TECHNISCHE
UNIVERSITÄT
MÜNCHEN

Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Entwurfsautomatisierung
Univ. Prof. Dr. -Ing. Ulf Schlichtmann

PhD-Thesis

Automated Assertion Transformation Across Multiple Abstraction Levels

Thomas Steininger

Lehrstuhl für Entwurfsautomatisierung
der Technischen Universität München

Automated Assertion Transformation Across Multiple Abstraction Levels

Thomas Steininger

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. sc.techn. Andreas Herkersdorf

Prüfer der Dissertation:

1. Hon.-Prof. Dr.-Ing. Wolfgang Ecker
2. Univ.-Prof. Dr.-Ing. Ulf Schlichtmann

Die Dissertation wurde am 29.10.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 30.09.2009 angenommen.

PhD-Thesis

Institute of Electronic Design Automation
Univ. Prof. Dr. -Ing. Ulf Schlichtmann
Department of Electrical and Information Technology,
Technische Universität München

in Cooperation with

Infineon Technologies AG Munich
IFAG COM BTS MT SD
Dr. -Ing. Matthias Bauer
Prof. Dr. -Ing. Wolfgang Ecker

Author: Thomas Steininger

Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Methodik zur automatisierten Transformation von Assertions, eine erweiterte Auswertungssemantik und eine dazugehörige Transformationssprache entwickelt. Unter Benutzung dieser Methodik ist es möglich, bestehende Assertions auf andere Abstraktionsebenen zu transformieren. Dies erlaubt die Wiederverwertung bestehender Assertions, die für abstrakte Modelle geschrieben wurden, für die Entwicklung von weniger abstrakten Modellen, wobei die Gefahr, manuell Fehler während der Transformation einzuführen, minimiert wird.

Abstract

In this work, a methodology for automated assertion transformation, an extended assertion evaluation semantics, and a corresponding transformation language has been developed. Using this methodology it is possible to transform existing assertions to other levels of abstraction. This allows reusing existing assertions, which were written for abstract models, for the development of less abstract ones while minimizing the danger of manually introducing errors during the transformation process.

Acknowledgment

This thesis was conducted in cooperation with Infineon Technologies AG, namely the department IFAG COM BTS MT SD headed by Dr. Matthias Bauer, and with Technische Universität München at the department Electrical Engineering and Information Technology at the institute for Electronic Design Automation, led by Professor Ulf Schlichtmann. I would like to thank Professor Schlichtmann for giving me the opportunity to carry out this thesis at his institute and also for taking over the role of co-advisor for this work. I also want to thank Dr. Bauer for making this thesis possible, as well as for his support and encouragement.

I owe special thanks to my advisor Professor Wolfgang Ecker. He was always willing to take time for discussions about all kinds of problems and new ideas. His continuous support and encouragement were a great motivation for this work. I also want to thank all my other colleagues at Infineon for a good collaboration.

I want to express my gratitude towards Professor Manfred Glesner and Dr. Thomas Hollstein from Technische Universität Darmstadt for their help.

I especially want to thank my girlfriend Yu Hu for her patience. She always supported and encouraged me. I would also like to thank my family who never failed to encourage me during this work and always helped to keep me calm.

Finally, I want to thank my colleague and friend Michael Velten who contributed a lot to this work and who spent a lot of time with discussing problems with me.

Thomas Steininger
Munich, November 27, 2009

Contents

List of Tables

List of Figures

List of Listings

1 Introduction

1.1 The Pervasiveness of Embedded Systems

Technical progress over the past decades has made high technology more and more part of people's every day life. Not only the number of devices, but also the complexity of these systems has increased drastically. Nevertheless, most people are only partly aware of the extent of this phenomenon, since in many cases the complex systems are part of seemingly simpler devices. Today, these systems can be found in almost all aspects of daily life: Domestic appliances like dishwashers or washing machines include them for controlling water supply or dosage of detergents in order to clean dishes or clothes as efficiently as possible. HiFi systems like CD players include controlling systems for the laser that reads the information stored on an audio CD. Mobile phones have rapidly spread and have already become a standard accessoir for most people. Clothes with built-in mp3 players are created. Cars include electronics for engine control or electrical control of locks, while current research aims at features as break-by-wire and warning systems that can alert the driver of potential dangers like collisions or leaving the road; these systems are intended not only alert the driver in the end, but also automatically correct wrong behavior. When compared with personal computers, these complex systems are not or not directly visible to the user. Instead they are embedded within a larger system and can only be accessed by a restricted interface.

1.2 Quality of Embedded Systems

In contrast to personal computers where a bug in a program can be corrected by a patch of the affected Software (SW) even after the product has been delivered to the customer, access to embedded systems for this purpose is difficult at best. Additionally, bugs in the Hardware (HW) that has already been delivered cannot be fixed at all.

Considering that at least some of these embedded systems control safety critical tasks like the break-by-wire system mentioned above, it is essential that the quality of embedded systems is checked according to strict standards during all stages of the

design process. Whenever a bug is detected during those tests, the reaction depends on the location, the severity of the encountered bug, and the stage in the design process. While SW bugs can usually be fixed without much additional effort, bugs in a HW component are not as easily fixed. In simple cases maybe the SW of the embedded system can be adapted to take the bug into account. In more serious cases this might lead to a complete respin of the system development and even fabrication if the system has already been shipped.

In order to minimize the immense costs of having to start the whole development process again from the beginning (even worse if the system has already been fabricated), the usual course of system development follows the so called VP model [1], which is shown in Figure 1.1.

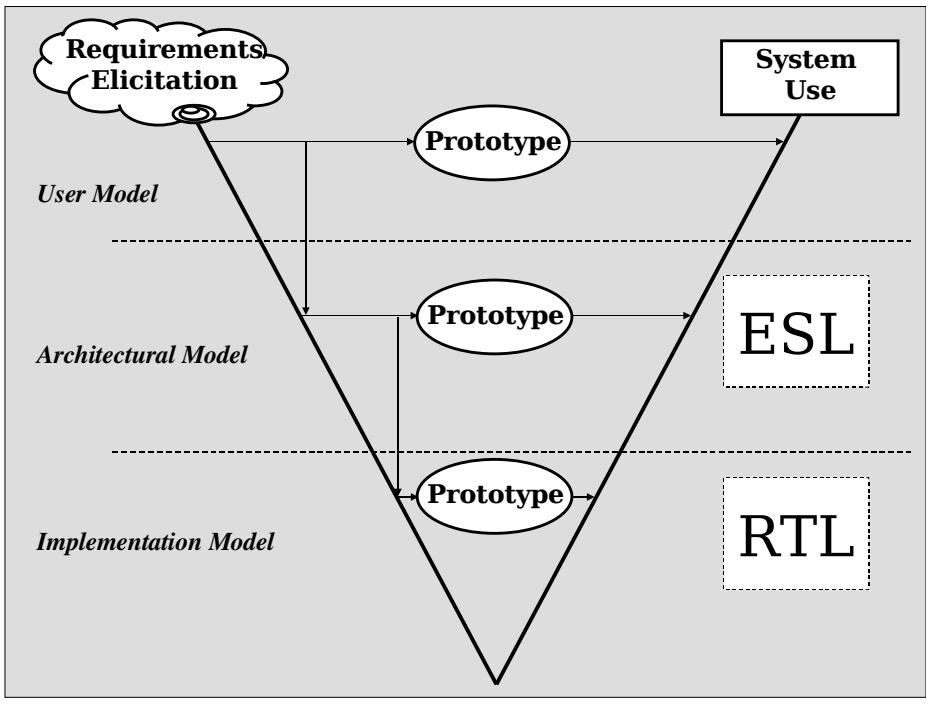

Figure 1.1: The VP Model [1] for System Development

According to this process, system development starts with a very abstract model that neglects many of the characateristics of the final design in order to focus on the important aspects as for instance the algorithms involved and the basic functionality coverd by the system. The corresponding prototype is then subjected to extensive tests concerning these features. If there are already bugs within the implementation, only a small part of the development work has to be redone. This early model can also help with architecture exploration, since several design decisions only have to be done for less abstract models.

After this model has passed an appropriate number of tests, a more complex model is developed. In this context, the more abstract models serve as golden reference for the newly developed ones. At the same time, the more abstract models can be used for early software development or system documentation, before the final version of the design is even ready.

1.3 Model Equivalence

If various models on different abstraction levels exist, a method for checking the functional equivalence of these models is needed. Several different technologies exist that can cover at least a part of this problem, depending on the exact application.

1.3.1 Connection Verfication

There are already efforts of standardizing the description of an Intellectual Property (IP) in order to ease its integration into a system of another company. The SPIRIT consortium [3] for instance already published a standard for the description of Register Transfer Level (RTL) model interfaces. At the moment, the standard is being extended in order to also cover Transaction Level (TL) models. A SPIRIT description captures the necessary data in several Extensible Markup Language (XML) files, which are then delivered to customers in combination with the precompiled IP. A specific IP may include a TL and an RTL implementation.

Using the SPIRIT description as a reference, it is possible to check that the interfaces of both models are consistent. Note that this does not require that they are identical (which will not be the case anyway), but just that the available connections are the same. If for instance, the TL model has a transaction port for a read access to a connected memory module with an address parameter and a data parameter, then the RTL implementation should at least contain an address output to the memory and a data input from the memory.

If the modules in question consist of several submodules then the correct interconnection of these submodules can be verified as well.

1.3.2 Formal Verification

Formal verification includes a variety of different technologies which all examine the model or models in question based on mathematical and logical criteria. In general, two different kinds of formal verification approaches can be distinguished:

1. Equivalence Checking: This technique examines two models and breaks down the corresponding functionality into small logical blocks. In order to be equivalent, both models have to behave in exactly the same way when they are faced with the same circumstances. For that purpose, the involved algorithms assume that both models are not equivalent and try to lead this assumption to a logical contradiction.

2. Property Checking: This technique tries to prove that a model fulfills a certain set of specified properties which reflect the original specification. When used to prove the equivalence of two models using this technique, it is required to check all properties with both models.

Formal techniques cover all possible transition paths through the design state space (in contrast to just a large number of sample tests). The larger the model in question, the more resources are necessary for a formal analysis. Due to the problem of state space explosion - the size of the state space increases exponentially with the number of design states - today's complex systems simply cannot be evaluated using formal verification.

Equivalence Checking

In order to at least reduce the problem of state space explosion, it is necessary to have efficient algorithms for the creation of the simplified models which are then compared.

There are already approaches for the formal equivalence checking between RTL models and gate level models. On the other hand, these concepts cannot be transferred easily to higher levels, as for instance the equivalence check between TL and RTL. This is partly the result of the very abstract modeling style on higher abstraction levels. Complex function calls and transfers of complex data objects cannot easily be reduced to logical representations as needed for the equivalence checks.

While a generalized formal equivalence check between TL and RTL is not possible yet, there have been first approaches for the application of formal equivalence checking in a more restricted context. The approach discussed in [4] for instance uses the equivalence checks in combination with a high-level synthesis tool. In other words, the less abstract model is not written by hand in this case and the additional data used or acquired during the model transformation is then used in order to formally verify the equivalence of original and generated model.

Property Checking

When applying property checking to models of different abstraction levels, checking the same property with all models might require the adaptation of the property description. This leads to similar problems as faced when using assertions (see below).

1.3.3 Simulation-Based Verification

Instead of analyzing a model mathematically, simulation-based verification uses a special testbench that generates stimuli sent into the model and receives the corresponding responses of the model. The generated stimuli might be either specify directly - so called directed tests - or created randomly, with or without constraints. The latter approach is much more efficient of finding bugs, since the randomization leads to checks that normally would not have been specified; thus, it is possible to detect bugs resulting from dependencies that nobody had considered before. The most common approach for simulation based verification is a combination of these two approaches: The testbench starts with generating random stimuli and gathers the results. These results are then stored in a database for several kinds of coverage data. Examples for coverage data include:

- Code coverage measures which parts of the code have been executed how often
- Functional coverage measures how often a certain functionality has been used
- Assertion coverage measures how often a certain assertion has fired or not resulted in a vacuous success (see below)

Since a complete check including every possible combination of values by simulation is impossible for realistic systems[1], certain coverage goals have to be defined (for instance, "every memory address has to be written three times"). As soon as a coverage goal has been achieved, this particular situation does not have to be checked anymore. As a result, the stimuli generation can be constrained to the remaining situations, which in turn improves the probability of achieving the remaining coverage goals faster. This approach is called *Constrained Random Simulation*.

Depending on the abstraction level of the Design Under Test (DUT), the interfaces between the testbench and the DUT are either method based for TL models, or signal based for RTL models.

[1] A chip with fifty one-bit registers has 2^{50} different states. When checking one million states per second, an exhaustive simulation would take about 850 years.

However, in order to ensure equivalence between the corresponding models, it would be beneficial if the same testbench can be used for the verification of both models. This is possible by using dedicated components called Bus Functional Model (BFM) or Transactor that "translate" between the high level and the low level protocol. While it does not matter for this method if the testbench is implemented on a higher or lower level, using an abstract testbench has two advantages:

1. It is easier to specify abstract test patterns, since the complex signal protocol does not have to be taken into account.
2. According to the VP model, the high-level implementation of the design is created first. Additionally, the creation of the Transactors can be postponed to the start of the low-level implementation. This means that the verification of the high-level model can be started sooner.

When using this method to check the equivalence of a TL model and an RTL model, the testbench sends the same stimuli to both models, and stores the response in two separate data bases. The corresponding structure is shown in Figure 1.2.

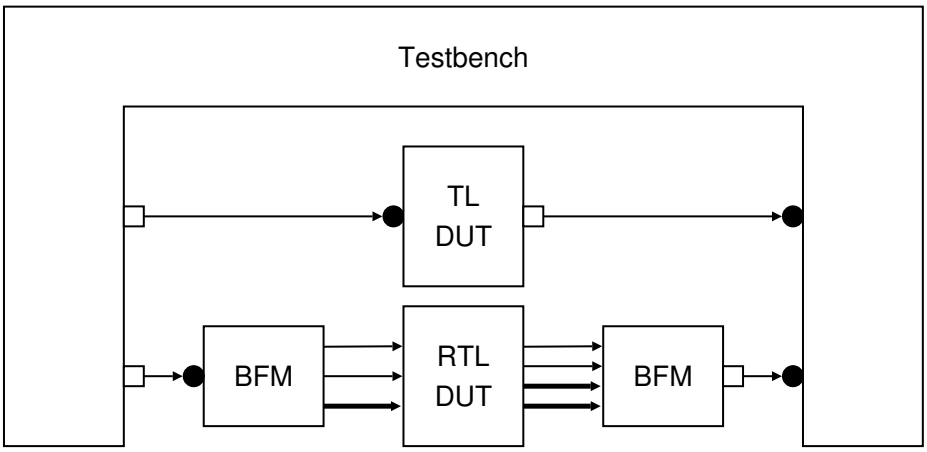

Figure 1.2: Use of Testbenches for Multi Abstraction Checks

While the stimulus generation can be used for both models in the same way, the coverage goals usually have to be redefined for the RTL model, since the modeling styles and paradigms differ greatly in some regards. Low-level structures might not even exist in high-level models, on the other hand, several distinct structures of the high-level model might be mapped on a shared structure in the low level model.

The use of Transactors is not restricted to the direct interaction with a testbench though. It is also possible to include an RTL component including the corresponding Transactors into an existing TL system, replacing the original TL component.

Assertions

Assertion Based Verification (ABV) uses specialized watchdog components called assertions that are either included in the design itself or attached to an existing design from the outside. Assertions specify a required temporal and logical behavior that must never be violated. Assertions can be used in addition to both formal verification (property checking) and simulation-based verification. In the former case, assertions are used for specifying both the properties of the design and assumptions about the behavior of external components. In case of simulation-based verification, assertions are simulated in combination with the model and continously check if the specified behavior is violated. If this is the case, the user is immediately informed about this error and - depending on the configuration of the assertion - the simulation is stopped.

In case of simulation-based verification the immediate notification of an error makes localization of the actual error much easier when compared to the tedious backtracking if an error shows up in the testbench (since the error had to propagate to the output first, it could have happened quite some time before it is noticed). Additionally, since assertions monitor the internal behavior of a model in contrast to the external monitoring a testbench does, they ar able to check behavior errors that only rarely become visible at an output. As a result of applying assertions to a design, the amount of coverage goals inside a testbench can be reduced.

In contrast to testbenches, assertions can and should also be written by designers while creating a model. This makes it possible to specify constraints of the model interface, for instance "port A and port B may never be one at the same time". Whenever this model is used within an existing system, every wrong usage would be notified immediately.

Since assertions can also be used in combination with simulation-based verification, it is also possible to apply them to equivalence checking. Besides adding corresponding assertions to both the TL model and the RTL mode, there is also the additional possibility of writing mixed level assertions that try to correlate the behavior of both models at certain matching points (for instance, in both models a register has to be written). Due to the variety of modeling styles on higher levels of abstraction and the fact that both models might differ greatly concerning the timing, finding these matching points within the various functional blocks of both models might be difficult.

While wide spread for RTL verification, only recently first approaches for applying ABV to TL models have been developed.

1.4 Motivation

It is vital to ensure consistency between the various models developed during the design process, which are written on different abstraction levels.

Right now, there is no methodology in place for the automated transformation of TL designs down to RTL. Instead, all models are transformed manually, which introduces additional sources for errors.

Unfortunately, verification efforts are mostly focused on checking the correctness of RTL designs. Simulation-based verification using testbenches is applied to TL models as well, in some cases the corresponding testbenches are even reused for RTL models later on. However, in many cases TL testbenches are much less sophisticated than their RTL counterparts. Furthermore, while the usage of assertions in simulation based verification for RTL is common, it is far from wide spread. On TL on the other hand it is almost non-existent (this is partly based on the fact that TL assertion languages are mostly still in development, however). Applying the same sophisticated test strategies already for TL models would allow an earlier identification of errors which in time reduces the overall verification effort. This is even more important since future designs will have a drastically increased complexity which in turn makes early identification of errors critical.

While TL testbenches can be reused for RTL in combination with Transactor components, it is not possible to reuse TL assertions in that way, since they are partly included within the design or access data not visible at the model interface.

As a result, even if assertions will be used more often in combination with TL models in the future, they will have to be manually transformed to a corresponding RTL equivalent, which may introduce additional errors, just as mentioned for the design refinement above.

This work addresses this lack of automated assertion transformation. Based on an existing assertion language capable of handling both RTL and TL assertions (including the various TL sublevels), a dedicated transformation language has been created that allows the automated transformation of assertions from and to all supported abstraction levels.

The peculiarities of all supported abstraction levels have been considered and the underlying assertion language has been enhanced by several features required for a consistent specification of assertions on these abstraction levels. As a consequence, consistent mixed level assertions are supported as both original and generated assertions. This allows the specification of assertions for the equivalence checking method based on matching points mentioned above.

The transformation descriptions can be formulated in a flexible way, which allows a reuse of existing transformation rules for other assertion transformation tasks as well.

The expected benefit is the efficient reuse of TL assertions for RTL verification, while only requiring little effort to adapt existing transformation rules. If desired, transformation of RTL assertions for the verification of TL models is possible as well.

1.5 Outline

This work is structured as follows:

Chapter 2 describes the problems an assertion transformation approach has to face and outlines possible solutions.

Based on this analysis, Chapter 3 introduces requirements for assertion transformations, split into requirements for the underlying assertion language and requirements for the transformation description.

Existing solutions for these are requirements are analyzed in Chapter 4 and the new work is motivated.

Chapter 5 describes the concepts introduced in this work, split into the extension of an existing assertion language and the development of a dedicated transformation language.

A formal representation of the modified assertion language is described in Chapter 6.

In Chapter 7 an overview over the application framework is given.

The results presented before are used in Chapter 8, which describes how to systematically apply the transformation language using an example that covers all relevant abstraction levels.

Finally, Chapter 9 summarizes the scientific contribution of this work and outlines further directions.

2 Problem Statement and Targeted Approach

This chapter discusses the problems an assertion transformation approach has to deal with and also outlines possible ways of solving these problems.

2.1 Today's Top-Down Design

Today, designs are created in a top-down process, beginning with a highly abstract model, containing only the information provided by the design specification that serves as a Virtual Prototype (VP) for SW validation and as a golden reference for later models. Partially, more abstract models are used to validate specific aspects, as for instance the algorithms behind a 3G receiver. Due to the high level of abstraction and the ensuing absence of details, the simulation of the VP can be done much faster and with greatly reduced computation effort. The following models gradually add more and more details, while still being consistent with the VP.

2.1.1 Abstraction Levels

In order to provide means for easier model comparison concerning the amount of details involved, several attempts of defining specific abstraction levels have been made. These abstraction levels differ in the way timing is modeled, how data is exchanged between different functional units, how this data is represented, etc. Most of these attempts differ in the names, number, and characteristics of levels defined. One example of how these differences can be categorized is presented in [5]. An example for a list of abstraction levels can be found in [6]. All in all, there are no established standards which clearly define all abstraction levels and what characteristics a model written at a certain abstraction has to show.

The most common abstraction levels are the following[1] (ordered from most abstract to most detailed):

[1] While the names may differ depending on the source, the underlying concepts are usually present in all approaches.

- Specification Level: This level only deals with the information relevant for correct system functionality. Timing is only considered when absolutely essential for the system behavior; similarly this levels rarely considers functional partitioning into several components, and thus, there is no concurrency either. Data processing is done in the most abstract way possible while using abstract data representation.

- TL: This level introduces partitioning of the model into different functional blocks which have to exchange data according to one or several protocols. This data exchange is handled by so called transactions, usually modeled as function calls, using abstract data types for parameters and return values; these transactions can be used to encapsulate certain functionality as well as to hide protocol details. Since a Transaction Level Model (TLM) is frequently used as VP, decisions concerning elements accessible by the SW have to be made here and kept for later design stages. In contrast to the other levels mentioned, this level does not represent one specific abstraction of a design but can be further divided into several sublevels. The Open SystemC Initiative (OSCI) TLM standard has introduced specific names for these sublevels, but unfortunately their characteristics are not clearly defined:

  - Programmer's View (PV): This sublevel is located only slightly below the Specification Level. PV models do not consider timing in any way, so synchronization between components is restricted to zero-delay wait statements and specific events in combination with conditional wait statements.
  - Programmer's View with Timing (PVT): This sublevel additionally introduces the notion of time, modeled by absolute time values. Every kind of functionality including transactions can consume a certain amount of time. Synchronization is done via timed wait statements, in addition to the methods available on PV.
  - Cycle Approximate (CA): This sublevel is very similar to PVT, but instead of using arbitrary time values within the components, all delays are replaced by cylic delays, expressed as a multitude of one (or several) clock periods. The available synchronization schemes match those of PVT.
  - Cycle Callable (CC): The final TL sublevel replaces the cylic time of the CA models by the introduction of one or several clock signals. The synchronization between different component happens only according to these clock signals.

- RTL: This level models both functionality and structure of the design in a very detailed way. Synchronization is done via clock signals as in Cycle Approximate (CA), but additionally the TL concept of transactions is replaced by signals and

corresponding protocols (e.g., handshake). These signal protocols exactly represent the behavior of the final silicon chip and can be automatically synthesized by corresponding tools. For that purpose, the abstract data types of the upper abstraction layers are replaced by HW data types. Since handling of complex protocols requires more effort than a simple high-level function call, RTL models often implement Finite State Machine (FSM)s for that purpose. The complete description can be easily mapped to simple logical circuits, like AND gates and multiplexors, while abstracting away everything but their functionality.

- Gate Level: This level introduces the other parameters of the low-level logical circuits comprising the RTL description. In adddition to their functionality parameters like throughput time and power consumption are modeled.

2.1.2 Refinement

In order to ensure continuity during the design process, whenever a less abstract model is created based on a more abstract one, the differences between the abstraction levels have to be kept in mind. In this context the term refinement is often used:

Definition 1 *Refinement describes the transition between different levels of abstraction[2], for example the transformation of a Programmer's View with Timing (PVT) model to an RTL model. This transition includes the removal of information no longer relevant on the new abstraction level, as well as the inclusion of additional information that was not available on the original level.*

Although number and exact details of the different abstraction levels differ depending on the source, the characteristics used for distinguishing the various levels are the same in all approaches (modeling of time, structure, etc.). As long as refinement processes are discussed in relation to these characteristics and not to particular abstraction levels, everything discussed is generally valid, independently of the definition used. For that reason, the remainder of this work will use the terms and abstraction levels presented above.

The effects of design refinement can be divided into effects on the module interfaces and effects on the module implementation. Interface changes foremost happen at the transition between TL and RTL since the abstract function calls of TL are replaced by complex signal protocols on RTL or vice versa. Implementation changes are less obvious and include the aforementioned topics of timing and data representation.

[2]This term is usually applied to a transformation towards a less abstract version, but it is not restricted to that.

Since an equivalence check between two different implementations of a complete system is very difficult and resource intensive, in many cases a BFM, also called transactor, is used which transforms the interface of a component modeled on one abstraction level to another one. This allows to create a less abstract version of a single component first and then to simulate this component within the existing abstract system.

When creating a design in a top-down process according to the abstraction levels described in Section 2.1.1, the following refinement steps have to be performed:

- Specification Level to Programmer's View (PV): The monolithic functionality of the Specification Level Model has to be partitioned into smaller components which exchange data through transactions.
- PV to Programmer's View with Timing (PVT): The purely untimed PV model is enhanced by the addition of timing information.
- PVT to Cycle Approximate (CA): The timing information expressed by absolute values is converted to cyclic timing.
- CA to Cycle Callable (CC): The cyclic timing is replaced by the usage of one or several dedicated clock signals.
- CC to RTL: Transactions are replaced by elaborate signal protocols. Additionally, data representation is changed to synthesizable data types. Finally, the functionality is modeled with great detail, including the use of state machines and the choice of specific computation architectures.
- RTL to Gate Level: All functional blocks are annotated with information concerning their timing and power consumption.

In most cases, there is not one prototype for each of these abstraction levels. Thus, the resulting refinement steps might combine several of the steps mentioned here. However, the underlying concepts are not touched by this.

It has to be noted that while some of these refinement steps can be automated (for some, automation has even become the standard procedure), all of them require that the details added on the less abstract level are provided to the automation tool. While this problem can be (partially) solved - for instance, most tools for the synthesis from RTL to Gate Level provide easy means for specifying the required data (choice of a suitable technology, clock frequency, etc.) - this approach also limits flexibility. Thus, a completely automated and flexible refinement without any kind of user interaction is - and very probably will always be - impossible.

2.2 Assertion Based Verification

ABV is one of the most effective way of checking designs for both intent (done by the designer during the creation of the model) and for correct functionality (done by a verification engineer after the design process has been completed). The approach uses so called assertions, code constructs that monitor a specific part of the design and issue an error message as soon as the observed behavior differs from a specified pattern.

The use of assertions within a design is complementary to the use of testbenches in simulation based verification and can also be used for formal verification. Assertions offer an easy way of specifying desired behavior and being notified immediately when this behavior is violated. Since assertions may be part of the design and thus may have immediate access to internal design objects, it is much easier to locate the origin of an error. In other cases the verification engineer has to rely on the error propagating out of the design, in which case the potential delay can complicate correlating the error to its reason.

While ABV is a frequently applied technology with regards to RTL design, only recently the first successful methods for applying assertions to higher levels of abstraction, namely the various TL sublevels, have emerged. This is based on the fact that most of the existing ABV approaches make heavy use of the specifics of RTL models: Assertions have to run synchronously to the design which is usually achieved by relying on clock signals for triggering the assertion evaluation. Due to this, most current approaches are ill equipped to deal with the very different concepts of timing on TL. Additionally, since they are mostly signal centric, the use of transactions as a means to abstract away parts of the design details causes further problems.

2.3 Assertion Refinement

Similarly to a design, high-level assertions can be refined into less abstract assertions in order to perform the same checks on the refined design. Since assertions are closely linked to a design, concerning both the behavior and the structure, assertion refinement follows the same steps and guidelines as design refinement.

