omitted, and definitions involving it are changed to use *gen_param_dfs* directly for presentation. This might also lead to seemingly useless involvements of the parameterization, like in *optimized_dfs* above.

4.3 Implementing the Search: A Specific State

After laying the foundation by specifying a generic search and a generic parameterized search, we are now continuing by introducing an explicit search instantiation. While this is more concrete than the generic versions, it is still aimed to be as abstract as possible by not caring about implementation details. Its goals are mainly to provide the final means to base an explicit DFS-using algorithm on, and to provide an extensive library of invariants. Therefore, this stage will be called the *abstract level* (opposed to the *generic level* we described so far, and the *concrete (implementation) level* which is still to come (Sections 4.5, 4.5.3)).

For an increased usability, one goal for specifiying this abstract level is to gather as much information as possible throughout the search process. This enables us to provide detailed lemmas about the search, which in turn may come in useful for the embedded algorithm. This is made possible by recent enhancements of the Isabelle Refinement Framework [31] which allows to remove information when projecting the abstract state onto the concrete state (details follow in Section 4.5.1), thereby enabling us to generate lean code where the auxiliary data from the abstract level is no longer present.

The content of the state, i. e., the information gathered, then can be divided into three categories:

Management information This includes those fields which are essential for conducting the search at all. This category consists of the *search stack* and the set of *pending edges*. While the first one is natural, the second one can be represented in multiple ways. One option, which was for example chosen in the first approach [35], is to use another stack, where each item represents the pending edges of the corresponding node on the search stack. Unfortunately, proving and enforcing this correspondence is rather inelegant and clutters the theory, as will be described in Section 4.8.1. A similar solution where both stacks are combined into one is, again, rather inelegant on the abstract level, as it needlessly merges two different concepts, but is a viable

option to be used in the implementation. The choice for the abstract level then is indeed the simple set of *all* the pending edges. While this would be a nightmare performance-wise in any implementation it is just right for abstract reasoning, as the pending edges of any node u can be simply queried by $pending `` \{u\}$.

Timing information A natural, and necessary, field to have in a depth-first search is the set of *discovered* nodes. A generalization is to not only mark the fact of discovery, but also the order. This information allows advanced reasoning as will be shown later on by the *Parenthesis Theorem* and the *White Path Theorem*, which in turn are needed for the correctness proof of Tarjan's algorithm as pictured in Section 4.7. Similarly, one adds the timing information for backtracking from a node, i. e., *finished* nodes. To enable gathering such information, one needs to also keep track of the current time, therefore a *counter* has to be added.

Search tree This category keeps track of the edges which have been traversed, differentiating between forward, cross, and back edges (cf. Section 4.1). While they can be generated from the timing information, we gather them explicitly. This ensures that there is no uncertainty about them representing the correct information. Also, due to the aforementioned ability to safely project the state, such additional information comes for free.

Combined, the state is thus defined as follows:

```
record 'v state =
  counter :: nat
  discovered :: 'v ⇒ nat option
  finished :: 'v ⇒ nat option
  pending :: 'v rel
  stack :: 'v list
  tree :: 'v rel
  back_edges :: 'v rel
  cross_edges :: 'v rel
```

Now, to help further use, we can already instantiate the type variable for the state in *gen_parameterization* to get a shorter type for parameterizations using our *state*:

```
type_synonym ('v,'es) parameterization
  = ('v, ('v,'es) state_scheme, 'es) gen_parameterization
```

We then start to specify depth-first search. The search needs two parameters: a graph, and the parameterization. We formalize graphs as a set of edges E and a set of initial nodes V_0[1]:

```
locale param_DFS = fb_graph E V₀
  for E :: 'v rel and V₀ :: 'v set +
  fixes param :: ('v,'es) parameterization
```

Inside this locale, the implementations for the search procedure are provided, using the just defined *state*. Thus, the discovery of a new node is implemented as:

[1] The *fb_graph* locale adds the assumption of the graph being *finitely branching*, i. e., each node having only a finite number of successors. This is a technical detail needed for being able to reason about ***foreach*** loops.

```
definition (in param_DFS) discover
  :: 'v ⇒ 'v ⇒ 'v state ⇒ 'v state
where
  discover u v s ≡ let
    d = (discovered s)(v ↦ counter s);
    c = counter s + 1;
    st = v#stack s;
    p = pending s ∪ {v} × E``{v};
    t = insert (u,v) (tree s)
  in s(| discovered := d, counter := c, stack := st, pending := p, tree := t |)
```

This implementation should not be surprising, as it updates the fields of the state in the expected way:

- record the discovery time of the new node and increase the timing counter
- push the new node onto the stack
- add all outgoing edges of the new node to the set of pending edges
- record the just taken edge (u is the predecessor of v) as forward edge

The other operations follow in a similar manner and will not be listed here.

Using those operations, we can finally define an instance of *gen_basic_dfs_struct*, where each of the fields of the struct is represented by an operation:

```
definition (in param_DFS) gbs ≡ (|
  gbs_init = λe. return (empty_state e),
  gbs_is_empty_stack = is_empty_stack,
  gbs_discover = λu v s. return (discover u v s),
  (* rest of fields omitted *)
|)
```

From this and the provided parameterization, it can then be shown that *param_DFS* is indeed a sublocale of *gen_param_dfs*: **sublocale** *gen_param_dfs gbs param* V_0 .

Example 4.3.1 (Cyclicity Checker continued)
From Example 4.2.1 on page 40, we recall our definition of the cyclicity checker as a parametrization to the DFS (fields omitted are *NOOP*):

```
definition cyc_checker where
  cyc_checker = (|
    on_init ≡ return (| cyc = False |),  (* initially no cycle has been found *),
    on_back_edge ≡ λu v s. return (| cyc = True |) (* cycle! *),
    is_break ≡ λs. cyc s (* break iff cycle has been found *)
  |)
```

We now instantiate our newly defined parameterized DFS with our formalization of cyclicity checking:

```
interpretation cyc!: param_DFS E V0 cyc_checker for E V0 .
```

As a result, we now have a fully initiated depth-first search: the search is completely specified by *param_DFS*, and the extension by *cyc_checker*. This enables us to refer to the final cyclicity checker by *cyc.dfs*.

4.4 Proof Architecture

Properties of the DFS algorithm are shown by establishing invariants, i. e., predicates that hold for all reachable states of the DFS algorithm. The standard way to establish an invariant is to generalize it to an inductive invariant, then show that it holds for the initial state, and is preserved by steps of the algorithm.

When using this approach naïvely, we face a problem: The invariant to prove an algorithm correct typically is quite complicated. Proving it in one go results in big proofs that tend to get unreadable and hard to maintain. For example, the version of Schimpf's verification of Tarjan's SCC algorithm (described in [49]) uses an invariant that spans more than 20 lines, including all the minor details that are needed throughout the proof. The proof in itself then uses another 1000 lines. Another example, from our own domain, is a previous version of this framework used in [10] (also see Section 4.8.2).

Such an approach is doable when dealing with only one algorithm (like it was the case for Schimpf), but poses a problem for our framework: The interesting properties for the framework are exactly those minor details which are encoded in the invariants (to keep with the example: e. g., properties about SCCs in itself). When using the framework one would expect to be able to use them in the proof, instead of having to prove them over and over again. Especially for a detailed state like ours (cf. Section 4.3), there is a large number of those invariants, like $\mathit{finished} \subseteq \mathit{discovered}$ or $\mathit{tree} \subseteq E$ or $\mathit{set\ stack} \cap \mathit{finished} = \{\}$. This set of problems will also be discussed in Section 4.8.2, where a previous version of the DFS framework is analyzed. Thus, we need a solution that allows us to establish invariants incrementally, re-using already established invariants to prove new ones.

A natural solution, which has already been chosen in the first version of the framework, is to phrase properties of the algorithm as properties over states, namely those states which can be reached during a run of the algorithm. That is, we fix an arbitrary state s for which we know it can be reached during the run and show the holding of the property. The difference to the general way of proving an inductive invariant here lies in having the fixed state a priori instead of obtaining it only during the proof. This allows the incremental proof setup as will be clear in a moment.

But before, we have to take one feature of the refinement framework into account that has been glimpsed over so far: Failure. As our refinement framework allows for failing assertions, we have to handle them, for we cannot establish any invariants if the algorithm may reach a failing assertion. Thus we can only establish invariants of the base algorithm under the assumption that the hook functions do not fail (*dfs* does not contain any assertions that may fail).

However, we would like to use invariants of the base algorithm to show that the whole algorithm is correct, in particular that the hook functions do not fail. Otherwise, we would have to re-prove any properties about depth-first search needed throughout the non-failure proof.

While this may be doable for small algorithms, it is a larger burden for more complicated algorithms like Nested DFS. But in both cases, it is contrary to the framework's goal of eliminating the need for duplicity. Therefore we need a solution that allows us to establish invariants for the non-failing reachable states only, and a mechanism that later transfers these invariants to the actual algorithm.

For this, we introduce another refinement relation[2] $\leq_n$ where

$m \leq_n m' \equiv m \neq FAIL \longrightarrow m \leq m'$.

Thus, $m \leq_n$ ***spec*** Φ expresses that m either fails or all its possible values satisfy Φ.

Using this new relation, we can exactly define when a property is an inductive predicate in our framework:

definition (***in*** *param_DFS*) *is_invar* ***where***
is_invar P $\equiv$
init $\leq_n$ ***spec*** *P*
$\wedge$ ($\forall$*s. P s* $\wedge$ *cond s* $\longrightarrow$ *step s* $\leq_n$ ***spec*** *P*).

Thus, any lemma showing a certain property P of the algorithm inductively is always of the form *is_invar P*.

Combining this definition of an inductive invariant with the idea of phrasing properties of the algorithm as properties over reachable states, we have to alter the latter slightly by restricting ourselves to states which can only be reached without any hook functions failing. We express the property of a reachable state without failing by the predicate *rwof* (**r**eachable **w**ith**o**ut **f**ailure):

inductive (***in*** *param_DFS*) *rwof* ***where***
init: $[\![$ *init = RES X; x*$\in$*X* $]\!] \Longrightarrow$ *rwof x*
| *step:* $[\![$ *rwof x; cond x; step x = RES Y; y*$\in$*Y* $]\!] \Longrightarrow$ *rwof y*

Of course, it trivially holds that *is_invar rwof*. Indeed, *rwof* is the *most specific invariant*, with any other invariant being a consequence thereof. Hence, we can show the following alternative definition of *is_invar*:

lemma (***in*** *param_DFS*) *is_invar_alt_rwof_def:*
is_invar P $\equiv$ $\forall$*s. rwof s* $\longrightarrow$ *P s*.

Any property P formalized in this way as *is_invar P* can then be discharged to apply to a specific state via:

lemma (***in*** *param_DFS*) *use_invar:*
$[\![$ *is_invar P; rwof s* $]\!] \Longrightarrow$ *P s*

In order to establish invariants of the algorithm, we show that they are inductive invariants when combined with *rwof*. This leads to the following rule:

lemma (***in*** *param_DFS*) *establish_invar:*
assumes *init* $\leq_n$ ***spec*** *P*
assumes $\bigwedge$*s.* $[\![$ *cond s; rwof s; P s* $]\!] \Longrightarrow$ *step s* $\leq_n$ ***spec*** *P*
shows *is_invar P*

This rule then can be used in the proof of any property, because, as stated earlier, the lemmas have the form *is_invar P*. It therefore serves as an introduction rule, i. e., it replaces the goal by its instantiated assumptions. In particular, during the process of proving a certain property we can use the aforementioned rule *use_invar* to use any other already proven property P'. This can be best expressed with an example:

[2] As one reviewer of our CPP paper [29] has remarked, this is not a partial order, even though the impression might arise from the usage of $\leq$. However, it is reflexive, and fulfills a restricted transitivity law: $a \leq_n RES\ X \leq_n c \implies a \leq_n c$

Example 4.4.1
Assume we have already proven some property *P'* as being an invariant:

lemma *P'_is_invar:*
is_invar P'

Now we would like to prove some other property *P* to be invariant, and make use of *P'* during that proof:

lemma *P_is_invar:*
is_invar P
proof (*rule establish_invar*)
fix *s*
assume *cond s* ***and*** *rwof s* ***and*** *P s*
from *P'_is_invar* ***and*** `*rwof s*` ***have*** *P' s* ***by*** (*rule use_invar*)

During the rest of the proof we can thus utilize the fact *P' s*. The same can be done for the *init* case, as *rwof init* holds trivially.

In order to use invariants to show properties of the algorithm, we can utilize the fact that at the end of a loop, the invariant holds and the condition does not:

lemma (***in*** *param_DFS*) *dfs_correct:*
dfs $\leq$ ***spec*** *s. rwof s* $\wedge$ $\neg$ *cond s*

Finally, we use the following rule to show that the algorithm does not fail:

lemma (***in*** *param_DFS*) *establish_nofail:*
assumes *init* $\neq$ *FAIL*
assumes $\bigwedge$*s.* $[\![$*cond s; rwof s*$]\!]$ $\Longrightarrow$ *step s* $\neq$ *FAIL*
shows *dfs* $\neq$ *FAIL*

To simplify re-using and combining of already established invariants, we define a locale *DFS_invar* as a sublocale of *param_DFS*, which fixes a state *s* and assumes that the most specific invariant holds on this state:

locale *DFS_invar* = *param_DFS E* V_0 *param*
for *E* V_0 *param* +
fixes *s*
assumes *rwof: rwof s*

Using the assumption that *rwof* holds, we can provide a version of *use_invar* with that assumption (read: future proof obligation) discharged and the state variable instantiated with the explicit state *s*:

lemmas (***in*** *DFS_invar*) *make_invar_thm* = *use_invar*[*OF _ rwof*]

Whenever we have established an invariant *P*, we also prove *P s* inside this locale. That is, after proving a theorem *is_invar P*, we lift this into the locale *DFS_invar* as another theorem *P s*. In a proof to establish some invariant, we may then interpret this locale. That way, any already established invariants of the form *P s* are available throughout this proof. This is useful in particular when dealing with multiple states at the same time, when we know for all of them that *rwof* holds: We can interpret the locale for each of the states yielding the right lemmas with the appropriate state being instantiated.

It also allows for proving properties that are consequences of other invariants and do not need the whole inductive proof-setup.

Example 4.4.2
In our parameterized DFS framework, we provide a version of *establish_invar* that splits over the different cases of *step*, and is focused on the hook functions:

lemma (***in*** *param_DFS*) *establish_invar:*
 assumes *init: on_init* $\leq_n$ ***spec*** *x. P* (*empty_state x*)
 assumes *new_root:* $\bigwedge v_0\ s\ s'$. *pre_on_new_root* $v_0\ s\ s'$
  $\Longrightarrow$ *on_new_root* $v_0\ s' \leq_n$ ***spec*** *x. P* $(s'(\!|more := x|\!))$
 assumes *finish:* $\bigwedge u\ s\ s'$. *pre_on_finish u s s'*
  $\Longrightarrow$ *on_finish u s'* $\leq_n$ ***spec*** *x. P* $(s'(\!|more := x|\!))$
 assumes *cross_edge:* $\bigwedge u\ v\ s\ s'$. *pre_on_cross_edge u v s s'*
  $\Longrightarrow$ *on_cross_edge u v s'* $\leq_n$ ***spec*** *x. P* $(s'(\!|more := x|\!))$
 assumes *back_edge:* $\bigwedge u\ v\ s\ s'$. *pre_on_back_edge u v s s'*
  $\Longrightarrow$ *on_back_edge u v s'* $\leq_n$ ***spec*** *x. P* $(s'(\!|more := x|\!))$
 assumes *discover:* $\bigwedge u\ v\ s\ s'$. *pre_on_discover u v s s'*
  $\Longrightarrow$ *on_discover u v s'* $\leq_n$ ***spec*** *x. P* $(s'(\!|more := x|\!))$
 shows *is_invar P*

Here, the *pre_*-predicates define the preconditions for the calls to the hook functions. For example, we have

pre_on_finish u s s' $\equiv$ *DFS_invar s* $\wedge$ *cond s*
 $\wedge$ *stack s* $\neq$ [] $\wedge$ *u* = *hd* (*stack s*)
 $\wedge$ *pending s* `` $\{u\} = \{\}$ $\wedge$ *s'* = *finish u s.*

That is, the invariant holds on state *s* and *s* has no more pending edges from the topmost node on the stack. The state *s'* emerged from *s* by executing the *finish*-operation on the base DFS state.

A typical proof of an invariant *P* has the following structure:

lemma *P_invar:*
 is_invar P
proof (*induction rule: establish_invar*)
 case (*discover u v s s'*)
 then interpret *DFS_invar s* ***by*** *simp*
 show *on_discover u v s'* $\leq_n$ ***spec*** *x. P* $(s'(\!|more := x|\!))$
  ...
next
 ...
qed
lemmas (***in*** *DFS_invar*) *P* = *P_invar*[*THEN make_invar_thm*]

The different cases that we have to handle correspond to the assumptions of the lemma *establish_invar*. The ***interpret*** command makes available all definitions and facts from the locale *DFS_invar*, which can then be used to show the statement. The second lemma just transfers the invariant to the *DFS_invar* locale, in which the fact *P s* is now available by the name *P*.

Note that this proof scheme is only suited for invariants with complex proofs. Simpler invariant proofs can often be stated on a single line. For example, finiteness of the discovered edges is proved as follows:

lemma *is_invar* (*λs. finite* (*edges s*))
by (*induction rule: establish_invar*) *auto*

Furthermore, we provide several special cases of *establish_invar* for the ease of the proving process, e. g., instances where handling cross and back edges has been collapsed.

4.4.1 Library of Invariants

In the previous section we have described the proof architecture, which enables us to establish invariants of the depth-first search algorithm. In this section, we show how this architecture is put to use.

Based on the extensive state given in Section 4.3, we provide a variety of invariants which use the information in the state at different levels of detail. Note that these invariants do not depend on the extension part of the state, and thus can be proven independently of the hook functions, which only update the extension part. Further note that we present them as they occur in the locale *DFS_invar*, which fixes the state and assumes that the most specific invariant holds.

Due to how maps work in Isabelle, we will use the shorthand notation $\delta\ s\ v$ for the discovery time of node v in state s, and $\varphi\ s\ v$ for the finishing time.

For the sets *dom* (*discovered s*) of discovered and *dom* (*finshed s*) of finished nodes, we prove, among others, the following properties:

lemma (***in*** *DFS_invar*) *disc_lt_fin:* $v \in$ *dom* (*finished s*) $\Longrightarrow \delta\ s\ v < \varphi\ s\ v$
lemma (***in*** *DFS_invar*) *stack_set_def: set* (*stack s*) = *dom* (*discovered s*) − *dom* (*finished s*)
lemma (***in*** *DFS_invar*) *finished_closed:* E``(*dom* (*finished s*)) $\subseteq$ *dom* (*discovered s*)
abbreviation (***in*** *DFS_invar*) *reachable* $\equiv E^*$``V_0
lemma (***in*** *DFS_invar*) *nc_finished_eq_reachable:*
¬*cond s* ∧ ¬*is_break s* $\Longrightarrow$ *dom* (*finished s*) = *reachable*

The first lemma states that for each finished node, the finishing time is smaller than the discovery time (the assumption is necessary, for there is no finishing time for unfinished nodes). The lemma thereafter states that the nodes on the stack are exactly those that have already been discovered, but not yet finished. The third lemma states that edges from finished nodes lead to discovered nodes, and the last lemma expresses that the finished nodes are exactly the nodes reachable from V_0 when the algorithm terminates without being interrupted.

We also prove more sophisticated properties found in standard textbooks (e. g., [5, pp. 606–608]), like the Parenthesis Theorem (the discovered/finished intervals of two nodes are either disjoint or the one is contained in the other, but there is no overlap) or the White-Path-Theorem (a node v is reachable in the search tree from a node u iff there is a white path from v to u, i. e., a path where all nodes are not yet discovered when v is).

lemma (***in*** *DFS_invar*) *parenthesis:*
  assumes $v \in dom\ (finished\ s)$ ***and*** $w \in dom\ (finished\ s)$
  and $\delta\ s\ v < \delta\ s\ w$
  shows $(*\ disjoint\ *)\ \varphi\ s\ v < \delta\ s\ w$
    $\vee\ (*\ v\ contains\ w\ *)\ \varphi\ s\ w < \varphi\ s\ v$

definition (***in*** *DFS_invar*) *white_path* ***where***
  white_path x y $\equiv x \neq y$
    $\longrightarrow (\exists p.\ path\ E\ x\ p\ y \wedge$
      $(\delta\ s\ x < \delta\ s\ y \wedge (\forall\ v \in set\ (tl\ p).\ \delta\ s\ x < \delta\ s\ v)))$

lemma (***in*** *DFS_invar*) *white_path:*
  assumes $v \in reachable$ ***and*** $w \in reachable$
  and $\neg cond\ s \wedge \neg is\_break\ s$
  shows *white_path v w* $\longleftrightarrow (v,w) \in (tree\ s)^*$

The Parenthesis Theorem is important to reason about paths in the search tree, as it allows us to gain insights just by looking at the timestamps:

lemma (***in*** *DFS_invar*) *tree_path_iff_parenthesis:*
  assumes $v \in dom\ (finished\ s)$ ***and*** $w \in dom\ (finished\ s)$
  shows $(v,w) \in (tree\ s)^+$
  $\longleftrightarrow \delta\ s\ v < \delta\ s\ w \wedge \varphi\ s\ v > \varphi\ s\ w$

This theorem expresses the relation between two discovered nodes v and w: There is a path in the search tree from v to w iff the discovery and finishing times of v create the eponymous parenthesis around the times of w.

From the location of two nodes in the search tree, we can deduce several properties of those nodes (e. g., the $\rightarrow$ direction of *tree_path_iff_parenthesis*). This can be used, for example, to show properties of back edges, as

lemma (***in*** *DFS_invar*) *back_edge_impl_tree_path:*
  $[\![\ (v,w) \in back\_edges\ s;\ v \neq w\ ]\!] \Longrightarrow (w,v) \in (tree\ s)^+.$

That is, for any back edge which is not a self-loop, there exists a path in the search for the other direction.

> **Example 4.4.3** (Cyclicity Checker: Proof)
> The idea of cycles in the set of reachable edges is independent of any DFS instantiation. Therefore we can provide invariants about the (a)cyclicity of those edges in the general library, the most important one linking acyclicity to the existence of back edges:
>
> ***lemma*** (***in*** *DFS_invar*) *cycle_iff_back_edges:*
> *acyclic* $(edges\ s) \longleftrightarrow back\_edges\ s = \{\}$
>
> Here, *edges s* is the union of all tree, cross, and back edges.
>
> The $\rightarrow$ direction follows as an obvious corollary of the lemma *back_edge_impl_tree_path* shown above. The $\leftarrow$ direction follows from the fact that *acyclic* $(tree\ s \cup cross\_edges\ s)$, the proof of which uses the Parenthesis Theorem.

Moreover, we need the fact that at the end of the search *edges s* is the set of all reachable edges:

lemma *nc_edges_covered:*
assumes ¬*cond s* ***and*** ¬*is_break s*
shows *E* ∩ *reachable* × *UNIV* = *edges s*

With those facts from the library, we recall the definition of the cyclicity checker in our framework as presented in Examples 4.2.1 and 4.3.1:

definition *cyc_checker* ***where***
cyc_checker = (|
on_init ≡ ***return*** (| *cyc* = *False* |), (∗ *initially no cycle has been found* ∗),
on_back_edge ≡ λ*u v s.* ***return*** (| *cyc* = *True* |) (∗ *cycle!* ∗),
is_break ≡ λ*s. cyc s* (∗ *break iff cycle has been found* ∗)
|)

interpretation *cyc!: param_DFS E* V_0 *cyc_checker* ***for*** *E* V_0 .

As the *cyc* flag is set when a back edge is encountered, the following invariant is easily proved:

lemma *i_cyc_eq_back:*
is_invar (λ*s. cyc s* ⟷ *back_edges s* ≠ {})
apply (*induct rule: establish_invar*)
apply (*simp_all add: cond_def cong: cyc_more_cong*)
apply (*simp add: empty_state_def*)
done

This happens to be the only invariant that needs to be shown for the correctness proof. Using the invariants mentioned above, we easily get the following lemma inside the locale *DFS_invar*, i. e., under the assumption *rwof s*:

lemma (***in*** *DFS_invar*) *cycc_correct_aux:*
assumes ¬*cond s*
shows *cyc s* ⟷ ¬*acyclic* (*E* ∩ *reachable* × *UNIV*)

Intuitively, this lemma states that the *cyc* flag is equivalent to the existence of a reachable cycle upon termination of the algorithm. Finally, we gain the correctness lemma of the cyclicity checker as an easy consequence:

lemma *cyc_correct:*
cyc.dfs E V_0 ≤ ***spec*** *s.*
cyc s ⟷ ¬*acyclic* (*E* ∩ *reachable* × *UNIV*).

Further examples of general properties (involving SCCs) are presented when proving the correctness of Tarjan's algorithm in Section 4.7.2.