Providing a methodical approach for assertion refinement that includes some checks for the correctness of the refined assertion could decrease the effort for the overall verification process. Since the refined assertion, as an additional source of simulation errors, has already passed these checks, the main effort can be used for the verification of the design refinement. In order to ensure the correct assertion refinement, the underlying structure of the assertion has to be kept as much as possible. If

different assertion languages are used for the different abstraction levels, this structure preservation gets more complicated due to the conceptual differences between most assertion approaches. Additionally, creating mixed-level assertions - especially useful for the aforementioned example of simulating a low-level component within a high-level environment using transactors - become almost impossible. As a result, assertion refinement is easiest when applied to a language supporting all abstraction levels involved.

Nevertheless, even given such an assertion language, certain problems arise due to the differences in the abstraction levels themselves. For instance, the different synchronization schemes on the various TL sublevels and RTL have to be matched somehow and the notion of TL transactions has to be converted to a comparable concept on RTL.

The refinement process should be captured in a way that allows easy reuse of the transformation description - in case the abstract assertion has been changed, so that the refined assertion does not have to be rewritten from scratch. Additionally, there might be similar assertions in different designs that have to be refined. If the assertion refinement can be described in a flexible way, it could be possible to apply the existing description to these similar assertions as well.

2.3.1 Example

```
1 property p_DATA_PIPE_tlm
2 int D1;
3 #1{PUT'END}{true, D1=PUT.X}
4 |->
5 #{1:6}{GET'END}{GET.X==D1};
6 endproperty
```

Listing 2.1: FIFO assertion for TLM

```
1 property p_DATA_PIPE_rtl
2 int D1;
3 #1{CLK'POS}{SND_ACK, D1=SND_DATA}
4 |->
5 #{1:6}{CLK'POS}{RCV_ACK && (RCV_DATA==D1)};
6 endproperty
```

Listing 2.2: FIFO assertion for RTL

Figure 2.1 shows a FIFO circuit, once modeled on Transaction Level Model (TLM) and once on RTL. The Listings 2.1 and 2.2 give an example of what behavior could

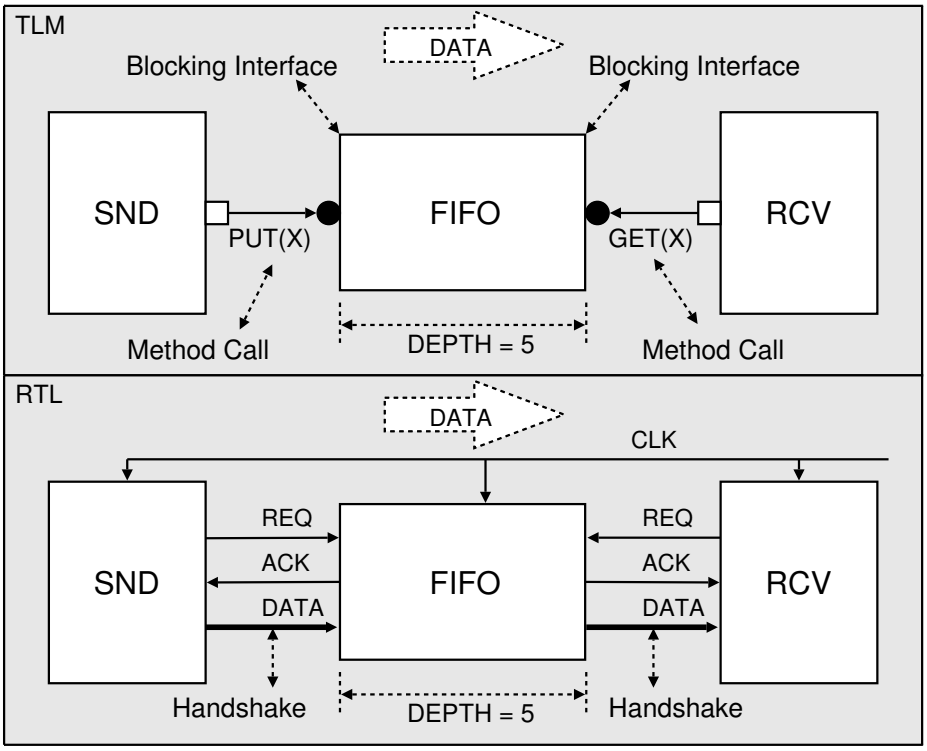

Figure 2.1: FIFO Example

be checked by an assertion in this example: With a given FIFO depth of five, both assertions check that each successful write access to the FIFO is followed by a read access providing the same data within the next six read accesses.

As can be seen, the underlying structure of both assertions are quite similar, although the details are necessarily different due to the different abstraction levels.

2.4 Targeted Approach

The approach presented in this work provides solutions for assertion refinement taking into acount the problems stated above. These solutions consist of two parts: First, an assertion language capable of TL, RTL, and mixed-level assertions is enhanced by several features which are required to map concepts from one abstraction level to another; second, a framework consisting of a new language for the description of the refinement process combined with a refinement generator is introduced.

This framework allows an easy refinement of existing assertions towards lower, higher, and mixed abstraction levels while preserving their structure. Additionally, refinement rules can be written in a flexible way so that they can easily be reused and applied to other assertions as well.

3 Requirements and Objectives for Assertions Refinement

In this chapter, the specific requirements for an assertion refinement approach covering all TL sublevels as well as RTL are listed and explained. First, requirements for the underlying assertion language are discussed, afterwards, the requirements for the transformation language are detailed. The text will provide references to the requirement summary at the end of the chapter (see Appendix A) as Rx where x represents the number of the requirement.

3.1 Requirements for the Assertion Language

In order to successfully refine existing assertions, the assertion language has to fulfill certain conditions as well. The following section details these requirements. Beginning with some general requirements which are valid for all kinds of assertions, requirements based on the covered abstraction levels, pipelined execution, and the connection to SystemC as the corresponding design language are listed.

3.1.1 General Requirements

Assertions are defined to monitor a design - either constantly or at certain points in time - for any mismatch between the actual behavior and a specified desired behavior. In order to fulfill this role as mere observers, assertions may not influence the design behavior in any way (R 1). Additionally, assertions only monitor a certain part of a design, so that changes of the remaining design parts that are entirely unrelated to the behavior checked by the assertion may not influence the assertion results (R 2); neither may these assertion results be influenced by adding further assertions to the design or removing them (R 3).

There are several different ways in which assertions can be used to support the verification of designs:

- Dynamic Verification: The Design Under Verification (DUV) is simulated in combination with a testbench module that applies stimuli to the design and evaluates the corresponding data provided by the design. This data can be stored in a coverage database and / or compared to the results of a golden reference model. This method has the advantage of easy use and low resource consumption, but it is almost impossible to guarantee that a design is absolutely error free.

  Assertions in Dynamic Verification can be used to immediately detect design errors during the simulation run; additionally they can be used to provide coverage data for the testbench.

- Formal Verification: This verification method does not rely on testbenches or other ways of stimulus generation. Instead, abstract properties are formulated which the design has to fulfill. The formal checker then tries to falsify the given properties. If these tries result in a contradiction, the corresponding property is proven. The advantage of proving that specified properties are fulfilled at all times without having to simulate every possible combination of inputs is offset by the amount of resources this method requires for formally verifying even a medium sized design.

  Assertions in Formal Verification are used for specifying both the properties to be proven and certain assumptions that can be made about the behavior of the environment.

- Semiformal Verification: This combination of the two methods mentioned above starts out with a normal simulation run during which "states of interest" are collected. Beginning with each of these states of interest, a bounded formal check[1] is then started. Basically this method enhances the probability of detecting design errors, but without the certainty of a pure formal check.

  Assertions in Semiformal Verification can serve for all purposes described for dynamic verification and formal verification.

Since the targeted approach deals with dynamic verification, assertion evaluation has to be done during the simulation run (R 4) while the assertions also have to be able to provide coverage information (R 5).

3.1.2 Abstraction Levels

As mentioned in Chapter 2.1.1, a typical design process spans several abstraction levels with corresponding prototypes created on these levels. Currently, there are no

[1] "Bounded" in this context means that the formal check is not completed, but only executed to a certain iteration depth, the bound.

attempts to bring ABV to the specification level yet, while gate level on the other hand introduces a lot of data to the models that are not present on the higher levels even in an abstract way. Thus, it is of low interest to cover these two abstraction levels in an approach for assertion refinement. The remaining levels on the other hand should be supported by a transformation approach (R 6). Since TLMs often contain parts on different TL sublevels and in order to allow verification of low-level components inside a high-level system (using transactors for connecting both parts), assertions containing parts on different abstraction levels have to be supported as well (R 7).

While it is possible to transform assertions of one abstraction level written in one language to another abstraction level and another language, this is no longer feasible if mixed level assertions have to be considered. Since any mix of abstraction has to be supported, the whole refinement approach should be based on only one assertion language that is able to cover all required abstraction levels by itself (R 8).

Transactions and Signals

All TL sublevels exchange data using remote function calls (so called transactions). In order to check TL models for correct behavior, these transactions and their arguments and return values have to be accessible for assertions as well. Thus, the assertion language has to be able to detect transaction occurrence as well as provide access to transaction parameters (R 9 and R 10).

Since RTL designs use signal protocols for data exchange, it is necessary to provide the assertions with access to all design signals (R 11). If assertions containing transactions have to be refined to RTL, it is necessary to provide a mechanism which maps TL transactions to a comparable RTL representation in order to ease the transformation (R 12).

Synchronization

Since the various abstraction levels differ greatly in how timing is modeled (or if it is modeled at all), there are also different synchronization schemes in place.

Programmer's View (PV) models do not consider timing at all. This does not necessarily mean that the model executes in zero time, but just that timing is of no importance for the interaction between the different components. As a result synchronization between these components and also synchronization between the design and the assertions is based on implicit events (zero delay wait statements) and explicit events (conditional wait statements) only (R 13).

Programmer's View with Timing (PVT) and Cycle Approximate (CA) models additionally include timing. Synchronization between design components and assertions can now happen based on simulation time (timed wait statements) as well (R 14).

Cycle Callable (CC) and RTL designs use clock signals for synchronization which as a consequence have to be usable for the synchronization of assertions as well (R 15).

Temporal Relations

As a consequence of different synchronization schemes, the abstraction levels also differ in what temporal relations between events, transactions and signal changes can be detected.

Since timing is not modeled in PV designs, and since the simulation kernel simulates the concurrent processes of a design in a sequential way, it is not possible to detect any kind of simultaneity between different events or transactions. However, it is possible to order events and transactions according to the execution order in the simulation kernel. This order might originate from causal dependencies between the corresponding events or transactions (in which case every simulator and every simulation run will produce this particular order), or by the fact that the kernel randomly processed first one event and then the other if there is no causal dependency (in which case the order might differ between simulation runs or simulators).

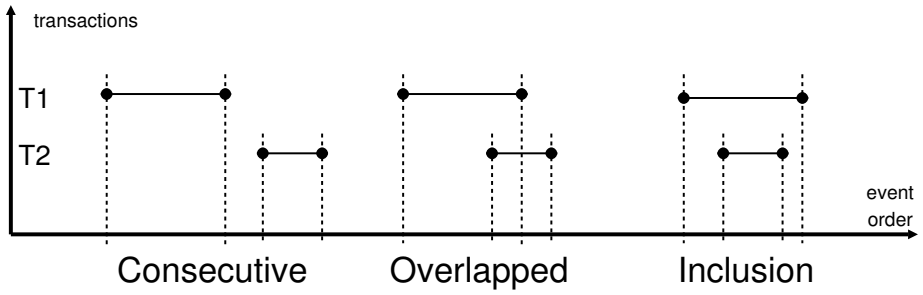

Figure 3.1: Transaction Relations

If care is taken to only reason about dependent transactions / events, then this order allows detecting the order of occurrence of events and of the different relations transactions have to each other (R 16) (see Figure 3.1).

Since every part of the design is able to produce events or call transactions, it is very likely that the use of a global event counter for determining the distance between two events would lead to dependencies between the assertion result and the completely unrelated parts of the design which are not checked by the assertion at

all; this behavior though would contradict R 2. Therefore, it is necessary to specify partial orders on events, which consider only those events relevant for a particular assertion and neglect the rest (R 17). One additional problem lies in the unreliable occurrence of specific events; in contrast to RTL where it can be easily assumed that the next clock edge will happen, the same is not true for TL. But if an expected event does not happen due to an error in the design, the assertion has to be able to detect this absence. The specification of strict partial orders on events allows the detection of event absences relative to the occurrences to other events (R 18).

PVT models introduce timing to the design. When comparing every occurrence of events or transations with the corresponding time stamp, it becomes possible to correlate these events and transactions not only with regard to their order, but also based on the simulation time. Since an arbitrary amount of events can happen at the same simulation time, it is now possible for several events to occur simultaneously. Additionally, events or transactions that happened one after the other may now happen simultaneously (e.g. "A before B" becomes "A before or simultaneous to B"). These possibilities have to be taken into account for the assertion evaluation, which implies that temporal relations based on simulation time have to be detectable as well (R 19).

In most cases the underlying paradigms of CA models are very similar to PVT. The one big difference concerning the detection of temporal relations is that in a CA model it is possible to change the frequency of the simulated clock by changing the corresponding clock period. As a consequence it is possible for time delays to change dynamically which means that assertions have to support dynamic temporal behavior as for example the dynamic change of a time delay (R 20).

CC and RTL models do not use direct time delays anymore. Instead clock signals are used for synchronization and execution. Therefore, assertions have to be able to correlate events, transactions, and signal changes according to the clock signals as well (R 21).

3.1.3 Pipelining

Pipelining in one way or other is a quite frequently used mechanism. When refining a TL design down to RTL for instance, a previously sequential computation is often transformed into a parallel one, either by introducing some sort of dedicated pipeline, or by using multiple resources of a kind. Similarly, many TL designs make use of FIFOs or other structures with a built-in pipelined evaluation. In order to capture this behavior with assertions, the assertion language has to support checks of pipelined behavior (R 22).

3.1.4 SystemC

Simulation Kernel

With the rise of Electronic System Level (ESL) and Transaction Level Modeling, several new languages have been developed. One of them, SystemC[2], has managed to establish itself as the de facto standard for creating TLMs while also supporting the modeling of RTL designs.

For that reason the assertion language should be able to check SystemC designs (R 23), which also means that all SystemC and C++ data types have to be supported for assertion notation and refinement (R 24).

A SystemC reference kernel including infrequent new releases is provided by Open SystemC Initiative (OSCI). It is possible to either use the unchanged simulation kernel of OSCI SystemC or to include some custom changes that have to be transferred on every new version, followed by extensive tests to ensure compliance to OSCI SystemC. Since the continuous update of various kernel versions is too tedious, it is essential that the assertion evaluation does not require changes of the basic simulation kernel (R 25).

OSCI also provides a standard for TL modeling, including the definition of basic communication protocols and corresponding transaction interfaces, in order to allow easier interoperability between different TL IPs. For obvious reasons, an assertion approach dealing with (among others) TL designs should comply to this standard as well (R 26).

In the same way as any Hardware Description Language (HDL), SystemC offers several features especially useful for modeling HW, for instance hierarchical designs. In order to check hierarchical designs with assertions, a mechanism for seamless access to modules and their internal objects has to be provided (R 27).

Event Handling

SystemC models concurrency through the use of processes running in parallel to each other. These processes communicate through and are scheduled for execution by events. SystemC offers basically three different kinds of events:

1. Signal change events: Whenever the value of a signal changes, it emits a corresponding event as a notification.

[2]To be precise, SystemC is merely an extension of C++ in the form of a class library.

2. Explicit events: SystemC provides several types of explicit events that can be emitted by the programmer after a specific simulation time has passed, in the next delta cycle, or immediately.

3. Implicit events: These events are used by processes to suspend themselves while at the same time marking themselves for waking up either after a specific simulation time has passed or at the beginning of the next delta cycle. Implicit events are generated by timed or zero-delay wait statements.

Processes can be made sensitive to certain events by the use of static or dynamic sensitivity lists or by wait statements, which allows influencing under which circumstances a process is to be evaluated. Thus, an assertion language has to be able to track the various kinds of events SystemC provides (R 28).

Transactions

SystemC allows the modeling of two different kinds of transactions:

- Blocking: Blocking transactions are able to suspend the calling process and only resume its execution when a condition is met (a certain time has passed, an event has occurred, etc.).

- Non-blocking: Non-blocking transactions cannot be suspended. The whole transaction is executed in an atomic manner within one delta cycle.

An assertion language has to be able to detect both blocking and non-blocking transactions (R 29). Since non-blocking transactions are executed within one delta cycle, the available time resolution has to be more granular than delta-time in order to monitor details within these transactions (R 30). For the same reason, it is necessary that assertions are notified immediately if a transaction has occurred (R 31).

Access to Data Objects

Objects storing data usually differ between RTL and TL. While RTL models make heavy use of signals for communication and also for storage, on TL this is usually replaced by the use of transactions for communication and variables for storage. In addition to the access to signals (R 11) and to transaction parameters (R 10), these variables have to be accessible as well (R 32). Furthermore, assignments on these variables have to be detectable (R 33).

Sampling

In addition to the general possibility of accessing the aforementioned objects, due to the different synchronization schemes used on the various abstraction levels, it is necessary to provide flexible ways of sampling the values of these objects, either with the occurrence of transactions or events (including signal change events of a clock signal) or at any particular simulation time (R 34).

3.2 Requirements for the Transformation Language

In addition to requirements for the assertion language, there are also certain requirements that have to be met by the transformation language. These requirements are discussed in the following section. After introducing the concept of structure preservation and its consequences, the special requirements brought up by the support of multiple abstraction levels, method based and signal based communication, timing, and reset mechanisms are discussed. Subsequently, assertions which would lead to ambiguous transformation results and the aspect of reusing existing assertion transformation descriptions are addressed.

3.2.1 Abstraction Levels

Since the main purpose of this approach is to provide flexible means for assertion refinement, the assertion transformation process should not be unnecessarily restricted concerning what can be refined and what cannot. As a result, transformations to both higher and lower abstraction levels have to be supported (R 35). For the same reasons, all abstraction levels that can be covered by assertions have to be covered by the transformation language as well. This includes RTL and the various TL sublevels as well as any mix of these (R 36 and R 37).

3.2.2 Behavioral Consistency

When trying to transform an assertion to another abstraction level, it has to be taken care that the transformed assertion still checks the same behavior as the original one. Since the various abstraction levels differ quite a bit concerning the amount of implementation detail they model, ensuring this fact is not trivial. While the corresponding behavior has to be consistent, the assertions can hardly be completely equivalent, since the abstraction levels involved use different models of computation, different notions of time, etc.

As a result, the general composition of the assertion may not be changed while changing specific elements is allowed (R 38). If the original assertion checks a handshake protocol for example then adding a check that deals with writing a register value is not consistent behavior anymore. On the other hand, changing the required time for the handshake from 10 ns to two clock cycles is based on the abstraction level and may thus still represent consistent assertion behavior.

3.2.3 Conversion of transactions and events

While all TL sublevels support the notion of transactions and events, on RTL transactions and most events (with the exception of clock events) are usually represented by a sequence of signal changes. Thus, the transformation language has to support the conversion between transactions and events on the one side and arbitrarily complex signal protocols on the other side (R 39 and R 40). Transaction arguments and return values have to be mapped to corresponding signals (R 41).

3.2.4 Timing

In order to cope with the different concepts of timing and synchronization on the various abstraction levels, the transformation has to provide the possibility to exchange one timing concept with another one, namely absolute delay values, cyclic delay values, and clock signals (R 42).

Additionally, it is necessary to be able to add timing information to an assertion or remove it again (R 43). This modification does not contradict R 38 if an untimed assertion is just considered as having unconstrained timing. Thus, addition of timing information can be regarded as replacing the unconstrained bounds (0 or ∞ respectively) by tangible values, while removing timing works the other way round.

3.2.5 Reset

While TL designs often do not contain a specific reset mechnanism, RTL designs on the other hand mostly do. Though evaluating an assertion during the time the design is reset does not cause problems in most cases, an assertion under evaluation at the exact moment the design is reset will almost certainly lead to a failure if the assertion is not reset as well. Thus, the transformation approach has to support the addition or removal of statements handling reset in the design to / from an assertion (R 44). Again, this modification does not contradict R 38 since the reset statement does nothing more than only constrain the time during which the assertion will be

actually evaluated. In case of a reset, all assertion evaluations that are currently running are terminated without having any effect.

3.2.6 Ambiguity

There are some situations where an assertion cannot be transformed correctly without violating the structure preservation. This might be due to some peculiar details of one of the designs in question or because a specific assertion construct simply cannot be mapped onto the target abstraction level. In this case the transformation language should provide the following three options (R 45):

1. Warning: The transformation issues a warning message stating that the desired transformation might createsible without creating behavioral discrepancies between the original and the target assertion.
2. Strict transformation: While the target assertion might produce false negatives, it avoids all false positives. In extreme cases this could lead to an assertion that is always violated.
3. Safe transformation: While the target assertion might produce false positives, it avoids all false negatives. In extreme cases this could lead to trivial assertions that will never fire.

While the results of the latter two choices are not exactly a correct refinement of the original assertion, it allows to at least get an estimation about the behavior of the refined design. It might even be useful to run the design with both the "safe" and the "strict" assertion and evaluate the corresponding coverage results. In any case, the transformation results can at least point in the right direction for manually creating an appropriate assertion.

3.2.7 Reuse

Using a transformation language provides two basic advantages when compared to manually transforming the assertions from one abstraction level to another:

1. Since the transformation description only specifies the changes that have to be done to the assertion, the user can be certain that everything not mentioned in this description will remain exactly as it was before. As a consequence, there will be less sources for potential errors than compared to the manual refinement process.

2. Some parts of that transformation description might - after slight adaptations - be useful for future refinement of similar assertions.

With some additional features for the transformation language this reuse of transformation descriptions could be made much easier. The language should support the combination of several transformation directives that serve a common goal into transformation rules (R 46). Providing these rules with parameters in order to dynamically adapt them for the task at hand would then reduce the modification effort when applying existing rules to a new project (R 47). Finally, the language should support the encapsulation of parameterizable rules in order to provide transformation libraries for frequently used transformation processes (R 48).

4 State of the Art and Related Work

This chapter discusses the two topics assertion refinement is based on. First, the various approaches for ABV and similar verification methods, and second, approaches dealing with automated or semiautomated refinement in other areas, be it related to design refinement or the concepts involved in crossing abstraction levels in general. This chapter discusses corresponding related work and motivates the choice of the underlying assertion language by comparing existing approaches with regard to the requirements discussed in Chapter 3.

After presenting current ABV approaches for both TL and RTL, refinement based on the use of transactors is discussed. Finally, general refinement approaches and other related work are covered.

4.1 Assertions in a Top-Down Design

As explained in Chapter 2, the design process covers several different abstraction levels. Assertions that are applied to a design during this process, differ according to the abstraction level of the corresponding design. Using ABV on TL requires the support of several features that are typical for TLMs, for instance a way of dealing with the abstract communication that is used at this level. Other requirements lie in the support of synchronization schemes that do not rely on clock signals and the possibility of untimed modeling. Compared to that, RTL assertions have to deal with completely different design features. RTL designs are based on signals, which are used for communication, storage, and synchronization (clocks), in contrast to the transaction and event based evaluation of TLMs. As a consequence, assertions on this level have to support a view centered on clock cycles. Since clock signals can be assumed to be regular and reliable, it is always guaranteed that assertion will be evaluated within a known time period. Several approaches for ABV have been created during the last years, with sometimes quite noticable differences concerning available functionality and applicability. In order to make a reasonable decision about which approach to base the refinement idea on, the following section details these different approaches and also compares their functionality in accordance to the requirements listed in Chapter 3.

4.1.1 SVA and PSL

SystemVerilog Assertions (SVA) [7][8] and Property Specification Language (PSL) [9] are two wide spread languages for ABV. While SVA is part of the Hardware Description and Verification Language (HDVL) SystemVerilog, PSL is an independent language completely focused on specifying assertions. Both are comparable concerning the included features (for a comparison of functionality and expressiveness of both languages, see [10]).

Both languages are very well equipped for verifying RTL designs due to the support of assertion evaluation based on clock signals, the ability to specify sequences of Boolean propositions to be checked, the ability to connect several sequences or properties by dedicated operators, etc. Both languages are only capable of very rudimentary support of TL designs by providing some features that allow the specification of rudimentary TL assertions (e.g., the use of events for triggering). Nevertheless, they lack several of the required features for specifying full fledged assertions on TL: For instance, neither language provides the ability to detect transactions (R 9), access to the corresponding transaction parameters (R 10), to synchronize assertion evaluation with delays in the design, for instance timed wait statements (R 14), or to detect dynamic or pipelined behavior (R 20 and R 22).

4.1.2 Research work

Finite Linear Temporal Logic

In [11][12] a bounded version of Linear Temporal Logic (LTL) called Finite Linear Temporal Logic (FLTL) and a SystemC implementation of the corresponding formulas is presented which is then applied to the verification of TLMs [13], [14]. While a notion of transactions exists, blocking transactions are not supported (R 29) which also prevents the detection of transaction relations (R 16). The detection of temporal relations based on simulation time is not supported (R 19). While event order can be detected, unfortunately, this event order is global. As a result all events of the design influence the distance between two events in question and thus, changing unrelated parts of the design influences the assertion behavior (R 2). Finally, neither RTL and mixed level assertions (R 6 and R 7), nor pipelining (R 22) are supported.

Transaction Level SystemC Assertions

The approach presented in [15][16] allows the specification of TL assertions in SystemC. While a notion of transactions exists here as well, again, there is no support for the detection of transaction relations (R 16). Linking to transaction parameters

(R 10) and the use of simulation time for synchronization or for detection of temporal relations is not supported either (R 14 and R 19). The detection of transactions in PV models is enabled by the use of callbacks. This approach does not support RTL and mixed level assertions either (R 6 and R 7). Detection of pipelined behavior is only possible if the designer provides a large amount of code annotations within the design and is furthermore restricted to PVT models (R 22).

Usage of SystemVerilog Assertions with SystemC Designs

In [17][18] SVA is used for specifying TL assertions on SystemC designs. Additional signals are introduced that indicate whether a specific transaction is active. An artificial clock signal is introduced which produces an edge with each edge of one of these auxiliary signals. The simulation run is stored in a Value Change Dump (VCD) file which is then read into a Verilog module representing that particular trace. The resulting Verilog module is simulated in combination with the SVAs. Due to the required update of the auxiliary signals, there is always an additional delta cycle between the actual transaction and the corresponding detection based on the signal. Thus, non-blocking transactions cannot be captured in this way (R 29). Since the artificial clock signal considers all transactions, the resulting event order is global, which leads to assertions depending on unrelated parts of the design (R 2). Since this approach is based on SVA it has to deal with all inadequacies this language shows concerning TL modeling. Thus, TL sublevels are not completely supported and as a direct consequence neither are mixed level assertions (R 6 and R 7).

Temporal Logic of Actions

The approach presented in [19] correlates so called actions (assignments to state variables) over time using operators similar to known LTL operators. Since neither a notion of events nor a notion of transactions exists, none of the corresponding requirements can be fulfilled (R 9, R 13, etc.). Finally, this approach does not support the detection of dynamic or pipelined behavior (R 20 and R 22).

Duration Calculus

In [20][21] an approach for using Duration Calculus (DC) to specify real time properties is presented. DC formulas allow reasoning about the duration and temporal correlations of so called phases (states). Thus, only detection of temporals relations based on simulation time is supported. On the other hand, detecting relations based on events or clock signals is not possible (R 17 and R 21), which makes it impossible to correlate transactions in pure PV models and requires additional effort to

do the same in CC models. Furthermore, both the DC formulas, and the Design Under Verification (DUV) need to be translated into so called phase event automata, which means that designs modeled according to the classical abstraction levels are not supported (R 6).

Logic of Constraints

In [22][23] an approach for simulation based checking of abstract SystemC models using a combination of LTL and Logic of Constraints (LOC) is presented. Temporal checks are specified using LTL while LOC is used for specifying performance checks. Using LOC formulas in combination with events that can be associated with specific values allows checking of pipelined behavior. Under certain conditions however, the results of these checks can be non-deterministic. This approach aims at higher levels of abstraction and lacks support for RTL designs (R 6 and R 7).

SystemC Implementation of SystemVerilog Assertions

In [24][25] a complete implementation of SVA on SystemC is presented. SVA are written natively in SystemC using a macro-style syntax. There is however no mentioning of how the differences between the sampling semantics of the simulation kernels of SystemC and SystemVerilog are resolved. Since SVA is used as the basis for this approach, most features necessary for the TL verification are missing (R 6 and R 7).

XML Based Assertion Generation

In [26][27][28] a generator based approach for creating assertion libraries in VHDL (VHDL), Verilog, SystemVerilog, and SystemC from an XML description is presented. In addition to classical RTL features this approach also allows the specification of assertion evaluation based on simulation time. However, this approach does not provide TL features (R 6 and R 7).

4.1.3 UAL

In [29][30][31][32] an assertion language for specifying assertions for SystemC designs is presented that covers TL, RTL, and mixed level assertions. This language is neither based on LTL, nor on Computation Tree Logic (CTL). Assertion evaluation happens on the basis of events; signal changes are directly converted to events, while specialized operators translate simulation time to events as well (in addition to that, operators for creating complex event expressions are provided). As a consequence,

sampling of design states can be done with regards to events, simulation time, and clock signals. Transactions are considered a pair of events, notified to the assertion by callbacks; this mechanism also allows the detection of transaction relations. But while it is possible to express the corresponding signal protocols on RTL, the language lacks direct support of an RTL transaction representation (R 12). The absence of events can be expressed by so called negative trigger events. Since the language is translated to native SystemC modules that are then co-simulated with the design, the language naturally is compliant to OSCI SystemC and also provides access to all internal elements, including signals, variables, and transaction arguments. Detection of pipelined behavior is supported as one of several evaluation modes.

While this language does not provide all required features for assertion refinement, it comes closest. The most important factor is that it does not only cover all abstraction levels in question, but also handles them in a consistent way. This in turn leads to greater similarities between high level and low level assertions (better for the refinement) and makes the specification of mixed level assertions much easier.

4.2 Transactor based Refinement

Several different verification approaches deal with the reuse of high level verification components, like assertions and testbenches. This reuse is accomplished by the use of transactor components which translate the low level protocol of the design to the high level protocol of the verification components or vice versa.

In the following, several approaches related to transactors are discussed.

4.2.1 Flexible Transactor Specification

SpiraTech, now a subsidiary of Mentor Graphics Inc., offers a methodology for easy specification of transactor components covering multiple abstraction levels[6]. Using SpiraTech's own language CY that was specifically developed for that purpose, the user can easily create transactors covering all abstraction levels he desires, and only those.

Using a generator, the CY description can be transformed into a simulatable transactor model which provides links to SystemC and RTL simulators as well as assertions concerning the correct protocol behavior.

This offers some benefit in transactor modeling productivity, but does not provide any new features to transactors at a glance.

4.2.2 Multi-Level Testbenches and Transactors

Each of the leading Electronic Design Automation (EDA) vendors has developed a methodology for the application of simulation based verification: the Advanced Verification Methodology (AVM) [2] by Mentor Graphics Inc., the Verification Methodology Manual (VMM) [33] by Synopsys Inc., and the Unified Verification Methodology (UVM) [34] by Cadence Inc.

All these methodologies are based on distributing the verification functionality into several different functional components (see Figure 4.1):

1. Stimulus Generator: Provides stimuli for the DUV (usually constrained random)
2. Response Checker: Compares the DUV responses with data from a golden reference model
3. Coverage Collector: Gathers coverage data
4. Test Controller: Uses coverage data to control stimuli generation

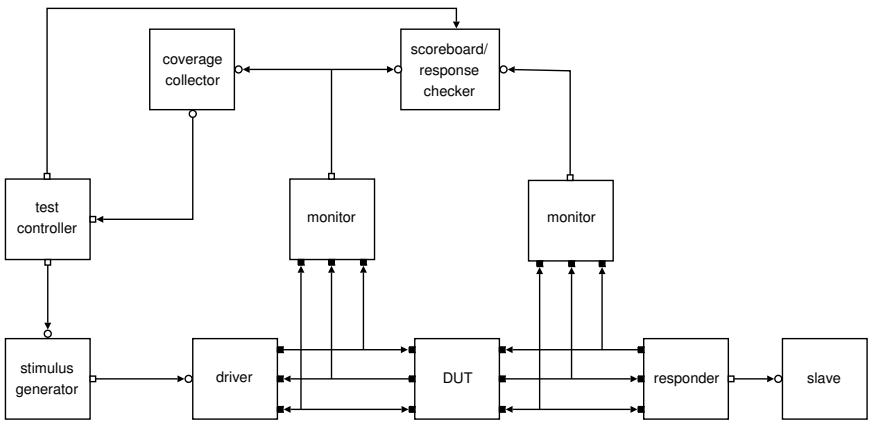

Figure 4.1: Sample Testbench System According to AVM[2]

When verifying an RTL design instead of a TLM, several additional transactor components translate the stimuli towards low level signal protocols and the response of the design back towards high level transactions. This way, the same testbench can be used for both models.

Assertions can be used in addition to this approach. They can help uncover design errors, while the assertion coverage information can be used in combination with the

conventional coverage for controlling the stimulus generation. Additionally, assertions allow to not only check behavior at the interface between different modules, but also the internal behavior, which in turn can reduce the overall verification effort.

Since assertions work in a way that is fundamentally different from testbenches - desired behavior is specified in an imperative way in testbenches, while assertions use a declarative approach - the combination of both approaches helps to achieve better verification results, since two different points of view are applied.

Another approach is presented in [35]. Here, an RTL component is embedded in a TL system. The communication between the different parts is again handled by a transactor. Both parts are co-simulated using two coupled simulators, one per abstraction level, in order to reduce the impact on simulation performance. The transactor component not only serves the purpose of translating high level protocols to low level signal interfaces and back, but also coordinates the information exchange between the two simulators (one simulator first receives data from the other one, then proceeds its own simulation for one step, and finally passes control back to the other simulator; only one simulator is active at any given time). Furthermore, the replacement of the RTL part by an HW emulator is also presented.

A different direction is persued in [36]. Here, existing RTL testbenches are reused for the verification of a TLM, either by means of mixed level (and potentially mixed language) simulation in combination with a transactor, or by using a transactor to record the signal protocols during the RTL simulation and to generate a corresponding TL testbench which will provide equivalent stimuli for the TLM.

4.2.3 Automated Transactor Generation

In [37], an approach for generating transactors from an Extended Finite State Machine (EFSM) is presented. A protocol description (for standardized protocols) or a given testbench (for non-standardized protocols) is transformed into an EFSM. This EFSM is then used to automatically generate interfaces and implementation for the corresponding transactor component.

A similar approach is presented in [38]. Interface protocols are specified using an enhanced version of PSL, specifically its Sequential Extended Regular Expression (SERE) subset. These expressions are transformed to an abstract syntax tree and from that to a Finite State Machine (FSM). Finally, the transactor implementation is generated from this FSM.

4.2.4 Transaction Level Assertions and Transactors

In [39][40], an approach for using TL assertions in combination with transactors and RTL designs is presented. The high level assertions are written in PSL without a real notion of transactions (R 9). Instead, the state of the design is represented by signals. Due to the additional delay for updating these signals, it is not possible to reason about behavior within a delta cycle (R 30). Furthermore, specifying temporal relations between different design behaviors based on simulation time is also not supported (R 19).

Since most of the inadequacies of this approach are based on its reliance on PSL for specifying the assertions, it is possible to use another assertion language, which is more suitable for TL, in combination with transactors. However, even if this language supports the transaction concept, due to the very nature of transactors, only the end of transactions can be detected. As a result, the different possible temporal relations of several transactions cannot be distinguished (R 16), since there is no way to reason about the start of the corresponding transaction calls.

4.3 General Refinement Approaches

This section covers various related topics, like various forms of design transformations and Aspect Oriented Programming (AOP). These topics either show some common properties to the assertion refinement presented in this work, or provide the basics on which the assertion refinement is built.

4.3.1 Model Driven Architecture

Model Driven Architecture (MDA) [41] describes an approach to specify systems with a separation between the system's functionality and the corresponding implementation. Due to this separation, it is possible to map one specific functionality on multiple corresponding implementations. In this context, a model represents a part of the function, structure and / or behavior of a system.

MDA requires a formal specification of a model, which means that the specification is done in a language / representation with clearly defined syntax and semantics. For instance, every simulatable HDL code would be a model.

In this context the terms of abstraction and refinement are used. Abstraction is defined as the suppression of irrelevant detail. Two models, one more abstract than the other, can be connected by a refinement model, which describes the abstraction steps taken from one model to the other while at the same time ensuring some basic

correlations. This refinement model could describe the transformation from a platform independent model to a platform specific model. Both are mostly based on XML description, with the actual transformation done via Extensible Stylesheet Language Transformations (XSLT) or a script language as for instance PERL.

4.3.2 Design Refinement

The topic of refining or transforming existing models is not new. In [42] the different aspects of model transformations are categorized in order to help programmers with approaching the transformation of their designs in a methodical manner.

There are already several different approaches for transforming an existing design to another abstraction level. Some examples are listed here:

- Abstraction: In [43] a methodical transformation of RTL models towards TL is described. RTL models are described as EFSMs which are then analyzed in order to extract the information relevant for the TL model.
- Systems:
  - The tool CatapultC from Mentor Graphics Inc. [44] as one example for a high level synthesis tool allows the creation of RTL code from high level C++ programs (functional to RTL)
  - In [4] an approach for high level synthesis is presented which also includes an equivalence check between the original and the generated models (TL to RTL)
  - Bluespec Inc. offers several approaches to describe hardware in a high level language (based on either SystemC or SystemVerilog) which can then be synthesized towards Verilog RTL code automatically [45] [46] (CC to RTL)
- Mapping: The IP-Xact standard developed by the SPIRIT consortium[3] is capable of describing the mapping information of RTL designs. Currently the standard is enhanced in order to also include TL designs and an automatic mapping of RTL and TL components.

4.3.3 Refinement Description Languages

For several other refinement tasks corresponding languages have been developed already.

For instance [47] presents a language for the stepwise refinement of software architectures. This language uses a set of refinement primitives that allow a formal

description of the refinement process. In this context, the language makes use of the fact that an abstract program does not specify all aspects of the implementation. These open aspects are then determined during the refinement. The language is formally based on the π-calculus, which was developed to describe concurrent computations with dynamically changing configurations.

In [48] a refinement of architectural specifications written in Common Algebraic Specification Language (CASL) is presented. The approach uses so called refinement trees which basically splits a refinement task in several independent subtasks that can be evaluated easier than the original one.

4.3.4 Aspect Oriented Programming

The transformation language presented in this work uses features known from AOP [49]. This programming technique tries to break down programs into parts with (in the best case) disjunctive functionality. If this is not possible, these so called cross cutting concerns can be implemented in separate modules called aspects. These aspects can then be added to all parts of the program where they are needed.

Usually, aspects fall into one of three categories:

1. New functionality is added to a program
2. Existing functionality is modified or replaced
3. Existing functionality is removed

Using similar methods, the new features of a less abstract level can be added to an assertion, while they are removed when moving to a more abstract description.

5 Assertion Refinement

This chapter introduces an approach for assertion refinement, allowing free transformation[1] between all TL sublevels and RTL. First, general concepts are discussed, followed by the changes to an existing assertion language named Universal Assertion Language (UAL) in order to provide the necessary features for the refinement process[2]. Afterwards a dedicated transformation language is presented, which allows the formal specification of the transformation process. The chapter concludes with an overview of the transformation file structure as well as general guidelines for the applicability and usage of the transformation rules.

5.1 Concept

Every approach of assertion refinement has to be based on a corresponding refinement of the underlying design. As a direct consequence, the peculiarities of design refinement have to be examined and - if possible - categorized in order to provide a conceptual basis for the transformation of assertions.

5.1.1 Basic Definitions

This section discusses several required definitions for the refinement process:

Concepts of Simultaneity

A concept connected to the notion of timing is simultaneity. Due to the different ways of modeling time within the various abstraction levels, there are also different possibilities to order events and the actions - operations or assignments on the states of a design - triggered by the occurrence of these events. This order can be either causal

[1] While the transformation language supports both refinement and abstraction, in the following sections we focus on a top-down design process and accordingly on assertion refinement.

[2] In order to illustrate some of the general concepts, code examples are used at several places. For convenience reasons these code examples also use UAL.

or arbitrary depending on the relation between those events. However, all actions invoked by the occurrence of one event can be considered to happen simultaneously.

Models without time can order events only by their occurrence, while others might additionally allow the use of simulation time values and even clock cycles. As a consequence, three different notions of simultaneity are defined:

Definition 2 *All events of a simulation run are unambiguously ordered concerning their occurrence. All actions occurring between two adjacent events are considered simultaneous to the first event. This relation is called Event Based Simultaneity (EBS).*

Definition 3 *Two events of a simulation run with the same simulation time stamp are considered simultaneous with regard to simulation time. All actions invoked by these events are hence considered simultaneous as well. This relation is called Time Based Simultaneity (TBS).*

Definition 4 *Two events of a simulation run occuring between two event occurrences of the same event object are considered simultaneous with regard to the corresponding event. All actions invoked by these events are hence considered simultaneous as well. This relation is called Cycle Based Simultaneity (CBS).*

Cycle Based Simultaneity (CBS) is most commonly used in combination with clock edges. In this case all events within the same clock cycle (for instance between two rising edges of the same clock signal) fulfill this requirement.

Note that all actions that are simultaneous according to Definition 2 are also simultaneous according to Definition 3 and Definition 4. The inverse of this relation does not hold.

There is no direct way to determine if actions that are simultaneous according to Definition 3 are also simultaneous according to Definition 4 or vice versa, since two adjoining occurrences of the relevant events for CBS might happen within one time step (for instance in different delta steps) or distributed on several different time steps.

When using a modeling style or abstraction layer which only uses one event per clock cycle (and thus also per time slot), all three definitions converge to the concept of simultaneity known from RTL[3].

As mentioned above, the modeling of time differs between the various abstraction levels. As a consequence, some forms of simultaneity are not detectable on all levels and thus cannot always be preserved when transforming an assertion.

[3]This assumes that the clock signal does not include any delta glitches.

Event Based Simultaneity (EBS) is always detectable but may have no sensible meaning in certain situations. It is always preserved when transforming an assertion to a lower abstraction level.

Time Based Simultaneity (TBS) is only detectable if the model includes time. It is always preserved when transforming an assertion to a lower abstraction level but dissapears when transforming to a more abstract level that does not model time anymore.

CBS is only detectable if the model includes the corresponding events or information about their occurrence (for instance a clock signal or a corresponding clock period). It is preserved as long as the target level includes information about these events.

Structure Preservation

As mentioned in Chapter 3, preserving the structure of an assertion during the transformation is necessary, since otherwise the transformation result cannot be correlated with the original assertion concerning the checked behavior any more. While an assertion transformation preserving the structure might still not be equivalent, checking the equivalence of transformed assertions violating this requirement might become very complicated. This can be easily seen when considering that checking this kind of equivalence includes the equivalence check of two Boolean expressions which is known to be an NP-complete problem [50].

Structure preservation is defined as follows:

Definition 5 (Structure Preservation) *Structure Preservation of an assertion implies that the structure of an assertion, including the exact number of subcomponents (properties, sequences, event expressions, Boolean expressions) and the way these subcomponents are connected, is not changed. Replacing an element by another element of the same type is allowed, while adding new elements, removing elements, or replacing elements by elements of another type is illegal.*

Since all operators entail a structure by their nature, they may not be replaced.

The following figures demonstrate this concept. Figure 5.1 shows a possible structure of an assertion: The assertion instantiates a property, the property connects two sequences with an implication operator, and each sequence includes two delay operators with their corresponding event expressions and Boolean expressions.

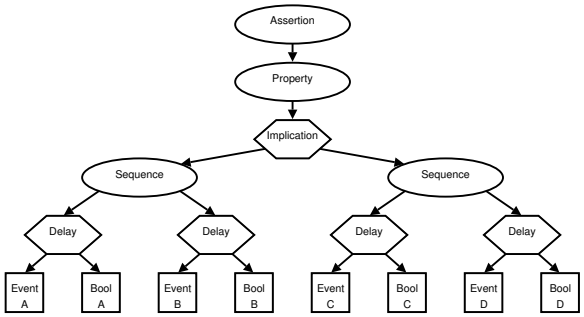

Figure 5.1: Sample Assertion Structure

In Figure 5.2 the event expression and the Boolean expression of one of the delay operators have been replaced, but the general structure of the assertion remains unchanged[4].

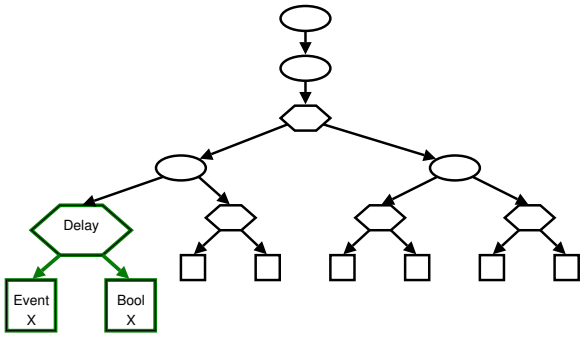

Figure 5.2: Transformed Assertion with Structure Preservation

The assertion shown in Figure 5.3 on the other hand changes the structure by adding a new delay operator to one of the sequences. An additional delay operator means that the corresponding assertion includes at least one additional check of the design behavior, which means that the transformed assertion cannot be equivalent to the original one any more.

Another example is shown in Listing 5.1. The topmost property shows the original version that has to be refined. Below, two possible results of this refinement are presented.

[4]Note that this example simplifies the issue a bit, since the exact composition of event and Boolean expressions is also part of the assertion structure. As a result, it is allowed to replace literals or objects within these expressions by other literals or objects, but not a complete expression by a different one.

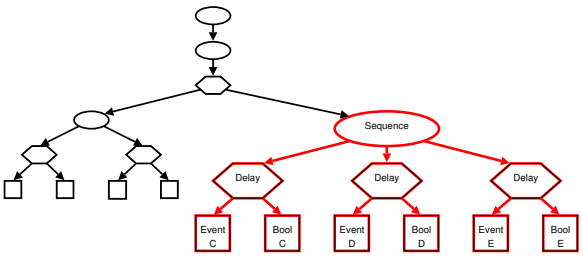

Figure 5.3: Transformed Assertion without Structure Preservation

```
1  property Orig_prop
2    int l_addr;
3    #1{read_mem 'END}{true, l_addr = read_mem.param2}
4    #1{write_mem 'END}{(write_mem.param1 == l_addr)};
5  endproperty
6
7  property Wrong_prop
8    int l_addr;
9    #1{clk 'POS}{RD_MEM == 1, l_addr = read_mem.param2}
10   #2{clk 'POS}{(WR_MEM == 1) && (write_mem.param1 == l_addr)};
11 endproperty
12
13 property Right_prop
14   int l_addr;
15   #1{read_mem 'END}{true, l_addr = DATA_IN}
16   #1{write_mem 'END}{(ADDR_OUT == l_addr)};
17 endproperty
```

Listing 5.1: Assertion Example for Structure Preservation

The assertion in the middle replaces the start and end of a transaction by a positive clock edge, changes the number of delay steps for the second delay, and modifies the structure of the Boolean checks - in the first check a simple `true` is replaced by a comparison, in the second check another subexpression has been added. While the replacement of the transaction events does preserve the assertion structure[5] the change of the delay steps and of the Boolean expressions does not.

The third assertion replaces references to transaction parameters by signals. Although the type of the underlying objects is different now, the assertion structure has been preserved.

[5]Note, that even though the structure is preserved, it cannot be guaranteed that this replacement is correct or legal.

5.1.2 Problem Classes for Assertion Refinement

When regarding the differences between the various abstraction levels as discussed in Chapter 2.1.2, assertion refinement can be divided into three different categories:

- Interface Type
- Timing
- Additional Information

These categories are explained in detail in the following, combined with a description of the difficulties for any transformation approach to handle them in assertions.

Interface Types

This category basically includes two different interface types: transaction based interfaces on the one hand, and signal based interfaces on the other. The transaction concept is mainly used for the modeling of TL designs, while signal interfaces are the prevalent concept for RTL designs.

Transactions are usually modeled as remote function calls, where one component provides the corresponding function which is then called by another component. Information is transmitted by the use of parameters and return values.

In order to transform a TL assertion, which is sensitive to transactions, to RTL, it is necessary to align TL transactions and the corresponding RTL counterparts.

One of the difficulties faced in this context lies in the fact that the TL transaction packages all relevant information in one method call, while the RTL signal protocols might share control signals between several transactions. Another problem, which is partially based on the first, is that detecting start and end of transaction calls becomes more difficult. Instead of one central location, several signals have to be monitored in order to correctly capture the corresponding events.

Timing

The second category concerns the modeling of time and temporal relations within the models. While the interface type as described above is uniform on all TL sublevels, they differ greatly in this regard. The less abstract the sublevel, the more timing details are added to the model - and accordingly also to the assertions.

Beginning with the abstract PV descriptions which do not contain any notion of time, PVT models introduce the notion of absolute timing, CA models use cyclic timing, and finally CC models employ clock signals in the same way as RTL models. When transforming assertions, it has to be possible to convert every one of these modeling styles into every other one.

This requirement seems to be at odds with the requirement of preserving the structure of a given assertion: How can an assertion that does not contain timing be transformed into one that does, since adding operators to an assertion is not allowed?

This dilemma can be avoided by introducing the concept of implicit timing information:

Definition 6 (Implicit Timing) *Every event expression within an assertion that does not already include an explicit time constraint can be considered to be implicitely constrained to occur within a time frame of immediately to infinity. In the same way, every event expression can be considered to produce a definite result as soon as the simulation time reaches infinity.*

According to Definition 5, changing the arguments of an operator does not change the structure of an assertion, as long as the new values are of the same type as the old ones. Consequently, using Definitions 5 and 6, transforming the timing information of one assertion to another abstraction level can be achieved easily by just replacing the corresponding values within a timer or time constraint, while still preserving the assertion structure.

Listing 5.2 shows an example for both the concept of implicit timing and for corresponding transformations.

```
1  sequence Orig_seq
2    #1{put 'END}{true}
3    #1{get 'END}{true};
4  endproperty
5
6  seqence Explicit_seq
7    #1{put 'END}{true}
8    #1{get 'END@($delta_t >= 0 && $delta_t <= $infinity);timer(
         $infinity)}{true}
9  endproperty
10
11 sequence Transformed_seq
12   #1{put 'END}{true}
13   #1{get 'END@($delta_t >= 3 && $delta_t <= 5);timer(6)}{true}
14 endproperty
```

Listing 5.2: Example for Implicit Timing

The first sequence looks for the occurrence of a completed put transaction followed by the occurrence of a get transaction without taking the temporal relation between those two into account (a typical situation for a PV design). The second sequence illustrates the concept of implicit timing by explicitely showing this information. The occurrence of the get transaction is constrained by a temporal distance between 0 and ∞ and if it does not occur within this time frame, the timer operator leads to a not match result as soon as the temporal distance equals ∞. Obviously, these time related operators do not change the behavior of the original sequence.

The third sequence shows a possible result after a transformation to PVT. Now, timing information is included in the model and thus, it also has to be included in the assertion. When comparing the second and the third sequence, it becomes clear that only the boundaries of the time constraint and the argument of the timer operator have been replaced by different values; the overall structure of the sequence has not been affected.

Additional Information

The last category is a catch all for the remaining differences between the abstraction levels, again mostly for the differences between all TL sublevels on one side and RTL on the other.

The most important issue in this context is the handling of design reset. While many RTL designs include a reset mechanism, this is not always the case for TL. When dealing with a resetable RTL design and a TL design without reset however, it is not possible to create an assertion with equivalent behavior if equivalence checks include the reset case. If the reset is excluded on the other hand, this problem is reduced to merely disabling the assertion during the reset phase.

As the reset statement merely disables the assertion evaluation for a certain time, it has no impact on the general assertion structure and thus, does not violate the requirement for structure preservation.

Other items in this category include the inclusion of scan or debug capabilities, additional registers containing status information or simulation specific data, etc.

Summary

Table 5.1 shows an overview of the possible combinations resulting from the first two categories[6].

Interface Type **Time**	**Transaction Based**	**Signal Based**
Ordered Events	PV	Behavioral (Zero Delay)
Token Cycle	PVT	Behavioral (Timed)
Cycle related (time)	CA	Behavioral (Cyclic)
Cycle related (clock)	CC	Behavioral (Clocked) / RTL

Table 5.1: Abstraction Levels

Transaction based interfaces are represented by the various TL sublevels, while RTL and behavioral modeling use signal based interfaces.

As can be seen, the transformation between models / assertions along the timing abstraction is orthogonal to transformations along the interface abstraction. As a consequence, the corresponding transformations can be considered independent from each other.

5.2 Implementation of Assertion Refinement

This section discusses the reasons for basing the assertion refinement on the language UAL and explains how the problem classes mentioned above are dealt with in UAL. A detailed discussion of both the problems refinement has to face, and of the corresponding solutions follows.

5.2.1 Reasons for choosing UAL

As discussed in Chapter 4, the assertion language UAL provides almost all features required for supporting assertion transformation between the various TL sublevels, RTL, and even mixed level assertions. Due to the fact that the language covers all necessary sublevels and even allows mixed level assertions, a seamless approach for the transformation framework is possible. With only slight changes or enhancements to the existing language, a UAL assertion written on any abstraction level can be transformed into another UAL assertion on any other abstraction level.

[6]Since the third category does not follow a specific theme, but basically just contains everything not covered by the first two, it is not considered here.

5.2.2 UAL Overview

This section gives a short overview over the organization and the features of UAL.

Layers

Like most assertion languages UAL is organized in several layers. Some of these layers are common to all assertion languages while others are new or at least greatly enhanced:

1. The Modeling Layer includes the structure of the assertion monitor file.
2. The Verification Layer includes all directives that interact with the simulator, which means assertions and coverage directives. Assertions notify the user in case of abnormal or erroneous design behavior, while coverage directives collect data on how often a certain behavior occurred.
3. The Property Layer allows the specification of properties. Properties describe desired design behavior, for instance in form of logical and temporal implications. Every property evaluation produces one of the following results:
   - Failure: The design violates the specified behavior.
   - Success: The design complies with the specified behavior.
   - Vacuous Success (only for implications): The antecedent condition of the implication is not met, so the consequent is not checked.
4. The Sequence Layer allows the specification of sequences. Sequences describe temporal patterns which are matched against the temporal behavior of the design. Every sequence evaluation produces one of the following results:
   - Match: The pattern has been recognized.
   - Not Match: The pattern has not been recognized.
5. The Event Layer provides operators for the logical combination of events which are used for the sampling of design states by UAL and for progression of the delay operator.
6. The Boolean Layer provides operators for the logical combination of Boolean propositions. This layer is responsible for defining the checks on design states.

The biggest difference between UAL and other assertion languages lies in the existence of an independent event layer that is responsible for handling the various abstraction levels.

Event Operators

The event layer offers a variety of operators which are used to generate events or combine existing events to event expressions. Table 5.2 lists the existing event operators and provides a short description of their functions.

Name	Syntax	Function
Single Event	*event*	fires if `event` occurs
TIMER	**timer**(*timeval*)	fires `timeval` time units later than the current evaluation time
ACCUMULATOR	*ev_expr*%(*expr*)	fires after `expr` occurrences of `ev_expr`
CONSTRAINT	*ev_expr*@(*expr*)	fires if `expr` is true on occurrence of `ev_expr`
OR	*ev_expr* | *ev_expr*	fires if either of the operand event expressions occurs
AND	*ev_expr* & *ev_expr*	fires if both operand event expressions occur at the same simulation time

Table 5.2: UAL Event Operators

Delay Operator

One of the key features of UAL is a general delay operator which works independently from the abstraction level and thus allows the specification of sequences across abstraction levels. Figure 5.4 depicts the overall structure and functionality of this delay operator.