4.5 Refinement

So far, we have described the general and the abstract framework. While the former defines the structure of the parameterized search algorithms, the latter adds an explicit state and

therefore allows to reason about its properties. But this is exactly where the scope of the abstract framework ends: It should only allow to help reasoning about algorithms implementing DFS, which is achieved by keeping a very high level view – therefore the designation *abstract*.

Of course, this is, in most cases, not the final phase. In particular for our goal of implementing a model checker, we need exectuable code. Directly executing the abstract definitions, given it was even possible, would yield very unfortunate timings, thereby rendering any actual use improbable.

The common way to handle this, and the reason the Refinement Framework was created, is to build a more optimized version of the algorithm (cf. Section 2.2), which can then be exported into executable code using the code generator of Isabelle/HOL [16]. This optimization process here can consist of different sub-steps, each optimizing a particular aspect of the algorithm. Such aspects include data refinement, i. e., replacing data structures by better fitted and more efficient versions (e. g., sets by lists or red-black trees), and structural refinement, i. e., optimizing the algorithm itself, for example by adding heuristics.

Both of those large aspects are also directly supported by the framework and detailed in this section. While those refinement steps are applied sequentially (first data refinement, then structural refinement, finally code generation), they are in themselves designed independent of one another: The first two are enabled by providing a library of possible refinements, which can be used in a sort of plug-and-play system. The last one, code generation, is done using the Autoref Tool [24] and thus done separately for each algorithm.

4.5.1 Data Refinement / Projection

To get a version of the algorithm over a state that only contains the necessary information, we use data refinement: We define a relation between the original *abstract* state, i. e., the state defined in Section 4.3, and the reduced *concrete* state, as well as the basic operations on the concrete state. Then we show that the operations on the concrete state refine their abstract counterparts. Using the refinement calculus provided by the Isabelle Refinement Framework, we lift this result to show that the whole concrete algorithm refines the abstract one.

In order to be modular w. r. t. the hook operations, we provide a library of standard implementations together with their refinement proofs, which are assuming a valid refinement for the hooks. As for the abstract state, we also use extensible records for the concrete state. Thus, we obtain templates for concrete implementations, which are instantiated with a concrete data structure for the extension part of the state, a set of concrete hook operations, and refinement theorems for them.

Such templates follow the approach for defining the search itself: At the bottom, we define data refinement for the generic search, followed by specializations to parameterized search and then our abstract search framework. For reasons of brevity, only the generic version will be described here, for the others follow straightforwardly.

First, let us define what a data refinement for a generic search is:

locale *gen_param_dfs_refine* =
c: gen_param_dfs gbs^i $param^i$ V_0^i +

a: gen_param_dfs gbs param V_0
for gbs^i $param^i$ V_0^i *gbs* $param^i$ V_0 +
fixes $\mathcal{V}$ $\mathcal{S}$ $\mathcal{X}$

At the very bottom, we have two instances of the generic search: the **a**bstract version *gbs* and its **c**oncrete counterpart (i. e., implementation) gbs^i, along with the corresponding parameterization *param* and $param^i$, respectively. They are also accompanied by different representations of the set of starting nodes (V_0 vs. V_0^i). Moreover, we fix two relations $\mathcal{V}$, $\mathcal{S}$, and $\mathcal{X}$ for the nodes, the search parts of the states, and the extension part of the states, respectively. Let us denote the relation of the combined state by $\mathcal{S}_\mathcal{X}$.

The locale is further equipped with a set of assumptions, detailing the exact nature of those relations and also of the two search instances.

We have added the parameter $\mathcal{V}$ to specify a refinement relation on nodes. This might become necessary, when the representation of the nodes in the abstract model cannot (or only inefficently) be used verbatim in the concrete version. An instance is to use natural numbers abstractly, but machine integers on the concrete level. The assumptions for $\mathcal{V}$ then are:

assumes *bijective* $\mathcal{V}$
assumes $(V_0^i, V_0) \in \langle \mathcal{V} \rangle set\_rel$

That is, the node relation must be bijective, for it must not alter the graph. For our example of naturals versus machine integers, this also implies that it has to be ensured that no overflow occurs, i. e., the number of nodes must be representable as a machine integer. The second assumption states that the two sets of starting nodes must be sets, where each member of V_0^i must have its counterpart in the abstract V_0 under the $\mathcal{V}$ relation (and vice versa because of the additionally required bijectivity).

For the operations defined by the generic search, we require the versions of the concrete gbs^i to be refinements of the abstract version, i. e., the concrete operation must be behaviorally equivalent to the abstract one. As a simple first example, we take the requirement for *gbs_is_discovered* (please refer to Section 2.2 for some explanation on the syntax):

assumes $(gbs\_is\_discovered\ gbs^i, gbs\_is\_discovered\ gbs) \in \mathcal{V} \to \mathcal{S}_\mathcal{X} \to Id$

This means, that for any node and state passed to the concrete *gbs_is_discovered* gbs^i, it must yield the same result as their abstract counterparts passed to the abstract *gbs_is_discovered gbs*. Other tests like *gbs_is_empty_stack* follow suite.

For state modifying operations like *gbs_discover*, the requirement is similar, but we weaken the requirement by adding preconditions. The reason is that certain operations are only valid for some restrictions: To give an example, when refining the set of discovered nodes by a distinct list, refining the insertion operation by simple appending is only valid when it is guaranteed not to be called on already inserted nodes. Hence, the requirement for *gbs_discover* is formalized as follows, where *a.pre_discover* is the set of constraints, containing $\neg$ *gbs_is_discovered gds v s* amongst others:

assumes $\bigwedge u\ v\ u^i\ v^i\ s_0\ s\ s^i.$
$[\![a.pre\_discover\ u\ v\ s_0\ s;\ (u^i,u) \in \mathcal{V};\ (v^i,v) \in \mathcal{V};\ (s^i,s) \in \mathcal{S}_\mathcal{X}]\!]$
$\Longrightarrow gbs\_discover\ gbs^i\ u^i\ v^i\ s^i \leq \Downarrow \mathcal{S}_\mathcal{X}\ (gbs\_discover\ gbs\ u\ v\ s)$

As already pointed out, this meaning is similar to the simple tests: For any arguments passed to the concrete function, its result must have at least one abstract counterpart in the results of the abstract version applied to abstract arguments.

Such assumptions are added in a similar fashion for the extensions contained in the parameterization:

assumes $(is\_break\ param^i,\ is\_break\ param) \in \mathcal{S}_\mathcal{X} \to Id$

assumes $\bigwedge u\ v\ u^i\ v^i\ s_0\ s\ s^i\ s'\ s'^i.$
$[\![a.pre\_discover\ u\ v\ s_0\ s;\ (u^i,u)\in\mathcal{V};\ (v^i,v)\in\mathcal{V};\ (s^i,s)\in\mathcal{S}_\mathcal{X};$
$(s'^i,s')\in\mathcal{S}_\mathcal{X};\ \textbf{return}\ s' \leq_n gbs\_discover\ gbs\ u\ v\ s]\!]$
$\Longrightarrow on\_discover\ param^i\ u^i\ v^i\ s'^i \leq \Downarrow\mathcal{X}\ (on\_discover\ param\ u\ v\ s')$

It should be noted that, theoretically, requirements on the simple test may be equipped with additional restrictions too (e. g., for any call of *gbs_is_finished v s*, it can be validly assumed that *gbs_is_discovered v s*). So far, this has not been done, as the need has not arisen.

Having defined all those assumptions, the setup of the locale *gen_param_dfs_refine* is finished and we show that under those assumptions, the concrete *dfs* refines the abstract one under the relation on the states $\mathcal{S}_\mathcal{X}$.

lemma (***in*** *gen_param_dfs_refine*) *dfs_refine:*
$c.dfs \leq \Downarrow\mathcal{S}_\mathcal{X}\ a.dfs$

Furthermore, it is shown that for any reachable concrete state, there exists a corresponding abstract state – given that the abstract version does not fail:

lemma (***in*** *gen_param_dfs_refine*) *rwof_refine:*
assumes *nofail* $(a.dfs)$
assumes *c.rwof s*
shows $\exists s'.\ (s,s')\in\mathcal{S}_\mathcal{X} \wedge a.rwof\ s'$

Example 4.5.1 (Simple State)
We now give an example on how this can be put to use to define a data refinement inside the DFS Framework.

For many applications, such as the cyclicity checker from Examples 4.2.1 and 4.3.1, it suffices to keep track of the stack, the pending edges, and the set of discovered nodes. We define a state type

record *'v simple_state =*
stack :: ('v × 'v set) list
on_stack :: 'v set
visited :: 'v set

and a corresponding refinement relation $\mathcal{R}_\mathcal{X}$

$(s^i,s) \in \mathcal{R}_\mathcal{X} \leftrightarrow$
$stack\ s^i = map\ (\lambda u.\ (u,pending\ s\ ``\ \{u\}))\ (stack\ s) \wedge$
$on\_stack\ s^i = set\ (stack\ s) \wedge$
$visited\ s^i = dom\ (discovered\ s) \wedge$
$(more\ s^i,\ more\ s) \in \mathcal{X}.$

Note that we store the pending edges as part of the stack, and provide an extra field *on_stack* that stores the set of nodes on the stack. This is done with the implementation in mind, where cross and back edges are identified by a lookup in an efficient set data structure and the stack may be projected away when using a recursive implementation. Moreover, we parameterize the refinement relation with a relation $\mathcal{X}$ for the extension state.

Next, we define a set of concrete operations. For example, the concrete *discover* operation is defined as:

definition *discover*^{i}
 :: *'v* ⇒ *'v* ⇒ (*'v*,*'es*) *simple_state_scheme* ⇒ (*'v*,*'es*) *simple_state_scheme*
where
 discover^i *u v s* ≡ *s* (|
  stack := (*v*, *E*``{*v*}) # *stack s*,
  on_stack := *insert v* (*on_stack s*),
  visited := *insert v* (*visited s*) |)

It is straightforward to show that *discover*^{i} refines the abstract *discover*-operation:

lemma *discover_refine:*
 assumes (*s*^{i},*s*)∈ $\mathcal{R}_\mathcal{X}$
 shows *discover*^{i} *u v s*^{i} ≤⇓$\mathcal{R}_\mathcal{X}$ *discover u v s*

Finally, we can collect these operations into a *gen_basic_dfs_struct*, say *simple_gbs*, to define a new generic DFS:

definition *simple_gbs* ***where***
  simple_gbs = (|
   ...,
   discover = *discover*^{i},
   ... |)

Using this, we create a new sublocale of the *gen_param_dfs_refine* locale and thus present a data refinement of the original generic DFS. As this refinement would be not very meaningful (we would refine some unknown generic search to our simple search), we also fix the abstract part as being the parameterized search given in Section 4.3:

locale *simple_impl* = *p!: param_DFS E* V_0 *param* +
  gen_param_dfs_refine
 where *gbs* = *p.gbs*
  and *gbs*^{i} = *simple_gbs*
  and *param* = *param*
  and V_0 = V_0
  and $\mathcal{V}$ = *Id*
  and $\mathcal{S}_\mathcal{X}$ = $\mathcal{R}_\mathcal{X}$
 for *E* V_0 *param param*^{i} $\mathcal{X}$

The main outcome of this locale is that we obtained a new generic DFS (here: *c.dfs*), which is a data refinement of the original DFS of Section 4.3 (here: *p.dfs*):

lemma (***in*** *simple_impl*) *simple_refine:*
c.dfs $\leq \Downarrow \mathcal{R}_{\mathcal{X}}$ *p.dfs*

This can now be used to provide data refinement for the cyclicity checker. We therefore define the concrete state by extending the *simple_state* record with a flag, analogously to Example 4.2.1. The extension state will be refined by identity, i. e., the refinement relation for the concrete state is $\mathcal{R}_{Id}$, as this is the only useful option for such a simple extension. We also define a set of concrete hook operations (which look exactly like their abstract counterparts) and name the resulting parameterization *cyc_checker*^{i}:

record *'v cyc_state_impl = 'v simple_state +*
cyc :: bool

definition *cyc_checker*^{i} ***where***
cyc_checker^i = (|
on_init $\equiv$ ***return*** (| *cyc = False* |), (* *initially no cycle has been found* *),
on_new_root $\equiv \lambda u$. *NOOP*,
on_discover $\equiv \lambda u\ v$. *NOOP*,
on_finish $\equiv \lambda u$. *NOOP*,
on_back_edge $\equiv \lambda u\ v\ s$. ***return*** (| *cyc = True* |) (* *cycle!* *),
on_cross_edge $\equiv \lambda u\ v$. *NOOP*,
is_break $\equiv \lambda s$. *cyc s* (* *break iff cycle has been found* *)
|).

Using this implementation of the cyclicity checker, we combine it with the *simple_impl* refinement for the search:

interpretation *cyc_impl!: simple_impl E* V_0
where *param = cyc_checker*
and *param*^{i} = *cyc_checker*^{i}
and $\mathcal{X}$ = *Id*
for *E* V_0

Once this is done, the DFS framework gives us a cyclicity checker over the concrete state (*cyc_impl.dfs*), and a refinement theorem relating it to the abstract version defined before:

lemma *cyc_impl_refine:*
cyc_impl.dfs $\leq \Downarrow \mathcal{R}_{Id}$ *cyc.dfs.*

This yields, as a general property of refinements, also the correctness of the implementation:

lemma *cyc_impl_correct:*
cyc_impl.dfs E $V_0 \leq$ ***spec*** *s.*
cyc s $\longleftrightarrow \neg$*acyclic* $(E \cap (E^{*}``V_0) \times UNIV)$.

We also provide further implementations (not listed here), which require the hooks for back and cross edges to have no effect on the state. Thus the corresponding cases

can be collapsed and there is no need to implement the *on_stack* set. As an additional optimization we pre-initialize the set of visited nodes to simulate a search with some nodes excluded[3]. As an example, this is used in the inner DFS of the Nested DFS algorithm (cf. Section 4.6.2).

4.5.2 Structural Refinement

Up to now, we have represented the algorithm as a while-loop over a step-function. This representation greatly simplifies the proof architecture. However, it is not how one would implement a concrete DFS algorithm. As an example, checking the loop condition would require iteration over all root nodes each time. For this reason, we want to replace the general algorithmic structure of the depth-first search by something more specific. That is, we want to refine on the structure. This topic has already been introduced in Section 4.2.1.

We are interested in making the structural refinement of the algorithm independent of the data refinement, such that we can combine different structural refinements with different data refinements, without doing a quadratic number of refinement proofs. For this purpose the structural refinements are formalized in the generic setting (cf. Section 4.2.1). As it was described there, depending on the desired structure, we have to add some minimal assumptions on the state and generic operations. The resulting generic algorithms are then instantiated by the concrete state and operations from the data refinement phase, thereby discharging the additional assumptions. The consequence of this is a slight loss of modularity, as indeed each module for data refinement needs to link itself to all those structural refinements it is going to work with. But we argue that this is a bearable burden, as in most cases this should be covered by a simple ***sublocale/interpretation***. Of course, if the refinement introduces new generic operations, the data refinement also needs to provide an implementation for those.

To help using the framework, we provide two standard implementations which can be used in lieu of a naïve code export of the generic algorithm: A tail-recursive one and a recursive one. The recursive implementation uses a recursive function and requires no explicit stack. The tail-recursive implementation still requires a stack and is in its general form very similar to the generic algorithm from Section 4.2. The most notable differences are the replacement of a non-deterministic choice for the next root node by an explicit iteration over all the nodes, and a more efficient loop condition, which does not require to check all root nodes on each iteration.

Tail-Recursive Implementation

To create the tail-recursive version, we follow the approach of Section 4.2.1 and create a new locale, inheriting from *gen_param_dfs*:

locale *tailrec_impl = gen_param_dfs gbs param* V_0

This locale is also fitted with some more assumptions, but we will omit them here for the moment. In this locale, we then define our new tail-recursive implementation as given in Alg. 4.6, using the generic operations defined by the *gen_param_dfs*.

[3]This is an optimization that saves one membership query per node.

Algorithm 4.6 Tail-Recursive DFS Implementation

```
definition (in tailrec_impl) tailrec_dfs where
 tailrec_dfs ≡ do {
  s₀ ← do_init;

  foreach V₀ s₀ (λs. ¬break s)
   (λv₀ s.
    if is_discovered v₀ s then return s
    else
     s' ← do_new_root v₀ s;
     while s' (λs. ¬break s ∧ ¬is_empty_stack s)
      (λs. do {
       ((u,nxt), s') ← get_pending s;
       case nxt of
         None ⇒ do_finish u s'
       | Some v ⇒
        if ¬ is_discovered v s' then
         do_discover gds u v s'
        else
         if is_finished v s' then
          do_cross_edge u v s'
         else
          do_back_edge u v s'
  }))}
```

This implementation iterates over all root nodes. For each undiscovered root node, it calls *do_new_root* and then executes steps of the original algorithm until the stack is empty again. Note that we effectively replace the arbitrary choice of the next root node by the outer foreach-loop. In order for this implementation to be a refinement of the original generic algorithm, we have to assume that

1) the stack is initially empty, such that we can start with choosing a root node, and
2) the same root node cannot be chosen twice, so that we are actually finished when we have iterated over all root nodes.

In order to ensure 2), we have to assume that *do_new_root* sets the node to discovered, and no operation can decrease the set of discovered nodes.

But we have to recall the definition of the *do_* functions: They effectively chain the operations from the search (of *gbs*) with the operations of the parameterization:

```
definition do_new_root v₀ s ≡ do {
   s' ← gbs_new_root gbs v₀ s;
   e ← on_new_root param v₀ s';
   return s'(|more := e|)
  }
```

As the extensions of the parameterization cannot modify the search part of the state and are therefore uninteresting for structural refinement, we have to lift those assumptions onto the general search algorithm, i. e., on the *gbs_* functions. We thus declare them as direct assumptions in the locale, as it was already hinted at above:

assumes *init_empty_stack:*
 $\bigwedge$*es. gbs_init gbs es* $\leq_n$ ***spec*** (*gbs_is_empty_stack gbs*)
assumes *new_root_discovered:*
 $\bigwedge v_0$ *s. pre_new_root* v_0 *s*
  $\Longrightarrow$ *gbs_new_root gbs* v_0 *s* $\leq_n$ ***spec*** *s'.*
  $\{v_0\} \cup \{v.$ *gbs_is_discovered gbs s* $v\} \subseteq \{v.$ *gbs_is_disovered gbs s' v*$\}$
assumes *finish_incr:*
 $\bigwedge$ s_0 *s u. pre_finish u* s_0 *s*
 $\Longrightarrow$ *gds_finish gds u s* $\leq_n$ ***spec*** *s'.*
  $\{v.$ *gbs_is_discovered gbs s v*$\} \subseteq \{v.$ *gbs_is_disovered gbs s' v*$\}$
(∗ *and so on for the other operations* ∗)

With these assumptions, we can use the infrastructure of the Isabelle Refinement Framework to show that the algorithm *tailrec_dfs* refines the original *dfs*:

theorem (***in*** *tailrec_impl*) *tailrec_dfs:*
 tailrec_dfs $\leq$ *dfs*

Recursive Implementation

Similarly, we approach the recursive implementation of DFS by introducing a new locale:

locale *rec_impl = fb_graph E* V_0 *+ gen_param_dfs gbs param* V_0
 for *E* ***and*** V_0 *:: 'v set* ***and*** *gbs param*
 fixes *choose_pending :: 'v* $\Rightarrow$ *'v option* $\Rightarrow$ *'s* $\Rightarrow$ *'s nres*

Note that we introduce an additional generic operation *choose_pending*, which shall have the implied semantics of removing the passed edge (first argument, second argument) from the set of pending edges, if the second argument is different from *None*. Also, we explicitly refer to *fb_graph* in this refinement, because we make explicit use of the successors of a node later on.

With this, we carry on to define the algorithm itself (cf. Alg. 4.7 on the next page): As in the tail-recursive implementation, we iterate over all root nodes. For each undiscovered root node, we enter a recursive block. Intuitively, this block handles a newly discovered node: It iterates over its successors, and for each successor, it decides whether it induces a cross or back edge, or leads to a newly discovered node. In the latter case, the block is called recursively (*D* (*v*,*s''*)) on this newly discovered node. Finally, if all successor nodes have been processed, the node is finished.

Intuitively, this implementation replaces the explicit stack of the original algorithm by recursion, i. e., the stack is now represented as the call stack of the recursive block.

Apart from the two assumptions from *tailrec_dfs*, we need some additional assumptions to show that this implementation refines the original algorithm:

Algorithm 4.7 Recursive DFS Implementation

```
definition (in rec_impl) rec_dfs where
 rec_dfs ≡ do {
   s₀ ← do_init;

   foreach V₀ s₀ (λs. ¬break s) (λv₀ s.
      if is_discovered v₀ s then return s
      else do {
       s' ← do_new_root v₀ s;
       if break s' then return s'
       else
        rec (v₀, s') (λD (u,s). do { (* D represents the recursive call *)
         s' ← foreach (E``{u}) s (λs. ¬break s) (λv s. do {
          s' ← choose_pending u (Some v) s;
          if is_discovered v s' then
           if is_finished v s' then do_cross_edge u v s'
           else do_back_edge u v s'
          else do {
           s'' ← do_discover u v s';
           if break s'' then return s'' else D (v,s'')
          }
         });
         if break s' then return s'
         else do {
          s'' ← choose_pending u None s';
          do_finish u s''
         }
       }) (* end rec *)
   }) (* end foreach *)
}
```

3) The operation *gbs_new_root gbs* v_0 initializes the stack to only contain v_0, and the pending edges to all outgoing edges of v_0; the operation *gbs_discover gbs u* pushes u onto the stack and adds its outgoing edges to the set of pending edges; the *gbs_finish*-operation pops the topmost node from the stack.

4) The *get_pending*-operation of the original algorithm must have the form of (or can be refined to) selecting a pending edge from the top of the stack, if any, and then calling the operation *choose_pending* for this edge, where *choose_pending* removes the edge from the set of pending edges.

Again, these assumptions are added to the locale *rec_dfs* – but we omit them here.

Eventually we can again show the corresponding refinement theorem:

theorem (**in** *rec_impl*) *rec_dfs: rec_dfs* $\leq$ *dfs*

While this proof requires the state to contain a stack, it is not used by the recursive algorithm. Provided that the parameterization does not require a stack either, a data refinement can be chosen where the stack is omitted (e. g., by using the provided *simple_state_ns* refinement).

Note that the assumptions introduced by the two structural refinements are, in general, natural for any set of operations on a DFS state, though 4) is a bit technical. The advantage of this formulation, i. e., the introduction as assumptions for the structural refinement instead of providing specifications a priori, is its independence from the actual operations. Thus, the same formalization for a final algorithm can be used to derive implementations for all states and corresponding operations, which reduces redundancies, and even makes proofs more tractable, as it abstracts from the details of a concrete data structure to its essential properties.

Example 4.5.2
Recall the simple state from Example 4.5.1. The simple implementation satisfies all assumptions required for the tail-recursive and recursive implementation, independent of the parameterization. Thus, upon refining an algorithm to *simple_state*, we automatically get a tail-recursive and a recursive implementation, together with their refinement theorems. In case of the cyclicity checker, we get:

lemma *cyc_impl_refine':*
$cyc\_impl.tailrec\_dfs \leq \Downarrow \mathcal{R}_{Id}\ cyc.dfs$ ***and***
$cyc\_impl.rec\_dfs \leq \Downarrow \mathcal{R}_{Id}\ cyc.dfs$

4.5.3 Code Generation

After projection and structural refinement have been done, the algorithm is still described in terms of quite abstract data structures like sets and lists. In a last refinement step, these are refined to efficiently executable data structures, like hash-tables and array-lists. To this end, the Isabelle Collections Framework [28] provides a large library of efficient data structures and generic algorithms, and the Autoref-tool [24] provides a mechanism to automatically synthesize an efficient implementation and a refinement theorem, guided by user-configurable heuristics.

Note that we do this last refinement step only after we have fully instantiated the DFS-scheme. This has the advantage that we can choose the most adequate data structures for the actual algorithm. The fact that the refinements for the basic DFS operations are performed redundantly for each actual algorithm does not result in larger formalizations, as it is done automatically.

The benefit of our DFS-framework here lies in supporting the user by passing information up-front to Autoref and the Isabelle Refinement Framework, so that the whole approach is more automatic. An example for such information is the refinement relation for the used data refinement. For in-depth details on the process of automatic refinement, we refer to its documentation [24, 23]. Instead, we will finish the example of the cyclicity checker to illustrate the process.