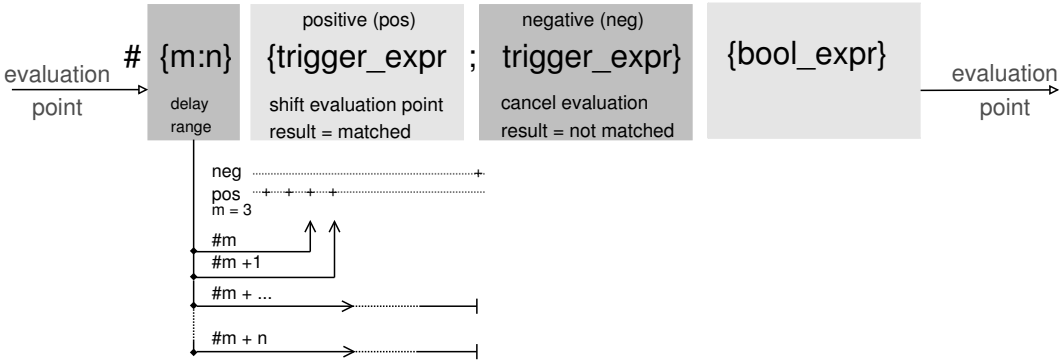

Figure 5.4: General Delay Operator

One evaluation attempt of a sequence built from this delay operator is called a thread. The specification of a delay range leads to a split into several so-called subthreads.

Listing 5.3 shows a sample use of this delay operator. The first set of curly brackets specifies the delay range, i.e. the required number of occurrences of a trigger expression. The trigger expressions are specified within the second set of curly brackets - divided into positive triggers in front of the semicolon which shift the evaluation and negative triggers after the semicolon which stop the evaluation - while the third set of curly brackets contains the Boolean propositions to be checked at the time of the trigger.

```
1 #{3:5}{(e1|e2)@($delta_t>=45 && $delta_t<=50);e3|timer(51)}{A ==
      B};
```

Listing 5.3: Sample Event Sequence

This configuration delays the evaluation until *e1* or *e2* have occurred between three and five times with a temporal distance of 45 to 50 time steps. If the positive trigger occurs for the third, fourth, or fifth time, the Boolean expression ($A == B$) to the right is evaluated. If this expression evaluates to "true" the delay operator results in a "match". The delay operator results in "not match" if either *e3* occurs, or the evaluation per delay step takes 51 time steps, or the Boolean expression evaluates to `false`.

Evaluation Modes

UAL provides several different modes which determine the way several evaluation threads of a property or sequence interact with each other.

Sequence modes only lead to different results if the sequence in question includes more than one subthread (introduced by ranged delays). The following sequence evaluation modes are available:

- AnyMatch: No interaction between different subthreads of an evaluation occurs. Thus, each subthread is treated separately and will produce a result of its own. A sequence running in this mode will produce exactly as many results as the product of all its range possibilities.
- FirstMatch: This mode introduces interaction between the different subthreads. As soon as one subthread of an evaluation attempt results in a match, all other subthreads are canceled (if more than one subthread result in a match at the

same time, only the subthread with the smallest delay value will match). Thus, a sequence running in this mode will always produce exactly one result.

- FirstMatchPipe: Additionally to the behavior of the previous mode, interaction also happens between different evaluation attempts of the same sequence. There are two additional restrictions concerning the matching of a subthread:

  1. A subthread may only match at a certain event index if no older thread tries to match at the same event index.
  2. Every matching thread "consumes" the Boolean expressions specified within its delay operators at the corresponding event indices[7]. A subthread may only match if the Boolean expressions it tries to consume have not already been consumed by an older thread.

Similarly, there are several modes for the evaluation of implication properties. The following property evaluation modes are available:

- Restart: If the antecedent sequence of the implication produces another match while the consequent sequence is still under evaluation, the current evaluation is canceled and a new one is started.
- NoRestart: If the antecedent produces another match during the consequent evaluation, this additional match is ignored.
- ReportOnRestart: This mode shows similar behavior to the previous mode, but additionally the restart attempt is reported by another assertion message.
- Overlap: Each additional match of the antecedent starts another evaluation attempt of the consequent, regardless of existing consequent evaluations. One consequent match is correlated to all antecedent matches that apply.
- Pipe: This mode shows similar behavior to the previous mode, but every match of the antecedent has to be followed by a separate match of the consequent.

UAL allows the specification of a sequence mode (for the antecedent sequence) and a property mode in case of implication properties, and the specification of a sequence mode in case of single-sequence properties. The mode for the consequent sequence of implication properties is derived from the property mode: Use of the *Pipe* property mode leads to the application of the *FirstMatchPipe* sequence mode, all other modes result in the use of the *FirstMatch* mode.

[7]The trivial true expression is never consumed since it does not represent any resource.

Mixed Level modeling

The modeling of the various abstraction levels in UAL is done by the use of different event operators within the event expressions of the various delay operators. The language does not include any restrictions concerning which operators can be combined or how. As a result, modeling mixed level assertions can be done in a quite straight forward way. Listing 5.4 shows an example for this.

```
1 sequence mixed_level
2  #1{T1'START}{true}
3  #1{T2'START;T1'END}{true}
4  #1{T1'END & T2'END;T3'START){true}
5  #1{T3'START@($delta_t == 10);timer(11)}{true}
6  #2{clk 'POS}{IRQ}
7  #1{clk 'POS}{!IRQ}
8  #1{T4'START@($delta_t == 2 * CLOCK_CYCLE)}{};
9 endsequence
```

Listing 5.4: Example for Mixed Level Assertions

The first delay operator matches with the start of transaction T1, the second checks that transaction T2 starts before T1 ends. T1 and T2 have to end at the same simulation time and before the start of transaction T3. T3 is required to start with a temporal distance of 10 time steps after the end of T1 and T2. At the second positive edge of signal clk the signal IRQ is checked and has to be `true`, followed by `false` at the next positive edge of clk. Finally, transaction T4 has to start with a temporal distance of two clock cycles after the second check of IRQ.

The following sections discuss the necessary enhancements of UAL in order to support the various concepts introduced by the assertion transformation.

In the following, the modified version of UAL will be referred to as UAL+.

5.2.3 Transformation Problem Classes and UAL

Interface Types

In order to detect transaction relations, it is necessary to recognize start and end of every single transaction call. UAL realizes this by the implementation of callbacks, that means an additional proxy module intercepts the transaction call, reports the start of the transaction to the assertion by an event, relays the transaction call to the target module, waits for the completion of the call, reports the end of the transaction to the assertion by a second event, and finally relays this information back to the calling module.

RTL designs on the other hand package this information in complex handshake protocols implemented with the help of signals. Changes of these signals can be tracked by the use of signal change events.

If TL assertions include transactions, then the corresponding refined assertion has to include an equivalent feature bundling the RTL signal protocols. This feature is not part of UAL and is introduced as a language extension.

Timing

UAL is exclusively based on events for sampling of design states, correlating actions, etc. All event based communication (as typically done on PV) can thus directly be captured by assertions. Those concepts that normally do not work with events are converted to events as well:

- As discussed above, start and end of transactions are converted to events by proxy modules.
- UAL provides an operator that is capable to generate events based on a certain amount of simulation time that has passed.
- Clock edges (as well as all other kinds of signal changes) directly produce events.

In most cases, the transformation of one timing scheme to another can be achieved by either replacing one event by another, or by exchanging specific delay values (for instance replace an absolute value by a multitude of a given clock period).

Since all relevant features for this are already part of UAL no extension in this context is necessary.

Additional Information

The various elements in this category do not require any specific adaptation of UAL.

5.2.4 Transaction Representation on RTL

Transactions are used for transporting data from one module to another and synchronizing their execution. A TL transaction is usually modeled as a remote function call which might be either blocking or non-blocking. On the other hand, communication in an RTL design is done via signals. These handshake signals lead to blocking behavior. The information bundled together in the transaction is usually distributed

among several signals which show a certain change pattern within one or several clock cycles. Hence, each TL transaction involved has to be converted to a sequence of signal changes representing that transaction.

In order to correlate a TL transaction to its RTL counterpart, both the signal sequence and the mapping of transaction parameters and return values to the corresponding signals have to be specified.

Here it has to be noted that it is necessary to still provide a mechanism for triggering the RTL assertions with events representing start and end of the different transactions. Otherwise, if clock edges are used instead, the structure of the assertion is changed which violates one of the basic requirements.

Detecting the end of a transaction is relatively easy. The simulator only has to check for the complete occurrence of the corresponding signal sequence. On the other hand detecting the start of an RTL transaction is a lot more complicated. While a TL transaction is called explicitly and can thus be easily identified from the very beginning, an RTL transaction might take several clock cycles for its complete execution; if there are more than one transaction starting with the same signal pattern in the first few clock cycles and only showing differences later on, it might be difficult to tell which transaction (if any at all) has been started by a certain signal sequence. The only possible solution to that problem lies in waiting until the check of the corresponding transaction sequence has been completed. At that time it becomes clear whether the transaction has really occurred. This problem will be discussed in detail below.

Parameters and return values of transactions have to be mapped to corresponding RTL registers or signals. Both, the mapping and the transaction sequence have to be provided by the designer, which has to be supported by the assertion language.

5.2.5 Trigger Concept

Introducing so called trigger sequences into UAL+ allows the specification of the signal sequence corresponding to an RTL transaction representation.

UAL+ files contain five sections for specifying different parts of the assertions[8]:

- The Ports section defines the interface of an assertion monitor and may include signals, events, and transactions.
- The Constants section defines constants of various data types that are visible within the whole monitor.

[8]For the original UAL grammar see Appendix B.

- The Sequences section allows the specification of UAL sequences.
- Similarly, the Properties section defines UAL properties.
- The Verification section specifies assertion and cover directives.

In order to use triggers, two steps have to be performed on the assertion. First, a sequence or property that should serve as the trigger has to be declared in the corresponding sections of the monitor. Instead of instantiating this sequence / property within an assertion however, it needs to be independent. As a consequence, the grammar of the UAL file gets extended by including an optional triggers section according to the following rule[9] (see also B.112):

monitor = `"monitor"` *identifier*
[*import_section*]
[*ports_section*]
[*constants_section*]
[*triggers_section*]
[*sequences_section*]
[*properties_section*]
verification_section
`"endmonitor"` ;

Within this section, it is possible to declare one or more triggers. The declaration of a trigger is defined by the following rule (see also B.113 and B.114):

triggers_section = `"triggers"`
trigger_declaration { *trigger_declaration* }
`"endtriggers"` ;

trigger_declaration = `"trigger"` *identifier* `"="` *trigger_kind* `";"` ;

Every trigger declaration consists of a unique identifier and a type which determines whether the trigger is a sequence or a property by using the corresponding keyword. The following rules show the correct declaration of triggers (see also B.115, B.116, and B.117):

trigger_kind = *sequence_trigger*
| *property_trigger* ;

sequence_trigger = `"sequence"` *sequence_instance* ;

property_trigger = `"property"` *property_instance* ;

Afterwards, the sequence / property is instantiated following the normal UAL syntax.

[9] The import section that has also been added is discussed in Section 5.3.2.

The problem of still needing a representation for transaction start and end when transforming transactions to sequences can be remedied by the introduction of several new attributes which can then be used in combination with trigger sequences and properties, either as parts of the event expressions triggering a delay operator or as part of its Boolean propositions. Table 5.3 shows an overview of the attributes producing events, while Table 5.4 details the attributes producing a Boolean result.

Attribute	**Produces an Event if ...**
seq'**ATTEMPT**	an evaluation attempt for *seq* has started
seq'**START**	*seq* has started
seq'**END**	*seq* has ended
seq'**MATCH**	*seq* has matched
seq'**NOTMATCH**	*seq* has finally not matched
prop'**SUCCESS**	*prop* has succeeded
prop'**FAIL**	*prop* has failed

Table 5.3: Trigger-related Attributes that produce Events

Since the resulting events might be used for the triggering of other sequences, these attributes have to be evaluated with the occurrence of every new event, in an analogous manner to how SVA starts new evaluation threads for all assertions with the occurrence of every new clock edge.

Attribute	**Evaluates to "true" if ...**
seq.**ATTEMPTED**	an evaluation attempt for *seq* has started
seq.**STARTED**	*seq* has started
seq.**ENDED**	*seq* has ended
seq.**MATCHED**	*seq* has matched
seq.**NOTMATCHED**	*seq* has finally not matched
prop.**SUCCEEDED**	*prop* has succeeded
prop.**FAILED**	*prop* has failed

Table 5.4: Trigger-related Attributes that produce Boolean Results

These attributes can be used within the Boolean expression of a delay operator or within the Boolean constraint condition within the sensitivity expressions of the delay operator. They are set whenever the corresponding events listed in the table above are generated and are cleared with the next occurrence of a primary event.

5.2.6 Categorization of Events

If the matching of a sequence is used for triggering other sequences another problem might occur for the implementation, though: If one sequence reacts not only to the matching of a second sequence but also to the event responsible for this match, it might be triggered several times instead of once.

```
1  sequence s1;
2    #1{e1}{A == B} #1{e2}{A == C};
3  endsequence
4
5  sequence s2;
6    #5{e2} true;
7  endsequence
8
9  sequence s3;
10   #2{e2 | s1 'END | s2 'END}{B == C};
11 endsequence
```

Listing 5.5: Example for Primary and Secondary Events

Listing 5.5 shows an example of this situation. Sequence *s3* can be triggered by the event *e2* as well as by the match of *s1* or *s2*. In some cases the occurrence of *e2* produces a match of *s1*, *s2* or even both, which might lead to an early match of *s3*, since one occurrence of *e2* basically triggers both delay steps of *s3* at once. Thus, it is essential that the delay operator is only triggered once in this case.

As a solution all events can be grouped into the following categories:

- Primary Events: This group captures all events that occur independently from all other events:
  - Design events: All events in this group originate in the design:
    - Annotated events (for instance sc_event)
    - Signal change events (for instance a positive clock edge)
    - Callback events (for instance start of a transaction)
  - Monitor events: This category includes all timer events that are not part of the design itself, but declared within the assertion monitor.

- Derived Events: This category includes all trigger events located in the assertion monitor. Each event in this group depends on exactly one specific primary event that happens simultaneously to all of its derived events according to Definition 2:

  - Sequence events: Start, end, matching, and not matching of sequences
  - Property events: Start, end, success, and failure of properties

This classification is shown in Figure 5.5.

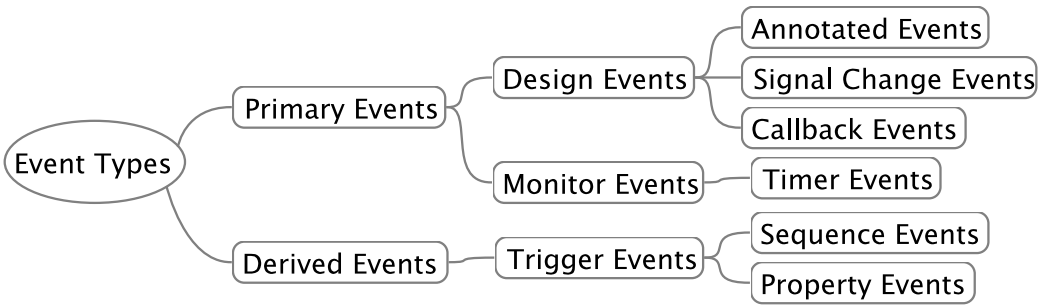

Figure 5.5: Classification of Events

If a primary event occurs during the normal execution, first all associated derived events are computed. A specific sequence evaluation thread may be triggered at most once per primary event, regardless of the number of occurring associated derived events.

In our example *e1* and *e2* are primary events while *s1'END* and *s2'END* are derived events associated with *e2*. An evaluation thread for *s3* may never be triggered twice, i.e. start and complete successfully, with only one occurrence of *e2*, even if this event also causes an occurrence of *s1'END* and / or *s2'END*.

In order to realize this behavior, the event handling of UAL has to be enhanced by an Evaluate-Update mechanism. The results of the various delay operators may not be passed to the next delay operator immediately; instead, the primary event and all corresponding derived events are processed by the assertion components and the results are stored temporarily. After all evaluations are finished, the results are propagated to the next delay operator. Thus, only one event can trigger any specific delay operator at a time.

5.2.7 Tentative Matching

As discussed above, the detection of transaction start on RTL can be difficult under certain conditions. If the corresponding signal protocol covers more than one clock cycle and several transactions start with the same signal signature, the real occurrence of the transaction can only be detected at a later point in time. In the most extreme cases, this might mean that the detection is possible only at the end of a long signal protocol.

Concept

The seemingly obvious solution for this scenario is to wait until the detection is complete and only then report the start of the transaction.

However, in this case an assertion might miss events located between the real and the apparent start of the transaction which results in overall wrong behavior.

In order to solve the problem without introducing the difficulties just mentioned, it is necessary to immediately issue an event with the first step of the signal protocol representing the transaction. Since it is not clear yet if this supposed transaction start is real or not, this event is declared tentative. As soon as a decision can be made on whether it is a real event or not, the evaluation thread is either allowed to continue or canceled. In all other regards, the tentative event is treated just as a normal event.

Interaction with Execution Modes

The various UAL execution modes for sequences and properties may cause additional problems when combined with the tentative matching concept:

- A sequence running in *FirstMatch* mode might lead to the situation that the shorter subthread contains a tentative event, while the longer one does not. If it is not clear whether the tentative event is a real event or not when the longer subthread tries to match, this match attempt has to be considered dependent on the final result of the shorter subthread. The tentative status is more or less inherited by this subthread.

- Pipelined evaluation might lead to the situation that a thread involving tentative events blocks the matching event index and / or consumption of Boolean expressions for other, non-tentative threads. Again, the latter becomes dependent on the status of the original, tentative thread.

- When using on of the various property modes dealing with additional antecedent matches during the consequent evaluation, an antecedent match including a tentative event requires a decision of whether or not to cancel existing evaluation threads, or ignore / report the new attempt.

Figure 5.6 illustrates these challenges. Events `E1`, `E2`, and `E3` represent normal events, while event `T` is a tentative event.

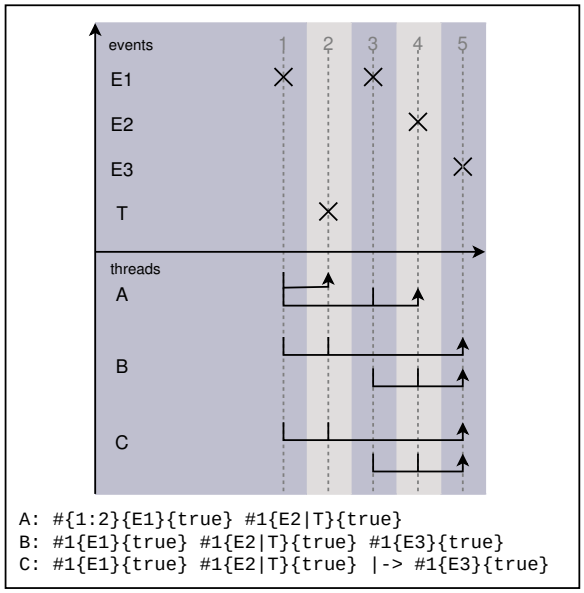

Figure 5.6: Interaction of Execution Modes with Tentative Concept

Assertion `A`, running in *FirstMatch* mode, includes a delay range and every evaluation thread is split into two subthreads. The upper subthread could match at time step 2 under the condition that `T` is validated. The lower subthread is not tentative and could produce a normal match at time step 4. If at time step 4 it is not clear yet, whether `T` will be validated or falsified, the final result has to be delayed.

Assertion `B`, running in *Pipe* mode, shows two independent evaluation threads. At time step 5 both could produce a match, but due to the pipelined evaluation, only the older thread may match. Since the older thread includes a tentative event however, it might not be clear yet at that time if this thread is even valid. If the older thread is discarded, the younger thread is allowed to match after all.

Assertion `C` is structured similar to assertion `B`, however it includes an implication. If this assertion is running in *NoRestart* mode, the younger thread has to be canceled before the consequent is evaluated. But due to the fact that the older thread again

includes a tentative event, it might be discarded later, which would imply that the younger thread has to be evaluated after all.

Implementation

In order to deal with the problems posed by the different evaluation modes, the implementation of tentative events consists of two parts:

1. Whenever a tentative event is encountered, all assertion evaluation threads waiting for this event are split at this point. One half assumes that the event is a real event and proceeds the evaluation accordingly while the other one assumes that the event is not real and continues waiting for the occurrence of the next instance (both threads store the result of their decision).

   There is one exception to this rule however: If the first delay operator of an antecedent sequence (or of the sole sequence of a single-sequence property) encounters a tentative event, no split happens. Instead, only the thread assuming the event is real is created (including its corresponding marking). The second thread waiting for the next occurrence is not created, since no evaluation would have been started without an event anyway. As a result, the start point of this assertion evaluation thread would be wrong if the tentative event occurrence proves to be false.

2. As soon as it becomes clear which decision was the right one - and accordingly which thread represents the correct assertion behavior - all evaluation threads using the wrong decision are canceled.

Figure 5.7 shows several examples for this splitting of threads. Again, events `E1`, `E2`, and `E3` represent normal events, while event `T` is a tentative event.

The evaluation thread of assertion `A` is split when encountering a tentative event. The upper thread (running under the premise that `T` will be validated) could match at time step 2, the lower one (under the premise that `T` will be falsified) could match at time step 3. Both threads are mutual exclusive. Whichever decision will be made concerning `T`, one thread will continue, the other will be canceled.

Assertion `B` is a little bit more complicated. After the first occurrence of `T`, the thread is split as before. While the upper thread continues as usual, the lower one encounters another occurrence of `T`. So again, the thread is split. With the next occurrence of `T`, this is repeated, and so on. Altogether it can be said that using a tentative event in an event expression without alternatives always leads to an infinite number of threads (unless one of the already created threads is validated in between).

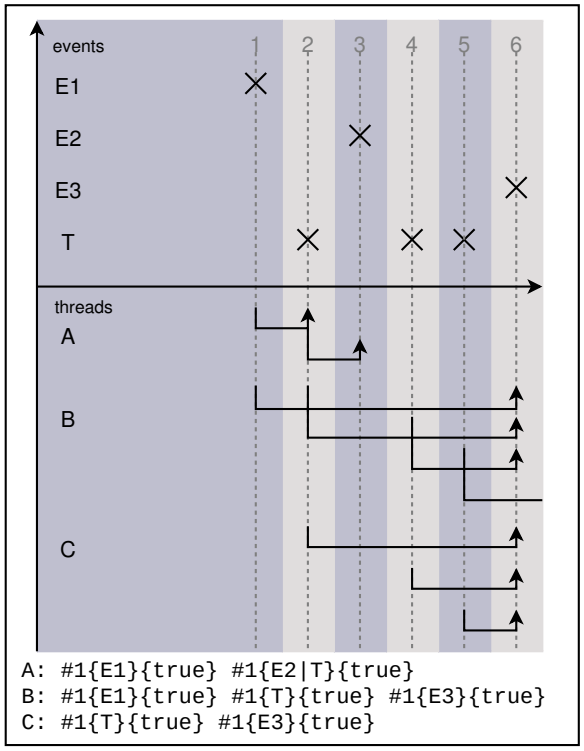

Figure 5.7: Split of Threads in case of Tentative Events

Assertion `C` looks similar to assertion `B`, however the tentative event is located in the first delay operator of the sequence. In this case, the thread is not split (since it should be started anyway if there is no real event). Instead the next occurrence of `T` starts a new evaluation thread (which is again tentative).

While the rejection of a tentative event might happen earlier, the confirmation is only possible after the whole signal protocol representing the transaction has occurred completely. In case of long protocols, this results in a huge computation overhead. However, it might be possible for the user to determine whether the transaction will occur or not even before that point. An example for that is a couple of transactions that start with the same protocol steps, can be differentiated by several signal values in the middle, and continue with the assignment of parameter values afterwards. Knowledge about the parameter values is not necessary for the identification of the transaction. So, if the user is willing to ignore the possibilities of incorrect implementation of the signal protocols, it is sensible to provide means for early decision on the tentative events. By adding the directive `$confirm` to the trigger sequence, the assertion engine is able to reduce the required computation effort for the confirmation of the tentative event.

```
1 sequence trig_seq
2   #1{clk 'POS}{WR}#1{clk 'POS}{!WR, $confirm}#1{clk 'POS}{true};
3 endsequence
```

Listing 5.6: Early Confirmation of Tentative Events

Listing 5.6 shows a sample trigger sequence consisting of three delay steps. The first two are relevant for the identification while the third might be used for the correct setting of parameter signals[10]. By including the `$confirm` directive in the second delay operator, the evaluation concerning the tentative status of start events can be finished sooner. As a result, the number of threads to be taken into consideration all the time is lower, which means the simulation performance is better.

In many cases, the decision concerning tentative events can be made before the corresponding evaluation thread produces a result. On the other hand, if this is not the case, then the presence of one or more tentative events in the sequence evaluation leads to a tentative sequence result, which in turn would propagate to the corresponding property and even the assertion. Since every assertion evaluation has to have a definite result, this is not acceptable. Thus, the propagation of these results has to be delayed until a definite decision is reached. In the worst case, the assertion is not able to report a possible violation immediately. But since the event index of the violation can be stored, so it is still possible to correlate the assertion violation with the actual cause.

5.2.8 Constrained RTL transactions

Another transaction related challenge for the refinement lies in the use of constraint operators referring to transaction parameters in combination with the start or end events of a transaction. An example for this is shown in Listing 5.7.

```
1 sequence s1;
2   #1{Write 'START@(Write.Addr == 1)}{true};
3 endsequence
```

Listing 5.7: Example for Constrained RTL transactions

On TL, it is very easy to check these kinds of constraints, since these values have to be known when calling the function. On RTL however, the signals corresponding to these parameters are not necessarily stable at the beginning or the end of the signal protocol.

[10] This step cannot be omitted though, since otherwise the transaction end event would be issued too early.

If a transaction call of this kind has to be refined towards RTL, a method for checking exactly this kind of behavior is required.

If the constraint is applied to the end of a transaction, then it is possible to simply store the values of all parameters in dedicated variables and then use these variables instead of the original signals.

In case of constrained start events however, the situation is a little bit more complicated: First of all, since the data is only available after it has to be checked, storing these values does not solve the problem. Second, start events per se are always tentative, which means that this has to be taken into account when dealing with the problem.

The solution for the start problem depends on the exact time when the required value or values are stable:

- If all values are stable at the first step of the signal protocol, no adjustment is necessary.
- If all values are stable at the same time, but after the first protocol step, the constraint expression can instead be moved into the trigger sequence[11]. Here, it is placed in the Boolean expression of the corresponding delay operator where it is connected to existing expressions via an `AND` operator. While strictly speaking the structure of the assertion is changed in this case (since a Boolean constraint operator vanished from an event expression), the missing part is merely moved to another place due to different timing notions of the designs / assertions involved. As such, it can be defined that this move does not affect the assertion structure in a wider sense.
- If all values are stable at different times but all constraints are connected together via `AND` operators, then the different parts can be distributed over the various Boolean expression parts of the trigger sequence, since all of them have to evaluate to `true` one after the other.
- If the stability points are distributed and the connection includes other operators than `AND` (for instance `OR`, `XOR`), then this approach does not work anymore. Instead several sequences have to be connected together using the corresponding sequence operators (`s_AND`, `s_OR`, `s_XOR`, etc.) where every of these subsequences contains one part of the constraint expression.

While the first three possibilities have been included into UAL+, the last one is not possible yet, since the language does not yet support the required sequence operators.

[11] If this trigger sequence is also used in other, unconstrained expressions, the trigger sequence has to be duplicated instead and the changes are applied to the copy only.

5.2.9 Enhancement of time related Operators and Functions

UAL provides one operator and one function that deal with simulation time:

- The `timer(X)` operator allows the scheduling of an event X timesteps after the assertion evaluation reaches the operator.
- The `$delta_t` function provides the number of time steps that have passed between the triggering of the previous delay operator and the current evaluation time point.

Both are used for assertions checking PVT and CA designs and both do not affect the structure of a given assertion (see Definition 6). On the other hand, CC and RTL designs use dedicated clock signals for the synchronization instead of simulation time. In order to keep the corresponding assertions as similar as possible on the one hand and to not cause additional problems concerning structure preservation both are enhanced to also support counting of clock edges.

When specifying an event in brackets directly after the `timer(X)` operator, the Xth occurrence of the specified event will cause the timer operator to emit an event. The following grammar rule demonstrates this (see also B.118):

trigger_timer = `"timer"` [`"["` *event_operand* `"]"`]
`"("` *timer_expression* `")"` ;

A sample timer expression could look like this:

```
timer[clk'POS](5)
```

Similarly, specifying an event in brackets after the `$delta_t` function causes the function to not count the passed simulation time since the last trigger event, but instead to count the number of occurrences of the given event.

Since these features are mainly intended for the counting of clock edges, tentative events are not allowed to be used in this context. This also results in an easier implementation.

5.2.10 Summary

Altogether, the assertion language was enhanced by the introduction of an RTL representation for transactions, the trigger concept and the corresponding categorization of events. Furthermore, the concept of tentative events and tentative matching was introduced, the challenge of dealing with constrained RTL transactions and event based time expressions were discussed.

A formal representation of both the tentative event handling and the event based timer operator introduced here will be shown in Chapter 6.

5.3 Assertion Transformation Language

This chapter introduces a new language developed for the transformation of UAL+ assertions. This transformation language is capable of handling assertions covering all TL sublevels, RTL, and any kind of mixed level assertions.

5.3.1 General Overview

The various features of the transformation language can be grouped into four layers:

1. Modeling Layer: Describes the structure of the transformation file
2. Rule Layer: Consists of rules - complex transformation commands - composed of directives, transactor calls, and / or other rules
3. Transactor Layer: Consists of transactors, used to change from signal based interfaces to transaction based ones and back
4. Directive Layer: Consists of directive calls, predefined transformation commands

These layers are described in the following sections. The formal grammar rules that are mentioned in this context are an excerpt of the complete grammar in Appendix B.

5.3.2 Modeling Layer

Every set of transformation descriptions can be encapsulated within a corresponding structure. Several of these structures can be placed within one refinement file. The following grammar rules illustrate this concept (see also B.1 and B.2):

refinement_file = *refinement_definition* { *refinement_definition* } ;

refinement_definition = `"refinement"` *identifier*
[*import_section*]
[*declarations_section*]
[*transactors_section*]
[*rules_section*]
apply_section
`"endrefinement"` ;

Every refinement definition consists of up to six different sections which are described below.

Import Section and Library Files

This section provides the possibility to include one or several library files with predefined transformation descriptions according to the following rules (see also B.5 and B.6):

import_section = *import_declaration* { *import_declaration* } ;

import_declaration = `"import"` *identifier* `";"` ;

The structure of the imported library files looks like this (see also B.3 and B.4):

library_file = *library_definition* { *library_definition* } ;

library_definition = `"library"` *identifier*
[*declarations_section*]
[*sequences_section*]
[*properties_section*]
[*transactors_section*]
[*rules_section*]
`"endlibrary"` ;

With the exception of an import section and an apply section, a library file may contain the same elements as a refinement file. The former is excluded in order to avoid recursive dependencies, the latter since a libray file is a passive element and thus, should not contain active directive calls.

Library files can be used to package generic, often used transformation descriptions, in order to allow easy reuse. A common application is the tranformation of assertions on standardized design behavior like communication protocols.

Additionally, it is possible to declare sequences and properties within a library file. In combination with port and constant declarations, all reusable parts of a monitor definition can be placed within a library in this way.

Declarations Section

This section allows the specification of various elements that can be used for the transformation process, namely constants, ports, and local variables. The declarations sections is structured as follows (see also B.7):

declarations_section = `"declarations"`
[*ports_section*]
[*constants_section*]
[*variables_section*]
`"enddeclarations"` ;

All elements declared here are usable within the apply section of the refinement file.

The constants section follows the same grammatic rules as the one within the UAL+ monitor file (see also B.49). Its main purpose is the possibility to specify clock periods for transformations from or to the CA sublevel. Other uses include the specification of reset modes, active levels for enable signals, etc.

In the ports section various monitor ports (signals, events, and transactions) can be specified. These ports might be added to an assertion interface during a transformation. The ports section follows the same grammar rules as the one within the UAL+ monitor file (see also B.47).

The variables section consequently allows the specification of local variables that are to be used within the transformation according to the following grammar rule (see also B.8):

variables_section = `"variables"`
localvar_declaration { *localvar_declaration* }
`"endvariables"` ;

The declaration of the variables themselves happens exactly as in UAL+ (see also B.107).

Transactors Section

This section allows the specification of transactors, that means the exact mapping of a TL transaction call to an RTL signal protocol. This section can hold an arbitrary number of transactors according to the following grammar rules (see also B.9 and B.10):

transactors_section = `"transactors"`
transactor_section { *transactor_section* }
`"endtransactors"` ;

transactor_section = `"transactor"` *identifier* *transactor_interface*
[*map_section*]
sequence_section
`"endtransactor"` ;

The transactor interface is defined according to the following rule (see also B.11):

transactor_interface = `"<"` *identifier* `">"` *formal_argument_list* ;

The list of formal arguments has to include one TL transaction definition (including name, return type, name and type of arguments), a name for the corresponding trigger sequence, and type and names of all RTL signals involved (either used within the signal protocol or for holding data).

The map section details which parameter or return value of the transaction has to be connected to which signal (see also B.12 and B.13):

map_section = `"map"`
map_declaration { *map_declaration* }
`"endmap"` ;

map_declaration = *identifier* `"."` (*identifier* | `"RET"`) `"=>"` *identifier* `";"` ;

The left side of the arrow contains the TL transaction parameters - the name of the transaction, followed by a dot, followed by the parameter name or the keyword *RET* for the return value - while the right side lists the corresponding RTL signals.

The sequence section on the other hand specifies the RTL sequence that represents the transaction. Since this sequence is a standard UAL+ sequence it follows the corresponding UAL+ grammar rules (see also B.68).

An example for a transactor specification can be seen in Listing 5.8.

Note that this transactor mechanism can also be used in order to specify an RTL equivalent for a TL design event. In this case, the map section is ommitted and only the sequence section is used. If the transactor represents an event then this event

```
1 transactor put_transactor<location>(transaction void PUT(int
    data),
2       trigger put_trigger,
3       signal sc_signal<bool> clk,
4       signal sc_signal<bool> en,
5       signal sc_signal<int> data)
6   map
7     PUT.data => data;
8   endmap
9   sequence put_trigger[FirstMatch](signal sc_signal<bool> clk,
      signal sc_signal<bool> en)
10    #1{clk'POS}{en} #1{clk'POS}{!en};
11  endsequence
12 endtransactor
```

Listing 5.8: Transactor Example

corresponds to the match of the transactor sequence.

Rules Section

The rules section is used for the specification of rules - complex transformation descriptions - in the following way (see also B.15):

rules_section = `"rules"`
rule_section { *rule_section* }
`"endrules"` ;

Every rule consists of a unique identifier, an interface, and the actual rule specification as can be seen in the following grammar rules (see also B.16, B.17, and B.22):

rule_section = `"rule"` *identifier rule_interface*
rule_specification
`"endrule"` ;

rule_interface = `"<"` *identifier* `">"`
`"("` [*rule_parameters*] `")"` ;

rule_specification = *rule_item* { *rule_item* } ;

Rule parameters consist of a type entry and an identifier (see also B.18, B.19, and B.20):

rule_parameters = *rule_parameter* { `","` *rule_parameter* } ;

rule_parameter = *rule_parameter_type identifier* ;

```
rule_parameter_type  =  "declaration"
                        | "name"
                        | "value"
                        | "type"
                        | "expression" ;
```

The various parameter types differ in how the object that is passed as parameter can be accessed and what information is available:

- Type *declaration* can only be used in combination with objects declared within the declarations section (constants, ports, variables). All elements of the corresponding declaration can be accessed by special functions which are defined as follows (see also B.44):

```
declaration_access  =  "name()"
                       | "type()"
                       | "value()" ;
```

- Type *name* refers to an existing object name, for instance the name of a variable, an event on a port signal, etc.
- Type *value* refers to a literal value, for instance the value "5".
- Type *type* refers to the data type of an existing object, for instance "sc_bit".
- Type *expression* refers to a complex expression composed of elements of types *name* and / or *value*.

Every rule item can be either the activation of another rule that has already been declared (or imported), the call of a transactor, or the call of a transformation directive (see also B.23):

```
rule_item  =  directive_instance
              | transactor_instance
              | rule_instance ;
```

Details concerning the usage and meaning of rules can be found in Section 5.3.3, details about the usage of transactors can be found in Section 5.3.4, and finally details concerning directives can be found in Section 5.3.5.

Apply Section

This section contains the actual refinement description. Predefined or imported rules, transactors, and directives can be applied here in order to perform the assertion transformation. The underlying grammar rule for the apply section can be seen here (see also B.21):

apply_section = `"apply"`
rule_specification
`"endapply"` ;

As can be seen, the apply section may contain the same elements as a rule. Thus, the transformation description within the apply section can be considered the master rule which in turn relies on "sub"-rules. The only restriction for this master rule is that it may not make use of parameters.

5.3.3 Rule Layer

The rule layer can be used to group several transformation directives and other rules in order to encapsulate related functionality. A rule could for instance include all necessary directives in order to transform the timing information within an assertion while another rule transforms the interface.

An additional benefit of rules is that they allow the use of parameters which can then be used for the included directives or sub-rules. By providing the possibility of parameterizing existing rules, it is easy to collect rules related to a specific goal and all additional information required by these rules (constants declarations, objects, transactors, etc.) and put them in a library file. Every refinement file using this library file can then provide different parameters for the given rule.

As mentioned above, rules may also use other rules. In order to avoid recursive calls, rules may only use rules that have been declared further up within the rules section. Rules imported from a library file may be used as well. Since library files cannot import other library files, recursive calls are prevented by default.

The declaration of rules was already discussed in the previous section. Declared rules are instantiated within either the apply section or other rules according to the following grammar rule (see also B.24):

rule_instance = *identifier* `"<"` *target_locations_list* `">"`
`"("` [*rule_arguments*] `")"` `";"` ;

The identifier specifies one of the rules declared within the rules section. The location parameter has to specify an existing sequence within the assertion monitor. For details on the location parameter see Section 5.3.6.

The rule arguments can be passed to transactor or directive calls within the rule or other instantiated rules. They follow these grammar rules (see also B.25 and B.26):

rule_arguments = *rule_argument* { "," *rule_argument* } ;

rule_argument = *string*
| *expression*
| *event_expression*
| (*identifier* "." *declaration_access*)
| *value* ;

An example for a rule instantiation can be seen in Listing 5.9.

```
1 my_transform<my_monitor.my_assertion>(clk, CLOCK_CYCLE.name());
```

Listing 5.9: Example for Rule Instantiation

5.3.4 Transactor Layer

The transactor layer covers the use of transactors in order to transform between a signal based interface on the one hand and a transaction based interface on the other hand.

Using a transactor happens similarly to the usage of rules or directives. Transactors fall in neither category, since they can be defined by the user, but at the same time they cannot include other functionality. Thus, they are located between rules and directives.

Instantiating a transactor causes the transformation of a sequence interface. An existing transactor is instantiated according to the following rule (see also B.14):

transactor_instance = *identifier* "<" *target_locations_list* ">"
"[" *refinement_mode* "]"
param_argument_list ";" ;

The identifier specifies one of the transactors declared within the transactors section. For details on the location parameter see Section 5.3.6.

The refinement mode paramter is defined as follows (see also B.45):

refinement_mode = "up"
| "down" ;

This parameter determines which interface transformation has to be performed:

- The value *up* represents a transformation from a signal based interface up to a more abstract transaction based interface.
- The value *down* specifies a transformation from a transaction based interface down to a less abstract signal based interface.

The parameter list finally is used to pass the TL transaction and the corresponding RTL signals to the transactor.

If the transactor is applied to a sequence, then all of the corresponding events and objects are replaced, the interfaces of the sequence itself, its instantiating property, the verification directive, and the monitor are adapted, and finally, a trigger sequence is added or removed.

Listing 5.10 shows an example for a transactor instantiation.

```
1 put_transactor<my_monitor.my_assertion.my_property.my_sequence>[
    down]
2   (PUT, put_trigger, clk, en, data);
```

Listing 5.10: Example for Transactor Instantiation

5.3.5 Directive Layer

This layer provides the built-in transformation functionality of the transformation language that consists of a list of predefined directives. In constrast to rules which can be specified by the user - which also allows the addition of new rules - the directives are an inherent part of the transformation engine and cannot be extended.

Activating a directive is done according to the following grammar rule (see also B.27):

directive_instance = *add_directive*
| *modify_directive*
| *remove_directive* ;

All directives fall into one of three groups: The first group adds an object to a monitor, the second group modifies an existing object, while the third group removes an object from a monitor.

Add Directives

All add directives are used for introducing new elements into a given assertion monitor, assertion, or part of an assertion. Add directives follow the grammar rule (see also B.30):

add_directive = "add" "." *directive_target*
"<" *target_locations_list* ">"
add_specification { *add_specification* } ;

The directive target specifies which add directive is to be executed (see also B.43):

directive_target = "boolean_expression"
| "constant"
| "parameter"
| "pass_copy"
| "pass_reference"
| "reset"
| "sensitivity"
| "time_constraint"
| "timer"
| "variable"
| "variable_assignment" ;

Note that not all possible values result in existing directives. Table 5.5 presents all available add directives including the location where these directives are applied (for details on location parameters see Section 5.3.6). All directives listed in parentheses do not exist.

Directive	**Target Location**
(add.boolean_expression)	n/a
add.constant	Monitor
add.parameter	Sequence / Property
add.pass_copy	Sequence / Property
add.pass_reference	Sequence / Property
add.reset	Verification Directive
(add.sensitivity)	n/a
add.time_constraint	Sensitivity
add.timer	Sensitivity
add.variable	Sequence / Property
add.variable_assignment	Delay Operator

Table 5.5: Available Add Directives

The corresponding add specification determines the content of the addition and in some cases it also provides additional information about the location of the addition (see also B.31):

add_specification = "{" [*refines*] *additives* "}" ;

The *additive* keyword specifies what element has to be added. This could be the declaration of a constant, a reset expression, etc. The corresponding grammar rules look like follows (see also B.32 and B.33):

additives = *additive* { *additive* } ;

additive = "additive" [*position_marker*] *directive_string* ";" ;

The position marker in this rule is only used in combination with the optional *refines* keyword. These constructs may only be used for the *add.time_constraint* and *add.timer* directives. The corresponding grammar rule looks like this (see also B.34):

refines = "refines" ["[" *event_operand* "]"] *directive_string*
{ "," *position_marker* "," *directive_string* } ";" ;

These two directives allow the specification of one or more parts of the corresponding event expression where the new element has to be inserted. Every specified identifier is then used with a separate *additive* expression in order to insert this particular string at the place of the identifier within the *refines* expression.

Listing 5.11 shows an example of how the *add.time_constraint* directive can be used.

```
1 add.time_constraint<location>
2 {
3     refines "E1",expr1,"|E2",expr2;
4     additive (expr1) "$delta_t==(",var_name,");";
5     additive (expr2) "$delta_t==(10);";
6 }
```

Listing 5.11: Example for Adding a Time Constraint

The directive in this case looks for all occurrences of the expression "E1|E2" at the specified location. After the "E1" a time constraint depending on a local variable is to be inserted, after the "E2" a time constraint of ten time steps.

Modify Directives

Modification directives change existing parts of an assertion monitor. Modify directives are specified according to the following grammar rule (see also B.35):

modify_directive = "modify" "." *directive_target*
"<" *target_locations_list* ">"
modify_specification { *modify_specification* } ;

The meaning of directive target and location parameters is the same as for add directives. Table 5.6 lists the legal directive targets for modify directives as well as the corresponding locations.

Directive	**Target Location**
modify.boolean_expression	Delay Operator
modify.constant	Monitor
(modify.parameter)	n/a
(modify.pass_copy)	n/a
(modify.pass_reference)	n/a
modify.reset	Verification Directive
modify.sensitivity	Sensitivity
modify.time_constraint	Sensitivity
modify.timer	Sensitivity
(modify.variable)	n/a
modify.variable_assignment	Delay Operator

Table 5.6: Available Modify Directives

The modify specification is defines as follows (see also B.36):

modify_specification = "{" *refines modification* "}" ;

In contrast to add directives, the *refines* branch is mandatory now, since it specifies the original expression that is to be modified. The new expression is specified using the modification keyword (see also B.37):

modification = "modification" ["[" *event_operand* "]"]
directive_string ";" ;

The specification of an event operand is only legal for *modify.time_constraint* and *modify.timer* directives. If it is left out, the corresponding operator - original or new - works on the basis of simulation time. Otherwise, the corresponding clock

event expression has to be specified and the operator works on the basis of this event expression (see Section 5.2.9 for details).

Listing 5.12 shows an example of how the *modify.timer* directive can be used.

```
1 modify.timer<location>
2 {
3   refines "10";
4   modification [clk 'POS] "1";
5 }
```

Listing 5.12: Example for Modifying a Timer

The directive looks for all occurrences of the value "10" within the timer operator at the specified location. This value is replaced by the value "1" and the timer is changed from being based on simulation time to counting positive edges of the *clk* signal.

Remove Directives

Remove directives are the opposite of add directives and delete existing elements from an assertion monitor. Remove directives are specified according to the following grammar rule (see also B.38):

remove_directive = "remove" "." *directive_target*
"<" *target_locations_list* ">"
{ *remove_specification* } ;

Table 5.7 shows an overview over the legal choices for directives.

The structure of the remove specification is defined according to the following grammar rules (see also B.39 and B.40):

remove_specification = "{" *removal* { *removal* } "}" ;

removal = "removal" *directive_string* ";" ;

After the *removal* keyword, the element that should be removed is specified. Remove directives do not contain a *refines* keyword, since it is not necessary to further narrow down the description of where to find the element that is to be removed.

In case of *remove.time_constraint* and *remove.timer* directives, even the *removal* keyword is not needed. These directives will simply remove all time constraints and

Directive	Target Location
(remove.boolean_expression)	n/a
remove.constant	Monitor
remove.parameter	Sequence / Property
remove.pass_copy	Sequence / Property
remove.pass_reference	Sequence / Property
remove.reset	Verification Directive
(remove.sensitivity)	n/a
remove.time_constraint	Sensitivity
remove.timer	Sensitivity
remove.variable	Sequence / Property
remove.variable_assignment	Delay Operator

Table 5.7: Available Remove Directives

timer operators they encounter at the specified location (it also does not matter whether these operators work based on time or on events).

Listing 5.13 shows an example of how the *remove.reset* directive can be used.

```
1 remove.reset<location>
2 {
3   removal "rst 'NEG";
4 }
```

Listing 5.13: Example for Removing a Reset

The directive removes the negative edge of the *rst* signal from the reset expression of the given assertion (specified by the location parameter).

5.3.6 Location Parameters

Every rule, every transactor, and every directive requires a location parameter that specifies at which point of a given monitor they are supposed to be applied. As shown in Tables 5.5, 5.6, and 5.7, the various directives require different locations: An *add.constant* directive for instance would only require the name of the monitor to which the constant declaration is to be added, in case of a *remove.timer* directive on the other hand, the location has to point to the correct sensitivity entry (that means choosing either positive or negative sensitivity of a given delay operator) where the timer operator should be removed.

The location parameter is defined according to the following grammar rule (see also B.28 and B.29):

```
target_locations_list  =  target_locations { "," target_locations } ;
target_locations       =  identifier
                          { "." ( identifier
                          | number
                          | "*" ) } ;
```

If the location is too specific - for instance if the directive requires a sequence location but is provided with the location of a specific delay operator within that sequence - then only the relevant part of the location is considered. In case of directives that can have several legal locations, the longest part of the specified location that still applies is considered.

If on the other hand the location is too unspecific - for instance if the directive requires a property location but is provided with only the location of an assertion - then the compiler will issue an error message and abort the transformation.

In some cases, it might be necessary to distinguish between a class of objects or a specific instance for the same object when transforming an assertion. One example is the use of a particular sequence in several properties. If one of these properties checks a part of the design that has been changed while the other checks an unmodified part, then only the sequence instance used in the first property has to be modified. On the other hand, if a transformation step changes the timing of a PVT model on a whole then all instances of a specific sequence have to be modified.

This problem is addressed by a flexible use of the location parameter. A specified location parameter always expects a hierarchical path to the destination, but there are two different ways this hierarchy could be specified:

1. When addressing a specific instance of a sequence, the location would start with the name of the monitor, followed by the name of the verification directive instantiating the sequence, followed by the name of the corresponding property, and finally by the name of the sequence itself (all elements are separated by a "."). Specifying the location of other design units (properties, etc.) works in a similar way.

2. When addressing all instances of a sequence, the location would still start with the name of the monitor. In this case the next part would be the keyword *sequences* (referring to the sequences section of the monitor file), followed by the name of the sequence.

If only one element of an object class is affected by the transformation, the transformation engine duplicates the original object declaration and modifies only the copy.

It is taken care that all corresponding references are updated to refer to the copy afterwards.

Some operations should be performed on more than one element - for instance the *remove.timer* directive mentioned above is supposed to remove all timer operators from all trigger expressions of all delay operators of a given sequence. For these purposes, several locations can be specified at once. Alternatively, the wildcard operator "*" can be used. This wildcard specifies that at this particular level of hierarchy all possible elements are affected.

```
1  monitor test
2
3  sequences
4   sequence read(...)
5    #1{clk 'POS;rst 'NEG} {en && !wr} #1{clk 'POS;rst 'NEG} {true};
6   endsequence
7
8   sequence write(...)
9    ...
10  endsequence
11 endsequences
12
13 properties
14  property read(...)
15   read[...](...) |-> write(...)
16  endproperty
17
18  property init(...)
19   ...
20  endproperty
21 endproperties
22
23 verification
24  assert_cover my_test(...) = read[...](...);
25 endverification
26
27 endmonitor
```

Listing 5.14: Location Example: Sample Monitor

```
1 test.my_test.read.write.1.*
2
3 test.sequences.read.*.positive
4
5 test.*.read
```

Listing 5.15: Location Example: Sample Locations

Listing 5.14 shows an excerpt from a monitor description (irrelevant parts were abbreviated by ...). The monitor contains one assertion directive as well as two

properties and two sequences. Listing 5.15 provides several possible location descriptions that could be used with this monitor.

The first location consist of the name of the monitor (*test*), followed by the name of the assertion directive (*my_test*), the name of the property (*read*), and the name of the sequence (*write*). The number "1" specifies the first delay operator within that sequence. The following wildcard addresses both the positive and the negative event expression of this delay operator.

The second location starts again with the name of the monitor (*test*), but now uses the keyword *sequences*, followed by the sequence name (*read*). The remainder of the location specifies the positive event expression of all delay operators within that sequence.

As can be seen, the wildcard operator can be used when targeting a class of objects. However, it is not possible to use a wildcard for replacing the keyword specifying the file section where the object is declared.

This is shown in the third location example. After specifying the monitor name, a wildcard is used, followed by the name *read*. This location does not target both the property object and the sequence object of this name. Instead, the wildcard stands for all assertion directives and thus, the location targets property instances of the name *read* that are instanciated within all assertion directives of this monitor.

This also means that it is not possible to affect objects of different kinds (for instance a sequence and a property) with the same directive activation by the use of wildcards. In order to do this, both locations have to be specified explicitely instead.

5.4 Constraints and Guidelines for Refinement

The following section summarizes several restrictions the transformation process has to face either due to the inherent differences of the abstraction levels involved or based on design complexity of some modeling styles. Additionally, there are some transformations that are not easily reusable while others might have a big impact on the simulation performance.

5.4.1 Ambiguity Issues

In some cases, differences between the various abstraction levels result in the fact that a structure preserving assertion transformation will produce errors even though the design refinement is correct. An example for this behavior can be seen in Listings 5.16 and 5.17.

```
1 sequence TLM
2  #1{PUT'END}{true}#1{GET'START}{true};
3 endsequence
```

Listing 5.16: Example for Transformation Ambiguity (1)

```
1  sequence PUT
2   #1{clk 'POS}{WR}#1{clk 'POS}{!WR};
3  endsequence
4
5  sequence GET
6   #1{clk 'POS}{!WR};
7  endsequence
8
9  sequence RTL
10  #1{PUT'MATCH}{true}#1{GET'START}{true};
11 endproperty
```

Listing 5.17: Example for Transformation Ambiguity (2)

The first listing shows a TL sequence checking for the successful termination of a *PUT* transaction followed by the start of a *GET* transaction.

In the second listing these transactions are replaced by trigger sequences (the actual trigger declaration is omitted, since it does not add any useful information to the example). It is clear that in the RTL model the end of the *PUT* transaction and the start of the *GET* transaction can happen at the same clock edge. According to the concept of derived events, assertions may be triggered at most once by any occurrence of a primary event and any of its derived events. As a consequence, the assertion would miss the start of the *GET* transaction and subsequently would produce results that do not match with the results of the TL assertion.

There are three ways of dealing with the situation which can be chosen by using corresponding options of the transformation engine:

1. The transformation engine produces a warning and states that this situation might lead to incorrect results. The assertion is then transformed preserving its structure.

2. The assertion is transformed in a strict manner. False positives are avoided at the cost of potential false negatives. In the example the second `#1` of the *RTL* sequence would have to be replaced by a `#0`.

3. The assertion is transformed in a safe manner. False negatives are avoided at the cost of potential false positives. In the example the second `#1` of the *RTL* sequence would have to be replaced by a `#{0:1}`.

Note that the last two choices violate the structure preservation.

Another example for this phenomenon is the transformation of a PVT model to CA. If the time values used within the PVT model are not an exact multitude of the CA clock period, an exact transformation is not possible. For this scenario no safe or strict way exists. Instead, the transformation engine will automatically issue a warning and allow the transformation of every time value to the closest multitude of the clock period, to a range of appropriate values, etc.

5.4.2 Detectability Issues

In Section 5.2.4, an RTL representation for transactions was introduced in order to allow a structure preserving transformation of TL assertions down to RTL. If this representation (including the use of triggers) is used when manually writing UAL+ assertions on RTL then a transformation to higher abstraction levels is possible without restrictions. If on the other hand RTL assertions include the signal protocols of RTL transactions directly within the assertion code, then structure preservation is no longer possible. Due to the great difficulty of identifying the distributed signal changes contributing to the RTL transaction in order to map them to a function call on TL, it is also very difficult to provide a method without regard to structure preservation as described in the previous section. This is especially true, since it is not uncommon that one signal is used for several different transactions.

Note that the remaining RTL features, like clock based timing, are not influenced by this restriction.

5.4.3 Restrictions on Reuse

The transformation language presented here was developed with a focus on potential reuse of the various transformation descriptions in mind. In some cases however, direct reuse is difficult at best and impossible at worst. The most obvious example for this problem is the transformation of PV assertions to a lower abstraction level if timing is to be introduced. If it is not possible to make some general statements about the timing (for instance "all time delays within the assertion have to be exactly ten time steps"), then all timing information introduced is probably highly specific to the model in use and will not fit another, similar model. This part of the problem could be avoided by extensive use of the parameterization option of rules, but as soon as several assertions with slightly different structures are to be transformed with the same description, even this approach will not work anymore.

5.4.4 Performance Issues

Another problem that is completely unrelated to the capabilities of assertion refinement concerns simulation performance. Due to the extensive computation effort for tentative events, simulations might take quite long if tentative events are used extensively, especially, if the corresponding trigger sequences are very long.

In order to keep this effect manageable, every verification engineer should try to avoid checking for the start of long transactions whenever possible. If this is not possible, the use of the `$confirm` command can help speed up the simulation by shortening the livetime of wrong decisions for tentative events.