Example 4.5.3
In order to generate an executable cyclicity checker, we start with the constant *cyc_impl.tailrec_dfs*, which is the tail-recursive version of the cyclicity checker, using

the *simple_state* (cf. Example 4.5.2). The state consists of a stack, an on-stack set, a visited set, and the *cyc*-flag. Based on this, we define the cyclicity checker by

```
definition cyc_checker where
 cyc_checker E V₀ ≡ do {
  s ← cyc_impl.tailrec_dfs E V₀;
  return (cyc s)}
```

To generate executable code, we first have to write a few lines of canonical boilerplate (omitted here) to set up Autoref to work with the extension state of the cyclicity checker. The executable version of the algorithm is then synthesized by the following Isabelle commands:

```
schematic_lemma cyc_impl:
 fixes V₀::'v::hashable set and E::('v×'v) set
 defines 𝒱 ≡ Id :: ('v × 'v) set
 assumes [unfolded 𝒱_def,autoref_rules]:
  (succi,E)∈⟨𝒱⟩slg_rel
  (V₀',V₀)∈⟨𝒱⟩list_set_rel
 notes [unfolded 𝒱_def, autoref_tyrel] =
  TYRELI[where R=⟨𝒱⟩dflt_ahs_rel]
  TYRELI[where R=⟨𝒱 × ⟨𝒱⟩list_set_rel⟩ras_rel]
 shows nres_of (?c::?'c dres) ≤⇓?R (cyc_checker E V₀)
 unfolding cyc_impl.tailrec_dfs_def[abs_def] cyc_checker_def
 by autoref_monadic
```

The first command uses the Autoref-tool to synthesize a refinement. The ***fixes*** line declares the types of the abstract parameters, restricting the node-type to be of the *hashable* typeclass. The next line defines a shortcut for the implementation relation for nodes, which is fixed to identity here, i. e., there should not be any translation on the nodes. The assumptions declare the refinement of the abstract to the concrete parameters: The edges are implemented by a successor function, using the relator *slg_rel*, which is provided by the CAVA automata library [26]. The set of start nodes is implemented by a duplication-free list, using the relator *list_set_rel* from the Isabelle Collections Framework.

Finally, the ***notes***-part gives some hints to the heuristics: The first hint causes sets of nodes to be implemented by array hash-tables (therefore the requirement to have nodes of class *hashable*). This hint matches the on-stack and visited fields of the state. The second hint matches the stack field, and causes it to be implemented by an array-list, where the sets of pending nodes are implemented by duplication-free lists of their elements. Again, the required datatypes and their relators *dflt_ahs_rel* and *ras_rel* are provided by the Isabelle Collections Framework.

Ultimately, the *autoref_monadic* method generates a refinement theorem of the shape indicated by the ***show***-part, where *?c* is replaced by the concrete algorithm, and *?R* is replaced by the refinement relation for the result.

Then we continue by defining a new constant for the synthesized algorithm, and also provide a refinement theorem with the constant folded. As this is a common operation, this is already provided by the Isabelle Refinement Framework:

concrete_definition *cycc_exec* ***uses*** *cycc_impl*

As the generated algorithm only uses executable data structures, the code generator of Isabelle/HOL [16] then can be used to generate efficient Standard ML code:

export_code *cycc_exec* ***in*** *SML*

4.6 An Application in Model Checking: Nested DFS

In the previous sections, we introduced the general setup of a framework for implementing depth-first search algorithms. In this section, we now want to detail the implementation of a Nested DFS algorithm inside this framework, as this algorithm is one choice for checking the language of the product automaton for emptiness, as was described in the introduction to this chapter.

4.6.1 Introduction to Nested DFS

The language of a Büchi automaton is empty if, and only if, there is a reachable cycle including an accepting state. Therefore the general idea of Nested DFS covers this behavior: use a DFS to find accepting states (the "blue DFS") and check whether one can build a cycle from there (the "red DFS").

The first Nested DFS proposal is due to Courcoubetis et al. [6] and is presented in Alg. 4.8[4]. Here the red search tries to find a path directly to the node it started from (*seed* in DFS-RED).

Note that the knowledge whether a node has been searched in a red DFS is shared globally by having the boolean *red* directly attached to the node (cf. line 11). This avoids searching subtrees that have been searched already in previous runs of the red DFS. Without this behavior, a plain DFS with linear runtime (in the size of the edges) is started

[4]The algorithm has been presented earlier in this chapter, as Alg. 4.3, albeit in a different way. The formalization in Alg. 4.3 was chosen to highlight the working of the framework. The current formalization in Alg. 4.8 on the other hand follows the literature.

Algorithm 4.8 Nested DFS by Courcoubetis et al. [6]

```
1:  procedure NESTED-DFS
2:      DFS-BLUE start

3:  procedure DFS-BLUE(s)
4:      s.blue ← true
5:      for all t ∈ SUCCS s do
6:          if ¬ t.blue then
7:              DFS-BLUE t
8:      if s ∈ 𝒜 then                ▷ s is accepting
9:          DFS-RED(s, s)

10: procedure DFS-RED(seed, s)
11:     s.red ← true
12:     for all t ∈ SUCCS s do
13:         if ¬ t.red then
14:             DFS-RED(seed, t)
15:         else if t = seed then
16:             report cycle
```

Algorithm 4.9 Nested DFS by Schwoon and Esparza [51]

```
1:  procedure DFS-BLUE(s)
2:      s.colour ← cyan
3:      for all t ∈ SUCCS s do
4:          if t.colour = cyan
                ∧(s ∈ 𝒜 ∨ t ∈ 𝒜) then
5:              report cycle
6:          else if t.colour = white then
7:              DFS-BLUE t
8:      if s ∈ 𝒜 then
9:          DFS-RED s s
10:         s.colour ← red
11:     else
12:         s.colour ← blue

13: procedure DFS-RED(s)
14:     for all t ∈ SUCCS s do
15:         if t.colour = cyan then
16:             report cycle
17:         else if t.colour = blue then
18:             t.colour ← red
19:             DFS-RED(t)

20: procedure NESTED-DFS
21:     DFS-BLUE start
```

for each accepting state. As the number of them is also linear, the resulting runtime for the whole algorithm would be quadratic. But with the approach shown in Alg. 4.8 the result is an overall linear runtime for the whole search: We traverse an edge two times in each colored DFS, once upon reaching the node and once upon backtracking.

Note that this omission of nodes visited in other red searches has the crucial prerequisite of triggering the red search while backtracking (see line 9). Running the red part directly in the discovery phase would result in an incomplete algorithm as possible cycles might be missed.

Enhancements of the basic Nested DFS are for example given by Holzmann et al. [19], where the red search succeeds if a path to the stack of the blue one has been found, or by Schwoon and Esparza [51], where also the blue search is utilizied for cycle detection.

For CAVA we chose to implement the version of Schwoon and Esparza, which is given in Alg. 4.9. The enhancement over the original approach is to also take the stack into consideration: Whenever we encounter a successor node which is currently coloured cyan (i. e., it is on the stack), we test whether the current node or its successor are accepting. If it is, we have succesfully identified a cycle (as the stack forms a path to the current node). Additionally, in DFS-RED the method introduced by [19] is also used, i. e., we check for a path onto the stack instead of looking exactly for the starting node.

Zhao et al. [58] lists and compares other implementations of Nested DFS, which are, in imperative implementations, faster than the one by Schwoon and Esparza. We nevertheless stuck to the latter, because it is quite straightforward to implement in a functional setting. Also, improvements that are cheap in an imperative setting (e. g., by adding additional flags for each node) can be expensive in the functional setting (e. g., flags would need some kind of dictionary structure, where lookups are expensive and memory consumption is heavier). Still, this has not been benchmarked by us.

4.6.2 Formalization – Inner DFS

From the structure of Nested DFS, it is obvious that one needs to instantiate the DFS framework twice: for the inner and the outer DFS. Taking a closer look at the inner DFS, one can notice that it is not necessary to use such a special case, but that the inner DFS represents something more general: It is a general safety checker, i. e., it looks whether there exists a reachable node with a certain property – in case of Nested DFS, the property would be lying on the stack.

Therefore we start by formalizing such a general property checker. For this instantiation we state that the extension state should be the path to a found node[5]:

```
record 'v fp0_state = 'v state +
  ppath :: ('v list × 'v) option
```

Here, *ppath* should then either be *None* if no node can be found, or *Some* (*p*,*v*), where v is the node found, and p is the path leading there. This path starts in V_0 and does not include v[6]. If one of the initial nodes V_0 already fulfills the search property, the path-element p is the empty list.

Using this state, we then assemble our search (hooks not mentioned are no-ops), where P represents the property we are searching for

```
definition fp0_params P ≡ ⦇
  on_init = return ⦇ ppath = None ⦈,
  on_new_root = λv0 s. return ⦇ ppath = (if P v0 then Some ([], v0) else None) ⦈,
  on_discover = λu v s. return ⦇ ppath = (
                if P v
                then (* v is already on the stack, so we need to pop it again *)
                  Some (rev (tl (stack s)), v)
                else None ) ⦈,
  is_break = λs. ppath s ≠ None ⦈
```

This instantiation can then be shown to fulfill its purpose:

```
locale fp0 = param_DFS E V0 (fp0_params P)
  for E V0 and P :: 'v ⇒ bool

lemma (in fp0) fp0_correct:
  dfs ≤ spec s. case ppath s of
        None ⇒ ¬(∃v0∈V0. ∃v. (v0,v) ∈ E* ∧ P v)
    | Some (p,v) ⇒ (∃v0∈V0. path E v0 p v ∧ P v)
```

Of course, this algorithm would work for a replacement of the inner DFS, but lacks an important property: It may visit a node that has already been visited by some other run of the inner DFS, thereby removing an advantage of Nested DFS. We counter this problem by adding a wrapper around the just defined algorithm. This wrapper will take the regular input of the safety checker plus an additional restriction set R. Here, R represents a set of nodes that are to be excluded from the search.

[5] *fp0* stands for "find path with length at least zero".
[6] This is the general path definition from Section 3.2. We use it for it allows easy concatenation.

The wrapper then returns the result of the search itself plus, if no node has been found, the union of the restriction set and the set of visited nodes (which just happens to fit the requirements of the inner DFS):

```
definition find_path0_restr where
 find_path0_restr E V0 P R ≡ do {
 s ← fp0.dfs (E ∩ (−R) × (−R)) (V0−R) P;
 case ppath s of
    None ⇒ return (Inl (R ∪ dom (discovered s))) (* new restriction *)
  | Some (vs,v) ⇒ return (Inr (vs,v)) (* path found *)
}
```

This shows that the restriction is done by removing the restriction set R both from the edges and the starting nodes. Of course, to still ensure correctness, the set R has to meet certain requirements:

- The restriction must be closed under E: $E \grave{}\grave{} R \subseteq R$. This ensures that no unrestricted nodes are "shielded off" by nodes in R. If this was not the case, there might be nodes fulfilling P, but only reachable by paths through R.
- The restriction must not contain any node fulfilling P: $Collect\ P \cap R = \{\}$.

Under these requirements, combined into a single predicate *restr_invar*, it can be shown that this restricted search is correct (for *restr_invar* to be an invariant, it is also demanded that it is also valid for the new updated restriction set):

```
lemma find_path0_restr_correct:
 assumes restr_invar E R P
 shows find_path0_restr E V0 P R ≤ spec r. case r of
     Inl R' ⇒ R' = R ∪ E*``V0 ∧ restr_invar E R' P
   | Inr (vs,v) ⇒ P v ∧ (∃v0∈V0−R. path (E ∩ (−R) × (−R)) v0 vs v)
```

Note, that $R' = R \cup E^*\grave{}\grave{}V_0 \wedge Collect\ P \cap R' = \{\}$ entails $\neg(\exists v_0 \in V_0.\ \exists v.\ (v_0,v) \in E^* \wedge P\ v)$.

This is then further extended into *find_path1_restr*, which bears the additional restriction that the final path must be at least of length 1. This is necessary for Nested DFS. We will omit its definition here.

4.6.3 Formalization – Outer DFS

Having laid the foundation for the inner "red" part of Nested DFS, we can continue with defining the outer part. Again, we begin by stating what the extension part of the state is going to look like. This time, it will consist of two parts: *red* being the set of nodes visited by red search, and *lasso*, which will be the counter-example we are looking for:

```
record 'v blue_dfs_state = 'v state +
 lasso :: ('v list × 'v list) option (* pr × pl *)
 red  :: 'v set
```

Note the form for *lasso*: it consists of the path to an accepting node (without this node), and the loop from that node to itself. Also, similarly to the formalization of the safety checker, it is embedded in an option to signal the non-existance of such a counter-example.

Again, we then start a new locale for the DFS[7]:

```
locale BlueDFS = fb_graph E V0
  for E and V0 :: 'v set +
  fixes accpt :: 'v ⇒ bool
```

Note the additional parameter *accpt* representing the acceptance property on nodes.

Now, inside this locale, we can combine our definition of state with the safety property checker to gain our equivalent to running the red search:

```
abbreviation (in BlueDFS) red_dfs R P x ≡ find_path1_restr E {x} P R
```

Here, $\{x\}$ is taken as the value for V_0 of the safety checker, because this is indeed the only node where paths should begin. Then we combine it further to also interpret the return value correctly:

```
definition (in BlueDFS) run_red_dfs where
 run_red_dfs u s ≡ case lasso s of
    None ⇒ do {
      redS ← red_dfs (red s) (λx. x = u ∨ x ∈ set (stack s)) u;
      return (mk_blue_witness s redS)
      }
  | _ ⇒ NOOP s
```

With this definition, we only call the red DFS if we have not found a counter-example yet. Also, we state in the property to check for that we are looking for a node which is either on the stack, or equals the current node. The latter part is needed, because it has already been popped from the stack when calling this function. The function *mk_blue_witness* is used to interpret the result and construct a counter-example, if necessary. Its definition is as follows:

```
definition (in BlueDFS) mk_blue_witness where
  mk_blue_witness s redS ≡ case redS of
      Inl R' ⇒ (| lasso = None, red = R' |)
    | Inr (vs, v) ⇒ let rs = rev (stack s) in
        (| lasso = Some (rs, vs@dropWhileNot v rs), red = red s|)
```

This definition states that in case the safety checker just returns a new restriction set, we take it to represent our new set of nodes visited by red searches. In case we found some node v on the stack and some path vs leading to it, we build the lasso: The leading path (to the current node u) is the reversed stack. The loop is then made up of two parts, the path vs from u to v, and this part of the reversed stack between v and u. While this is not necessarily the shortest counter-example, it is the easiest one to construct.

We now can already instantiate our DFS framework to yield the outer DFS (omitted fields again are no-ops):

```
definition (in BlueDFS) blue_dfs_params where
blue_dfs_params = (|
   on_init = return (| lasso = None, red = {} |),
```

[7] One might notice, that this time the new locale does not extend *param_DFS*. This is due to the fact that we first need some setup to actually create the DFS definition. The sublocale relation is established later on.

on_finish = $\lambda u\ s.$ ***if*** *accpt u* ***then*** *run_red_dfs u s* ***else*** *NOOP s*,
is_break = $\lambda s.$ *lasso s* $\neq$ *None*,

But this only represents the Nested DFS version of Holzmann et al. [19]. To get the one of Schwoon and Esparza [51], we also need to handle back edges:

on_back_edge = *se_back_edge* ⦉

The function *se_back_edge* handles the cases of one of the ends of the edge being accepting. In any of those cases we get a lasso constructed by the stack and the back edge which is then created. Of course, nothing is done in case a counter-example has already been found:

```
definition (in BlueDFS) se_back_edge where
se_back_edge u v s ≡ case lasso s of
  None ⇒
    (* it's a back edge, so u and v are both on stack *)
    (* we differentiate whether u or v is the 'culprit'
      to generate a better counter example *)
    if accpt u then
      let rs = rev (tl (stack s));
        ur = rs;
        ul = u#dropWhileNot v rs
      in return ⦇ lasso = Some (ur,ul), red = red s ⦈
    else if accpt v then
      let rs = rev (stack s);
        vr = takeWhileNot v rs;
        vl = dropWhileNot v rs
      in return ⦇ lasso = Some (vr,vl), red = red s ⦈
    else NOOP s
| _ ⇒ NOOP s
```

To prove correctness for this formalization, we define three different invariants:

- In case we have not found a counter-example yet, we have to show that *red* is a restriction set fulfilling the requirements stated earlier, and that it only contains finished nodes:

  lasso s = *None* $\longrightarrow$ *restr_invar E* (*red s*) ($\lambda x.$ *x*$\in$*set* (*stack s*)) $\wedge$ *red s* $\subseteq$ *dom* (*finished s*)

- If we have not found a counter-example yet, there is no accepting finished node which is part of a cycle:

  lasso s = *None* $\longrightarrow$ ($\forall x.$ *accpt x* $\wedge$ *x* $\in$ *dom* (*finished s*) $\longrightarrow$ (*x*,*x*) $\notin E^+$))

- In the case where we have found a counter-example, it indeed is a lasso:

  $\forall$*pr pl. lasso s* = *Some* (*pr*,*pl*) $\longrightarrow$
  pl $\neq$ []
  $\wedge$ ($\exists v_0 \in V_0$. *path E* v_0 *pr* (*hd pl*))
  $\wedge$ *accpt* (*hd pl*)
  $\wedge$ *path E* (*hd pl*) *pl* (*hd pl*) (* *cycle* *)

After we have proven that these invariants hold on our definition (via *is_invar*), they can be combined into the correctness property of the algorithm:

case *lasso s* ***of***
  None $\Rightarrow \neg(\exists v_0 \in V_0.\ \exists v.\ (v_0,v) \in E^* \wedge accpt\ v \wedge (v,v) \in E^+)$
| *Some* $(pr,pl) \Rightarrow \exists v_0 \in V_0.\ \exists v.$
    $path\ E\ v_0\ pr\ v \wedge accpt\ v \wedge pl \neq [] \wedge path\ E\ v\ pl\ v$

Similarly to our approach on the safety property checker, we can create a definition outside of the locale to have a stand-alone function for Nested DFS and further lift the correctness property on this definition:

definition *nested_dfs E* V_0 *accpt* $\equiv$ ***do*** {
 s $\leftarrow$ *blue_dfs.dfs E* V_0 *accpt;*
 return (*lasso s*)
}

lemma *nested_dfs_correct:*
 assumes *fb_graph E* V_0
 shows *nested_dfs E* V_0 *accpt* $\leq$ ***spec*** *r.* ***case*** *r* ***of***
  None $\Rightarrow \neg(\exists v_0 \in V_0.\ \exists v.\ (v_0,v) \in E^* \wedge accpt\ v \wedge (v,v) \in E^+)$
| *Some* $(pr,pl) \Rightarrow (\exists v_0 \in V_0.\ \exists v.$
    $path\ E\ v_0\ pr\ v \wedge accpt\ v \wedge pl \neq [] \wedge path\ E\ v\ pl\ v)$

The development is finished by a straightforward refinement to an implementation and the eventual export to code.

4.7 An Advanced Application: Tarjan's Algorithm

The DFS framework has been developed as part of a model checker, but it is intended to be used in a broad range of applications. To showcase our framework in a more advanced setting than Nested DFS, we implemented Tarjan's Algorithm [54] for generating the set of strongly-connected components (SCCs) of a graph using the framework. While Tarjan's algorithm itself is actually used inside the model checker as part of the LTL to GBA translation [49], our implementation is not, although it would be feasible.

The algorithm has already been mentioned earlier in the introduction to depth-first search (Section 4.1). For a first overview we repeat the pseudo-code of the algorithm in Alg. 4.10 on page 73. This section is going to analyze the algorithm futher and to provide a general picture about its correctness. Thereafter, we detail the implementation inside our framework and show parts of the proof process leading to the final lemma of correctness.

The most important concepts of Tarjan's algorithm are the map *lowlink* and the accompanying "Tarjan stack" $stack_{tj}$: The "Tarjan stack" contains the current stack plus all nodes of all SCCs that are currently *in progress*. That is, any SCC where at least one node has been discovered, but the SCC is not explored fully, or at least it is unknown whether further contained nodes exist. Whenever the *root of an SCC* (the first discovered node of an SCC) is finished, the whole SCC is popped from the Tarjan stack: By construction, when a root of an SCC is about to be popped from the normal stack, all elements placed on top of it on the Tarjan stack form the corresponding SCC.

The *lowlink* map on the other hand represents the current most probable root for the SCC of each node: Each node is mapped to the identification of the corresponding node that is (at the current state of exploration) most likely to serve as the root. In our example the time of its discovery is used, because we have been using this concept already throughout our DFS formalizations. Other formalizations may choose an explicit index set, or any other means.

The restriction mentioned above, to only consider the current state of exploration, should be understood as: Until the SCC is completely determined, its root cannot be identified. Therefore nodes not equal to the root can (and will) remain equipped only with partial information. This is not an issue, because, as we will show later on, when the root of the SCC is to be finished the information in *lowlink* will be correct, i. e., that node will point to itself.

Initially (line 12) each node starts out by denoting itself as a possible root, because there is no knowledge yet about its successors. When encountering a back or cross edge (line 25) the choice is two-fold:

- When a back edge is encountered (i. e., $u \in \mathit{stack}$, which entails $\mathit{discovered}\ u \leq \mathit{discovered}\ x \wedge u \in \mathit{stack}_{tj}$) and it is not the top of the stack itself (i. e., a self-loop), the node u is taken as a new candidate for the root of the current SCC, given that we haven't found anything better yet. The latter is ensured by taking the minimum, i. e., the node highest up in the search tree.

- When a cross edge is encountered and the discovery time of the cross-edge's target (u) is greater or equal to the current top of the stack, nothing is done: Even if it were part of the same SCC it cannot be its root, for the root is defined as having the lowest discovery time of the SCC.

- When a cross edge is encountered and the edge's target (u) is not contained on the Tarjan stack anymore, its SCC has already been found. Therefore the current node cannot be part of the same SCC and nothing is done.

- When a cross edge is encountered and both of the previous points do not apply, the target node u is part of the same SCC as the current node. Therefore it serves as a candidate for the SCC's root.

On backtracking (line 17), the current information about the SCC's root is propagated back up the stack, that is the lowlink of the node on the stack eventually is the minimum of the lowlink of its successors and its own discovery time.

When, during backtracking, the lowlink of a node is equal to its own discovery time (line 21), the current node is a root of an SCC and therefore the other nodes of the SCC can be collected from the Tarjan stack (line 22).

4.7.1 Implementation in the Framework

For the course of this section, we apply some short-hand notation: Similarly to writing $\delta\ s\ v$ for the discovery time of node v in state s and $\varphi\ s\ v$ for the finishing time (cf. Sec.4.4.1), we will write $\zeta\ s\ v$ when referring to the lowlink value[8] of node v in state s.

[8] ζ looks somewhat similar to an l.

Algorithm 4.10 Tarjan's Algorithm (repetition of Alg. 4.4)

```
1:  discovered ← {}
2:  stack ← []
3:  stack_tj ← []
4:  sccs ← {}
5:  lowlink ← {}
6:  time ← 0

7:  procedure DFS(u)
8:      if u ∉ discovered then
9:          stack ← PUSH u stack
10:         stack_tj ← PUSH u stack_tj
11:         discovered ← discovered ∪ {(u, time)}
12:         lowlink ← lowlink ∪ {(u, time)}
13:         time ← time + 1
14:         for all v ∈ SUCCESSORS u do
15:             DFS v
16:         stack ← POP stack

17:         if stack ≠ [] then
18:             let x = TOP stack
19:             let t' = MIN (lowlink x)(lowlink u)
20:             lowlink ← lowlink ∪ {(x, t')}
21:         if lowlink u = discovered u then                    ▷ Root of SCC
22:             let (tj, scc') = COLLECT AND POP EVERYTHING UNTIL u stack_tj
23:             stack_tj ← tj
24:             sccs ← sccs ∪ {scc'}
25:     else
26:         let x = TOP stack
27:         if discovered u < discovered x ∧ u ∈ stack_tj then
28:             let t' = MIN (lowlink x)(discovered u)
29:             lowlink ← lowlink ∪ {(x, t')}

30: procedure TARJAN
31:     for all v0 ∈ V0 do
32:         DFS v0
33:     return sccs
```

As usual, the implementation starts by specifying the search state, which equals the additional variables given in the pseudo-code:

```
record 'v tarjan_state = 'v state +
  sccs :: 'v set set
  lowlink :: 'v ⇒ nat option
  stack_tj :: 'v list
```

This state is initialized empty:

```
definition tarjan_init ≡ return ⦇ sccs = {}, lowlink = Map.empty, stack_tj = [] ⦈
```

On discovering a new node, that node is pushed onto the Tarjan stack and its discovery time is registered as an initial lowlink, as described earlier:

```
definition tarjan_disc where
  tarjan_disc v s ≡ return ⦇
                       sccs = sccs s,
                       lowlink = (lowlink s)(v ↦ δ s v),
                       stack_tj = v#stack_tj s⦈
```

The function for encountering a back or cross edge is also similar to Alg. 4.10:

```
definition tarjan_back where
  tarjan_back u v s ≡ (
    if δ s v < δ s u ∧ v ∈ set (stack_tj s) then
      let ul' = min (ζ s u) (δ s v)
      in return ⦇ lowlink := (lowlink s)(u↦ul'), ... ⦈
    else NOOP s)
```

The last phase, finishing the current top of the stack, is modeled as:

```
definition tarjan_fin where
  tarjan_fin v s ≡ do {
      let ll = (if stack s = [] then lowlink s
                else let u = hd (stack s) in
                  (lowlink s)(u ↦ min (ζ s u) (ζ s v)));

      if ζ s v = δ s v then do {
          assert (scc_root E s v (scc_of E v));
          (tjs,scc) ← tj_stack_pop (stack_tj s) v;
          return ⦇ stack_tj := tjs, sccs := insert scc (sccs s), lowlink := ll⦈
      } else do {
          assert (¬ scc_root E s v (scc_of E v));
          return ⦇ lowlink := ll, ... ⦈
      }}
```

Here, *tj_stack_pop* $stack_{tj}$ v takes care of popping all nodes from $stack_{tj}$ until (and including) v and returns the resulting stack and the set of popped nodes, building a new SCC. Its definition will be omitted. The two ***assert***s will be discussed later on.

From those building blocks, we construct our DFS implementation, together with its locale:

definition *tarjan_params* ≡ (|
on_init = tarjan_init,
on_new_root = tarjan_disc,
on_discover = λu. tarjan_disc,
on_finish = tarjan_fin,
on_back_edge = tarjan_back,
on_cross_edge = tarjan_back,
is_break = λs. False |)

locale *tarjan = param_DFS E* V_0 *tarjan_params*
for *E* V_0

Note that *is_break* is always *False* as the intention is to explore the whole graph and collect all SCCs.

Eventually, we wrap the "generated" *dfs* function from the *tarjan* locale into a simple function that, given a graph, should output the set of SCCs of that graph:

definition *tarjan E* V_0 ≡ ***do*** {
s ← *tarjan.dfs E* V_0*;*
return (*sccs s*) }

On it, we are going to prove the following correctness theorem:

lemma *tarjan_correct:*
tarjan E V_0 ≤ ***spec*** *sccs.* ∀*scc* ∈ *sccs.*
is_scc E scc
∧ ⋃*sccs* = *reachable E* V_0

4.7.2 Prerequisites for the Correctness Proof

To establish the correctness theorem about Tarjan's algorithm, we have to develop some basic concepts beforehand.

As laid out earlier in Section 4.4, the framework is meant to be extended easily, even with general properties about depth-first search without having to modify the original theories[9]. For this reason, the following parts are modeled as being part of *DFS_invar*, that is, any other DFS-based algorithm only needs to import the respective theory to gain access to those properties.

Root of an SCC

The first such concept is formalizing the *root of an SCC*, i. e., the node of an SCC with the highest position in the tree (or, equivalent: the lowest discovery time of the SCC). This is expressed as all discovered nodes of the SCC being reachable from the root in the search tree:

[9] Of course, exceptions apply for any properties that need additional (general) information. For those, either an extension to *param_DFS* and *DFS_invar* needs to be created, or the original must be extended.

definition (***in*** *DFS_invar*) *scc_root* ***where***
scc_root s v scc $\longleftrightarrow$ *is_scc scc*
$\wedge$ $v \in scc$
$\wedge$ $v \in dom\ (discovered\ s)$
$\wedge$ $scc \cap dom\ (discovered\ s) \subseteq (tree\ s)^*$ `` $\{v\}$

Of course, this entails the existence of a path in the search tree from the root of the SCC to x, for any discovered node x, asserted that x and the root are not identical:

lemma (***in*** *DFS_invar*) *scc_root_scc_tree_trancl:*
$[\![$ *scc_root s v scc*; $x \in scc$; $x \in dom\ (\delta\ s)$; $x \neq v$ $]\!]$
$\Longrightarrow$ $(v,x) \in (tree\ s)^+$.

It can also be shown that a root is unique:

lemma (***in*** *DFS_invar*) *scc_root_unique_root:*
$[\![$ *scc_root s v scc*; *scc_root s v' scc* $]\!]$
$\Longrightarrow$ $v = v'$

Utilizing the knowledge about the search tree, we can eventually show that a node of an SCC is the root iff it has the minimum discovery time of that SCC.

lemma (***in*** *DFS_invar*) *scc_root_iff_Min_disc:*
$[\![$ *is_scc scc*; $r \in scc$; $r \in dom\ (discovered\ s)$ $]\!]$
$\Longrightarrow$ *scc_root s r scc* $\longleftrightarrow$ $\delta\ s\ r = Min\ \{\delta\ s\ v \mid v \in scc \cap dom\ (discovered\ s)\}$

This is an important fact, and a future building block for the proof of our correctness theorem: It allows to deduce the root of the SCC from the set of discovery times of that SCC.

Another important property is that during the search the (determined) root of an SCC does not change. The following lemma proves that given our state s, in any possible future state s' the root remains stable. The assumptions model the "possible future", i. e., it is not built differently from s. Naturally, the root r must be discovered in s:

lemma (***in*** *DFS_invar*) *scc_root_transfer:*
assumes $r \in dom\ (discovered\ s)$
assumes *future:*
DFS_invar G param s'
$dom\ (discovered\ s) \subseteq dom\ (discovered\ s')$
$\forall x \in dom\ (discovered\ s).\ \delta\ s\ x = \delta\ s'\ x$
$\forall x \in dom\ (discovered\ s') - dom\ (discovered\ s).\ \delta\ s'\ x \geq counter\ s$
$tree\ s \subseteq tree\ s'$
shows *scc_root s r scc* $\longleftrightarrow$ *scc_root s' r scc*

Lowlink

The second concept to introduce before the correctness proof is a formalization of the *lowlink* that is used in Tarjan's algorithm. While in the main algorithm, the *lowlink* is a simple map, it lacks any semantics. Therefore we are going to define an expressive version of lowlink that can be used to define what lowlink-value any node will have at any point in time of the exploration.

For this, we develop the concept of a *lowlink_path*[10]:

definition (***in*** *DFS_invar*) *lowlink_path* ***where***
lowlink_path s v p w ≡ *path E v p w* ∧ *p* ≠ []
∧ (*last p*, *w*) ∈ *cross_edges s* ∪ *back_edges s*
∧ (*length p* > 1 ⟶
p!1 ∈ *dom* (*finished s*)
∧ (∀*k* < *length p* − 1. (*p!k*, *p!Suc k*) ∈ *tree s*))

From this definition, a *lowlink_path* is a (non-empty) path along the search tree – except for the final edge, which is either a cross or back edge. It denotes those paths that are inherently necessary to build non-trivial SCCs: Every non-trivial SCC needs a cross or back edge, for else there is no cycle.

We can then collect the set of nodes reachable via such paths, resulting in the *lowlink_set*:

definition (***in*** *DFS_invar*) *lowlink_set* ***where***
lowlink_set s v ≡ {*w* ∈ *dom* (*discovered s*).
v = *w*
∨ (*v*,*w*) ∈ E^+ ∧ (*w*,*v*) ∈ E^+ ∧ (∃*p*. *lowlink_path s v p w*)}

Here, the set *lowlink_set s v* denotes the set of possible candiates for the root of the SCC of v, given the current search state s. This time, we also include trivial one-node SCCs by having the additional condition $w = v$.

Finally, we define the property *LowLink s v* to be the minimum discovery time of all such possible candidates:

definition (***in*** *DFS_invar*) *LowLink s v* ≡ *Min* (*δ s* `` *lowlink_set s v*)

From the basic understanding of lowlink follows that it cannot point further down the tree, thus:

lemma (***in*** *DFS_invar*) *LowLink_le_disc:*
v ∈ *dom* (*discovered s*) ⟹ *LowLink s v* ≤ *δ s v*

A further intuition about lowlink is that whenever *LowLink s v* = *δ s v*, then v is a root of its SCC. Of course, this does not hold at any time: Initially, when no successors of v have been discovered, the equality holds trivially, while the implication does not.

We can show this intution in its own lemma where the assumptions reflect the state shortly before a node is popped from the stack, or where it has been popped already. This is not incidentally.

lemma (***in*** *DFS_invar*) *LowLink_eq_disc_iff_scc_root:*
v ∈ *dom* (*finished s*) ∨ (*stack s* ≠ [] ∧ *v* = *hd* (*stack s*) ∧ *pending s* `` {*v*} = {})
⟹ *LowLink s v* = *δ s v* ⟷ *scc_root s v* (*scc_of E v*)

The proof of this lemma is pretty straightforward in the ← direction, using the fact that *LowLink s v* ≤ *δ s v*. The → direction on the other hand is more involved: We need to show that every node of the SCC is reachable from v via a path in the search tree. This proof

[10]The formalization of paths is again the same introduced for automata (Section 3.2) and also the same as used earlier for Nested DFS (Section 4.6.2). That is, the predicate *path E v p w* denotes a path from v to w in E, where p contains all the nodes visited *except for the final node*. Thus w is not contained in p, given we do not visit it twice.

then makes heavy use of (consequences of) the Parenthesis Theorem, which allows to assume paths through the tree using timing information, for instance:

lemma (***in*** *DFS_invar*) *parenthesis_impl_tree_path:*
assumes $v \in dom\ (finished\ s)$ ***and*** $w \in dom\ (finished\ s)$
and $\delta\ s\ v < \delta\ s\ w$ ***and*** $\varphi\ s\ v > \varphi\ s\ w$
shows $(v,w) \in (tree\ s)^+$

A final important lemma is a transfer lemma, i. e., showing that the *LowLink* value of a node does not change, under certain conditions, when developping a state further:

lemma (***in*** *DFS_invar*) *LowLink_eqI:*
assumes *DFS_invar G param s'*
assumes *discovered s* $\subseteq$ *discovered s'*
assumes *lowlink_set s w* $\subseteq$ *lowlink_set s' w*
and *lowlink_set s' w* $\subseteq$ *lowlink_set s w* $\cup$ *X*
and $w \in dom\ (discovered\ s)$
and $\bigwedge x.\ [\![x \in X; x \in lowlink\_set\ s'\ w]\!] \Longrightarrow \delta\ s'\ x \geq LowLink\ s\ w$
shows *LowLink s w = LowLink s' w*

4.7.3 Correctness Proof

After the two building blocks of the formalization of roots of SCCs and an expressive lowlink variant in the previous subsection, we tackle the final correctness proof of Tarjan's algorithm in our framework. In the process, we will only highlight the final parts and skip over the (mostly) technical lemmas on leading to them.

We use the process described in Section 4.4 for establishing invariants. To ease reading, we will show here the invariants after being lifted into our locale *tarjan* declared above, instead of the ones for the actual proof[11].

sccs s are SCCs

Firstly, we show that, as an invariant, all SCCs found during the process are indeed strongly-connected components:

lemma (***in*** *tarjan*) *sccs_are_sccs:*
$scc \in sccs\ s \implies is\_scc\ E\ scc$

For the proof, we first need to recall the part of our definition of *tarjan_fin* where new SCCs were added to *sccs s*:

assert $(scc\_root\ s\ v\ (scc\_of\ E\ v))$;
$(tjs,scc) \leftarrow tj\_stack\_pop\ (stack_{tj}\ s)\ v$;
return $(\!|\ stack_{tj} := tjs, sccs := insert\ scc\ (sccs\ s), lowlink := ll\ |\!)$

The important part is the first line: we annotated an assertion that v is indeed the root of its SCC. The Refinement Framework allows to use those assertions in the proof process as

[11] Please refer to the aforementioned Section 4.4 for how those differ.

additional assumptions (cf. Section 2.2). Only later on, in an additional proof step, it has to be shown that the assertions hold.

The proof itself boils down to showing that the nodes popped from the $stack_{tj}$ are equivalent to *scc_of E v*. The $\subseteq$-direction follows from the Parenthesis Theorem and properties about the Tarjan stack including:

lemma (***in*** *tarjan*) *tj_stack_reach_hd_stack:*
$v \in set\ (stack_{tj}\ s) \implies (v, hd\ (stack\ s)) \in E^*$

For the $\supseteq$-direction we use another invariant stating that whenever a root of an SCC is finished, there are no nodes of the SCC left on the Tarjan stack:

lemma (***in*** *tarjan*) *no_finished_root:*
$[\![\ scc\_root\ s\ r\ scc;\ r \in dom\ (finished\ s);\ x \in scc\ ]\!]$
$\implies x \notin set\ (stack_{tj}\ s))$

Moreover we know that all other nodes of the SCC already must be finished due to the following lemma:

lemma (***in*** *tarjan*) *scc_root_finished_impl_scc_finished:*
$[\![\ v \in dom\ (finished\ s);\ scc\_root\ s\ v\ scc\ ]\!]$
$\implies scc \subseteq dom\ (finished\ s)$

And thirdly, it has been shown that finished nodes are either on the Tarjan stack or contained in one of the collected SCCs:

lemma (***in*** *tarjan*) *finished_ss_sccs$_{tj}$_stack:*
$dom\ (finished\ s) \subseteq \bigcup sccs\ s \cup set\ (stack_{tj}\ s)$

On combining those three properties, we can deduct that all other nodes of the SCC must either be removed from the Tarjan stack with the current step, or already be part of the collected SCCs. The latter is not possible, because due to the induction hypothesis, the set *sccs* can only hold complete SCCs, which would imply that also the root of that SCC is already collected and thereby *finished*. This is definitly not the case, as we are just in the process of finishing it. Also, as a node cannot be part of two different SCCs, and roots are unique, no other SCC or root can serve instead.

Thus, ultimately we can follow that the set of popped nodes equals the SCC of v.

Lowlink Equivalence

In the previous section, we introduced a semantically meaningful version of the *lowlink*: *LowLink*. We also showed that, if the *LowLink* is equal to the discovery time for a node, under certain assumptions, this node is a root of its SCC.

In our implementation of Tarjan's algorithm, we fill a *lowlink* map without any inherent meaning. We must therefore prove that the value of a node in that map is equal to its *LowLink* value, thereby showing the equivalence of those two representations:

lemma (***in*** *tarjan*) *lowlink_eq_LowLink:*
$x \in dom\ (discovered\ s) \implies \zeta\ s\ x = LowLink\ s\ x$

The definition of *LowLink*, and especially *lowlink_path* uses constructs from all possible phases of the DFS, in particular both *discovered* and *finished*. As a consequence, the proof

for the lemma is very large (about 500 lines), for we have to show the equivalence holds throughout all phases (in regular cases, lemmas only have to deal with one or two cases). Additionally, when processing one node, it influences the *LowLink* value of other nodes: For instance, when declaring a node as finished, the *lowlink_set* of other nodes higher up on the stack can increase, because all the back and cross edges originating from the finished node are now possible to be used as the final part of a *lowlink_path*.

For those reasons, the proof of the equivalence is large, convoluted and very technical. We are therefore omitting any further discussion of that proof.

The useful consequence stems from its combination with *LowLink_eq_disc_iff_scc_root*, yielding

lemma (***in*** *tarjan*) *lowlink_eq_disc_iff_scc_root:*
$v \in dom\ (finished\ s) \vee (stack\ s \neq [] \wedge v = hd\ (stack\ s) \wedge pending\ s \grave{`}\grave{`} \{v\} = \{\})$
$\Longrightarrow\ \zeta\ s\ v = \delta\ s\ v \longleftrightarrow scc\_root\ s\ v\ (scc\_of\ E\ v)$

Finalizing the Proof

To finalize the correctness proof, we first need to show that the algorithm does not fail (for the refinement notion of failing). As no possibly failing operations are used in the algorithm itself except for the assertions, it needs to be shown that they hold. Obviously, they follow from the lemma *lowlink_eq_disc_iff_scc_root*.

The correctness proof also included the property of covering all nodes during the search, i. e., $\bigcup sccs\ s = reachable$. This is proven with the following lemma, which we will include in full, as it is self-explanatory and mainly relies on the fact that we already know that the set of reachable nodes is equal to the set of finished nodes on completion (cf. 4.4.1):

lemma (***in*** *tarjan*) *nc_sccs_eq_reachable:*
assumes *NC:* $\neg$ *cond s*
shows *reachable* = $\bigcup$*sccs s*
proof
from *nc_finished_eq_reachable NC* ***have*** [*simp*]*: reachable = dom* (*finished s*) ***by*** *simp*
with *sccs_finished* ***show*** $\bigcup sccs\ s \subseteq reachable$ ***by*** *simp*

from *NC* ***have*** *stack s* = [] ***by*** (*simp add: cond_alt*)
with *stacks_eq_iff* ***have*** $stack_{tj}\ s = []$ ***by*** *simp*
with $finished\_ss\_sccs\_stack_{tj}$ ***show*** *reachable* $\subseteq \bigcup sccs\ s$ ***by*** *simp*
qed

Finally, we gain the correctness lemma from a combination of the properties above:

lemma *tarjan_correct:*
$tarjan\ E\ V_0 \leq tarjan\_spec\ E\ V_0$.

4.7.4 Concluding Remarks

In the previous sections, we have shown how Tarjan's algorithm can be implemented in our DFS framework and that its correctness can be proven. We have not shown, as it has not

been done yet, how the framework can be used to generate resulting code. While a trivial code export is straightforward (no additional non-deterministic functionality is used and the data structures involved are already covered in the Isabelle Collections Framework), an efficient code takes more work: Especially we are, for the first time, using the timing information and thus cannot simply map it away. Hence, the current data refinements provided by the framework (cf. Section 4.5.1) are not sufficient and additional refinements need to be developed.

Moreover, the proofs are in some instances unpolished and can be, very probably, fine-tuned and collapsed. Especially the parts about roots of SCCs and *LowLink* share proof-work, mainly when it comes to constructing paths between nodes in the search tree. A more abstract notion about roots of SCCs and lowlink might be desirable, reducing the need for tedious construction proofs.

Finally, this implementation of Tarjan's algorithm is not the first one inside the CAVA-Project: There already exists one by Schimpf [49], that is used for the LTL-to-GBA translation. Schimpf's version is not created using the DFS framework, relying instead on the Refinement Framework alone. As a consequence, the invariants about the algorithm are encoded in a twenty lines block, followed by more than 900 lines of proof showing all those invariants at the same time. While Schimpf's version is shorter as a whole (and does provide an additional implementation), we prefer the incremental way of building and proving the algorithm that is allowed by the DFS framework. In particular, refactoring parts of the proof for new versions of libraries or changed assumptions can require a whole rewrite in Schimpf's case (see also the discussion about a former version of the framework in Section 4.8.2 for related arguments).

4.8 Comparison to Previous Approaches

Our framework for formalizing depth-first search based algorithms that we presented so far is the result of multiple design iterations. In this section, we want to give an overview about the lessons learnt from the previous approaches and highlight the differences.

Unfortunately, we cannot point out differences to related work, as there are, to our knowledge, no similar frameworks existing. The other works so far are merely dealing with DFS directly, like the work of Nishihara and Minamide [40], where two variants of a basic DFS are given (one with explicit stack, one without) and their equality is shown. Furthermore a couple of basic invariants are proven and code export is possible. But there is neither parameterization (it can solely compute the set of reachable nodes) nor flexible representation of the graph: It is fixed as a list of pairs. Another basic approach is given by Pottier [45], where DFS is formalized in Coq to prove correct Kosaraju's algorithm for computing the strongly connected components. This formalization also allows for program extraction, but does not allow easy extension for use in other algorithms.

There are also instances of DFS inside the CAVA project, which are not covered by the framework. This includes a version of Nested DFS by Peter Lammich, and also his verified implementation of Gabow's SCC algorithm [27]. Another notable instance is the version of Tarjan's algorithm by Alexander Schimpf [49].

Having stated those usages of DFS directly, we go back to examine the differences, and the consequences those differences entail, of previous versions of the framework. The

versions we want to take as reference are the one described in [35] and the one mentioned (without going into details there) in [10]. Also, we compare to Lammich's templating approach as mandated for example in [27].

4.8.1 DFS-Framework, the ATX Approach

The version of the framework published in the ATX-Paper [35] is the first published approach to the idea of a framework for DFS-based algorithms. At its start, there was no support for refinement in Isabelle/HOL and the idea not incorporated in its design.

The general idea of this framework is not different from the one described so far: It adds extension points to the search algorithm, which can be used by implementations to enrich the search with their functionality. That is, a search-based algorithm is expressed as a record of implementations for those hooks:

record $('S, 'n)$ *dfs_algorithm* =
dfs_cond :: $'S \Rightarrow$ *bool*
dfs_action :: $'S \Rightarrow ('S,'n)$ *dfs_sws* $\Rightarrow$ $'n \Rightarrow 'S$
dfs_post :: $'S \Rightarrow ('S, 'n)$ *dfs_sws* $\Rightarrow$ $'n \Rightarrow 'S$
dfs_remove :: $'S \Rightarrow ('S, 'n)$ *dfs_sws* $\Rightarrow$ $'n \Rightarrow 'S$
dfs_start :: $'n \Rightarrow 'S$
dfs_restrict :: $'n$ *set*

The type variables $'n$ and $'S$ refer to the type of the nodes and the extension state, respectively. Also $('S,'n)$ *dfs_sws* refers to the search state, which will be described later on.

Problem: Deterministic hooks

While the exact naming and number of hooks differ (for instance, there is no differentiation between back and cross edges, both are covered by *dfs_remove*), the main difference is the result type of the hooks: While the current framework allows for non-deterministic results (the non-deterministic monad is part of the Refinement Framework), the hook functions of the ATX-version are deterministic. This has drastic consequences for the expressiveness of the framework: As an example, it is not possible for a Nested DFS algorithm in this framework to pass back a counter-example, as the counter-example of the inner DFS depends on the exact run. As the search algorithm itself is, like we will describe in a moment, formulated in a non-deterministic way, the resulting counter-example is also not deterministic.

Problem: Feature overload

Another difference is the parameter *dfs_restrict*: It allows to specify a set of nodes that the search will ignore. The motivation for this parameter was the Nested DFS algorithm, where the inner DFS does not look at nodes that were visited by any of the former runs of the inner DFS. This approach was chosen, as the alternative would have been to modify the graph instead – which was found not feasible, as in practice such an operation would be too costly. The disadvantage of this approach, namely wiring restriction into the core of

the algorithm, was the obfuscation of the properties of the DFS, since any property had to take this restriction into account. Examples of such properties are:

lemma *finished_implies_succs_discovered:*
dfs_constructable dfs s $\Longrightarrow$ *v* $\in$ *finished s* $\Longrightarrow$ *succs v* $-$ *dfs_restrict dfs* $\subseteq$ *discovered s*

lemma *start_restr_reach_discovered:*
assumes *constr: dfs_constructable dfs s*
and *stack: stack s* $\neq$ []
and *discovered: v* $\in$ *discovered s*
and *ne: v* $\neq$ *start s*
shows *start s* $\rightarrow_{\setminus dfs\_restrict\ dfs}{}^{+}$ *v*

Here, the latter lemma states that each discovered node which is not the start node is reachable in the graph *without visiting any of the restricted nodes*.

Problem: Non-abstract state

Similarly to the setup of the hooks, the idea of the DFS state is not far from the current framework. Again, we have a general state, which is extended by the final algorithm:

record *('S,'n) dfs_sws =*
start :: 'n
stack :: 'n list
wl :: 'n set list
discover :: ('n, nat) map
finish :: ('n, nat) map
counter :: nat
state :: 'S

The biggest difference to the current setup is the field *wl*, which encapsulates the waiting set for each node on the stack. This formulation stems from the earlier version of having a list of lists and formulate depth-first search as induction over the stack and the waiting list.

Going with this approach, instead of using a general *pending* set, induced a complex handling. For example, it was necessary to have lemmas detailing the relation between the nth position in the waiting set list with the nth position on the stack:

lemma *wl_subset_succs_all:*
dfs_constructable dfs s $\Longrightarrow$ $\forall n <$ *length* (*stack s*). *wl s ! n* $\subseteq$ *succs* (*stack s ! n*)

Or induction schemes on the stack, which, as a prerequisite, needed to show that *stack* and *wl* are of a certain form, which required substantial additional proof work for every application:

lemma *stack_wl_visit_induct*
assumes *stack s = x#xs* ***and*** *wl s = w#ws*
and *stack s' = e#x#xs* ***and*** *wl s' = succs e#*$(w - \{e\})$*#ws*
...

Problem: Proof by state construction

The search itself is expressed as a while-loop over a step-function. The latter returns for each step the set of the next possible steps. This formulation is then used to introduce the predicate *dfs_constructable dfs s*, stating that *s* is reachable from the starting state by iterations of *dfs_step*. This is therefore equivalent to the predicate *rwof* of the current framework, and also used for the same purpose: As already visible in the lemmas above, every property of the depth-first search fixes a state and assumes it is *dfs_constructable*. Unfortunately, the ATX-Framework also uses it in an unnecessarily complex way for specifying correctness properties. As an example, we want to take a look at the correctness property for a cyclicity checker inside this framework:

lemma *dfs_cycle_correct:*
 assumes $x \in V$
 shows *dfs cdfs x* $\leq$ ***spec*** *s. state s* $\longleftrightarrow$ *cycle*

From the looks of this property, there is nothing unusual to it: For the resulting state of the algorithm, the cyclicity flag should be set if, and only if, there is a cycle. But, as it turns out, *cycle* is not defined that way. Instead its definition is:

definition *cyclic* ***where*** *cyclic s* $\equiv$ *stack s* $\neq$ [] $\wedge$ *hd* (*stack s*)$\rightarrow^{+}$*hd* (*stack s*) $\wedge$ *state s*
definition *cycle* ***where*** *cycle* $\equiv \exists$*s. dfs_constructable dfs s* $\wedge$ *cyclic s*

That is, we stated the correctness by requiring the existence of a constructable state, such that there exists a cycle for the node on top of the stack. While this is not wrong, because we also show

lemma *cycle_is_cycle:*
 cycle $\longleftrightarrow$ ($\exists$*v. start* $\rightarrow^{*}$ *v* $\wedge$ *v* $\rightarrow^{+}$ *v*),

it is unnecessarily complicated: For a proof, we have to actually *construct* a state where *cyclic* holds and show that this state can be reached from the starting point. These proofs were a burden, for one had to construct parts of a search process. Thus it was deemed the wrong way of specifying properties about searches, and we chose a different way in the next version of the framework, the CAV version detailed in the following section.