5.4.5 Further Issues

Additional problems for assertion refinement stem from the fact that very different design architectures probably show many differences concerning the temporal and logical interactions of different actions, while the overall behavior is still identical.

An example for this is the implementation of a pipelined execution within an RTL CPU, whereas the corresponding TLM does not use any pipeline. For simple assertions, this might not lead to any problems, since both models are still supposed to provide the same results. More complex assertions that take the pipeline into account might not be transformable with a preserved structure. In some cases this requirement could be dropped as seen in the previous sections - for instance an assertion could be changed from Overlap mode to Pipe mode while adapting the delay steps in some delay operators - but for many cases even this is not sufficient.

5.4.6 Summary

While the assertion refinement approach presented here cannot take into account all possible scenarios for the configuration of designs and / or assertions, it is capable of handling all standard situations. Timing transformation between the various levels can be done very easily, similarly, it is possible to change between signal based interfaces and transaction based interfaces.

Some of the unresolved challenges faced are not based on lacking capabilities of the transformation engine, but instead stem from the inherent properties of the abstraction levels involved.

6 Formal Semantics

This chapter describes the formal semantics of UAL+ while considering all necessary modifications presented in the last chapter.

While the actual changes to UAL+ seem to be only minimal - mere addition of some useful features - these changes require fundamental changes in the underlying execution engine. As a result, the formal model presented here is still similar in structure to the original one but almost all elements show major changes. This is primarily a consequence of the introduction of the tentative matching concept.

First a formal trace representation, a formal specification of data that is checked by UAL+ assertions is defined. Afterwards, the UAL+ language itself is mapped to an High-Level Colored Petri Net (HLCPN). After explaining the representation of various language elements by HLCPN components, a description of how to assemble these components in order to represent a complete assertion is given.

6.1 Trace Semantics

Due to the similarities in the formal basics of this work and the dissertation of Volkan Esen [51], the definitions presented in this section have been aligned with the definitions presented there.

This section presents a trace definition that is capable of holding the necessary data for the UAL+ assertion evaluation. Afterwards, it is explained why neither approaches like LTL nor finite automatons are sufficient for checking UAL+ assertions. Instead, a HLCPN is introduced as formal representation on top of the trace.

6.1.1 Classical Trace

While traces are used in many contexts and for many different purposes, there is no uniform definition of what a trace has to show and what not. Different definitions and explanations can be found in [9], [52], [53], and [23] for instance.

The common point of all these definitions and descriptions of a trace can be summarized as follows:

A trace is an array which stores a sequence of values within a design in ascending temporal order beginning from an initial value. Depending on the model this initial value can be either arbitrary or fixed.

The description however does not include any information on whether the trace contains all state transitions that happen during the simulation, or if the trace is created by sampling the design state while ignoring everything that happens in between two sampling events. In the latter case, the sampling might both reduce and increase the numbers of elements within the trace when compared to the asynchronous version:

- The number gets reduced if the sampling causes many state transitions to be left out (because they happen in between two sampling events).
- On the other hand, if the design state is sampled without the occurrence of state transitions, the trace might include several elements with the exact same values.

In RTL designs, a trace may be obtained by using a dedicated clock signal for this sampling process. Very often combinational behavior, such as asynchronous output reaction to changes on inputs, is not caught in this case in order to reduce the number of trace entries. The other case - sampling introducing several identical trace elements - can still happen, but is quite rare with most designs.

It has to be noted that some RTL traces include the clock signal while others do not. If the clock is included then in most cases the trace includes about double the number of entries than required (since most synchronous designs use only one kind of clock edge). If the clock is not part of the trace on the other hand, the correlation of trace elements to the clock edges is no longer possible (unless the trace was specifically generated by sampling the design state with every relevant clock edge).

Traces are mostly used in combination with temporal formulas such as LTL. An LTL formula specifies a Boolean condition on one or several elements of the trace, beginning with a specific element. An LTL formula holds if it evaluates to `true` starting at every trace element. LTL based assertion approaches formulate these kind of conditions.

6.1.2 UAL Trace Definition

UAL+ uses sampling for the generation of a trace as well. The sampling events are not restricted to only clock edges, but can come from a variety of sources, for instance callback events, annotated SystemC events, etc. In order to distinguish the event source, events are also part of the trace.

Due to the importance of events for the definition of the UAL+ trace, the term event has to be formally defined first. In this context it is important to distinguish between an event object (a design element just like signals and transactions), and event occurrences. The use of the term *event* means the latter.

Definition 7 *An event object $e \in \boldsymbol{V_e}$ is a two-state valued object. The value set $\boldsymbol{V_e}$ is defined as follows:*

$$\boldsymbol{V_e} := \{Fire, Idle\} \tag{6.1}$$

Event objects can be inputs, outputs, and internal objects of a design.

An event occurrence is an infinitely short impulse which indicates a single emission of the corresponding event object. During an event occurrence, the value of an event object is Fire. *As long as an event object does not occur, its value is* Idle. *The occurrence of an event does not consume time and disappears immediately. Every event object can emit an arbitrary number of event occurrences during simulation.*

Note that if two adjacent trace entries show the same event object as firing then two event occurrences happened and not just one, since an event occurrence has no duration. In this, events differ from signals or other value objects.

Definition 8 *The event object tuple* E *consists of all existing event objects observed by assertions. The element t_x is a special assertion timer event object which is part of the assertion engine.*

$$\begin{aligned} E := (e_1, \ldots, e_i, t_x) \quad & \textit{with } e_1, \ldots, e_i \in \boldsymbol{V_e} := \textit{ design event objects,} \\ & i := \textit{ number of event objects in a design and} \\ & t_x \in \boldsymbol{V_e} := \textit{ assertion timer event object} \end{aligned} \tag{6.2}$$

The value set $\boldsymbol{V_E}$ of the event object tuple E *is the product of the value sets of all event objects in* E.

$$\boldsymbol{V_E} := \boldsymbol{V_e}^{(i+1)} \tag{6.3}$$

Occurrences of events are always ordered but the order may be one of a set of non-deterministically given choices. Any two events are defined to never occur concurrently. Hence, occurrences of event objects are disjunctive and only one event object in tuple E *may have the value* Fire *at a time.*

The timer event object t_x is used within the assertion engine. A UAL timer operator is capable of scheduling the firing of this event object at a specific point in time.

In order to represent variable and signal values as well as transaction arguments and return values in a trace, it is necessary to define value objects.

Definition 9 *A data object is an object which stores data values. Such objects can be function arguments and return values, variables, and signals of different data types* $\boldsymbol{V_{d_i}}$ *and represent inputs, outputs, or internal objects of a design.*

Definition 10 *The data object tuple* D *consists of all existing data objects.*

$$D := (d_1 \in \boldsymbol{V_{d_1}}, \ldots, d_i \in \boldsymbol{V_{d_i}}) \quad \textit{with } i := \textit{number of data objects in a design} \tag{6.4}$$

The value set $\boldsymbol{V_D}$ *of the data object tuple* D *is the product of the value sets of all data objects in* D*:*

$$\boldsymbol{V_D} := \quad \boldsymbol{V_{d_1}} \times \boldsymbol{V_{d_2}} \times \ldots \times \boldsymbol{V_{d_i}} \tag{6.5}$$

Definitions 7 and 9 correspond to the UAL+ port kinds *event* and *state*. The port kinds *transaction* and *signal* map to transaction objects and signal objects respectively. These objects can be considered as compounds of the objects defined in Definitions 7 and 9:

Definition 11 *A transaction object represents a transaction including its associated events and values. Every transaction object includes two distinct event objects in* E *representing its start and end. In addition, every transaction object includes data objects in* D *for each transaction parameter and - if present - its return value.*

Definition 12 *A signal object represents any data object in a design which is capable of emitting value-change events. Every signal object hence, consists of a data object in* D *and a corresponding value-change event object in* E*.*

Definition 13 *The simulation time* $\boldsymbol{T}$ *is represented by a natural number.*

$$T = \mathbb{N}_0 \tag{6.6}$$

The current state $t_s \in \boldsymbol{T}$ *shall be observable at the occurrence of any event of the event object tuple* E*.*

Based on these definitions the UAL+ trace τ can be defined.

Definition 14 *The alphabet* Σ *is defined as follows:*

$$\Sigma := \boldsymbol{V^E} \times \boldsymbol{V^D} \times \boldsymbol{T} \tag{6.7}$$

The UAL+ trace τ *reveals the succession of symbols* $s \in \Sigma$*:*

$$\tau := \langle s^1, s^2, \ldots \rangle \qquad s^1, s^2, \ldots \in \Sigma \tag{6.8}$$

A symbol consists of one valuation of the event object tuple E*, the data object tuple* D*, and a time stamp. The trace contains one symbol per occurrence of any event in* E*.*

The superscript is the index value of the trace. It denotes the global count of event occurrences in E*.*

$$\begin{array}{lcll} s^j & := & (E^j, D^j, {t_s}^j) & \\ E^j & := & ({e_1}^j, {e_2}^j, \ldots, {e_n}^j) & \textit{with } e_{1\ldots n}\textit{: event objects in } E \textit{ and} \\ & & & {e_{1\ldots n}}^j : \textit{ current value of event objects at index } j \\ & & & n \in \mathbb{N} : \textit{number of all event objects in } E \\ D^j & := & ({d_1}^j, {d_2}^j, \ldots, {d_m}^j) & \textit{with } d_{1\ldots n}\textit{: data objects in } D \textit{ and} \\ & & & {d_{1\ldots n}}^j : \textit{ current value of data objects at index } j \\ & & & m \in \mathbb{N} : \textit{number of all data objects in } D \\ {t_s}^j \in \boldsymbol{T} & & & \textit{with } {t_s}^j : \textit{current time at index } j \end{array} \tag{6.9}$$

The current value of E^j *yields which event object has the value* Fire*. The according event occurrence marked by* Fire *is the occurrence at which the values in* D *and the value in* T *are sampled leading to the creation of symbol* s^j *in* τ*.*

Definition 15 *The UAL+ trace* τ *can be considered as a compound of three sub-traces formed by the tuple items of* s *over the index* j*:*

1. ϵ *sub-trace:* $\epsilon := \langle E^1, E^2, \ldots \rangle$ *with* $\epsilon(j) \equiv E^j$
2. ω *sub-trace:* $\omega := \langle D^1, D^2, \ldots \rangle$ *with* $\omega(j) \equiv D^j$
3. χ *sub-trace:* $\chi := \langle {t_s}^1, {t_s}^2, \ldots \rangle$ *with* $\chi(j) \equiv {t_s}^j$

Combined together, the trace holds information about the order of event occurrences as well as simulation time stamps and the sampled design state at these event occurrences.

6.1.3 Benefits and Problems of LTL for Trace Evaluation

Using LTL formulas in combination with traces allows the specification and verification of temporal properties beginning from a certain state. In this case, the difficulties introduced by the tentative matching concept disappear, since all information about a simulation run is available and thus, the final decision for every tentative event can be done via lookup-checks in the trace.

This greatly reduces computation effort and - as a result - performance impact, since it is not necessary to double up all evaluation threads as soon as a tentative event is encountered. On the other hand, all information of the trace has to be stored. In bigger designs the required storage space can easily exceed sensible amounts.

Using LTL in combination with UAL+ has several additional restrictions.

First of all, every assertion evaluation - and even every delay operator within an assertion evaluation - can specify its own sampling events. In other words, every delay operator is sensitive to a specific event expression and ignores all remaining events. As a result, every delay operator works on a specific reduced version of the whole trace that filters out all irrelevant information. When an assertion evaluation passes through the various delay operators, the decision about required information changes. In order to keep track of this behavior, it would be necessary to provide means of dynamically reducing the complete trace every assertion. While it is possible to do this with LTL, it is quite complicated: A trace entry has to be created for every potentially triggering event. Auxiliary steps have to be introduced to step over trace entries not of interest. Manual counting of potentially occurring events is required.

Another problem lies in the various different evaluation modes of UAL+. Some of these modes allow interaction between different evaluation threads, as for instance the *Restart* or *Pipe* modes. The first mode cancels running evaluation threads of the consequent sequence of an implication property when another match of the antecedent sequence is detected. The second mode allows the parallel evaluation of threads, but places restrictions concerning legal ending time and resource consumption. LTL formulas on the other hand only provide access to the future values of the current evaluation thread and do not allow references to other threads, regardless of whether it is older or newer ones. Thus, expressing pipelined or retransmission behavior using LTL is impossible. For the same reason, a mapping of UAL+ to FSMs as a whole is not possible.

6.2 Petri Net Semantics

As described in the section above, the use of FSMs as the only formal representation of the complete set of UAL+ functionality is impossible. This section describes an alternative model for formally describing the behavior of the language. An HLCPN is introduced and it is shown how the various UAL+ elements are mapped to this structure.

Every assertion is mapped to a specific petri net representation where the tokens represent the various evaluation threads. Due to the necessity of distinguishing between different evaluation threads, the petri net has to be colored, while a high level petri net was chosen, since every token is required to hold information about the current evaluation thread. As a result, every token contains an internal structure.

The HLCPN uses several levels of hierarchy, since this allows the easy assembly of assertions from modular components.

Using petri nets enables a runtime evaluation of assertions on the trace items which means that no knowledge about future events or even the complete simulation trace is required.

6.2.1 Definitions of Global Functions

Throughout the description of the HLCPN, several functions are used that either refer to elements of the trace or to objects or parameters within the HLCPN components:

- The function *C_IDX* returns the current trace index (which equals the global count of design events and timer events at the current point of time and is thus also called event index).
- The function *C_EV* returns the current event object with value *Fire* stored in ϵ(C_IDX)
- The function *C_TIME* returns the current time value χ(C_IDX)
- The implication operator includes a counter object that is used to determine ID values for consequent evaluation threads. The current value of the object can be accessed by *Cons_Cnt*.
- The evaluation mode of antecedent sequence, consequent sequence, and property can be accessed by *AMode*, *CMode*, and *PMode*, respectively.

6.2.2 Petri Net Definition

An HLCPN is defined as a tuple:

$$\text{HLCPN} := (P, TR, A, M_0, C) \tag{6.10}$$

P	:=	Finite set of places
TR	:=	Finite set of transitions
A	:=	Finite set of arcs $A \subseteq ((P \times TR) \cup (TR \times P))$
M_0	:=	Initial marking of the net
C	:=	Set of colors

6.2.3 Token Structure

Tokens in this HLCPN represent assertion evaluation attempts, while the different elements of the HLCPN represent the various parts of the assertion itself. Several of these assertion parts provide information to the evaluation attempt that is necessary for determining the final result of the evaluation. Thus, this information is to be stored within the token itself.

The token is defined as follows:

Definition 16 (Token) *A colored token - with the exception of black tokens - represents one evaluation attempt (also referred to as evaluation thread) of a given assertion specification. Every token* TK *stores information, part of which comprises the structured color type C_t while the rest comprises the structured information type I_t.*

$$C_t := (ATID, ASTID, CTID, CSTID) \tag{6.11}$$

$$I_t := (ASTS, CSTS, S, IDX, TS, ACC\_LST, AMI, CMI, ACA\_LST, CCA\_LST, TE\_LST) \tag{6.12}$$

The various structure items are defined as follows:

$ATID \in \mathbb{N}_0$	:=	*AntecedentThreadID as unique identifier of one antecedent evaluation attempt; unused in single sequence properties*
$ASTID \in \mathbb{N}_0$	:=	*AntecedentSubThreadID as unique identifier of one alternative of one antecedent evaluation attempt; unused in single sequence properties*
$CTID \in \mathbb{N}_0$	:=	*ConsequentThreadID as unique identifier of one consequent evaluation attempt; also used for single sequence properties*
$CSTID \in \mathbb{N}_0$	:=	*ConsequentSubThreadID as unique identifier of one alternative of one consequent evaluation attempt; also used for single sequence properties*
$ASTS \in \mathbb{N}_0$	:=	*Available Subthread ID space in antecedent; unused in single sequence properties*
$CSTS \in \mathbb{N}_0$	:=	*Available Subthread ID space in consequent; also used for single sequence properties*
$S \in R$	:=	*Current state of the evaluation attempt, with* $R := \{Match, NotMatch, VacuousMatch, Cancel\}$
$IDX \in \mathbb{N}_0$	:=	*Trace index at the arrival of the token in a place*
$TS \in \mathbb{N}_0$	:=	*Time stamp; used for evaluating timeout conditions as required by TIMER and AND operators*
ACC_LST	:=	*List of accumulator data elements* $acc_i \in \mathbb{N}_0$ *with* $i \in \mathbb{N}_0$ *being the unique ID of a given ACCUMULATOR operator while* acc_i *holds the corresponding accumulation counter*
$AMI \in \mathbb{N}_0$	:=	*Trace index at the match of the antecedent; unused in single sequence properties*
$CMI \in \mathbb{N}_0$	:=	*Trace index at the match of the consequent; also used in single sequence properties*
ACA_LST	:=	*List of antecedent consumption attempts; each attempt is represented by a tupel (Index,Bool_ID), with Index, Bool_ID* $\in \mathbb{N}_0$ *where Index specifies the trace index at the occurrence of the consumption attempt while Bool_ID identifies the Boolean expression to be consumed; unused in single sequence properties*
CCA_LST	:=	*List of consequent consumption attempts; each attempt is represented by a tupel (Index,Bool_ID), with Index, Bool_ID* $\in \mathbb{N}_0$ *where Index specifies the trace index at the occurrence of the consumption attempt while Bool_ID identifies the Boolean expression to be consumed; also used for single sequence properties*
TE_LST	:=	*List of tentative event occurrences; each occurrence is represented by a tupel (Index,Event_ID,Decision), with Index, Event_ID* $\in \mathbb{N}_0$*, Decision* $\in \mathbb{B}$ *where Index specifies the trace index at the moment of the tentative event occurrence, Event_ID identifies the occurred tentative event, and Decision marks whether the token assumes the event really occurred or not*

Tokens are distinguished by their color $c \in C$. A specific color corresponds to a valuation of the structured color type C_t. Since, the value set of ATID*,* ASTID*,* CTID*, and* CSTID *is the set of natural numbers, the set of possible colors* C *is infinite:*

$$C := {\mathbb{N}_0}^4 \tag{6.13}$$

Definition 17 (Structure Item Reference) *References to a structure item in* TK *are written using a "." operator. Since all structure items of both the color type C_t and the information type I_t have a unique name a structure item is referenced directly. Referring to item* ATID *for instance is written in the form* TK.ATID*.*

In some parts of the assertion evaluation however, another kind of token is required. This token is used for the implementation of semaphore-like mechanisms, for interacting with various other tokens, and so on. On the other hand, every assertion evaluation needs a special iniator token which is used for starting the first evaluation thread. Thus, two token valuations with a special meaning are introduced:

Definition 18 (Black Token) *Black tokens do not carry any information. A black token is identified by its valuation:*

$$TK^{Black} = ((0,0,0,0),(0,0,Match,0,0,\emptyset,0,0,\emptyset,\emptyset,\emptyset)) \tag{6.14}$$

Definition 19 (Initial Token) *The initial token is only needed for starting the assertion evaluation. Otherwise it is treated as any other colored token. The valuation of the initial token is defined as follows:*

$$TK^0 = ((1,1,0,0),(\text{ASTS},\text{CSTS},Match,0,0,\emptyset,0,0,\emptyset,\emptyset,\emptyset)) \tag{6.15}$$

The values for ASTS *and* CSTS *depend on the assertion structure. They can be calculated as follows:*

$$ASTS = \prod_{i=1}^{M}(max\_delay_i - min\_delay_i + 1) \tag{6.16}$$

M equals the number of range-delay operators within the antecedent sequence, while max_delay_i *and* min_delay_i *specify the maximum and minimum number of delay steps of the* i^{th} *delay operator.*

$$CSTS = \prod_{j=1}^{N}(max\_delay_j - min\_delay_j + 1) \tag{6.17}$$

N equals the number of range-delay operators within the consequent sequence, while max_delay_j *and* min_delay_j *specify the maximum and minimum number of delay steps of the* j^{th} *delay operator.*

The subthread space calculation takes into account that every delay range multiplies the number of possible subthreads. Thus, the overall number equals the product of all delay ranges.

In some cases, two or more tokens have to interact during the assertion evaluation, while in other cases they must not. Thus, it is necessary to define how tokens can be distinguished from each other:

Definition 20 (Token Color Equality) *Two tokens are considered to have the same color if the valuation of their structured color type C_t is equal, regardless of possibly different valuations of the structured value type I_t.*

One exception to this rule are black tokens. A black token is defined to have the same color as a token of any color.

When referring to functions that change color of information of a token, it is often important to differentiate between the original valuation of a token and the new valuation.

Definition 21 *In combination with actions that change the color of a token, the original valuation of the token is referred to as* TK *while the valuation after the action is indicated by* TK'.

6.2.4 Places

Places are the storage elements of petri nets, since tokens may stay in a place for a while. Every place may have one or more incoming transitions as well as one or more outgoing transitions. Every incoming transition may put additional tokens in the place, independent from other incoming transitions. If there are several outgoing transitions, then every token can only use one of them. The HLCPN presented here uses two different kinds of places, normal places and hierarchical places (used to build up hierarchical nets). Both are defined as follows:

Definition 22 (Place) *A Place $p \in P$ can hold an unconstrained number of tokens of any combination of colors. A place* p *stores the current trace index in each token arriving in* p*, which means a change of the token's additional information while the color is not affected:*

$$TK'.IDX \;=\; C\_IDX \tag{6.18}$$

A place may also include an additional action to be performed uniformly on all tokens arriving in p*. This action may also change color or additional information of a*

token. If no action is specified, a NULL action is performed which does not modify a token. Whenever several tokens leave a place simultaneously - according to EBS (see Definition 2) - they are removed from the place in the same order they arrived.

Definition 23 (Hierarchical Place) *A Hierarchical Place* p *is an encapsulation of a petri sub-net. Hierarchical places allow a concise description of the overall petri net out of primitives that are closely related to grammar nodes. In order to comply with the rules of petri net connectivity, all transitions connected to a hierarchical place must be connected to a normal place inside the hierarchical place.*

Figure 6.1 shows the corresponding symbols that are used in the remainder of this chapter.

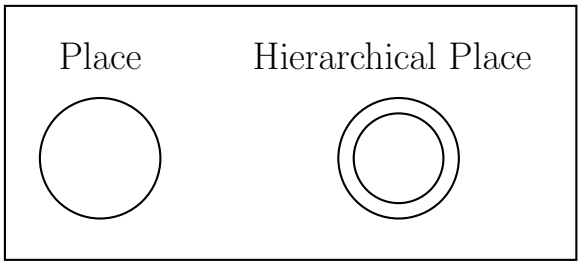

Figure 6.1: Types of Petri Net Places

Definition 24 (Marking) *The set of tokens residing in a non hierarchical place* p *is called the marking* $m(p)$*. Similarly, the marking* $m_c(p) \subseteq m(p)$ *specifies the set of tokens of a specific color* $c \in C$ *in a place* p*. The marking of a place can vary over time.*

The marking of a place $m(p)$ *can be changed to a new marking* $m'(p)$ *by two basic operations, subtraction and addition.*

$$m'(p) = m(p) \pm k\ TK_x \tag{6.19}$$

These operations add or remove k *tokens of kind* TK_x *to / from* $m(p)$*.*

Since markings of a hierarchical place as such (without considering internals) is not needed, the corresponding definition is omitted.

6.2.5 Transitions

Transitions are the elements that allow propagation of a token from one place to another. In contrast to places, tokens do not stay within a transition, they just pass through. Every transition can have one or more input places as well as one or more output places. A transition can only be activated if there is a token in each of its input places. After removing these tokens from the input places a transition will place a token into each of its output places. It is not unusual that transitions possess an additional condition for their activation or show different behavior with regard to how tokens are removed from input places or placed into output places, so petri nets frequently contain several different kinds of transitions. The HLCPN presented here uses five different kinds of transitions. Figure 6.2 shows the corresponding symbols that are used in the remainder of this chapter.

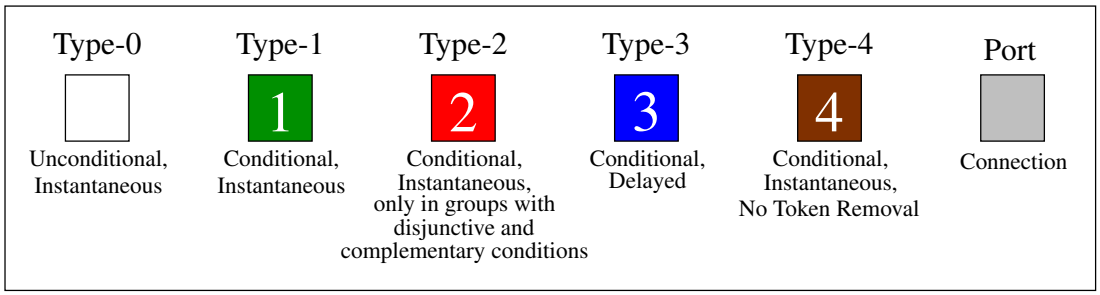

Figure 6.2: Types of Petri Net Transitions

The general behavior of all transition types is defined as follows:

Definition 25 (Enabling and Firing of Transitions) *Every transition* $tr_i \in \mathrm{TR}$ *has an enabling condition* tr_i^e *which describes a precondition for the firing of the transition. Every enabling condition includes a requirement concerning the markings of all input places of the transition. The set of all input places of a transition* tr_i *is described by* ${}^\bullet tr_i$*, the set of all output places by* $tr_i{}^\bullet$*. While the firing of a transition is defined to be atomic, it can be split into three logical phases:*

- *Removing Phase* tr_i^-*: Removing tokens of the same color* c *from all input places* $p \in {}^\bullet tr_i$
- *Action Phase* tr_i^a*: Absorption or transformation of the removed tokens; if* ${}^\bullet tr_i = \emptyset$ *black tokens are created instead*
- *Adding Phase* tr_i^+*: Adding the modified or created tokens to all output places* $\in tr_i{}^\bullet$

If one or more (but not all) of the tokens removed in the Removing Phase are black tokens, both color and additional information of the resulting token are determined by the non-black input tokens.

All five transition types used in this HLCPN differ only in their corresponding enabling conditions and Removing Phases. Action Phase and Adding Phase have the same behavior. In the following, first the enabling conditions for the various transition types are defined, followed by the definitions of the various Removing Phases, and finally by the definitions of the shared Action Phase and Adding Phase behavior.

Port transitions are just placeholders for transitions on the next higher level of hierarchy connected to the hierarchical sub-net. A port transition can represent any other transition type and thus, there is no specific behavior associated with it.

Type-0 transitions are the most common transitions within the HLCPN. They do not contain an additional Boolean condition.

Definition 26 (Type-0 Transitions (Enabling)) *A Type-0 transition tr_i is enabled for a color $c \in C$ if all input places hold at least one token of this color:*

$$\begin{aligned} &tr_i^e : M \mapsto \mathbb{B} \\ &\exists c \in C, \forall p \in {}^\bullet tr_i : |m_c(p)| \geq 1 \end{aligned} \tag{6.20}$$

Type-1 transitions introduce an additional Boolean condition for enabling. Type-2 transitions are identical to Type-1 transitions in all execution related regards, but they can only occur in groups. The corresponding enabling conditions within a group are disjunctive and complementary. Hence, exactly one transition of a group fires immediately. Type-2 transitions are introduced for clarity reasons - due to the nature of their conditions, Type-2 transitions never cause a delay of the execution. Type-4 transitions also use an identical Boolean enabling condition.

Definition 27 (Type-1 / Type-2 / Type-4 Transitions (Enabling)) *A Type-1, Type-2, or Type-4 transition tr_i is enabled for a color $c \in C$ if all input places hold at least one token of this color and if an additional Boolean condition* G *evaluates to true:*

$$\begin{aligned} &tr_i^e : M \mapsto \mathbb{B} \\ &\exists c \in C, \forall p \in {}^\bullet tr_i : (|m_c(p)| \geq 1) \wedge (G = true) \end{aligned} \tag{6.21}$$

Type-3 transitions combine the Boolean condition introduced with Type-1 transitions with an additional delay condition. The enabling condition checks that at least one additional event has occurred after the token arrived at the corresponding input place.

Definition 28 (Type-3 Transitions) *A Type-3 transition tr_i is enabled for a color $c \in C$ if all input places hold at least one token of this color, if the current trace index is bigger than the trace index stored in the token (expressed by the additional Boolean term* D*), and if an additional Boolean condition* G *evaluates to true:*

$$\begin{aligned} &tr_i^e : M \mapsto \mathbb{B} \\ &\exists c \in C, \forall p \in {}^\bullet tr_i : (|m_c(p)| \geq 1) \wedge (D = true) \wedge (G = true) \end{aligned} \tag{6.22}$$

with

$$D \equiv (TK^c.IDX < C\_IDX \text{ for at least one token of color } c \text{ in all } p \in {}^\bullet tr_i) \tag{6.23}$$

Most transition types remove one token from all input places.

Definition 29 (Type-0 / Type-1 / Type-2 / Type-3 Transitions (Removing Phase)) *The Removing Phase for Type-0, Type-1, Type-2, and Type-3 transitions extracts one token TK^c from all input places for color c:*

$$\begin{aligned} &tr_i^- : M \mapsto M,\ m_c \mapsto m'_c \\ &\forall p \in P, c \in C : m'_c(p) = \begin{cases} m_c(p) - 1\,\mathrm{TK}^c & \text{if } p \in {}^\bullet tr_i \\ m_c(p) & \text{else} \end{cases} \end{aligned} \tag{6.24}$$

Type-4 transitions just copy the tokens from the input places instead of removing them.

Definition 30 (Type-4 Transitions (Removing Phase)) *The Removing Phase for Type-4 transitions copies one token TK^c from all input places for color c:*

$$\begin{aligned} &tr_i^- : M \mapsto M,\ m_c \mapsto m'_c \\ &\forall p \in P, c \in C : m'_c(p) = m_c(p) \end{aligned} \tag{6.25}$$

As discussed above, Action Phase and Adding Phase show the same behavior for all transition types:

Definition 31 (All transitions (Action Phase)) *The Action Phase uses the following rules to determine the additional information of the output token* TK_+^c *depending on the information of all input tokens* $\mathrm{TK}_{1\ldots n}^c$*:*

$$
\begin{array}{lcl}
TK_+^c.ASTS & = & \max(TK_{1\ldots n}^c.ASTS) \\
TK_+^c.CSTS & = & \max(TK_{1\ldots n}^c.CSTS) \\
TK_+^c.S & = & \begin{cases}
Cancel & if & \exists TK_i^c : TK^c.S = Cancel \\
VacuousMatch & if & (\nexists TK_i^c : TK^c.S = Cancel) \wedge \\
 & & (\exists TK_i^c : TK^c.S = VacuousMatch) \\
NotMatch & if & (\nexists TK_i^c : TK^c.S = Cancel) \wedge \\
 & & (\nexists TK_i^c : TK^c.S = VacuousMatch) \wedge \\
 & & (\exists TK_i^c : TK^c.S = NotMatch) \\
Match & else &
\end{cases} \\
TK_+^c.IDX & = & \max(TK_{1\ldots n}^c.IDX) \\
TK_+^c.TS & = & \max(TK_{1\ldots n}^c.TS) \\
TK_+^c.ACC\_LST[i] & = & \max(TK_{1\ldots n}^c.ACC\_LST[i]) \qquad \forall i \in \mathbb{N}_0 \\
TK_+^c.AMI & = & \max(TK_{1\ldots n}^c.AMI) \\
TK_+^c.CMI & = & \max(TK_{1\ldots n}^c.CMI) \\
TK_+^c.ACA\_LST & = & \bigcup_{i=1}^n TK_i^c.ACA\_LST \\
TK_+^c.CCA\_LST & = & \bigcup_{i=1}^n TK_i^c.CCA\_LST \\
TK_+^c.TE\_LST & = & \bigcup_{i=1}^n TK_i^c.TE\_LST
\end{array}
\tag{6.26}
$$

Whenever the additional information of two or more tokens is merged as indicated above, all integer values are either identical or will not be used anymore. Thus, the maximum function could also be replaced by any other mathematical function that keeps the original value if all incoming tokens share the same value[1].

The action phase may also include an additional operation which is to be uniformly performed on the tokens removed in the previous phase. This action may also change color or additional information of the tokens. If no action is specified, a NULL action without any side effects is performed.

Definition 32 (All transitions (Adding Phase)) *The Adding Phase then sends one token* TK_+^c *to all output places.*

$$
\begin{array}{l}
tr_i^+ : M \mapsto M,\ m_c' \mapsto m_c'' \\
\forall p \in P, c \in C : m_c''(p) = \begin{cases} m_c'(p) + 1\,\mathrm{TK}_+^c & \text{if } p \in tr_i{}^\bullet \\ m_c'(p) & \text{else} \end{cases}
\end{array}
\tag{6.27}
$$

[1]The implementation just takes one of these values in order to reduce computation effort.

6.2.6 Lists

The match filter component (see Section 6.2.11) and the AND operator (see Section 6.2.8) need to store auxiliary information which is independent from the data in the tokens. While it is possible to store this information in a single token of an appropriate type and provide arcs from the storage place of this token to all transitions where it is needed and back again, this would make the graphical representation of the petri net overly complicated. Due to this reason, all of this additional information is stored in two lists in the match filter and one list in the AND operator which are visible to all transitions and places of the corresponding component.

Definition 33 (Tentative List (Match Filter)) *The tentative list stores occurrences and evaluations for all tentative events in the simulation. It is used to determine whether a specific token using tentative events is to be discarded in the end or not. Each list element is of the following structure type TL:*

$$TL := (Index, Event\_ID, Decision) \tag{6.28}$$

with the structure items

$$\begin{array}{lcl} Index \in \mathbb{N}_0 & := & \textit{Trace Index of the tentative event occurrence} \\ Event\_ID \in \mathbb{N}_0 & := & \textit{Identifies the occurred tentative event} \\ Decision \in \mathbb{B}_+ & := & \textit{Marks if the tentative event was a real event or not, or} \\ & & \textit{if the status is still undecided, with} \\ & & \mathbb{B}_+ := \{TRUE, FALSE, UNKNOWN\} \end{array}$$

Definition 34 (Match List (Match Filter)) *The match list keeps track of the various assertion evaluation attempts and stores their final results. The list entries can be used to decide whether all subthreads of an evaluation attempt have already been completed, to see which threads have already matched and when, to keep track of the Boolean propositions being consumed by matching threads, etc. Each list element is of the following structure type ML:*

$$ML := (ATID, ASTID, CTID, CSTID, S, AMI, CMI, ACA\_LST, CCA\_LST) \tag{6.29}$$

with the structure items

$ATID \in \mathbb{N}_0$	:=	*AntecedentThreadID as unique identifier of one antecedent evaluation attempt; unused in single sequence properties*
$ASTID \in \mathbb{N}_0$	:=	*AntecedentSubThreadID as unique identifier of one alternative of one antecedent evaluation attempt; unused in single sequence properties*
$CTID \in \mathbb{N}_0$	:=	*ConsequentThreadID as unique identifier of one consequent evaluation attempt; also used for single sequence properties*
$CSTID \in \mathbb{N}_0$	:=	*ConsequentSubThreadID as unique identifier of one alternative of one consequent evaluation attempt; also used for single sequence properties*
$S \in R$	:=	*Current state of the evaluation attempt, with* $R :=$ {*Match,NotMatch,VacuousMatch,Cancel*}
$AMI \in \mathbb{N}_0$	:=	*Trace index at the match of the antecedent; unused in single sequence properties*
$CMI \in \mathbb{N}_0$	:=	*Trace index at the match of the consequent; also used in single sequence properties*
ACA_LST	:=	*List of antecedent consumption attempts; each attempt is represented by a tupel (Index,Bool_ID), with Index, Bool_ID* $\in \mathbb{N}_0$ *where Index specifies the trace index at the occurrence of the consumption attempt while Bool_ID identifies the Boolean expression to be consumed; unused in single sequence properties*
CCA_LST	:=	*List of consequent consumption attempts; each attempt is represented by a tupel (Index,Bool_ID), with Index, Bool_ID* $\in \mathbb{N}_0$ *where Index specifies the trace index at the occurrence of the consumption attempt while Bool_ID identifies the Boolean expression to be consumed; also used for single sequence properties*

Definition 35 (AND operator list) *This list stores occurrences of completed AND connections. Based on this information it can be decided whether a new AND combination should be propagated or discarded. Each list element is of the following structure type AL:*

$$AL := \quad (ATID, ASTID, CTID, CSTID, TE\_LST) \tag{6.30}$$

with the structure items

$ATID \in \mathbb{N}_0$	:=	*AntecedentThreadID as unique identifier of one antecedent evaluation attempt; unused in single sequence properties*
$ASTID \in \mathbb{N}_0$	:=	*AntecedentSubThreadID as unique identifier of one alternative of one antecedent evaluation attempt; unused in single sequence properties*
$CTID \in \mathbb{N}_0$	:=	*ConsequentThreadID as unique identifier of one consequent evaluation attempt; also used for single sequence properties*
$CSTID \in \mathbb{N}_0$	:=	*ConsequentSubThreadID as unique identifier of one alternative of one consequent evaluation attempt; also used for single sequence properties*
TE_LST	:=	*List of tentative event occurrences; each occurrence is represented by a tupel (Index,Event_ID,Decision), with Index, Event_ID* $\in \mathbb{N}_0$*, Decision* $\in \mathbb{B}$ *where Index specifies the trace index at the moment of the tentative event occurrence, Event_ID identifies the occurred tentative event, and Decision marks whether the token assumes the event really occurred or not*

6.2.7 Mapping Concept

In the following section, the different elements of UAL+ are mapped to petri net representations. Mostly, this is done according to the UAL+ layer the corresponding element belongs to. Some components cannot be placed within one specific layer though, so they are discussed at the end.

Note that neither the verification layer nor the Boolean layer are discussed here, since the UAL+ elements of these layers would only appear as an action to be performed or a Boolean condition to be checked in a petri net.

Some general methods and functions are used to modify or check the status of an evaluation thread and to replace a colored token by a black token. The corresponding methods are listed in Table 6.1, while Table 6.2 contains the functions.

Symbol	**Definition**
SetBlack()	TK' = TK^{Black}
SetCancel()	TK'.S = Cancel
SetNotMatch()	TK'.S = NotMatch
SetVacuousMatch()	TK'.S = VacuousMatch

Table 6.1: General Methods

Symbol	Definition
IsCancel()	TK.S = Cancel
IsMatch()	TK.S = Match
IsNotMatch()	TK.S = NotMatch
IsVacuousMatch()	TK.S = VacuousMatch

Table 6.2: General Functions

6.2.8 Event Layer

Every UAL+ event operator has its own HLCPN representation. These operators can be interconnected in a flexible way in order to form event expressions. As a result, it is necessary to also provide a representation for the occurrence of a single event.

Every operator has an enable port where tokens from a higher level of hierarchy (another event operator or a delay operator) can come in and a result port where tokens can be passed back to that hierarchy level.

Several of these event operators require functions for checks of certain conditions and methods for the manipulation of tokens. The methods are listed in Table 6.3, while Table 6.4 contains the functions.

Symbol	Definition
Acc()	TK'.ACC_LST[AccID] = TK.ACC_LST[AccID] - 1
ClearTokenTime()	TK'.TS = 0
Count()	TK'.TS = TK.TS - 1
SetAccValue(X)	TK'.ACC_LST[AccID] = X
SetTentativeMark()	TK'.TE_LST = TK.TE_LST $\cup$ (C_IDX,e_x,`true`)
SetTentativeMarkLeft()	TK'.TE_LST = TK.TE_LST $\cup$ (C_IDX,e_x,`false`)
SetTentativeMarkRight()	TK'.TE_LST = TK.TE_LST $\cup$ (C_IDX,e_x,`true`)
SetTokenTime()	TK'.TS = C_TIME
SetTokenTime(X)	TK'.TS = X

Table 6.3: Event Layer Methods

The method *SetAccValue(X)* sets the list entry in the *ACC_LST* token item that is associated with the current ACCUMULATOR operator to X, while *Acc()* decrements it by one.

The method *SetTokenTime()* sets the *TS* token item either to the current simulation time (if called without parameter) or to the value of a given parameter. The method *Count()* decrements this token item, while *ClearTokenTime()* sets it to zero.

The methods *SetTentativeMark()* and *SetTentativeMarkRight()*[2] mark the token representing the assumption that the tentative event is a valid event. For this purpose, a tuple containing the current trace index, the ID of the tentative event, and the term `true` marking the decision is added to the token item *TE_LST*. Similarly, *SetTentativeMarkLeft()* marks the token representing the assumption that the tentative event is not valid. This decision is marked by using the term `false` in the tupel.

Symbol	**Definition**
AccDone()	TK.ACC_LST[AccID] = 0
CountDone()	TK.TS = 0
ListEmpty()	$\nexists q \in$ AND_LST: (q.ATID = TK.ATID) $\wedge$ (q.ASTID = TK.ASTID) $\wedge$ (q.CTID = TK.CTID) $\wedge$ (q.CSTID = TK.CSTID)
TimeOut(X)	C_TIME - TK.TS = X $\wedge$ C_EV = t_x
TokenInList()	$\exists q \in$ AND_LST: (q.ATID = TK.ATID) $\wedge$ (q.ASTID = TK.ASTID) $\wedge$ (q.CTID = TK.CTID) $\wedge$ (q.CSTID = TK.CSTID) $\wedge$ (q.TE_LST = $\bigcup_{i=1}^{n}$ TK$_i$.TE_LST)

Table 6.4: Event Layer Functions

The function *AccDone()* checks if the list entry in the *ACC_LST* token item that is associated with the current ACCUMULATOR operator has the value zero after the decrement. The function *CountDone()* does the same for the token item *TS*. The function *ListEmpty()* checks whether the list of the AND operator contains at least one entry of the same color as the current token. The function *TimeOut(X)* checks whether the difference between the current time value and the value stored in the token equals X and if the current event is the timer event. Finally, the function *TokenInList()* checks whether the token resulting from the merger of all input tokens is already stored in the list of the AND operator.

[2] Although these two methods are identical, different names were chosen to simplify the explanation of the corresponding event operators.

Single Event Occurrence

The HLCPN representation of a single event occurrence is depicted in Figure 6.3. Every instance of this representation features a parameter specifying to which event e_x it is sensitive. When a token arrives via the enable port it resides in the place *Input* until the result of the function *C_EV* equals e_x; at this time the token is transported to the place *Output* and consequently leaves via the result port.

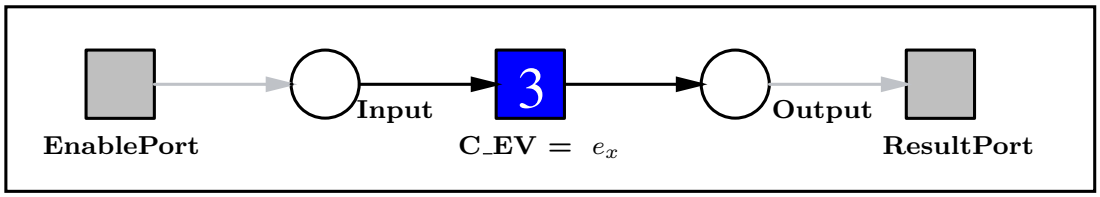

Figure 6.3: Single Event

The introduction of the tentative matching concept however has led to two variants of this HLCPN representation. Figure 6.4 depicts the version dealing with a tentative event. In contrast to the original representation the Type-3 transition duplicates the token while passing through in order to represent the two possible decisions for the event status. The token that is sent to the *Output* place assumes that the tentative event is a real event, while the token traveling back to the *Input* place assumes it is not. This information is stored within the token according to the formulas listed in Table 6.4.

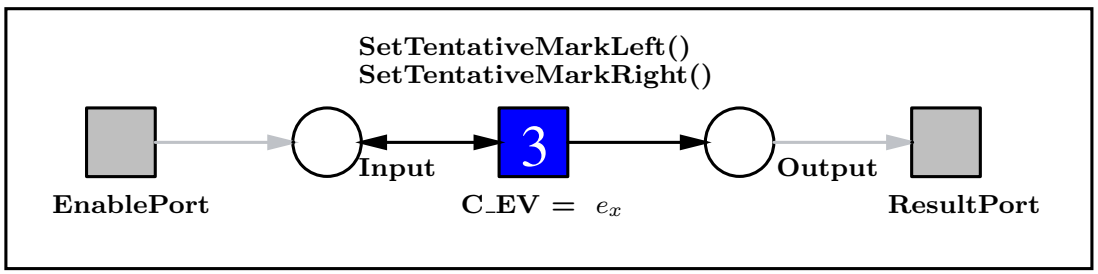

Figure 6.4: Tentative Event

Figure 6.5 shows the special case of a tentative event within the first delay operator of a sequence (in case of implication properties this only affects the antecedent sequence). The token is not duplicated, but the token is still marked as seen in Table 6.4.

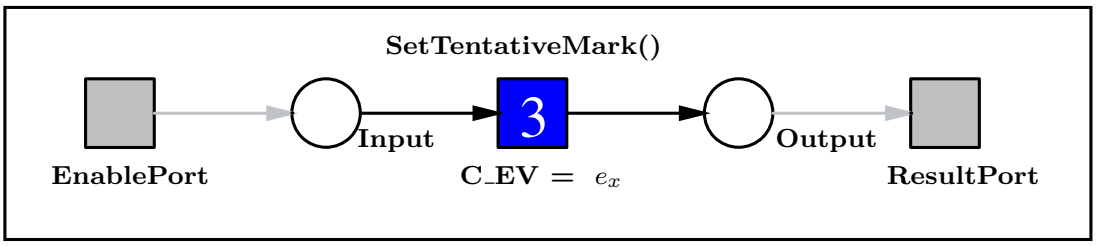

Figure 6.5: Tentative Event in First Delay Operator

TIMER Operator

The enhanced UAL+ TIMER operator has two HLCPN representations. The first one deals with the classical occurrence of a timer event, while the second one handles the count of clock events.

The first representation is depicted in Figure 6.6. Every instance of this representation features a parameter specifying after which time X the timer event has to occur. When a token arrives via the enable port it resides in the place *Input* where the current simulation time is stored within the token and a timer event t_x is scheduled X time steps later. As soon as this event is detected the token is transported to the place *Output*, where the stored time value is cleared in the token, and consequently leaves via the result port. The required functions are described in Table 6.4.

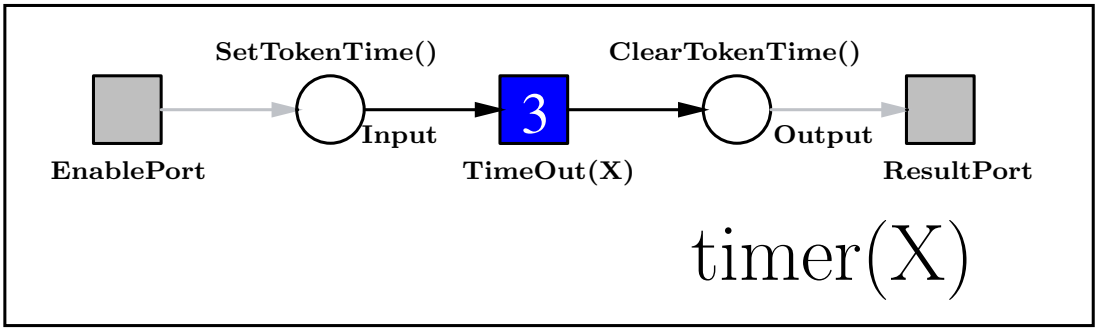

Figure 6.6: TIMER Operator

The second representation is depicted in Figure 6.7. Every instance of this representation features two parameters: One specifies which event e_x is to be counted, the second specifies the required number of occurrences. When a token arrives via the enable port, the value X is stored within the token at the *Input* place. In the *Loop* place it is checked whether the required number of occurrences has already been reached, in which case the token leaves via the result port. Otherwise, the token

reaches the *Check* place, where it awaits the occurrence of an event. Afterwards, the value stored within the token is decremented and the token returns to the *Loop* place. The required functions are described in Table 6.4.

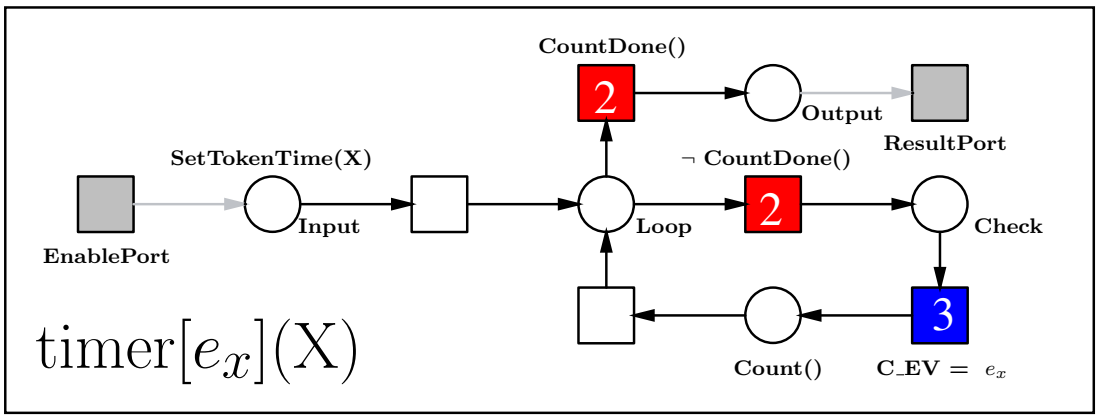

Figure 6.7: Event-based TIMER Operator

ACCUMULATOR Operator

The HLCPN representation of the ACCUMULATOR operator is depicted in Figure 6.8. As can be seen it is very similar to the event-based TIMER operator[3]. But instead of counting the occurrences of a single event, this operator is capable of counting the occurrence of a complex event expression, represented by a hierarchical place. Since it is possible that several ACCUMULATOR operators are nested within the same event expression, every operator is assigned a unique ID *AccID*. The accumulator counters are no longer stored in the TS element of the token, instead, the list ACC_LST is used, where every active accumulator gets a speparate entry. The remaining behavior is identical to that of the event-based TIMER operator. The required functions are described in Table 6.4.

CONSTRAINT Operator

The HLCPN representation of the CONSTRAINT operator is depicted in Figure 6.9. The CONSTRAINT operator checks that whenever a given event expression (again represented by a hierarchical place) matches, a certain Boolean condition has to hold. When a token arrives via the enable port, it is sent to the hierarchical place of the

[3]The reason for this similarity is that the event-based TIMER operator has been derived from the ACCUMULATOR operator and simplified.

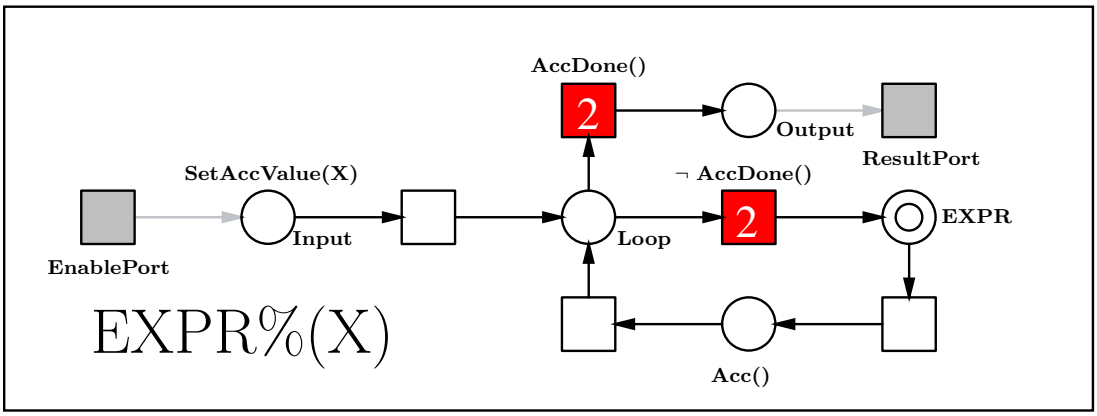

Figure 6.8: ACCUMULATOR Operator

event expression. After the expression has been processed and the token has arrived in the place *Check*, the Boolean constraint condition is evaluated. If it does not hold, the token is sent back to the hierarchical place. Otherwise, the token leaves via the result port.

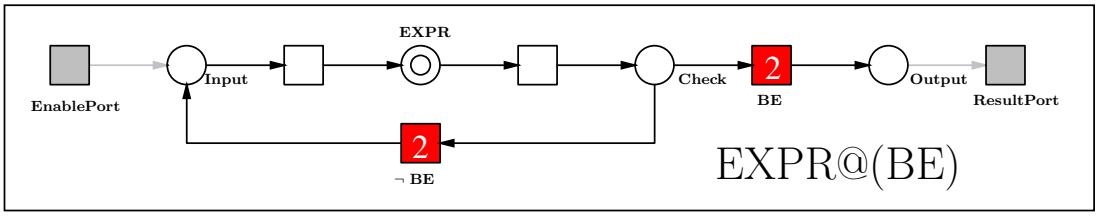

Figure 6.9: CONSTRAINT Operator

OR Operator

Figure 6.10 shows the HLCPN representation of the OR operator. The structure of this operator is very simple. Every token arriving via the enable port is just split up and sent to both alternative event expressions (again represented as hierarchical places). Both expressions can produce independent results which are then propagated out of the operator via the result port.

AND Operator

Normally, the AND operator has to check for the occurrence of two events or event expressions within the same time step and then propagate a positive result. The orig-

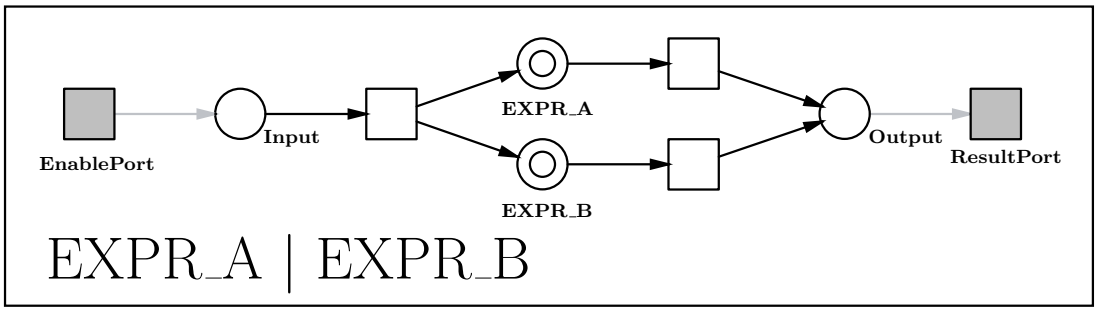

Figure 6.10: OR Operator

inal implementation of UAL+ solved this by waiting for exactly one such combination and discarding all further ones (within the same time step). Due to the introduction of the tentative event concept the AND operator has to consider additional situations. It is possible that the first transmitted combination is only tentative and gets invalid later on. Thus, all further combinations may not be discarded. In order to keep track of the information relevant for this, the AND operator implements one list introduced in Section 6.2.6. The corresponding update functions are defined as follows:

Adding a token to the List:

$$\begin{aligned} &\text{AND\_LST'} = \text{AND\_LST} \cup \\ &\{(\text{TK.ATID,TK.ASTID,TK.CTID,TK,CSTID,TK.TE\_LST})\} \end{aligned} \tag{6.31}$$

Update of List - all entries of the current token color are removed:

$$\begin{aligned} &\forall\, \text{j} \in \text{AND\_LST} \mid (\text{j.ATID} = \text{TK.ATID}) \wedge (\text{j.ASTID} = \text{TK.ASTID}) \wedge \\ &(\text{j.CTID} = \text{TK.CTID}) \wedge (\text{j.CSTID} = \text{TK.CSTID}): \\ &\text{AND\_LST'} = \text{AND\_LST} \setminus \text{j} \end{aligned} \tag{6.32}$$

Figure 6.11 shows the HLCPN representation of the AND operator. Similarly to the OR operator, every token arriving via the enable port is split up and sent to both event expressions (again represented as hierarchical places). Both expressions have to produce a result. The corresponding token resides in either place A or place B and a timer event t_x is scheduled for the next time step. If the other expression produces a result before this event is detected, it is checked whether the merger of both tokens has already been processed this time step (as indicated by a corresponding entry in the AND operator list). If not then both tokens are merged, the resulting token is added to the list of the AND operator and then propagated out of the operator via the result port. If the timer event is detected, it is checked whether the list contains elements of the same color as the token (this represents that the operator already produced a result this time step). If this is not the case the waiting token is sent

back to the corresponding hierarchical place. Otherwise, all list entries of the same color as the token are removed and the token is destroyed. The required functions are described in Table 6.4.