Problem: Two incompatible ways to formulate invariants

As it turned out, this was not the only misconception. Another formalization mistake was adding the field *dfs_invar* to the record defining dfs-instances. Its purpose was to allow the instances to add properties about its extension state and then show "with a simple proof" that those properties actually hold. To reach this, we defined the additional predicate *dfs_preserves_invar*, stating that *dfs_invar* is an invariant of the algorithm. To keep the example of the cyclicity checker, its *dfs_invar* was defined as follows:

definition *dfs_invar s* $\equiv$ (
 (*stack s* $\neq$ [] $\longrightarrow$ $\neg$ *state s* $\longrightarrow$ ($\forall$*n* < *length* (*stack s*).
 (*succs* (*stack s* ! *n*) $-$ *wl s* ! *n*) $\cap$ *set* (*drop n* (*stack s*)) = {}))
 $\wedge$ (*stack s* $\neq$ [] $\longrightarrow$ ($\forall$*x* $\in$ *finished s.* $\forall$*y* $\in$ *set* (*stack s*). $\neg$ *x* $\rightarrow^{+}$ *y*))
 $\wedge$ ($\forall$*x* $\in$ *finished s.* $\neg$ *x* $\rightarrow^{+}$ *x*))

The corresponding proof for *dfs_preserves_invar* then covers about 100 lines. Thus, we built something we wanted to avoid from the beginning: large inductive invariants. This stems from the fact, that we overlooked the link between *dfs_constructable* and *dfs_preserves_invar*. That is, the link that we established between *rwof* and *is_invar* in the current framework (cf. Section 4.4): *dfs_constructable* is also an invariant, namely the most specific invariant. In particular it also implies all the specific properties of a DFS instance. Thus we got stuck with the split into using *dfs_constructable* for general properties about the search and the large *dfs_invar* for properties about the specific instance.

Problem: Reusing large libraries for simple topics

A third misconception was the idea that re-use is better than re-write. While being the very idea this framework builds upon, it turned out that for easier data structures, in Isabelle, re-write is better than re-use. The data structure of concern here is graphs. To avoid having to define our own abstraction over graphs, we used the abstraction by Noschinski [42, 41] and adapted it to our needs. We were not able to see that for those basic principles of graphs it would have been easier to start from scratch. Because, as it turned out, Noschinski's formalization was more general than ours, in particular it allowed multi-edges. To fit our needs, we thus had to put a façade in front of this formalization so that the complicated parts were hidden. This took over 700 lines of proofs, for instance we had to add conversions between paths defined by edges to paths defined by vertices (the latter is not sufficient on multi-edge graphs but useful for our use case). This is larger than the size of the complete graph formalization for the current framework.

Problem: Missing tooling support

Finally, the ATX version of the framework suffered from a limitation of the Refinement Framework of that time: It required that a refinement relation has to be single valued, i. e., that each valid concrete value must have *exactly* one abstract value. As a consequence, fields of the concrete state could not be dropped when there was no way to re-construct them from the remaining fields. Thus, any additional information in the state introduced a performance penalty – which had to be taken into consideration when designing the abstract state, as to not make the penalty too large.

Example 4.8.1 (Problem of Single Valued Abstraction)
The problem incured by the restriction of the Refinement Framework's limitation is best explained with a small example. Assume the abstract state consists of a stack and the timings for discovery and finishing. Now, the timing information is in general an advantage for formulating proofs on the abstract level (examples are given in Section 4.4.1), but seldomly needed by the algorithm itself. Therefore, gathering the timing information is unecessary and only costs performance without gain. Hence, one might be inclined to define a concrete state without this information and use it as a data refinement, just as described in Section 4.5.1.

But the single-value constraint of the Refinement Framework of that time does not allow such a refinement: Abstraction must yield at most one value for some concrete state.This is not possible for the case presented, as one cannot construct the timing information from the remaining data. It works in the current version of the

framework, for the Refinement Framework has been extended to lift the single-value restriction. Thus, abstraction may now yield more than one abstract state – in this example, all possible timings can be injected.

Conclusion

To conclude, the version of the framework presented in this section already was equipped with useful ideas that are to be re-used later. But sometimes connections between the introduced concepts have been overlooked, leading to unnecessary complexity.

4.8.2 DFS-Framework, the CAV Approach

After the experiences with the ATX-framework and its deficiencies (some of which just were due to oversights, as pointed out before), it was decided to rewrite from scratch. This time, the framework should be created around the Refinement Framework. While keeping the general idea of a search algorithm with extension points, a version building on explicit inductive invariants was chosen, borrowing from a formalization of Nested DFS by Lammich for CAVA (which in turn borrowed from the ATX version). This version is the one presented in [10].

The differences to the other approaches are best described by starting to look at the formalization of the DFS itself:

```
definition dfs_algo_body
 :: ('S, 'n) dfs_sws ⇒ 'n ⇒ ('S, 'n) dfs_sws nres
where
 dfs_algo_body s₀ v₀ ≡ rec (s₀,v₀) (λD (s, v). do {
     assert pre_dfs s v ∧ dfs_pre_cond dfs s v ∧ dfs_cond dfs (state s);
     if v ∈ discovered s then dfs_rem_step v s
     else do {
       s' ← dfs_disc_step v s;
       s'' ← foreach (E``{v}) s' (λs. dfs_cond dfs (state s)) (λ v s. D (s, v));
       assert ∃S. dfs_cond dfs (state s'') ⟶
                    inres (dfs_visit dfs s v) S
                  ∧ fe_inv_dfs (disc_step_upd v s S) v {} s''
                  ∧ dfs_fe_inv dfs (disc_step_upd v s S) v {} s'';
       s''' ← dfs_fin_step v s'';
       assert post_dfs s s''' v ∧ dfs_post_cond dfs s s''' v;
       return s'''
   }})
```

The first thing to notice is: We are now using a recursive implementation, the reason of which will be explained later. What is more interesting though, is that we assert certain invariants to hold. Those invariants (*pre_dfs*, *fe_inv_dfs*, and *post_dfs* for the general search; *dfs_pre_cond*, *dfs_fe_inv*, and *dfs_post_cond* for the instantiation) are the vehicles used to prove properties about the algorithm. That is, any property which is necessary must be a consequence of one of those invariants. In particular, to show something about the final

algorithm, it must be entailed by *post_dfs* and *dfs_post_cond*[12].

This approach is very friendly to use with the Refinement Framework, as the framework in essence is tailored towards invariant propagation.

Problem: Complex invariants and proofs

As everything, in general, must already be covered by the mentioned invariants, those invariants need to be very complex. As an example, the following predicate *common_inv* is part of all the aforementioned invariants:

definition *common_inv* ***where***
common_inv s v $\longleftrightarrow$
$v \in V$
$\wedge$ *distinct* (*stack s*)
$\wedge$ ($\forall$ *n* < (*length* (*stack s*)) $-$ *1*. (*stack s ! n*) $\in$ *succs* (*stack s ! Suc n*))
$\wedge$ (*stack s* $\neq$ [] $\longrightarrow$ *last* (*stack s*) = *start s*)
$\wedge$ (*stack s* = [] $\longrightarrow$ *start s* = *v*)
$\wedge$ (*stack s* $\neq$ [] $\longrightarrow$ *vwalk* (*rev* (*stack s*)) (*E*,*V*))
$\wedge$ *discovered s* $\subseteq$ *V*
$\wedge$ *finished s* $\cup$ *set* (*stack s*) = *discovered s*
$\wedge$ *set* (*stack s*) $\cap$ *finished s* = {}

This also influences the way the proofs are written. As the lemmas to show are of the form "invariant A implies invariant B after some action", and each of those invariants consist of different, often unrelated properties, their proofs inherit the same style. Most consist of different unrelated blocks to show that each of the subproperties of invariant B is actually fulfilled. Also, if during such a proof it turns out that either of the invariants is not sufficient (A is not strong enough, B is too strong) it has to be modified. This often leads to the existing proofs needing adaption. Compare this to the approach of the ATX-version and the current version, where each property stands for itself. In the latter case, easy extension is possible: For instance, adding properties about SCCs is done by defining what an SCC is, and then adding lemmas to prove certain properties are preserved throughout the run. On the other hand, adding such properties to this CAV-version would require modifying all the current invariants and also to touch a lot of existing proofs. Hence it would not be possible to add such additional concepts in additional theories.

Problem: Not mitigating most of the older problems

Apart from this change in concept, the version does not differ from the version before. Thus, it also inherited its drawbacks, namely the usage of a non-fitting graph representation and the non-ability to remove fields of the abstract state when refining into the concrete world.

[12]Of course, from a theoretical point of view, there is no real difference between these approaches, as one could encode something like *rwof* into this model. But what this section is about, is the *practical* usability from an ITP-view.

Excursus: Recursive formulation

Let us explain why a recursive formulation had been chosen, instead of the iterative one used in ATX and in the current version. The iterative approach is ideal to reason about single steps, thus allowing to phrase something like reachability of states very naturally (the transitive closure of the step-function). On the other hand, for the approach used here, in the CAV-version, it is important to keep some history. That is, we want to be able to relate states to some certain points in the past, so that the invariants can make use of those relations. For instance, it is quite natural to relate the state at the beginning of the dfs-body with the state at its end, i. e., the states marking the points in time *before* a particular node is being processed, and the one right *after* it is finished. A property relating those two would be, for example, that the stack must be the same for both[13].

Conclusion

This version served for learning how to write such a framework tailored towards the Refinement Framework. It also visualized the advantages of approaches with a most specific invariant, which allows easier modularization, as it does not need one to explicitly build large all-encompassing invariants. Especially for our use case to build a framework that supplies a large set of different properties about different aspects of an algorithm, monolithic invariants are inconvenient, especially when one searches for properties to use. This also concerns the proofs: Due to them being a large collection of subproofs for the different parts of invariants, they are cluttered and do not serve very well for giving insights into the general working of the algorithm. Hence, the approach used in this version works very good, especially due to the automation of the Refinement Framework, for algorithms which are not going to be extended. But it shows the mentioned weaknesses when used in a framework.

4.8.3 DFS-Framework, a Templating Approach

A third approach is mandated by Lammich: It builds around the general possibility of the Refinement Framework to automatically determine for some function to have the same properties (modulo data refinement) as some other function. Again, one starts with some abstract function as a basis for showing general properties. But, as the framework works best on syntactically similar functions, the specializations (i. e., the refinements) are created by copying the literal definition of the abstract version and modifying it where needed. If phrased right, the verification condition generator (VCG) of the Refinement Framework is then able to lift the properties of the abstract function onto the new one without much manual work. As this approach starts with some generic template that is copied for the specializations, we dub it the *Templating Approach*.

An example usage of this approach is presented by Lammich in [27] to formalize Gabow's algorithm for computing the strongly connected components of a graph [12]. We will use this example to show-case the templating approach. Because we are mainly

[13] This also serves as a good example why writing general invariants is hard: Of course, this only holds under the assumption that the search was not aborted in between. In case of abortion, the stack of the latter state would be an extension of the stack from the beginning.

Algorithm 4.11 Skeleton Algorithm

```
definition skeleton :: 'v set nres where
skeleton ≡ do {
  let D = {};
  r ← foreach^outer_invar V₀ D (λv₀ D₀. do {
    if v₀∉D₀ then do {
      let s = initial v₀ D₀;

      (p,D,pE) ← while^invar v₀ D₀ s (λ(p,D,pE). p ≠ []) (λ(p,D,pE).
      do {
        (* Select edge from end of path *)
        (vo,(p,D,pE)) ← select_edge (p,D,pE);

        case vo of
          Some v ⇒ (* Found outgoing edge to node v *)
            if v ∈ ⋃set p then (* Back edge: Collapse path *)
              return (collapse v (p,D,pE))
            else if v∉D then (* Edge to new node. Append to path *)
              return (push v (p,D,pE))
            else (* Edge to done node. Skip *)
              return (p,D,pE)
        | None ⇒  (* No more outgoing edges from current node on path *)
            return (pop (p,D,pE))
      }); (* end while *)
      return D
    } else return D₀
  }); (* end foreach *)
  return r
}
```

interested in the general way this approach is applied, we will not discuss the specific purpose of this algorithm. This is already covered by Lammich's article [27].

The algorithm is used and verified for two different goals: The first one is calculating the set of SCCs (as it was originally intended to do); the second goal is to be able to determine whether the language of a generalized Büchi automaton is empty. While it is possible to implement the latter just as an extension of the SCC-calculation, this is not very efficient, for it calculates far more information than is needed: If we find an SCC containing nodes from all of the acceptance classes of the GBA, there is no reason in carrying on calculating the following sets.

To counter this, Lammich presents two different algorithms, each tailored towards only one of the goals. Reasonably, he intends to share as much work as possible between those two algorithms. Thus, while not offering a full-fledged framework for general DFS, it presents an insight about other ways of how such a framework could be set up.

The Skeleton

As laid out above, Lammich starts with some abstract common denominator of the two algorithms (we want to follow Lammich's notation here and call this denominator the *skeleton algorithm*). This skeleton algorithm, given with Alg. 4.11 on the previous page, represents the algorithm for a *path based SCC algorithm*, which is an implementation of DFS that contracts found cycles into single nodes. It is to note that the skeleton algorithm itself does not act on the result – it just computes.

We explained that the templating approach relies on copying the original abstract function (i. e., the skeleton algorithm in this example). Therefore, there is no need for hooks or other kinds of parameterization – hence the functions referred to throughout this algorithm (*initial*, *select_edge*, *collapse*, *push*, and *pop*) are actual parts of the algorithm, but outsourced into auxiliary functions (whose definitions are omitted here).

Proving properties

Proving properties of the algorithm is supported by the Refinement Framework's ability to annotate invariants and assertions directly inside the function code. Those invariants are directly employed by the VCG in trying to solve encountered goals, at least partially. They are also presented as assumptions for visible goals, thereby yielding a well-defined interface.

This mechanism of annotation is used here for the two loops (additional assertions were omitted): The two annotated invariants are *outer_invar* and *invar* v_0 D_0. Again, those invariants are not parameters, but explicitly defined. But instead of regular definitions, the invariants are stated as locales. As an example, *outer_invar* is defined as follows, where *fr_graph* is another locale for graphs (with a finite set of reachable nodes), introducing the variables V_0 and E for initial nodes and edges, respectively:

locale *outer_invar* = (∗ *Invariant of the outer loop* ∗)
fr_graph +
fixes *it* :: *'v set* (∗ *Remaining nodes to iterate over* ∗)
fixes *D* :: *'v set* (∗ *Finished nodes* ∗)
assumes $it \subseteq V_0$ (∗ *Only start nodes to iterate over* ∗)
assumes $V_0 - it \subseteq D$ (∗ *Nodes already iterated over are visited* ∗)
assumes $D \subseteq E^{*}``V_0$ (∗ *Done nodes are reachable* ∗)
assumes $E``D \subseteq D$ (∗ *Done is closed under transitions* ∗)

Choosing a locale to express invariants has multiple advantages: For one, inside the locale the invariant itself is always an implicit assumption, thereby simplifying the lemmas. Second, locales can easily be extended and combined:

locale *some_more_specific_invar* = *outer_invar* + *some_other_invar* +
assumes ...

For convenience, lemmas defined inside the parent locale are available directly inside the child locale. This feature is used extensively in the templating approach, because in general each new refinement requires a new invariant, which of course needs to imply the one in the refined definition.

The usage of a locale has no immediate disadvantage, as a locale also introduces a predicate under the same name. This predicate expresses that the assumptions of the locale are met – which is the reason that the annotations work as presented.

For the presented skeleton algorithm, it can then be shown that the invariant *outer_invar* holds on the result:

```
theorem skeleton_outer:
 skeleton ≤ spec (λD. outer_invar {} D)
   unfolding skeleton_def select_edge_def select_def
   by (refine_rcg WHILEIT_rule[where R=abs_wf_rel v0 for v0])
     (vc_solve solve: invar_preserve simp: pE_fin' finite_V0)
```

From this follows that all consequences of this invariant also hold on the result, for example that the set of finished nodes is equal to all the reachable ones:

```
lemma fin_outer_D_is_reachable:
  outer_invar {} D  ⟹  D = E*``V0

corollary skeleton_is_reachable:
  skeleton ≤ spec (λD. D = E*``V0)
```

Specialization

After having defined the skeleton algorithm and having shown some general properties, Lammich continues and defines the two final algorithms. This works by taking the exact definition of the skeleton version and change those parts that are different. We only show one version here, the search for a counter-example in Alg. 4.12 on the following page.

It can be seen that the general structure of the algorithm is kept intact, as the refinement automatisms work best when for each step on the one side, there is exactly one step on the other side. One notable addition is the *brk* field: It either signals that no counter-example has been found yet (*brk = None*), or that one has been found (*brk = Some p*), where *p* contains information about the counter-example. Note that, due to the node collapsing in the algorithm, this is not simply the path, but needs to be expanded later on (details are omitted here).

The two invariants used here, *fgl_outer_invar* and *fgl_invar*, are defined in two different ways: The latter is an extension of the *invar*-locale from the original algorithm:

```
locale fgl_invar = invar G v0 D0 p D pE + igb_graph G
  for G v0 D0 brk p D pE +
  assumes (* No accepting cycle over visited edges *)
    brk = None ⟹
      ¬(∃v pl. pl≠[] ∧ path lvE v pl v ∧  (∀i<num_acc. ∃q∈set pl. i∈acc q))
  assumes brk = Some (Vr,Vl) ⟹ ce_correct Vr Vl
```

Recall from Section 3.2 that *igb_graph* is the locale introducing generalized Büchi automata, i. e., stating that *G* is a generalized Büchi automaton, and *ce_correct* is the correctness property of the counter-example (omitted here).

Algorithm 4.12 Specialization for counter-example search

```
definition find_ce :: ('v set × 'v set) option nres where
 find_ce ≡ do {
   let D = {};
   (brk,_)←foreach^fgl_outer_invar V₀ (None, D) (λ(brk,_). brk=None) (λv₀ (brk,D₀). do {
     if v₀∉D₀ then do {
       let s = (None,initial v₀ D₀);

       (brk,p,D,pE) ← while^fgl_invar v₀ D₀ s (λ(brk,p,D,pE). brk=None ∧ p ≠ []) (λ(_,p,D,pE).
       do {
         (* Select edge from end of path *)
         (vo,(p,D,pE)) ← select_edge (p,D,pE);

         case vo of
           Some v ⇒ do {
             if v ∈ ⋃set p then do {
               (* Collapse *)
               let (p,D,pE) = collapse v (p,D,pE);

               if ∀i<num_acc. ∃q∈last p. i∈acc q then
                 return (Some (⋃set (butlast p),last p),p,D,pE)
               else
                 return (None,p,D,pE)
             }
             else if v∉D then (* Edge to new node. Append to path *)
               return (None,push v (p,D,pE))
             else return (None,p,D,pE)
           }
         | None ⇒ (* No more outgoing edges from current node on path *)
           return (None,pop (p,D,pE))
       }); (* end while *)
       return (brk,D)
     }
     else return (brk,D₀)
   }); (* end foreach *)
   return brk
 }
```

Algorithm 4.13 Introducing acceptance sets

```
(* original version *)
definition push v PDPE ≡
  let
    (p,D,pE) = PDPE;
    p = p@[{v}];
    pE = pE ∪ (E∩{v}×UNIV)
  in
    (p,D,pE)
```

```
(* new, enhanced definition *)
definition gpush v s ≡
  let (a,s) = s
  in (a@[acc v],push v s)
```

The other invariant, *fgl_outer_invar*, is not an extension of the original *outer_invar* as *outer_invar* would not hold when a counter-example has been found and the algorithm aborts. Thus it is defined on its own, simply leveraging the original invariant:

```
definition fgl_outer_invar ≡ λit (brk,D). case brk of
    None ⇒ outer_invar it D ∧ no_acc_over D
  | Some (Vr,Vl) ⇒ ce_correct Vr Vl
```

Refinement to executable code

After having defined the specialization of the original skeleton algorithm to get an algorithm for finding counter-examples, Lammich continues with data-refinement: Step-by-step he introduces better suited data structures (or structures to gather more information) with the goal of having a basis that allows for an easy and effective refinement into an exectutable version. As the main data-manipulating functions are the auxiliary functions (i. e., *collapse*, *push* etc.), those are the parts mainly involved in the process: For example, a first step by Lammich is the introduction of explicit acceptance sets. Thus all auxiliary functions are equipped to deal with those, as is illustrated by the excerpt given in Alg. 4.13.

The new definitions are then shown to be refinements of the original versions:

```
lemma gpush_refine:
  ⟦(v',v)∈Id; (s',s)∈gstate_rel⟧ ⟹ (gpush v' s',push v s)∈gstate_rel
```

The algorithm itself only needs to be changed to use the newly defined functions, making the refinement proof of the overall algorithm very simple. Of course, some refinements are done to make the algorithm more efficient. When this is the case, the changes can be more invasive and the complexity of the resulting refinement proof increases too. For illustration, the final algorithm before code-generation (the code-algorithm is generated using the Autoref-Framework) is given by Alg. 4.14 on page 95.

Evaluation

The main advantage of Lammich's approach is a better variability in what extensions can change in the underlying skeleton algorithm. In our general DFS framework, the extension points are fix and everything that intends to use the foundation laid by the framework must implement its functionality inside this corsett. Quite often this is possible, but as

we learned when implementing Tarjan's algorithm, the algorithm needs to be massaged to fit this corset. With Lammich's approach, the boundaries are defined solely by the capabilities of the refinement framework and on how much work one intends to invest into the refinement proofs.

At the same time, this is also the greatest weakness of this approach: To know the boundaries of the Automatic Refinement Framework, a somewhat intrinsic knowledge about the internals of it are needed. While Lammich posseses them, for obvious reasons as the author of the framework, the general public may not. And then the approach can quickly lose the advantage of working in most ways automatically. While this is also possible with the DFS framework, as it equally relies on the Automatic Refinement Framework, the refinement proofs entailed by the extension points are, in general, simpler as what an arbitrary skeleton algorithm can produce.

A second disadvantage of the approach is harder maintainability: In Software Engineering, copy-and-paste is being frowned upon, and for good reason: Whenever it is needed to change something at one point, this change needs to be applied to all places where the code has been pasted into. It is not possible to abstract changes away, for there is no separation between some agreed-upon interface and internal implementation.

Algorithm 4.14 Implementation of counter-example search

```
definition find_ce_impl :: ('v set × 'v set) option nres where
 find_ce_impl ≡ do {
   let os = goinitial_impl;
   os←foreach^{λit os. fgl_outer_invar it (goGS_α os)} V_0 os go_is_no_brk_impl (λv_0 s_0.
   do {
     if go_is_done_impl v_0 s_0 then return s_0
     else do {
       let s = (None, ginitial_impl v_0 s_0);

       (brk,s)←while^{λ(brk,s). fgl_invar v_0 (oGS_α(goD_impl s_0)) (brk,snd(gGS_α s))} s
                (λ(brk,s). brk=None ∧ ¬gpath_is_empty_impl s)
                (λ(l,s).
       do {
         (* Select edge from end of path *)
         (vo,s) ← gselect_edge_impl s;

         case vo of
           Some v ⇒ do {
             if gis_on_stack_impl v s then do {
               s←gcollapse_impl v s;
               b←last_is_acc_impl s;
               if b then ce_impl s
               else return (None,s)
             }
             else if ¬gis_done_impl v s then (* Edge to new node. Append to path *)
               return (None,gpush_impl v s)
             else (* Edge to done node. Skip *)
               return (None,s)
           }
         | None ⇒ do {
             (* No more outgoing edges from current node on path *)
             s←gpop_impl s;
             return (None,s)
           }
       });
       return (gto_outer_impl brk s)
     }
   });
   return (goBrk_impl os)
 }
```

5 Model

The final aspect when talking about model checking is to discuss what is going to be checked. That is, one needs to take the *model* part of model checking into account: How is the model specified? Possibilities include using the source code directly, i. e., the final Java or C sources, or another abstract specification.

Using the source code directly is done in different projects, but has the very general disadvantage that even simple properties might be not decidable, due the complexity of the languages involved. Often, projects deal with this by approximation, that is they only support a subset of the languages semantics and ignore or fail on those parts of the project which are not covered by this subset. This has the advantage that the final implementation is directly checked, although the necessary restrictions on the language might negate this effect. Such a negation can occur when the necessary restrictions hinder the language to be used for more complex applications. An example of such an impractical restriction is "C without using arrays or pointers".

When introducing an intermediate modeling language, another layer is built. This is prone to errors, both in understanding the original implementation and in handling the modeling language. In particular, absence of errors in the model does not directly imply absence of errors in the original implementation. On the other hand, such a modeling language, like Promela [48], may help verification by introducing abstractions (e. g., for concurrency or communication). They are thus mainly fit for protocols: Protocols are always specified abstractly, for example as a written natural language specification. Thus transferring them into a specification language is not a loss, but a gain of concreteness. Further, there exist approaches, for example by Sharma [53], to directly create executables from such a model.

Finally, there also exist hybrid approaches, where an implementation is combined with an additional layer of abstraction. See Holzmann and Joshi [18] as an example.

For CAVA, we provide two different modeling languages. Both are thought to be used as intermediate languages and not for directly implementing programs, as such intermediate languages are easier to implement. While formalizations for C [32] or Java [21] are not outside the scope of a higher-order interactive theorem prover, they are, so far, out of scope for a verified model checker. It must be noted though, that in general this is not a restriction of CAVA: Any system is just a system for our model checker, and can thus be plugged into the model checker anytime.

The first modeling language we provide, a language for Boolean programs, similar to Dijkstra's guarded command programs [8], is mainly used as a proof of concept. The second one is the aforementioned Promela, which was chosen for it is the modeling language of the well-known explicit state model checker SPIN [17]. Thereby, we allow model equivalence between SPIN and CAVA [37, 36].

5.1 Introduction – Boolean Programs

Boolean Programs are our first approach to modeling in CAVA. They operate, as the name suggests, only on finite fields of Booleans (i. e., bits). This has the advantage of being easily implementable, which was the main reason why they were chosen: Their main intent is to serve as a proof-of-concept language, to show the functionality of our model checker. They were introduced into CAVA in our overview paper [10, 11].

As foundational programming constructs, the language offers the following constructs, which are phrased very similar to Dijkstra's guarded command language [8]:

- *SKIP*
- simultaneous assignment $v_1,\ldots,v_n := b_1,\ldots,b_n$ (where the b_i are Boolean expressions)
- sequential composition $c_1; c_2$
- conditional statements in two variants:
  - *IF* $[(b_1, c_1), \ldots, (b_n, c_n)]$ *FI*, that is, a list of condition/code pairs, offering non-deterministic choice in case multiple conditions are met
  - *IF b THEN* c_1 *ELSE* c_2, which is an optimization of the previous version (equal to *IF* $[(b, c_1),(Not\ b, c_2)]$ *FI*), allowing to omit one condition check
- loops *WHILE b DO c*.

The general idea is to embed the language shallowy into Isabelle. That is, a Boolean Program is always a function in Isabelle/HOL. As a consequence, a program has to be embedded into CAVA itself and cannot be loaded at runtime. Embedding a language into a functional setting is a well-known approach, and as old as ML. A general pointer to how this is done in Isabelle/HOL can for example be found in Nipkow/Gerwin [38]. In this chapter, we only want to give a very short introduction how this is implemented.

Generally, a program is represented as a nested data structure, the syntax tree. Leaving notation aside, for our Boolean Language, we have the following program structure:

```
datatype
  com = SKIP
      | nat list ::= bexp list          (* assignment *)
      | com;  com                       (* sequential composition *)
      | IF (bexp × com)list FI          (* conditions; variant 1 *)
      | IF bexp THEN com ELSE           (* conditions; variant 2 *)
      | WHILE bexp DO com               (* loops *)
```

The expressions, *bexp*, are simple expressions on booleans:

```
datatype bexp = TT | FF | V nat | Not bexp | And bexp bexp | Or bexp bexp
```

Here, *TT* and *FF* represent true and false, respectively. *V nat* represents a variable as an offset, because the data any Boolean Program operates on is a field of booleans, and the sole basic data type we support is a boolean.

Example 5.1.1 (Simple Boolean Program)
To give an example, a simple program toggling one of two-bits non-deterministicly ad infinitum is thus represented as:

```
definition toggle :: com where
  toggle = WHILE TT DO IF [
     (V 0, 0 ::= [FF]),
     (Not (V 0), 0 ::= [TT]),
     (V 1, 1 :: = [FF]),
     (Not (V 1), 1 ::= [TT])]
```

In this example we see the outer loop without any abortion condition, and then one larger choice: For each variable we check whether it is true or false, and set it to the opposite value.

Due to the nature of shallow embeddings, everything more complex than the basic operations has to be established using functions inside Isabelle/HOL:

- As protocols must often be sized dynamically (e. g., the number of consumers and producers), this is represented in the programs by also constructing them according to a parameter, often using recursion on the parameter(s). For example, the program for the n dining philosophers is defined as:

  ```
  fun dining :: nat ⇒ (bexp × com) list where
    dining 0 = []
  | dining (Suc n) = ... @ dining n

  definition philosophers n = WHILE TT DO IF dining n FI
  ```

  That is, *dining* constructs the different choices for each philosopher in turn, returning a list of pairs of Boolean expressions and commands. The top *philosophers* then simply chooses one of them ad infinitum.

- Advanced concepts like bounded natural numbers are given as macros, i. e., functions that just expand to a syntax tree which can be inserted at the current place. For our Boolean Programs, we provide macros to represent bounded natural numbers in unary notation, for example. That is, a variable representing a natural number is made up of *bound* boolean variables. Thus a macro *counter_eq pos bound n* comparing the value of a counter at position *pos* with bound *bound* to a natural number n for equality has to generate code that, generally, needs to count the number of set bits between positions *pos* and *pos* + *bound*. Given the simple encoding of naturals and the restricted set of operations for our language, the code can grow very quickly.

- Similarly, named variables are represented as constants or functions returning the variable at a specific position:

  ```
  definition Q_error = V 3
  definition Eat n i = V (3 + i)
  definition One n i = V(n + 3 + i)
  ```

  In this example, we see the variable *Q_error* is located at offset 3; the parameterized variable (working as an array) *Eat* at the positions thereafter; and another array *One*, with offsets just behind *Eat*, given that *Eat* has length n. It can be seen that accessing arrays out of bounds is possible, bearing the well-known problems.

The semantics of this language is then formalized by a translation into a simple interpreted assembly language. The reason is speed: executing commands on the source code level is slow. Because the execution has to be interleaved with the state space exploration this would slow down the model checker considerably. The resulting assembly language only consists of four different operations: assignment, test-else-goto, choice, goto:

datatype *instr* =
 AssI nat list bexp list |
 TestI bexp int |
 ChoiceI (*bexp* × *int*) *list* |
 GotoI int

Thus, we provide a function to compile a list of commands into a list[1] of instructions:

fun *comp :: comp list* ⇒ *instr list* ***where***
  comp SKIP = []
| *comp* (*vars ::= vals*) = [*AssI vars vals*]
| *comp* (*c*$_1$; *c*$_2$) = *comp c*$_1$ @ *comp c*$_2$
| ...

Further, we allow to optimize the generated assembly code, by providing the function *opt :: instr list* ⇒ *instr list*, which, for example, resolves chains of gotos. This reduces the state space quite drastically. It has to be noted, that we refrained from showing its correctness, i. e., preservance of semantics.

A compiled program, as used in the model checker, thus is represented by a list of instructions. The runtime state, the configuration, is represented as a program counter into the program array, and the array of booleans.

The semantics of an assembly language program is given by a function *nexts* that computes the list of possible next configurations from a given configuration. The function *nexts* always returns a nonempty list (in the worst case by cycling, also known as "stuttering"), which means that every program has a run and all runs are infinite. Semantics of a whole boolean program, as seen by our model checker, is only specified by this *nexts* function. That is, we define the language of a Boolean program as:

definition *bpc_is_run bpc r* ≡ ***let*** (*bp*,*c*$_0$)=*bpc* ***in*** *r 0* = *c*$_0$ ∧ (∀*i*. *r* (*Suc i*) ∈ *set* (*nexts bp* (*r i*)))
definition *bpc_props c* ≡ ***let*** (*pc*,*vals*) = *c* ***in*** *vals*
definition *bpc_lang bpc* ≡ {*bpc_props* ∘ *r* | *r*. *bpc_is_run bpc r*}

That is, a run *r* is a run of a Boolean program *bp* and a start configuration *c*$_0$, if it starts with *c*$_0$ and can further be constructed by *nexts*. The atomic propositions of the run, i. e., what is represented at the logic level for forming propositions about a program, are the variables. The language is hence defined as all valid sequences of variable assignments.

The language for Boolean Programs worked as a proof-of-concept approach to show-case the model checker and provide first evaluations on the runtime, as was done in [10]. But that paper also highlighted problems with the approach: Evaluations were not comparable, as different formalizations of the same problem were used. Also, programs have to be embedded into the model checker itself, which is not practical.

[1] For performance reasons arrays are used in the theory, because array access is faster than walking a list.

Algorithm 5.1 Example Promela Code [47]

```
// a small example spin model
// Peterson's solution to the mutual exclusion problem (1981)

bool turn, flag[2];          // the shared variables, booleans
byte ncrit;                  // # procs in critical section

active [2] proctype user()   // two processes
{
    assert(_pid == 0 || _pid == 1);
again:
    flag[_pid] = 1;
    turn = _pid;
    (flag[1 − _pid] == 0 || turn == 1 − _pid);

    ncrit++;
    assert(ncrit == 1);      // critical section
    ncrit−−;

    flag[_pid] = 0;
    goto again
}
```

5.2 Promela

After introducing the development of a modeling language for our model checker based on Boolean Programs, in this section we will formalize an already widely-used language: Promela. This language is used mainly in the model checker SPIN [17], which is a well-known explicit state model checker. By supporting its input language, we can re-use models written for it, which in turn allows easier comparison with it.

As an additional benefit, our formalization [37, 36] serves as the first executable (partial) formalization of Promela. This will be discussed in more detail in Section 5.2.7.

5.2.1 Introduction to Promela

Promela [48] is a modeling language, mainly used in the model checker SPIN [17]. It offers a C-like syntax and allows to define processes to be run concurrently. Those processes can communicate via shared global variables or by message-passing via channels. Inside a process, constructs exist for non-deterministic choice, starting other processes and enforcing atomicity. It furthermore allows different means for specifying properties: LTL formulas, assertions in the code, never claims (i. e., an automaton that explicitly specifies unwanted behavior) and others. A small example is given in Alg. 5.1.

Some constructs found in Promela models, like `#include` and `#define`, are not part of the language Promela itself, but belong to the language of the C preprocessor. SPIN does

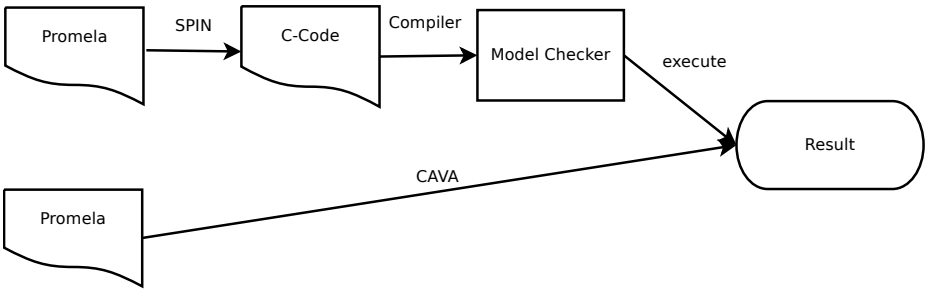


Figure 5.1: Workflow of SPIN vs CAVA

not process those, but calls the C compiler internally to process them. In CAVA we do the same. This is therefore not implemented in the Isabelle part of CAVA, but instead the SML frontend passes the input file through `cpp` before parsing it.

This coupling with C is not a lone instance. Instead, large parts of the language are modeled after C. This stems from SPIN compiling a Promela model into a C program, which then serves as the model checker (compare Fig. 5.1). Thus, for efficiency reasons, parts of Promela are passed verbatim to C (e. g., data types), or others are only very shallow wrappers around C concepts (e. g., Promela `typedef` vs C `struct`). It even allows to embed C code directly.

As discussed in depth later (Section 5.2.7), there is no usable formal semantics of Promela. Instead, the main sources for formalization were the Promela manpages [48], and the descriptions of semantics by Holzmann [17, chap. 3+7]. While those are extensive in general, covering all of Promelas behavior from a programmer's point of view, it lacks a detailed (i. e., formal) description on how constructs are converted into a Kripke structure. Therefore, observing the output of SPIN and examining the generated graphs (via `pan -d`) often is the only way of determining the semantics of a certain construct. This is complicated further by SPIN unconditionally applying optimizations, for example resolving chained gotos for nested loops. Reading the sources of SPIN was deemed not to be a viable option, for it turned out to be rather unreadable. In the end, it is C-code generating C-code.

As we have to model our formalization using SPIN as some sort of black box, we needed an easy measure to spot deviations. One such useful measure is the number of possible configurations reachable in a generated system. We therefore tailored the formalization to generate identical numbers, which comes with the cost of also having to copy the unconditional optimizations of SPIN.

As a consequence of the lack of a formal basis, the formalization presented in this work is not necessarily congruent with SPIN's implementation. Still, we have not yet seen semantic differences so far, besides the ones covered in Section 5.2.5, which are also of a conservative nature. In any case, the need remains for authorative formal semantics of Promela. That is, a semantics where SPIN is modeled after and not vice versa. Our formalization hence aims to be solely a substitute to that end. Furthermore, our formalization should allow for rather easy modifications of the semantics, enabling users to test different approaches and concepts.

5.2.2 Formalization and Implementation

The general structure of the formalization and implementation of Promela follows the general structure already used for the Boolean Programs: We start from an abstract syntax tree, use this to compile some program representation, and finally have some means of computing successor states to formalize what a run on such a program represents.

AST Handling

The abstract syntax tree is built by a hand-written SML parser[2]. The tree is then passed on to the parts implemented in Isabelle. As a first step there, the simple AST is enriched with semantic information and some constructs are replaced by semantic equivalents ("de-sugaring"). Examples for such de-sugaring are simple replacements like i++ by $i = i + 1$, or more involved substitutions like

for (i : lb .. ub) steps

being replaced by

```
i = lb;
do
 :: i =< ub −> steps; i = i+1
 :: else −> break
od
```

Also, different syntactical elements with similar semantics are combined into one single construct bearing different variants, which can for example be expressed with additional flags. For example, both *c!v* and *c!!v* send v on channel c, but while the former simply appends, the latter inserts in correct order. Both can thus be collapsed in to one *Send*-node with a *sorted* flag.

The reason for such transformations is to shrink later stages of the formalization by reducing the set of constructs they have to support. Similarly, the enrichment with semantic information (mostly types) moves some of the burden of static checking to an earlier stage. For example, usage of unknown variables or using wrong typed arguments can already be caught at this state. Later stages then can rely on the type information gathered. Currently, whenever static errors are encountered, the whole model checker terminates by throwing an exception. Ideally, it would return an error which the model checker would gracefully handle. Up to now, this has not been implemented in CAVA, due to the additional overhead of wrapping everything in the option monad.

Compilation

After generating this enriched AST, the real core of the formalization follows: Compiling the AST into an executable program representation. For an understanding of the requirements of the compilation process, we need to look one step ahead and have to analyze how the set of successors is eventually going to be computed: For each running process

[2] As a consequence, any other language going to be supported by CAVA and offering the Promela formalization needs a new implementation of the parser.

we collect the set of statements which are executable at the current state. Then, for each of those statements, we evaluate its effects to obtain a modified state. The set of resulting modified states now represents the set of successors, which is returned.

Promela also knows *atomic* blocks: In general, states inside such a block are not visible outside. Therefore, whenever a statement is marked atomic (i. e., it is part of an atomic block), we have to re-iterate before returning until we find ourselves in a non-atomic block, or until we are stuck and cannot process any further. The latter, allowing the middle of an atomic block as a valid configuration, is an unusual addition to atomicity, which regularly states that either everything is executed or nothing at all.

This general concept on compilation is taken from [17, Chap. 7], and thus we also use the name *semantic engine* coined there for the successor function of the Promela interpreter.

From this behavioral description, we take that the important pieces we need to encode are: the conditions under which a statement is executable, the effect of this statement and whether it is atomic or not. Moreover, we need to encode the order of statements in a process. While this is, strictly speaking, already part of the effect/action pair, it is very natural to consider this as extra properties. We are therefore going to represent a process' behavior as a transition system, obtaining the ordering. Each transition (edge) is then to be annotated by the effect/action pair, amongst others:

record *edge* =
  cond (∗ *Necessary condition* ∗)
  effect (∗ *Effect on states* ∗)
  target (∗ *Next state* ∗)
  prio (∗ *Priority* ∗)
  atomic (∗ *Atomicity information* ∗)

The system itself is then composed of a list of lists of transitions, where each position in the list represents a program point, and the list of edges at that position encodes the outgoing transitions from that point. The target of a transition is thus a simple offset into the transition system denoting the subsequent program point.

In this formalization, condition and effect are encoded abstractly, for a later interpretation. For example, the type for conditions is defined as follows:

datatype *edgeCond* =
    ECElse
  | *ECTrue*
  | *ECFalse*
  | *ECExpr expr*
  | *ECRun String.literal*
  | *ECSend chanRef*
  | *ECRecv chanRef recvArg list bool*

A complete process is then put together from this transition system, plus some more information like the expected argument list. As a Promela program is mainly a collection of process definitions, this already amounts to our representation of the program itself[3].

[3]The reality adds some more information, mainly for debugging purposes or for efficiency reasons. For example, the transition systems are kept in the program, and process definitions just refer to the offset.

Having the representation of a program, we need to model the compilation process. At its heart lies a translation from statements in the AST to edges. This translation is done backwards through the AST. This is motivated by the need to calculate targets of transitions as offsets into lists: By having later statements at known positions, the target is already fixed. This works very well for all statements, except gotos, so that the target position is simply threaded through the compilation.

This compilation of the AST into the transition system is formalized as two mutually recursive functions *stmntToState* and *stepToState* (the difference between step and statement is minor and not of interest here). In the following, we want to illustrate the compilation process by giving some example cases (there is one case per AST node type). The first three examples, assignment, condition, and goto, are straightforward, followed by the more involved handling of the *unless*. Note that atomicity is handled later on and is thus not taken into account in any of those cases, i. e., they default to *NonAtomic*:

Example 5.2.1 (Assignment Statement)
An assignment can always be executed, therefore the condition is simply "true". The assignment needs to be evaluated at runtime, thus it is again stored as such. It also does not change anything regarding control flow, labels or priority and hence simply uses the one provided by the vicinity:

$$\begin{aligned}&\mathit{stmntToState}\ (\mathit{StmntAssign}\ v\ e)\ (\mathit{lbls}, \mathit{pri}, \mathit{pos}, \mathit{nxt}, \_) = \\ &\quad ([[\langle\!|\ \mathit{cond} = \mathit{ECTrue}, \mathit{effect} = \mathit{EEAssign}\ v\ e, \mathit{target} = \mathit{nxt}, \mathit{prio} = \mathit{pri}, \\ &\qquad \mathit{atomic} = \mathit{NonAtomic}\ |\!\rangle]], \mathit{Index}\ \mathit{pos}, \mathit{lbls})\end{aligned}$$

Example 5.2.2 (Condition Statement)
The condition is in most respects the counterpart to the assignment, as it does not have any effect, i. e., the effect is the identity. Instead, it denotes a condition that must be satisfied. Again, this condition must be evaluated at runtime and is therefore simply stored; no further changes are done:

$$\begin{aligned}&\mathit{stmntToState}\ (\mathit{StmntCond}\ e)\ (\mathit{lbls}, \mathit{pri}, \mathit{pos}, \mathit{nxt}, \_) = \\ &\quad ([[\langle\!|\ \mathit{cond} = \mathit{ECExpr}\ e, \mathit{effect} = \mathit{EEId}, \mathit{target} = \mathit{nxt}, \mathit{prio} = \mathit{pri}, \mathit{atomic} = \mathit{NonAtomic}\ |\!\rangle]], \\ &\quad \mathit{Index}\ \mathit{pos}, \mathit{lbls})\end{aligned}$$

Example 5.2.3 (Goto Statement)
The *goto* has neither effect nor condition, but nevertheless receives its own effect marker to make it distinguishable. Instead, it modifies the control flow. This manifests twofold: First, the target is set to $\mathit{LabelJump}\ l\ \mathit{None}$, i. e., a jump to the destinated label l as specified by the *goto*. But this is not sufficient, for it would render an additional state at runtime (first the goto would be reached, and only thereafter the jump is taken). So, secondly, the jump target is directly propagated to the preceding statement[4] instead: Note that in the previous examples, we set the *target* to the parameter *nxt*. Note that in those examples, we also return *Index pos*, which denotes our current position in the transition system. This is then fed to the

[4] It is possible to land at the goto indirectly, for instance when the goto is behind another label. Thus we cannot just remove the gotos from the transition system. Also, in those indirect cases, the back propagating might not work, so we have to use the two-step jumping semantics regardless.

preceding statement as *nxt*. In case of our goto, we therefore also return the jump[5]:

```
stmntToState (StmntGoTo l) (lbls, pri, pos, _) =
  ([[⦇ cond = ECTrue, effect = EEGoto, target = LabelJump l None, prio = pri,
       atomic = NonAtomic ⦈]], LabelJump l (Some pos), lbls)
```

Allowing the back propagation of targets as detailed here, is another reason why processing the AST in a backward manner is of advantage.

Example 5.2.4 (Unless Statement)
A more advanced example is the compilation of the *{s} unless {u}* construct, which serves as an exception handling in Promela: From each step in the sequence *s*, control can go the the *unless*-part *u* as soon as the first expression in *u* becomes true. At this point, the field *prio* of the *edge* record becomes important: The priority of the edges going from some point in *s* to *u* is not equal to the priority of the edges advancing in *s*, but higher in order to be examined first.

The translation is now done in two steps: First we compile *u*, which returns the compiled partial transition system. Due to our backwards processing, the last element is actually the first transition, i. e., the one carrying the necessary condition. After having also compiled the *s*-part (with a lower priority), we add the first transition of the unless-part to each program point to the transition system of *s*:

```
stepToState (StepStmnt s (Some u)) (lbls, pri, pos, nxt, onxt, _) = (
 let
  (* the 'unless' part *)
  (ues,_,lbls') = stmntToState u (lbls, pri, pos, nxt, onxt, True);

  (* 'u' is the guard for the whole unless; 'ues' the rest *)
  u = last ues; ues = butlast ues;
  pos' = pos + length ues;

  (* find minimal current priority *)
  pri' = min_prio u pri;

  (* the main part --
     priority is decreased, because there is now a new unless part with higher prio *)
  (ses,spos,lbls'') = stmntToState s (lbls', pri' - 1, pos', nxt, onxt, False);

  (* add an edge to the unless part for each generated state *)
  ses = map (λ s. u@s) ses
 in (ues@ses,spos,lbls''))
```

In general, all constructs influencing the control flow are more complex, as their main consequence is not setting condition and effect, but generating the right number of edges and setting the targets accordingly. To a great degree, the complexity is increased due to SPIN's semantic trying to minimize the use of intermediate states, something commonly

[5]The difference in the second parameter of *LabelJump* is needed for atomicity calculation and is going to be explained later.

happening with nested loops – even more when (nested) *unless* is involved. And as we have laid out in Section 5.2.1, our formalization aims to keep step with SPIN here.

Another rather complex matter for constructing the control flow originates from atomicity. While generally the concept is simple (everything inside an *atomic* block is executed atomically), it becomes complex when *goto*s are involved: When jumping from one atomic block to another one, atomicity is preserved. This does not hold if it is done via a non-atomic intermediate label. Therefore, the state of atomicity of an edge can only be determined reliably, when a process is compiled completely and labels are going to be resolved. Thus, it is required to not only cover the two possibilities *Atomic* and *NonAtomic*, but also *InAtomic*. The latter states that an edge *starts* in an atomic block. Hence, if a label inside an atomic block is resolved to a position where all outgoing edges are *InAtomic* or *Atomic*, the edge itself becomes *Atomic*, otherwise it remains as *InAtomic*.

```
primrec resolveLabels
 :: edge list list ⇒ labels ⇒ edge list ⇒ edge list where
 resolveLabels _ _ [] = []
| resolveLabels edges lbls (e#es) = (
   let check_atomic = λpos. fold (λe a. a ∧ inAtomic e) (edges ! pos) True in
   case target e of
    Index _ ⇒ e
   | LabelJump l None ⇒
      let pos = resolveLabel l lbls in
       e(|target := Index pos,
         atomic := if inAtomic e then
                     if check_atomic pos then Atomic
                     else InAtomic
                   else NonAtomic |)
   | LabelJump l (Some via) ⇒
      let pos = resolveLabel l lbls in
       e(|target := Index pos,
         (* NB: isAtomic instead of inAtomic, cf atomize() *)
         atomic := if isAtomic e then
                     if check_atomic pos ∧ check_atomic via then Atomic
                     else InAtomic
                   else atomic e |)
  ) # (resolveLabels edges lbls es)
```

The third case (*LabelJump l (Some via)*) has already been shortly mentioned in the example on compiling *goto* (Ex. 5.2.3). This case occurs whenever we end up at a *goto* indirectly, for example with another jump. In order to match SPINs behavior (cf. Section 5.2.1), we try to avoid intermediate states and thus have to change the edge targeting a *goto* to point to the target of the jump. While not a problem in general, it complicates atomicity handling, because now the atomicity state of the intermediate label itself (the parameter *via* in the function above) is also accounted for.

Note that there are also other means of passing atomicity, especially between processes. But those are resolved at runtime and will be explained there.

The construction of the transition system of a process is finished by adding a final

ending state, which just self-loops. Every transition which causes a process to terminate is then changed to target there. This construction ensures that we always have infinite runs in the system.

Combining the transition system with further information (e. g., the labels, the names of the defined processes, ...) we receive the final data structure *program*. We will not give the exact definition of this structure here, because it is very technical and mostly depends on what debugging/pretty-printing information is needed.

Execution

Now, after having covered the compilation of Promela programs, the next important part is the actual execution of such a program. For that matter, we want to start with the definition of a configuration of a program: As already covered before, a Promela program consists of a number of concurrent processes. Thus, we also need to keep a state for each running process:

record *pState* =
pid :: *nat* (∗ *Process identifier* ∗)
vars :: *var_dict* (∗ *Dictionary of variables* ∗)
pc :: *nat* (∗ *Program counter* ∗)
channels :: *integer list* (∗ *List of channels created in the process* ∗)
idx :: *nat* (∗ *Index defining the underlying transition system* ∗)

The only notable field here is *channels*: As channels in Promela are by definition global, they are not accounted for in the states. The only reason to carry the list of local channels is to mark them as invalid after termination of a process (it is not clear whether channel numbers are to be reused in SPIN, thus we do not reuse them in CAVA). The transition system of the process is not stored in the *pState*, for it would yield unnecessary duplication (multiple instances of the same process would all contain a copy of the transition system). For that reason, only an *idx* is stored which is the offset into the global list of transition systems.

The global state then contains the list of all program states (the *pid* being the index). It further contains the global variables and, as explained, the channels. Additionally, a flag *timeout* is added, which is part of Promela's semantics and is set when no process can progress.

record *gState* =
vars :: *var_dict*
channels :: *channels*
timeout :: *bool*
procs :: *pState list*

This record then serves as the configuration for a Promela program. Promela defines two more flags, namely *else* and *handshake*. As their setting is invisible to an observer, they are not part of *gState* but are set internally.

The idea of the semantic engine, that is the function computing the set of successing configurations, has already been explained earlier: Calculate the set of computable edges taking priorities into account, and for each of those edges apply the effect onto the current

state, obtaining the set of resulting states. With the defined representation of the program, this becomes straigthforward, with g and l holding the global and the local (= process) state, respectively:

1. For each edge, check wheter its condition *cond* holds. This is achieved by the function *evalCond* getting the condition, the global and the process-local state as input. Example evaluations are:

   $$\begin{aligned}
   &\mathit{evalCond}\ (\mathit{ECExpr}\ e)\ g\ l \leftrightarrow \mathit{exprArith}\ g\ l\ e \neq 0\\
   &\mathit{evalCond}\ (\mathit{ECRun}\ _)\ g\ l \leftrightarrow \mathit{length}\ (\mathit{procs}\ g) < 255\\
   &\mathit{evalCond}\ (\mathit{ECSend}\ v)\ g\ l \leftrightarrow \mathit{withChannel}\ v\ (\lambda_\ c.\ \textbf{\textit{case}}\ c\ \textbf{\textit{of}}\\
   &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Channel\ cap\ _\ q} \Rightarrow \mathit{length\ q} < \mathit{cap}\\
   &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{HSChannel\ _} \Rightarrow \mathit{True})\ g\ l
   \end{aligned}$$

   These examples amount to: expressions must evaluate to something non-zero; spawning a new process requires the number of currently running processes to be below some upper bound[6]; and for sending something over a channel, the capacity of this channel must not be exhausted.

2. For each pair of process and edge, we then compute its effect: The function *evalEffect* takes the effect of an edge, the program itself (for lookup tasks) and again the global and local state. It returns the updated global and local state:

   $$\begin{aligned}
   &\mathit{evalEffect}\ (\mathit{EEAssign}\ v\ e)\ _\ g\ l = \mathit{setVar}\ v\ (\mathit{exprArith}\ g\ l\ e)\ g\ l\\
   &\mathit{evalEffect}\ (\mathit{EERun}\ \mathit{name}\ \mathit{args})\ \mathit{prog}\ g\ l = \textbf{\textit{let}}\ (g,_) = \mathit{runProc}\ \mathit{name}\ \mathit{args}\ \mathit{prog}\ g\ l\ \textbf{\textit{in}}\ (g,l)\\
   &\mathit{evalEffect}\ (\mathit{EEAssert}\ e)\ _\ g\ l = \ \textbf{\textit{if}}\ \mathit{exprArith}\ g\ l\ e = 0\\
   &\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{\textit{then}}\ \mathit{setVar}\ \mathit{__assert__}\ 1\ g\ l\\
   &\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{\textit{else}}\ (g,l)
   \end{aligned}$$

   In this excerpt, a variable is set to the correct value; a new process is started; the `__assert__` variable is set, if the assertion holds (cf. Section 5.2.5 for assertion handling). For sending and receiving (not shown) more effort is necessary, stemming mostly from the zoo of different variants and from the fact that receiving can compare values, evaluate variables, and set variables at the same time.

This general handling is then enhanced to take atomicity into account. Whenever we handle an edge with the atomic bit set, we feed it again into our semantic engine with the additional modification of setting an *exclusive* field to the process where the edge was part of, thus barring the other processes from being run. The idea is to complete the whole atomic block in only one visible global configuration. In case the process cannot perform the whole atomic block, the state is served as-is, i. e., in the middle of that block. It has to be noted that this may lead to non-termination.

So, for example, *atomic { do :: skip; od }* will not terminate, as the successor function loops forever. This is the intended Promela semantics.

Additional complexity is added by allowing to pass atomicity between processes:

[6] A necessary condition for a finite state-space.

```
chan hs = [0] of { int };

active proctype rcv () {
  int r;
  atomic { hs?r; ... }
}

active proctype send () {
  atomic { ...; hs!1 }
}
```

Here, the process *send* can pass its atomicity to *rcv* via the handshake (or rendezvous) mechanism: A channel with 0 capacity is a so called handshake channel. Whenever a process wants to write to it, there must be another process which immediately reads from it (as there is no capacity). If both the send and the receive operation are inside an atomic block, as in the example, the atomicity is passed from the sender to the receiver.

Please observe that this definition of the handshake channel also adds the need for a limited back-tracking, because we cannot execute the sending without having a receive at the next step. The implementation of this rendezvous mechanism therefore is similar to the one for atomicity, and thus invisible from the outside.

The semantic engine, including non-covered features like *timeout* and the similar *else*, is exported by the function

definition *nexts :: program ⇒ gState ⇒ gState set nres*

As noted earlier, resolving atomic blocks may lead to non-termination. Therefore it is not possible to refine this *nexts* function into an implementation which solely returns the *gState set*. Instead we, we are stuck with *gState set dres*, which is the deterministic monad of the Refinement Framework which still allows non-termination. But as the model checker framework requires a pure function outside of monads, we have to work around it:

```
definition dSUCCEED_abort where
 dSUCCEED_abort msg dm m = (
   case m of
    dSUCCEEDi ⇒ Code.abort msg (λ_. dm)
   | _ ⇒ m)

definition nexts_code
 :: program ⇒ gState ⇒ gState set
where
 nexts_code prog g =
   the_res (dSUCCEED_abort (STR ''The Universe is broken!'')
                           (dRETURN {g})
                           (nexts_code_aux prog g))
```

What this definition expresses is abortion in case of non-terminiation. While this is of no influence to the actual code, it circumvents some logic by failing with exceptions. But as this is done anyway inside Promela to signal runtime errors, we find this an acceptable solution.

5.2.3 Abstract Language

The last and final step is defining the abstract language of such a Promela program. We define the atomic properties as the set of all possible expressions, and thus positive properties are those expressions which evaluate to true under the current configuration (and, as the evaluation requires a process state, an empty process state):

definition *promela_props :: gState ⇒ expr set* ***where***
promela_props g = {e. exprArith g emptyProc e ≠ 0}

A run of the program is defined naturally over the *nexts_code* function:

definition *promela_is_run'*
:: program × gState ⇒ gState word ⇒ bool
where
promela_is_run' progg r ≡
let *(prog,g₀)=progg* ***in***
r 0 = g₀
∧ (∀i. r (Suc i) ∈ nexts_code prog (r i))

But this definition only defines runs based on the compiled program, which leaves the compilation to the descretion of the user. As we need our program to be well-formed, we extend this definition to be defined on the AST, so that compilation is part of the semantics:

definition *promela_is_run ≡ promela_is_run' ∘ compile*

Combining this, we define our language as:

definition *promela_language ast ≡ {promela_props ∘ r | r. promela_is_run ast r}*

For technical reasons, the definition of the language used in the model checker is slightly different: We preprocess the LTL formula to extract the included expressions, and only use those expressions as the set of atomic properties (*APs*), i. e., if the formula were `GF((eat[0] ⟶ ¬fork[0]) ∨ ¬__assert__)`, the atomic properties are `eat[0]`, `fork[0]` and `__assert__`. The LTL formula then is changed to only include pointers to the expressions instead of the expressions itself. We show that this processed variant behaves equivalently to the original definition:

lemma *promela_run_in_language_iff:*
assumes *ltl_convert φ = (APs,ψ)*
shows *promela_props ∘ ξ ∈ ltlc_language φ*
⟷ promela_props_ltl APs ∘ ξ ∈ ltlc_language ψ

A last thing to take into account is the counter-example generation. The run returned by the model checker itself only consists of the externally visible states. If those were presented to the user as-is, it would not be possible to retrace the run and find the problem. For instance, atomic intermediary states would not be visible. Therefore, we introduce another function

definition *replay :: program ⇒ gState ⇒ gState ⇒ (edge × pState) list nres*

This function takes a program definition together with a state s_1 and its successor s_2 and returns the list of edges with their associated local process which were taken to go from

s_1 to s_2. Although this function may return an execution different from the one taken originally, this does not pose a problem, as the result of that execution is identical to the original executaion (i. e., s_2).

5.2.4 Space State Finiteness

As was the case for the Boolean Programs, we do not have some abstract semantics which we can prove to adhere to. Instead, the semantic engine and the compilation *is* the semantics. But nevertheless we have to show a very important property about the Kripke structure that is defined by our language: It must be finite. Therefore, the set of all states reachable from some initial state *constructed by our compilation process* must be finite.

In this section, we will introduce a set of invariants, which our formalization is proven to honor. These invariants will establish, eventually, the finiteness of the generated state space, given that the compiled program adheres to some invariant. We will show that our compilation does adhere to it.

A natural requirement for this to hold is the need for bounds to any variable type, number of variables, and number of processes. While the number of variables is finite from restricting the number of processes, we additionally need to enforce bounds on the length of arrays and the range of variables. This is not a restriction when compared to SPIN, where, due to relying on C datatypes, the ranges are inherent. But it still can pose a problem, as the defined bounds are arbitrary. Ideally they could be set as parameters on the command line when calling the model checker; currently they require modifying the theories, although without needing to touch any proof.

Using those bounds, we then go forward on defining invariants that have to hold on the generated program. For example, variables and channels have to adhere to the bounds:

```
fun varType_inv :: varType ⇒ bool where
  varType_inv (VTBounded l h)
  ⟷ l ≥ min_var_value ∧ h ≤ max_var_value ∧ l < h
| varType_inv VTChan ⟷ True

fun variable_inv :: variable ⇒ bool where
  variable_inv (Var t val)
  ⟷ varType_inv t ∧ val ∈ {min_var_value..max_var_value}
| variable_inv (VArray t sz ar)
  ⟷ varType_inv t
    ∧ sz ≤ max_array_size
    ∧ IArray.length ar = sz
    ∧ set (IArray.list_of ar) ⊆ {min_var_value..max_var_value}

fun channel_inv :: channel ⇒ bool where
  channel_inv (Channel cap ts q)
  ⟷ cap ≤ max_array_size
    ∧ cap ≥ 0
    ∧ set ts ⊆ Collect varType_inv
    ∧ length ts ≤ max_array_size
```

$\wedge$ *length q* $\leq$ *max_array_size*
$\wedge$ ($\forall$*x* $\in$ *set q. length x = length ts*
$\wedge$ *set x* $\subseteq$ {*min_var_value..max_var_value*})
| *channel_inv* (*HSChannel ts*)
$\longleftrightarrow$ *set ts* $\subseteq$ *Collect varType_inv* $\wedge$ *length ts* $\leq$ *max_array_size*
| *channel_inv InvChannel* $\longleftrightarrow$ *True*

It can then be shown that the set of all variables and all channels adhering to those invariants is finite (recall that *Collect* $\Phi \equiv \{p.\ \Phi\ p\}$) :

lemma *variables_finite:*
finite (*Collect variable_inv*)

lemma *channels_finite:*
finite (*Collect channel_inv*)

Similar, we define an invariant, depending on the compiled program, for process states. Besides some bounding properties (amongst others we have to show that the names of the variables are bounded!), we also introduce necessary correlation properties. For instance, we also show that the program counter cannot run past the end of the array representing the transition system:

definition *pState_inv :: program* $\Rightarrow$ *pState* $\Rightarrow$ *bool* ***where***
pState_inv prog p
$\longleftrightarrow$ *pid p* $\leq$ *max_procs*
$\wedge$ *pState.idx p* < *IArray.length* (*states prog*)
$\wedge$ *IArray.length* (*states prog*) = *IArray.length* (*processes prog*)
$\wedge$ *pc p* < *IArray.length* ((*states prog*) ! *pState.idx p*)
...

Again, we can show that the number of possible program states is finite:

lemma *pStates_finite:*
finite (*Collect* (*pState_inv prog*))

This can be further extended to global states, which follow naturally:

definition *gState_inv :: program* $\Rightarrow$ *gState* $\Rightarrow$ *bool* ***where***
gState_inv prog g
$\longleftrightarrow$ *length* (*procs g*) $\leq$ *max_procs*
$\wedge$ ($\forall$*p* $\in$ *set* (*procs g*). *pState_inv prog p*)
$\wedge$ *length* (*channels g*) $\leq$ *max_channels*
...

Unfortunately, *Collect* (*gState_inv prog*) no longer is finite. Instead we create some relation (definition omitted), called *gState_progress_rel*, for which we will eventually show that our semantic engine is in that relation. We can then show that the number of successors of a state under this relation is finite:

lemma *gStates_finite:*
fixes *g :: gState*
shows *finite* ((*gState_progress_rel prog*)* `` {*g*})

Finally, we also give an invariant to hold on the compiled program:

definition *program_inv* ***where***
program_inv prog $\longleftrightarrow \ldots$

Thereafter, we show that our compilation respects those invariants:

lemma *compile_program_inv:*
assumes *compile ast* = (*prog*,g_0)
shows *program_inv prog*

lemma *compile_gState_inv:*
assumes *compile ast* = (*prog*, g_0)
shows *gState_inv prog* g_0

Also, our semantic engine only creates successors which are related to the input via *gState_progress_rel*:

lemma *nexts_SPEC:*
assumes *gState_inv prog g*
and *program_inv prog*
shows *nexts prog g* $\leq$ ***spec*** *gs.* $\forall g' \in gs.\ (g,g') \in$ *gState_progress_rel prog*

This property carries over, by refinement, onto our executable version:

lemma *nexts_code_SPEC:*
assumes *gState_inv prog g*
and *program_inv prog*
shows $g' \in$ *nexts_code prog g*
$\Longrightarrow (g,g') \in$ *gState_progress_rel prog*

Therefore, we can finally show that the set of reachable states over the *nexts_code* function is finite, given the invariants on the program and the initial state hold. But as was shown earlier, this is indeed the case when using the provided compilation function:

lemma *reachable_states_finite:*
assumes *program_inv prog*
and *gState_inv prog g*
shows *finite* (*reachable_states prog g*)

5.2.5 Differences between SPIN and CAVA

It has been established in the beginning of this section on Promela that there is no real and useful formal description of the language. For this reason we had to see the implementation in SPIN as the specification. For some reason or other, we had to deviate from those SPIN-specific specification. We claim, though, that those deviations are conservative: Any model which runs with CAVA also runs with SPIN and yields the same result. For models not using those unsupported constructs, we generate the very same number of states as SPIN does. An exception applies for large `goto` chains and when simultaneous termination of multiple processes is involved, as SPIN's semantics is too vague here, i. e., it does not

specify whether those processes die at the same time or only one after the other in different visible states.

In the following, we describe the known differences between our formalization and what is described for SPINs implementation.

d_step The deterministic step *d_step* is an alternative to *atomic*. Everything inside is collapsed into one visible change, also removing non-determinism. This construct is not supported by CAVA. Instead one should replace *d_step* by *atomic*, which yields the same semantics (the other direction does not hold), even though generating larger models. Because CAVA already uses the C preprocessor, this replacement can be easily done by passing `-D d_step=atomic`.

***run* is a statement** Following the Promela specification, *run* is "the only operator allowed inside expressions that can have a side effect" (from [48]). Of course, having *run A()* as a full-fledged expression would be a disaster (and what would *run A() + run B()* express?). Thus SPIN enforces some, again rather underspecified, restrictions. As the sole reason for this semantics is to get hold of the process ID of the spawned process (*pid = run A()*), for which there are also other measures, we omitted this feature and made *run* a statement. This also improves the formalization, for we now can assume all expression to be free of side-effects.

***assert*s do not abort** If SPIN comes across a violated assertion, the model checking process aborts and a counter-example is printed. As our model checker is tailored towards getting an LTL property as input and take this as its sole responsibility, such behavior could not be implemented properly inside CAVA. As an alternative, we added the general atomic property, read: variable, `__assert__` which is available in each program. Hence, it is necessary to actively check this variable in the LTL formula if assertion violation should be paid attention to.

Other property specifications For the same reason given in the assertion case, property specifications besides *ltl* and *assert* are ignored. In the SPIN-world, a very widely used approach for specifiying properties are the never-claims. This is an additional process which expects a property that must not happen. LTL formulas are normally converted into such a never-claim by SPIN.

Types are bounded As described in the section on finiteness 5.2.4, all types are bounded. While this does not deviate from the general SPIN behavior, the bounds can be different. SPIN does not specify any explicit semantics here, but solely refers to the underlying C-compiler and its semantics. This might result in two models behaving differently on different systems when run with SPIN, while CAVA, due to the explicit bounds in the semantics, is not affected. Moreover, in CAVA we do not allow overflowing of values except for well-defined types like booleans[7].

[7] We found one model while benchmarking which was wrongly specified. Due to an integer underflow, certain paths were never reached. This passed silently in SPIN.

Certain variable declarations SPIN has strange semantics when it comes to variable declarations. In earlier versions of SPIN, all variable declarations inside a process definition were evaluated implicitly as if they were placed at the top of the process definition. That is, their actual positions were irrevelant, though the order mattered. While this behavior has been removed, there are occurences where the position is not taken into account. For instance the following program yields 0, which is expected:

```
int i
goto L;
i =5;
L: printf("%d", i)
```

On the other hand, the following yields 5, which is unexpected. An error might have been better:

```
goto L;
int i = 5;
L: printf("%d", i)
```

In CAVA, the latter version is not allowed and would result in an error "unknown variable i".

typedef The *typedef* allows to specify larger structures in Promela models. While this is an important feature, we thought it not feasible to implement it correctly in CAVA, due to the rather complicated restrictions on them:

> It is not possible to assign the value of a complete typedef object directly to another such object of the same type in a single assignment. A typedef object may be sent through a message channel as a unit provided that it contains no fields of type unsigned and no arrays of typedef'ed structures. A typedef object can also be used as a parameter in a run statement, but in this case it may not contain any arrays. (Source: [48])

Binary Operations and *unsigned* Currently, binary operations on integers are not supported by CAVA, as is *unsigned*, which implements bit-fields. Using the native Word theories by Lochbihler [33] this should be possible to be implemented, though.

Remote References Remote references allow to access other processes' internal variables, both from LTL properties and from processes itself. This has not been implemented in CAVA, because it seems rather obscure and might not be trivial to implement.

Ignored Constructs Various constructs like `xr`, `xs`, advanced variable scoping, priorities, and visibility of variables are ignored. This does not alter the behavior of the model, but is mostly used for creating more efficient models. Especially advanced variable scoping is also, again, very underspecified. For example, it seems like even though variables can be scoped, there must not be two different variables with the same name in one process but different scopes. Also, all constructs of the *print** family are ignored, as they are not expressible in SML.

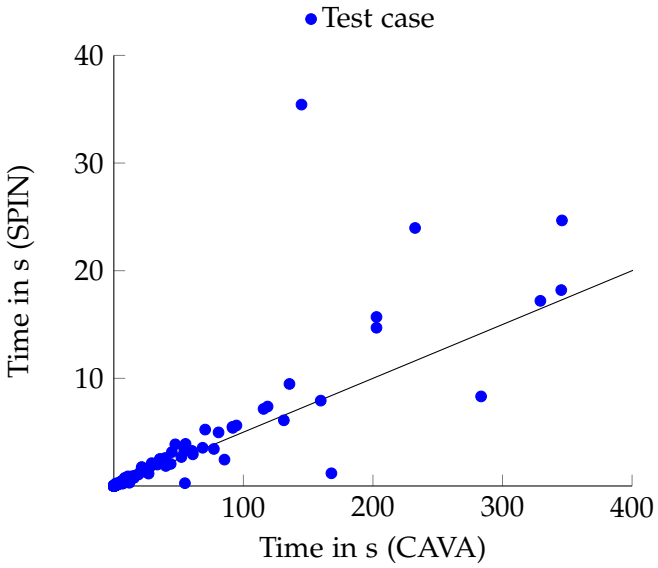


Figure 5.2: G true benchmark on 157 tests

5.2.6 Evaluation

With the support for Promela, it is now possible to test the very same models in both SPIN and CAVA. For this, we used models from [43, 46], with some minor modifications to match modern Promela syntax. The tests were performed on a Core i7 with 2.7 GHz, memory being hard-limited to 6.5 GB. Also, a timeout of 800 seconds was set for each run.

CAVA was compiled with MLton 20130715 and used an ad-hoc implementation of Lammich for Nested DFS. SPIN (version 6.2.5) was used without optimizations, especially partial-order reduction: `spin` was run with `-o1 -o2 -o3` and the code compiled with `-DNOREDUCE`. During the benchmark, SPIN's search depth was set to $6 * 10^7$ (`-m60000000`).

Further, `-D d_step=atomic` was passed to both SPIN and CAVA, replacing `d_step` blocks by `atomic` blocks, as the former is not supported by CAVA (cf. Section 5.2.5). Since `d_step` is an optimized and restricted form of the latter (collapsing the sequence into one state), this is semantically sound, but influences the size of the state space.

The benchmark consists of 306 single tests, 4 of which got removed, as they contained failing asserts which CAVA ignores by default (cf. Sect. 5.2.1). Further, 50 tests included features not supported in CAVA, 77 led to failures in SPIN (most often out-of-memory and exhausted search depth), 94 timed out on CAVA (a test may occur in multiple of those categories). In total 157 tests performed successfully on both tools. To ensure a complete search of the state space the property used together with these tests is G true. Each test was run 5 times, the worst and best time removed and the remaining three averaged. Two timed out runs mark the whole test as timed out.

This benchmark shows that overall CAVA is about 20 times slower than SPIN. Fig. 5.2 plots the results of the benchmark: the line represents $t_{\text{SPIN}} = 20t_{\text{CAVA}}$, so anything above represents a test where CAVA was less than 20 times slower than SPIN (dots below analogously). Tests on which it timed out had a mean run time of 89.18 seconds in SPIN, lying far above $\frac{\text{timeout}}{20}$. This is a good result for a verified and generated software,

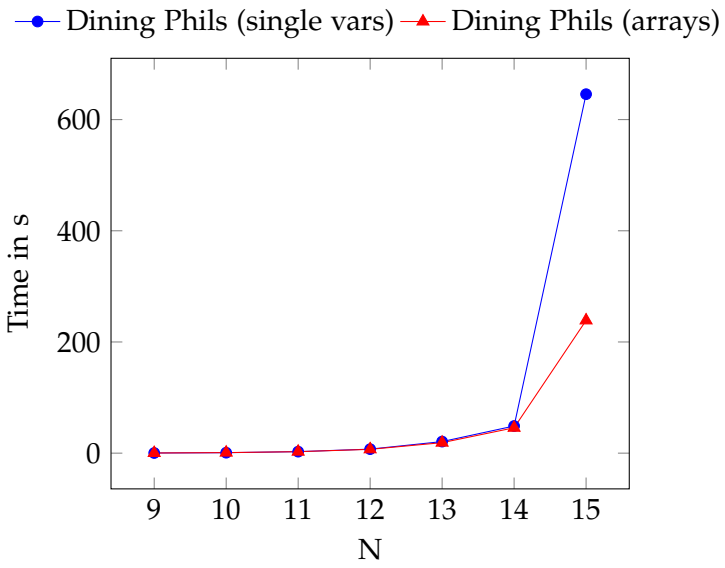


Figure 5.3: Comparison array vs. single variables

especially as SPIN builds a tailored checker for each model, whereas CAVA's is general.

Further, we tested multiple properties on scaling versions of the leader election protocol and the "Dining Philosophers". Here, the LTL-to-Büchi translation is important. As of this time, the implementation in CAVA is tailored to verification, not efficiency. This leads to larger-than-necessary state spaces, in particular for formulas containing U. Therefore the slowdown is a factor between 9 and 70. For negative properties, SPIN found 75 of 77 counter-examples in less than 10 seconds, CAVA 70 of 77. In general, those numbers are hard to compare though, as the counter-example search heavily depends on choices made throughout the search process.

The main reason for the difference in performance is the lack of destructive updates in a purely functional program. In particular we must use trees as our main data structure, yielding a logarithmic overhead. Arrays can only be used when updates are seldom, as they cannot be updated in-place but need to be copied in full. Moreover we cannot utilize pointers for keeping a reference to a changing structure, but have to look up information each time. The consequences are shown in Fig. 5.3. We ran "Dining Philosophers" modeled in two different ways:

- using three arrays of length N, i. e., one array for each general variable to keep track of, where the values for process i is stored at position i in each array
- using $3N$ different variables, i. e., one variable for each combination of process and general variable

As can be seen in the figure, the amount of variables has a very notable impact on performance, even though this does not influence the state-space.

5.2.7 Related Formalizations

Other formalizations of Promela exist [57, 13, 53]. Unfortunately, it was not possible to use any of them as a template for our use.

Weise [57] presents a small-step operational semantics for Promela, based on earlier works of Natarajan and Holzmann [34]. While the rules presented in the paper make it possible to understand the general concepts, they only represent an old version of Promela[8] – and this one only partially. Furthermore, the semantics described there (for example regarding atomicity) are not exactly what SPIN is doing, though the possibilty remains for the semantics having changed in the meantime. Additionally, it suffers from the main problem of most formalizations, the lack of an implementation. Thus it remains unclear whether all deviations from SPIN are being mentioned in that paper.

A more recent approach is done by Gallardo et al. [13]. In their paper, they present a small step operational semantics, which is built using different layers: intra-procedural, two different notions of inter-procedural, and the observable semantics. While their goal is to create a semantics which makes it possible to use Promela in abstract model checking, this paper makes it possible to gain a more detailed view into the behavior of Promela. It also handles the difficult edge-cases of the language, like passing atomicity via handshakes.

We tried to use the approach by Gallardo for our own formalization. Unfortunately, it often remained too abstract, making it hard to implement it efficiently. Additional complexity was introduced by the multiple layers. Thus, we changed back to the approach of Holzmann [17]. But it may be reasonable to use [13] as the abstract layer atop of the current formalization.

The last formalization we took into account is the fairly recent work by Sharma [53]. The author presents a refinement framework for refining Promela models to C, in order to generate implementations of the models. The author also shows that their translation is sound regarding the LTL properties. Sharma starts with a core part of the language to establish semantic rules. This is then extended to the overall language. Unfortunately, the author does not link the two concepts properly, but instead just gives the extension as a set of syntax rules. It is not explained why this now models all of Promela, and also why this does not interfere with the correctness properties of the refinement. Also, similar to Weise [57], the semantics of atomicity defined in the core Promela language is wrong; due to the missing explanation on the extension to the language proper, this seems to carry over to the whole model.

5.2.8 Conclusion

Our formalization is, to our awareness, the only executable formalization of Promela so far. While still lacking some features of the original language (cf. Section 5.2.5), the generated system automaton does not differ from the one generated by SPIN. We were able to run large parts of a benchmark suite using this formalization, proving its practicability. As we can interpret the model on the fly, the workflow is even simpler compared to SPIN, where three steps are necessary (compile to C, compile to executable, run – cf. Fig. 5.1 on page 102).

[8] We cannot find any usage of versioning for Promela after this paper, so it is hard to relate the version mentioned there to the current one.

But naturally, the presented formalization is not perfect. The most important point is the missing features. Of those, only *typedef* and the binary operations represent features that would really increase usability. The *d_step* is definitly a nice-to-have perfomance-wise, but not essential as it can be replaced, as mentioned, by *atomic*.

A larger problem with this formalization is its lack of abstraction. Currently, a lot of the functions involved already take efficiency into account, at least partially. As a consequence, they have to deal with both their inherent use and the correct use of more efficient datastructures. A good example for illustration is the use of lists throughout the whole formalization, whereas sets would have been easier to model. A different illustration is the function for calculating the set of executable edges: It internally has two different code paths depending on whether there are priorities involved. While the effect on readability is neglegible, it has two further consequences: proofs become more involved, and optimizations are more difficult.

Another important topic to work on are the additional non-trivial optimizations of SPIN, including partial-order reduction [44]. This technique is an important optimization used in SPIN to drastically reduce the size of the state-space. This technique is currently implemented as part of CAVA [4], but needs to be get adapted for Promela. Currently, it is only formalized for another simplified, but abstract, modeling language that can be seen as a subset of Promela.

6 Assembling the Model Checker

From the different building blocks (System Representation from Chapter 5, Automata Intersection from Chapter 3, Finding of Counter-Examples from Chapter 4) given in this thesis, plus the translation from an LTL property into a Büchi automaton described by Schimpf [49], a complete model checker can be assembled and proven to be correct.

In this chapter we will describe how the different parts are to be assembled to form the resulting model checker CAVA. Something similar is also given in our former overview paper [10], where older versions of the different parts where used. Also, in the following, the assembly makes heavy use of the Refinement Framework, while in the paper, a more manual approach has been chosen. Using the Refinement Framework now is favorable, as this is also used in the different parts and therefore the final step, the code generation, can be done nearly automatically; the assembling is transparent for the Framework and it can therefore rely on the implementation work already existing as part of the different building blocks.

For each of the basic building blocks (except for System Representation: it is just encoded as a system automaton), an abstract function is defined. Each is defined in terms of a ***spec***, that is, only the property of the outcome is defined. Later on, the implementations are shown to be a refinement of the abstract definitions, so that the correctness property follows by transitivity.

For the LTL-to-Büchi translation, we require the resulting GBA to be language equivalent to the formula passed, and also to have only finitely many states reachable from the initial states:

definition *ltl_to_gba_spec*
 :: 'prop ltlc ⇒ ('q, 'prop set) igba nres
 where *ltl_to_gba_spec φ ≡* ***spec*** *gba.*
  igba.lang gba = ltlc_language φ ∧ igba gba ∧ finite ((E gba) `` V₀ gba))*

The types involved define the input to be an LTL formula, where the propositions are given by some type variable *'prop*, and the output to be a GBA (in the form of the new automata library as described both in [26] and Section 3.2).

Second, the intersection is given by the following definition, returning two things: First, what the automata framework calls the *graph* of the product automaton, i. e., the automaton without labelling information, and second, a projection from the state of the product automaton onto the original state space of the system automaton. The latter is needed for presenting the counter-example:

definition *inter_spec*
 :: ('s,'prop set) sa
 ⇒ ('q,'prop set) igba
 ⇒ (('prod_state) igb_graph × ('prod_state ⇒ 's)) nres
 where *inter_spec sys ba ≤* ***spec*** *(G, project).*

```
    igb_graph G
  ∧ finite ((E G)* `` V0 G)
  ∧ (∀r.
          (∃r'. igb_graph.is_acc_run G r' ∧ r = project ∘ r')
      ⟷ (graph_defs.is_run sys r ∧ L sys ∘ r ∈ igba.lang ba))
```

The requirements for the result already state most of the main property of the model checker: An accepting run on the product exists iff its projected version is a run in the original system and the trace is in the language given by the property.

Lastly, the search for a counter-example is defined. It shall return *None* when there is no accepting run, else *Some* plus an optional counter-example. Here, the counter-example is optional to allow for algorithms that are fast for stating the existence, but unusable in construction of such a counter-example.

```
definition find_ce_spec
 :: 'q igb_graph ⇒ 'q word option option nres
 where find_ce_spec G ≤ spec result. case result of
    None ⇒ ¬(∃r. igb_graph.is_acc_run G r)
  | Some None ⇒ (∃r. igb_graph.is_acc_run G r)
  | Some (Some r) ⇒ igb_graph.is_acc_run G r
```

Eventually, they can be combined into a definition for a model checker, which takes a system and an LTL formula as input:

```
definition abs_model_check
 :: ('s,'prop set) sa
 ⇒ 'prop ltlc
 ⇒ 's word option option nres
 where
 abs_model_check sys φ ≡ do {
  gba ← ltl_to_gba_spec (LTLcNeg φ);
  (Gprod, project) ← inter_spec sys gba;
  ce ← find_ce_spec Gprod;

  case ce of
    None ⇒ return None
  | Some None ⇒ return (Some None)
  | Some (Some r) ⇒ return (Some (Some (project ∘ r)))
 }
```

This can be proven to have the expected property of returning *None* iff all runs of the system are a model of φ (or equivalent: the language of the system is a subset of the language of φ). If a counter-example is returned, the run is valid but not a model of φ:

```
theorem abs_model_check_correct:
 abs_model_check sys φ ≤ spec result. case result of
    None ⇒ sa.lang sys ⊆ ltlc_language φ
  | Some None ⇒ ¬ sa.lang sys ⊆ ltlc_language φ
  | Some (Some r) ⇒ graph_defs.is_run sys r ∧ L sys ∘ r ∉ ltlc_language φ
```

Having defined the abstract view of such a model checker, it remains to construct an actual model checker. For each of the different building blocks, it is shown that they are a refinement of their corresponding abstract definition as stated above. Those proofs mostly follow from the correctness properties as shown individually plus technical setup for the Refinement Framework. We will omit the definitions and proofs here, for they are mainly boilerplate.

Also, as there exist, for example, multiple implementations for counter-example search, an additional configuration is added that allows to choose at runtime between the different implementations. Eventually, a system-agnostic version of the model checker is defined, named *cava_sys_agn*. Also, the correctness is shown by refinement[1]:

theorem *cava_sys_agn_correct:*
$cava\_sys\_agn \leq abs\_model\_check$

From the system agnostic model checker, the last step is the addition of a system. While we have earlier mentioned that the system is "merely" given as a system automaton, this still has to be implemented. We will highlight this for the Promela setup.

As a first step, we define the system automata given an already translated Promela program (cf. Sections 5.2.2 and following). Such a translated Promela program consists of the state-machine representing the program *prog*, the set of atomic propositions *APs*, and an initial configuration g_0. The mapping to a system automaton is then given by:

states Since we cannot define the states in general, we simply take the universe of the underlying type of configurations. This is sufficient, for only finitely many reachable states are required – which is proven to hold for our Promela implementation (cf. Section 5.2.4).

transitions The Promela implementation provides a successor function *nexts* (which was refered to as *semantic engine* in the Promela formalization). This is sufficient to define the transition relation.

initial states The initial state is the given initial configuration.

labels The labels are defined to be those atomic propositions which are true in the current state.

In code, this reads:

definition *promela_to_sa promg* $\equiv$ ***let*** $(prog, APs, g_0) = promg$ ***in***
$(\!|$
$V = UNIV$,
$E = E\_of\_succ\ (Promela.nexts\ prog)$,
$V_0 = \{g_0\}$,
$L = promela\_props\_ltl\ APs$
$|\!)$

Requiring a compiled program before stating correctness is unsatisfactory. Therefore we implement also the compilation from an AST to the Promela program as part of the

[1] In the actual theories, this refinement is implicit by use of locales. Hence, the theorem itself does not exist, only the lifted property (as above).

model checker. Recall from Section 5.2.3 that for efficiency reasons propositions inside the program are only expressed as pointers into this set. To that end, the Promela compiler also needs to inspect and modify the given LTL property.

```
definition cava_promela cfg ast φ ≡
  let
    (promg,φ_i) = PromelaLTL.prepare cfg ast φ
  in
    cava_sys_agn (fst cfg) (promela_to_sa_impl promg) φ_i
```

Finally, we can show that the resulting model checker is still correct, hiding the translation phases of both the program and the formula.

```
lemma cava_promela_correct:
  case cava_promela cfg ast φ of
    None ⇒ promela_language ast ⊆ ltlc_language φ
  | Some None ⇒ ¬(promela_language ast ⊆ ltlc_language φ)
  | Some (Some ce) ⇒ promela_is_run ast (run_of_lasso ce)
                     ∧ promela_props ∘ run_of_lasso ce ∉ ltlc_language φ
```

Thus, we get a fully verified model checker that is able to understand Promela as part of its input.

As a last step, the code can be exported

```
export_code cava_promela
```

and compiled. Additional setup (like parsing Promela code from some input or parsing LTL) and outputting the counter-example in usable format is not part of theories and has to be implemented in the final code wrapper. This is mostly technical and can be found in the accompanying theories [11].

7 Conclusion

In this thesis, we used the Interactive Theorem Prover Isabelle/HOL to develop a reference implementation of a SPIN-style explicit state LTL model checker. This model checker then is not only proven to be correct, but also executable. Even more, the created model checker is comparative with SPIN – both in input and in runtime (cf. Chapter 5). To our knowledge, on presenting the original paper [10], our work, the CAVA project, was the first successful approach to achieve this.

Further, as part of this model checker, we formalized the semantics of SPINs modelling language Promela (cf. Section 5.2). Again, to our knowledge, on original publication [37] we presented the first result that is also executable.

Other subprojects/developments of this model checker were different formalizations of automata (cf. Chapter 3) and a generic framework for developing and proving algorithms based on Depth-First Search in Isabelle/HOL (cf. Chapter 4).

But besides those usable end-products, this thesis also took an in-depth look at different developing approaches for some of them. We compared various approaches to the formalization of the automata-theoretic foundations in Chapter 3, and different implementations of how to create a framework that covers depth-first search based algorithms in Chapter 4. Comparing those different approaches and implementations, as well as taking into account the lessons learnt from creating and using these approaches and implementations with a particular goal – the model checker – in mind, led to an insight: In the field of Interactive Theorem Proving, the more comprehensive approach is often not the one best-suited.

The insight that too comprehensive libraries or toolkits can be obstacles more than enablers is not new in software development in general[1]. The, in our opinion, interesting observation is that it applies to the world of theorems, too. In our understanding providing comprehensive libraries of theorems is essential for increasing the usefulness of a particular ITP ecosystem, allowing its usage in more circumstances, in turn leading to more formalizations. We cannot – and do not want to – refute this. But as we had to learn, there is a difference between formalizing a particular concept and writing a particular tool.

In the latter case, the goal is not to cover as much as possible about a certain topic but to implement a certain algorithm. What is more, this algorithm must be exportable to code with the resulting code being as efficient as possible; an unusable tool would be the result otherwise. When incorporating a concept or a data structure into such a tool, this concept, or any algorithms on it, thus needs to be expressed in a way as not to decrease runtime performance.

In general, most of the libraries available follow the goal of providing abstract concepts together with their properties. The libraries very seldomly provide an exportable and efficient[2] implementation. And even if they do, the optimizations chosen may not be in

[1] A compilation of libraries/protocols with too broad a goal and the resulting too complex API given at a talk at the 26C3: `https://events.ccc.de/congress/2009/Fahrplan/events/3691.en.html`

[2] "Efficient" here is meant as efficiency of the execution. Proof-efficiency on the other hand often *is* a goal of

line with what is needed by the tool one is developing. For that reason, it is safe to assume that one only has the abstract notions at hand.

Starting with such an abstract notion one has to provide an efficient implementation. On the other hand, quite often one needs only a few of the properties that are provided by the framework. Putting those two points together, it shows that the work needed to shape the provided library in such a way that it can be used successfully in the development of a tool often is larger than the benefit it provides.

Also, we learnt that when developing a framework and its application side-by-side, some general design principles of the framework often have to be rethought. This then led to larger overhauls of the framework itself. This also has to be taken into account for the cost-benefit-analysis of creating a central framework.

Lammich has shown a slightly different way by providing additional automatization for usage in the theorem prover. The main example is the Isabelle Refinement Framework [24] detailed in Section 2.2, and an approach for building DFS-based algorithms was shown in Section 4.8.3. With this approach, there is no need for a central framework. Instead, the manual labor needed to derive the specific results for a particular project is reduced. While we find this still to have some downsides (also given in Section 4.8.3), it serves well as another starting point.

such a library.

Bibliography

[1] Accompanying downloads of CAVA publications. `https://cava.in.tum.de/downloads`. Accessed: 2016-12-22.

[2] R.-J. Back. *On the correctness of refinement steps in program development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978.

[3] R.-J. Back and J. von Wright. *Refinement Calculus — A Systematic Introduction*. Springer, 1998.

[4] J. Brunner and P. Lammich. Formal verification of an executable LTL model checker with partial order reduction. In S. Rayadurgam and O. Tkachuk, editors, *Proc. NASA Formal Methods (NFM '16)*, volume 9690 of *Lecture Notes in Computer Science*, pages 307–321. Springer, 2016.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. MIT Press, 2009.

[6] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3): 275–288, 1992.

[7] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In J. Wing, J. Woodcock, and J. Davies, editors, *Proc. Formal Methods (FM '99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271. Springer, 1999.

[8] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[9] Documentation of Isabelle/HOL. `https://isabelle.in.tum.de/documentation.html`. Accessed: 2016-12-22.

[10] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable LTL model checker. In N. Sharygina and H. Veith, editors, *Proc. Computer Aided Verification (CAV '13)*, volume 8044 of *Lecture Notes in Computer Science*, pages 463–478. Springer, 2013.

[11] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable LTL model checker. *Archive of Formal Proofs*, May 2014. URL `http://isa-afp.org/entries/CAVA_LTL_Modelchecker.shtml`. Formal proof development.

[12] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3):107–114, 2000.

[13] M. d. M. Gallardo, P. Merino, and E. Pimentel. A generalized semantics of PROMELA for abstract model checking. *Formal Aspects of Computing*, 16(3):166–193, 2004.

[14] F. Gava, J. Fortin, and M. Guedj. Deductive verification of state-space algorithms. In E. B. Johnsen and L. Petre, editors, *Proc. Integrated Formal Methods (IFM '13)*, volume 7940 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2013.

[15] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *Proc. Int. Symp. Protocol Specification, Testing, and Verification*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1996.

[16] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Proc. Functional and Logic Programming (FLOPS '10)*, volume 6009 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2010.

[17] G. J. Holzmann. *The Spin Model Checker — Primer and Reference Manual*. Addison-Wesley, 2003.

[18] G. J. Holzmann and R. Joshi. Model-driven software verification. In S. Graf and L. Mounier, editors, *Model Checking Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2004.

[19] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors, *Proc. of the 2nd SPIN Workshop*, volume 32 of *Discrete Mathematics and Theoretical Computer Science*, pages 23–32. American Mathematical Society, 1997.

[20] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[21] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Prog. Lang. Syst.*, 28(4):619–695, 2006.

[22] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In J. N. Matthews and T. E. Anderson, editors, *Proc. ACM Symp. Operating Systems Principles*, pages 207–220. ACM, 2009.

[23] P. Lammich. Automatic data refinement. *Archive of Formal Proofs*, Oct. 2013. URL `http://isa-afp.org/entries/Automatic_Refinement.shtml`. Formal proof development.

[24] P. Lammich. Automatic data refinement. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Proc. Interactive Theorem Proving (ITP '13)*, volume 7998 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2013.

[25] P. Lammich. The CAVA automata library. *Archive of Formal Proofs*, May 2014. URL `http://isa-afp.org/entries/CAVA_Automata.shtml`. Formal proof development.

[26] P. Lammich. The CAVA automata library. In *Isabelle Workshop 2014*, 2014.

[27] P. Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In G. Klein and R. Gamboa, editors, *Proc. Interactive Theorem Proving (ITP '14)*, volume 8558 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2014.

[28] P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In M. Kaufmann and L. C. Paulson, editors, *Proc. Interactive Theorem Proving (ITP '10)*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.

[29] P. Lammich and R. Neumann. A framework for verifying depth-first search algorithms. In *Proc. Certified Programs and Proofs (CPP '15)*, pages 137–146, New York, 2015. ACM.

[30] P. Lammich and R. Neumann. A framework for verifying depth-first search algorithms. *Archive of Formal Proofs*, July 2016. URL `http://isa-afp.org/entries/DFS_Framework.shtml`. Formal proof development.

[31] P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In L. Beringer and A. Felty, editors, *Proc. Interactive Theorem Proving (ITP '12)*, volume 7406 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2012.

[32] X. Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43:363–446, 2009.

[33] A. Lochbihler. Native word. *Archive of Formal Proofs*, Sept. 2013. URL `http://isa-afp.org/entries/Native_Word.shtml`. Formal proof development.

[34] V. Natarajan and G. J. Holzmann. Outline for an operational semantics of PROMELA. In J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors, *Proc. of the 2nd SPIN Workshop*, volume 32 of *Discrete Mathematics and Theoretical Computer Science*, pages 133–152. American Mathematical Society, 1997.

[35] R. Neumann. A framework for verified depth-first algorithms. In A. McIver and P. Höfner, editors, *Proc. Workshop on Automated Theory Exploration (ATX '12)*, pages 36–45. EasyChair, 2012.

[36] R. Neumann. Promela formalization. *Archive of Formal Proofs*, May 2014. URL `http://isa-afp.org/entries/Promela.shtml`. Formal proof development.

[37] R. Neumann. Using Promela in a fully verified executable LTL model checker. In D. Giannakopoulou and D. Kroening, editors, *Proc. Verified Software: Theories, Tools and Experiments (VSTTE '14)*, volume 8471 of *Lecture Notes in Computer Science*, pages 105–114. Springer, 2014.

[38] T. Nipkow and G. Klein. *Concrete Semantics — With Isabelle/HOL*. Springer, 2014. URL `http://concrete-semantics.org`.

[39] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[40] T. Nishihara and Y. Minamide. Depth first search. *Archive of Formal Proofs*, June 2004. URL `http://isa-afp.org/entries/Depth-First-Search.shtml`. Formal proof development.

[41] L. Noschinski. Graph theory. *Archive of Formal Proofs*, Apr. 2013. URL `http://isa-afp.org/entries/Graph_Theory.shtml`. Formal proof development.

[42] L. Noschinski. A graph library for Isabelle. *Mathematics in Computer Science*, 9(1): 23–39, 2015.

[43] R. Pelánek. BEEM: Benchmarks for explicit model checkers. In D. Bošnački and S. Edelkamp, editors, *Model Checking Software*, volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer, 2007.

[44] D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. L. Dill, editor, *Proc. Computer Aided Verification (CAV '94)*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 1994.

[45] F. Pottier. Depth-First Search and Strong Connectivity in Coq. In D. Baelde and J. Alglave, editors, *Vingt-sixièmes journées francophones des langages applicatifs (JFLA '15)*, Jan. 2015.

[46] Promela Database. URL `http://www.albertolluch.com/research/promelamodels`. Accessed: 2016-12-22.

[47] Promela Homepage. `http://spinroot.com/spin/whatispin.html`. Accessed: 2016-12-22.

[48] Promela Manual Pages. `http://spinroot.com/spin/Man/promela.html`. Accessed: 2016-12-22.

[49] A. Schimpf. *Eine vollständig verifizierte, ausführbare Formelübersetzung à la SPIN*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 2015.

[50] A. Schimpf, S. Merz, and J.-G. Smaus. Construction of Büchi automata for LTL model checking verified in Isabelle/HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Proc. Theorem Proving in Higher Order Logics (TPHOLs '09)*, volume 5674 of *Lecture Notes in Computer Science*, pages 424–439. Springer, 2009.

[51] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In N. Halbwachs and L. Zuck, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2005.

[52] N. Shankar. Trust and automation in verification tools. In S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, editors, *Proc. Automated Technology for Verification and Analysis (ATVA '08)*, volume 5311 of *Lecture Notes in Computer Science*, pages 4–17. Springer, 2008.

[53] A. Sharma. A refinement calculus for Promela. In *Proc. International Conference on Engineering of Complex Computer Systems (ICECCS '13)*, pages 75–84. IEEE, 2013.

[54] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[55] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. Logic in Computer Science (LICS '86)*, pages 332–344. IEEE, 1986.

[56] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2: 461–478, 1992.

[57] C. Weise. An incremental formal semantics for PROMELA. In *Proc. of the 3rd International SPIN Workshop*, 1997.

[58] L. Zhao, J. Zhang, and J. Yang. Advances in on-the-fly emptiness checking algorithms for Büchi automata. In *Proc. International Conference on Advanced Computational Intelligence (ICACI '12)*, pages 113–118. IEEE, 2012.