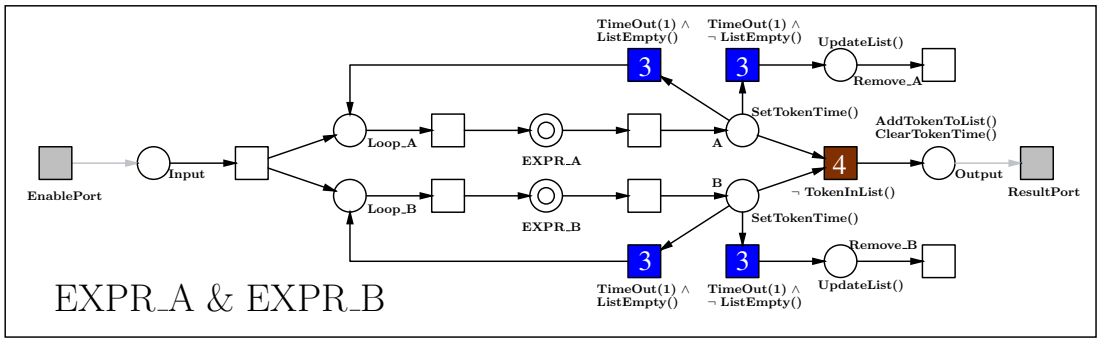

Figure 6.11: AND Operator

6.2.9 Sequence Layer

The UAL+ sequence layer includes mainly the various delay operators. Additionally, a token generator component is located on this layer.

Sequence operators also require methods for token manipulation which are listed in Table 6.5.

Symbol	**Definition**
AddCA()	TK'.CA_LST = TK.CA_LST ∪ (C_IDX,Bool_ID)
AssignSTID()	TK'$_i$.STID = TK.STID + i * (TK.STS / N) TK'.STS = TK.STS / N
IncreaseTID()	TK'.TID = TK.TID + 1

Table 6.5: Sequence Layer Methods

The method *AddCA()* stores the consumption attempt in the token. For this purpose, a tupel containing the current trace index and the ID of the Boolean expression is added to the token item *CA_LST*.

The method *AssignSTID()* is needed whenever a ranged delay operator splits the current thread into several subthreads. In order to distinguish the various subthreads all corresponding tokens have to be assigned a different ID. The corresponding subthread ID is calculated as follows:

1. Divide the available subthread space by N (the number of range steps), the result specifies the number of subthreads each token represents
2. For the i^{th} token multiply this by the number i, with $0 \leq i < N$
3. Add this offset to the current subthread ID

Additionally, the available subthread space will be decreased accordingly.

The *IncreaseTID()* method increments the ID of the current token by one.

Delay Operator

The main elements of the sequence layer are the UAL+ delay operators. Due to differences in their structure, it is not possible to include every possible configuration within one HLCPN representation. Instead, these various possibilities are reduced to three different representations: a zero-delay operator which does not include any event expressions, a single-delay operator, and a ranged delay operator. All other configurations can be built up from these three components. The following rewriting rules are used for this purpose:

Definition 36 *The minimum delay of a delay range always has to equal zero (m le n):*
`#{m:n}{...}{BE} =>`
`#m{...}{true} #{0:(n-m)}{...}{BE}`

If $m = n$ then a non-ranged delay operator is implemented.

Definition 37 *A multi-step delay consists of several single step-delays:*
`#N{...}{BE} =>`
`#1{...}{true}`$_0$ `#1{...}{true}`$_1$ ... `#1{...}{true}`$_{N-1}$`#1{...}{BE}`$_N$

Basically a `#N` operator is replace by N `#1` operators which all contain the same event expressions as the original one. The first N-1 operators contain a simple `true` as Boolean proposition, while the last one uses the same expression as the original operator.

Similarly to the event operators, each delay operator has an enable port and a result port.

Figure 6.12 depicts the HLCPN representation of the zero-delay operator. Tokens arriving via the enable port are evaluated concerning their status. If the status does not equal *Match* then they just leave via the result port. Otherwise the Boolean condition of the delay operator is checked. If it evaluates to `true` then the ID of the Boolean condition (with the exception of the trivial true proposition every Boolean proposition is assigned a unique ID) together with the current trace index is stored within the token in order to mark the consumption attempt of this thread. If it evaluates to `false` the status of the token is set to *NotMatch*. The functions used in the zero-delay operator are listed in Table 6.5.

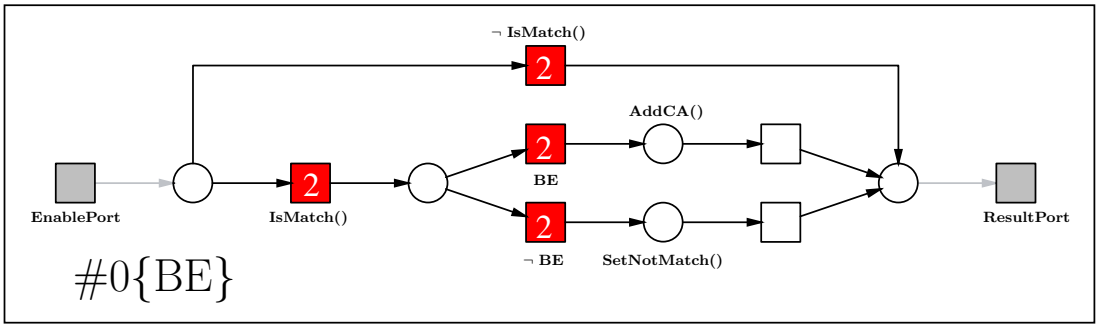

Figure 6.12: Zero-Delay Operator

Figure 6.13 depicts the HLCPN representation of the single-delay operator. In contrast to the zero-delay operator, it contains several hierarchical places representing various event expressions. UAL+ supports the specification of a positive and a negative event expression per delay operator. The positive expression propagates the evaluation thread while the negative expression cancels it. In addition UAL+ allows the specification of a reset expression per assertion directive. Due to the addition of the tentative mechanics (early cancelation of threads might lead to wrong results, depending on the result of tentative events) this reset expression has to be included in every delay operator of all sequences used within that assertion directive.

Tokens arriving via the enable port are first distributed to all three event expressions. The status of tokens from all event expressions is checked and if it does not equal *Match* they are propagated to the result port without further processing[4]. Otherwise, the status of tokens from the negative expression is set to *NotMatch* while for tokens from the positive expression the Boolean condition of the delay operator is checked. If it evaluates to `true` then the ID of the Boolean condition together with the current trace index is stored within the token in order to mark the consumption

[4]This only applies to tokens that were already set to a status other than *Match* in a previous delay operator. It is necessary for these tokens to still pass the event expressions, since otherwise tokens might overtake each other.

attempt of this thread, in the same way as in the zero-delay operator. If it evaluates to `false` the status of the token is set to *NotMatch*. The functions used in the single-delay operator are listed in Table 6.5.

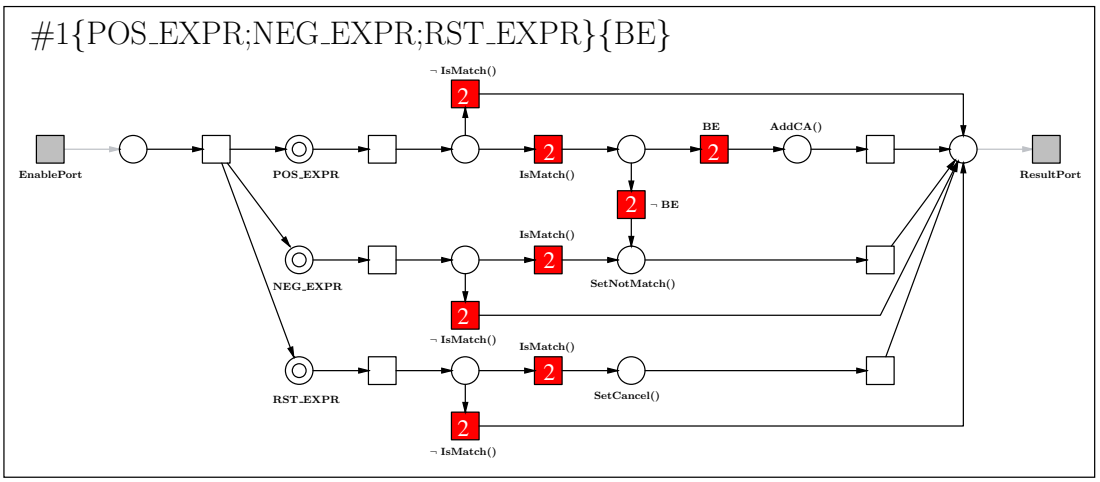

Figure 6.13: Single-Step-Delay Operator

Every event expression can consist of an arbitrary number of event operators (as detailed in the previous section). In addition, up to two of the three event expressions can also be left empty which represents the absence of positive or negative triggers or of a reset mechanism. The corresponding HLCPN component is shown in Figure 6.14. Tokens sent in there are just consumed.

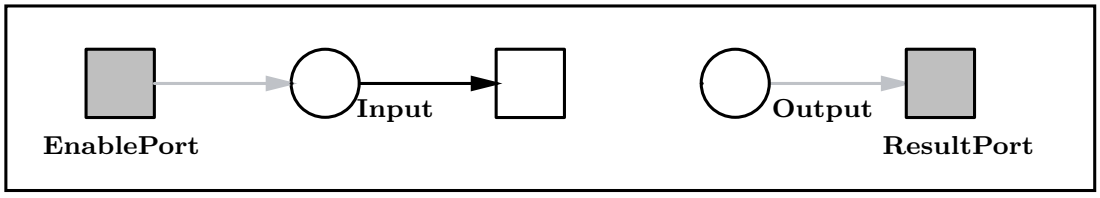

Figure 6.14: Empty Event Expression

Finally, a ranged delay operator is shown in Figure 6.15. Basically, this operator instantiates a number of delay operators according to the width of the delay range. Each of these operators represents one specific delay value of that range. These operators are either zero-delay, single-delay, or longer delay operators (the latter can be constructed by single-delay operators as shown above). The only important thing done within the range operator is the assignment of new subthread IDs to the various tokens, in order to be able to distinguish them later (this is needed in order to decide

on the correct result for the various execution modes). The corresponding functions are listed in Table 6.5.

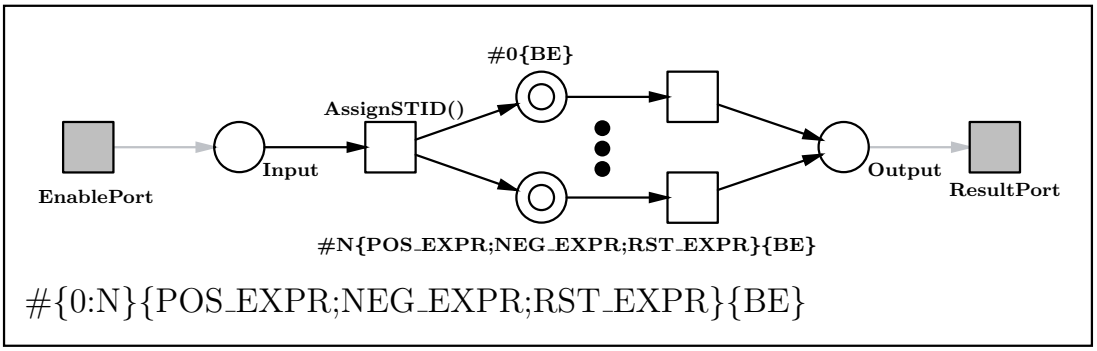

Figure 6.15: Ranged Delay Operator

Token Generator

Figure 6.16 shows the HLCPN representation of the token generator component. Its purpose is to provide evaluation threads for the given assertion represented by tokens. To that purpose, the generator should not produce more tokens than absolutely necessary.

The output of the token generator is connected to the input of the first antecedent delay operator (or to the first delay operator of the single sequence in a single sequence property).

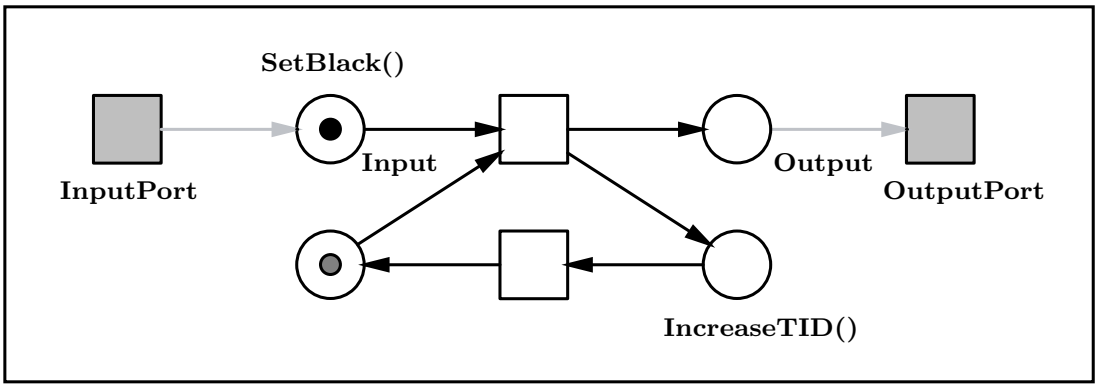

Figure 6.16: Token Generator

The token generator contains two tokens initially, a black token TK^{Black} and an initial token TK^{0} (represented by the gray colored token).

At the beginning of the simulation, the central transition of the token generator is enabled, since both input places contain a token. The transition places one colored token in the *Output* place, which is then propagated to the first delay operator. The second colored token's thread ID is increased. After passing an unconditional transition, it reaches the initial place again. Since the black token is not present anymore, the transition cannot be enabled.

The input of the token generator is connected to the output of the first delay operator. If this delay operator produces a result, a copy of the corresponding output token is sent back to the token generator and colored black; this restores the initial configuration (with the exception of the color of the second token) and thus re-enables the evaluation of the connected assertion. As a consequence, there will never be two completely parallel evaluation threads present within the first delay operator.

6.2.10 Property Layer

The UAL+ property layer is represented by an implication operator. The additional methods required by this operator for token manipulation are listed in Table 6.6.

Symbol	**Definition**
CreateNewTID()	TK'.CTID = Cons_Cnt Cons_Cnt = Cons_Cnt + 1 TK'.CSTID = 1
SetAMI()	TK'.AMI = C_IDX

Table 6.6: Property Layer Methods

The *CreateNewTID()* method is responsible for generating an ID for the consequent evaluation thread, since these are independent from the antecedent IDs. The new ID is determined by the current value of the counter *Cons_Cnt* that is incremented after the assignment. Finally, the subthread ID is set to one.

The method *SetAMI()* stores the current trace index in the *AMI* token item, in order to mark when the antecedent has matched.

Similarly to the event operators, an implication operator has an enable port and a result port.

Figure 6.17 depicts the HLCPN representation of the implication operator. Tokens arriving via the enable port are evaluated concerning their status. If the status equals

Cancel then they just leave via the result port (since the thread is already canceled no further processing is required). Tokens with the status *NotMatch* represent a vacuous success of the implication property already. Thus, the status is set to *VacuousMatch*. Finally, tokens with the status *Match* require a complete evaluation of the consequent sequence. For this purpose every one of these tokens is assigned a unique consequent ID, since consequent evaluation threads have to be evaluated independently from each other. The functions used in the implication operator are listed in Table 6.6.

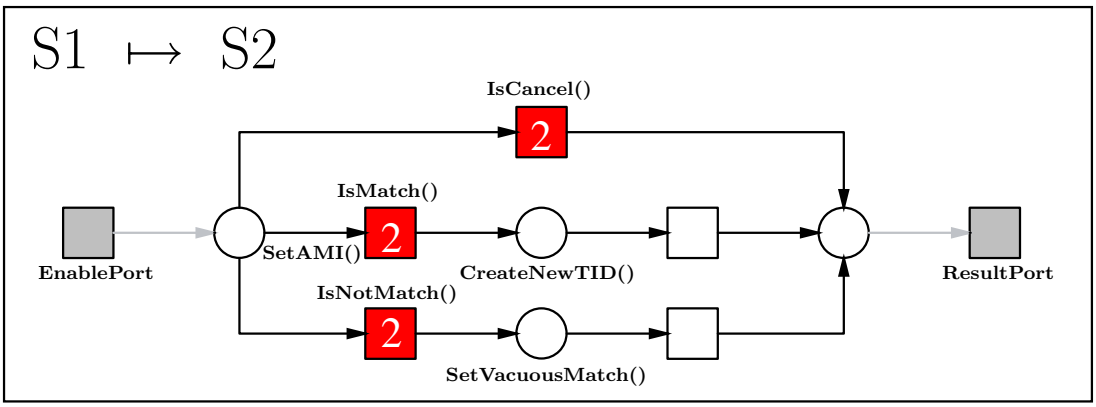

Figure 6.17: Implication Operator

Note that in case of single sequence properties no implication operator is needed.

6.2.11 Hybrid Component: Match Filter

A UAL+ key component is the so-called match filter which is responsible for computing the final assertion results based on the data aquired by the various tokens on the one hand and the different evaluation modes of the assertion on the other hand. Depending on the execution modes in use, it is possible that a token representing a successful evaluation attempt is still discarded because another thread / subthread prohibits it from being validated. In contrast to the original implementation of UAL+, where the match filter was part of the sequence layer, the inclusion of the tentative matching concept makes an immediate evaluation of the antecedent sequence impossible. Instead, this evaluation has to be delayed until all results of all involved tentative events are available, which can easily take more time then the complete evaluation of antecedent and consequent sequence including the corresponding property as well.

Thus, the match filter has to take the property results into account as well (including the property execution modes). As a result, the modified match filter does not

clearly belong to only the sequence layer. Instead it includes functionality of both the sequence and the property layer.

Instead of having one match filter at the end of both antecedent and consequent sequence, the revised match filter is to be attached to the end of the consequent sequence only. It includes the functionality for antecedent checks as well.

The match filter needs to know the maximum subthread space for both the antecedent and the consequent sequence. Similarly, information about the evaluation modes of antecedent, consequent, and property is required. This information is stored in global constants. The match filter implements two lists introduced in Section 6.2.6 in order to store additional information that has to be kept even after the evaluation of a token is completed. The corresponding update functions are defined as follows:

Update of Match List:

$$\begin{aligned}&\text{MA\_LST'} = \text{MA\_LST} \cup \{(\text{TK.ATID,TK.ASTID,TK.CTID,TK,CSTID,TK.S,}\\&\text{TK.AMI,TK.CMI,TK.ACA\_LST,TK.CCA\_LST})\}\end{aligned} \tag{6.33}$$

Update of Tentative List (Occurrence of Tentative Event) - a tentative event occurrs during simulation:

$$\text{TE\_LST'} = \text{TE\_LST} \cup \{(\text{C\_IDX,Event\_ID,UNKNOWN})\} \tag{6.34}$$

Update of Tentative List (Decision of Tentative Event) - a previously occurred tentative event can be decided:

$$\begin{aligned}&(\text{TE\_LST'} = \text{TE\_LST} \setminus \{(\text{C\_IDX,Event\_ID,UNKNOWN})\}) \cup\\&\{(\text{C\_IDX,Event\_ID,Decision})\}\end{aligned} \tag{6.35}$$

Table 6.7 details the remaining methods used in the match filter, while Tables 6.8, 6.10, 6.9, and 6.11 contain all functions.

Symbol	**Definition**
CancelOldThread()	$\forall j \in \text{MA\_LST} \mid (\text{j.AMI} \leq \text{TK.AMI} < \text{j.CMI}) \wedge$ ((j.S = Match) $\vee$ (j.S = NotMatch)): j'.S = Cancel
SetCMI()	TK'.CMI = C_IDX

Table 6.7: Match Filter - Methods

The method *CancelOldThread()* looks up all entries in the match list with a status of *Match* or *NotMatch*[5] with an antecedent match index before and a consequent match index after the antecedent match of the current token. These entries represent older threads that should be canceled by a restart, since their consequent was still in evaluation when the antecedent represented by the token matched. As a consequence, the status of these entries is changed to *Cancel*.

The method *SetCMI()* stores the current trace index in the *CMI* token item, in order to mark when the consequent has matched.

Symbol	**Definition**
ACConflict()	$\exists j \in$ MA_LST: (j.ACA_LST $\cap$ TK.ACA_LST $\neq \emptyset$) $\wedge$ ((j.S = Match) $\vee$ (j.S = NotMatch))
ALastSubthread()	$\forall i \in N \vert (0 < i \leq$ ASTS) $\wedge (i \neq$ TK.ASTID): $\exists j \in$ MA_LST: (j.ATID = TK.ATID) $\wedge$ (j.ASTID = i)
AMConflict()	$\exists j \in$ MA_LST: (j.AMI = TK.AMI) $\wedge$ ((j.S = Match) $\vee$ (j.S = NotMatch))
AThreadMatched()	$\exists j \in$ MA_LST: (j.ATID = TK.ATID) $\wedge$ ((j.S = Match) $\vee$ (j.S = NotMatch))
A_AM()	AMode = AnyMatch
A_FMP()	AMode = FirstMatchPipe

Table 6.8: Match Filter - Antecedent related Functions

The function *AMConflict()* checks whether there is a match conflict between the antecedent thread represented by the current token and an older one. For this purpose, it is checked whether there is already an entry in the match list with the same antecedent match index with status either *Match* or *NotMatch*, which represents an antecedent match.

The *ACConflict()* function checks whether there is a consumption conflict between the antecedent thread represented by the current token and an older one. For this purpose, it is checked whether the intersection of the antecedent consumption attempt list within the token and the antecedent consumption attempt list of any element within the match list representing an antecedent match is not empty.

The *ALastSubthread()* function checks whether the current token represents the last antecedent subthread for a given thread ID, which means whether there is already an entry for all other subthread IDs of the current thread ID in the match list.

[5]Both represent a match of the antecedent sequence.

The function *AThreadMatched()* checks whether the current thread has already matched, which means whether there is already an entry for the current thread ID in the match list with the status of *Match* or *NotMatch*, which represents an antecedent match.

The functions *A_AM()* and *A_FMP()* check whether the antecedent is evaluated using a specific mode.

Symbol	**Definition**
CCConflict()	$\exists j \in$ MA_LST: (j.CCA_LST $\cap$ TK.CCA_LST $\neq \emptyset$) $\wedge$ (j.S = Match)
CLastSubthread()	$\forall i \in N \mid (0 < i \leq$ CSTS) $\wedge (i \neq$ TK.CSTID): $\exists j \in$ MA_LST: (j.CTID = TK.CTID) $\wedge$ (j.CSTID = i)
CMConflict()	$\exists j \in$ MA_LST: (j.CMI = TK.CMI) $\wedge$ (j.S = Match)
CThreadMatched()	$\exists j \in$ MA_LST: (j.CTID = TK.CTID) $\wedge$ (j.S = Match)
C_AM()	CMode = AnyMatch
C_FMP()	CMode = FirstMatchPipe

Table 6.9: Match Filter - Consequent related Functions

The functions *CMConflict()*, *CCConflict()*, *CLastSubthread()*, *CThreadMatched()*, *C_AM()*, and *C_FMP()* work similarly, but check the corresponding consequent conditions.

Symbol	**Definition**
OldThread()	$\exists j \in$ MA_LST: (j.AMI $\leq$ TK.AMI $<$ j.CMI) $\wedge$ ((j.S = Match) $\vee$ (j.S = NotMatch))
P_NR()	PMode = NoRestart
P_O()	PMode = Overlap
P_P()	PMode = Pipe
P_R()	PMode = Restart
P_ROR()	PMode = ReportOnRestart

Table 6.10: Match Filter - Property related Functions

The *OldThread()* function checks whether there is an older thread in the match list that might influence the behavior of the current one. It is checked, whether there is

a match list entry with an antecedent match index prior to and a consequent match index later than the current antecedent match index and whether the status of this entry is either *Match* or *NotMatch*, since both represent an antecedent match.

The functions *P_NR()*, *P_O()*, *P_P()*, *P_R()*, and *R_ROR()* check whether the property is evaluated using a specific mode.

Symbol	**Definition**
Finished()	(TK.S = VacuousMatch) $\vee$ ((TK.S $\neq$ Cancel) $\wedge$ ((PMode $\neq$ Restart) $\vee$ ($\exists j \in$ MA_LST:j.AMI $\geq$ TK.CMI)))
Ignore()	(TK.S = Cancel) $\vee$ ((PMode = Restart) $\wedge$ (TK.S $\neq$ VacuousMatch) $\wedge$ ($\exists j \in$ MA_LST: ((TK.AMI $\leq$ j.AMI $<$ TK.CMI) $\wedge$ ((j.S = Match) $\vee$ (j.S = NotMatch)))))
Invalid()	($\exists q \in$ MA_LST: (q.ATID = TK.ATID) $\wedge$ (q.ASTID = TK.ASTID) $\wedge$ (q.CTID = TK.CTID) $\wedge$ (q.CSTID = TK.CSTID)) $\vee$ ($\exists i \in$ TK.TE_LST, $j \in$ TE_LST: (i.Index = j.Index) $\wedge$ (i.Event_ID = j.Event_ID) $\wedge$ (i.Decision $\neq$ j.Decision) $\wedge$ (j.Decision $\neq$ UNKNOWN))
Valid()	($\nexists q \in$ MA_LST: (q.ATID = TK.ATID) $\wedge$ (q.ASTID = TK.ASTID) $\wedge$ (q.CTID = TK.CTID) $\wedge$ (q.CSTID = TK.CSTID)) $\wedge$ (TK.TE_LST $\subseteq$ TE_LST)

Table 6.11: Match Filter - General Functions

The function *Invalid()* checks whether a token has to be discarded instead of entering the match filter. This happens if there already exists an entry in the match list which uses the same thread IDs and subthread IDs for both antecedent and consequent as the current token (this represents the fact that a token from another trigger expression has reached the matchfilter, after a first one has already been processed), or if the token includes a decision for a tentative event that has proven wrong.

The function *Valid()* checks whether a token is ready to proceed into the match filter evaluation. This is the case if there is no existing entry in the match list with identical values for thread ID and subthread ID for antecedent and consequent, and if the decisions for all encountered tentative events have proven correct.

The *Ignore()* function checks whether the token has to be discarded instead of leaving the match filter. This happens to all tokens with status *Cancel* since they do

not provide useful information for the assertion. In mode *Restart*, this also happens to all tokens with a status different from *VacuousMatch* if there is at least one match list element representing an antecedent match (i.e. status is either *Match* or *NotMatch*) with an antecedent match index that lies between antecedent match index and consequent match index of the token. This fact represents the situation that the current token is canceled due to a successful restarting of the assertion evaluation.

The *Finished()* function checks whether the token may leave the match filter and provide a final result to the assertion. This happens to all *VacuousMatch* tokens and to all tokens with a status different from *Cancel* if not running in *Restart* mode or if there is already a match list entry with an antecedent match index later than the token's consequent match index. This last condition represents the fact that all tokens that could restart the evaluation have already been processed.

Initial Validation / Discarding of Tokens

The general HLCPN structure of the match filter is shown in Figure 6.18.

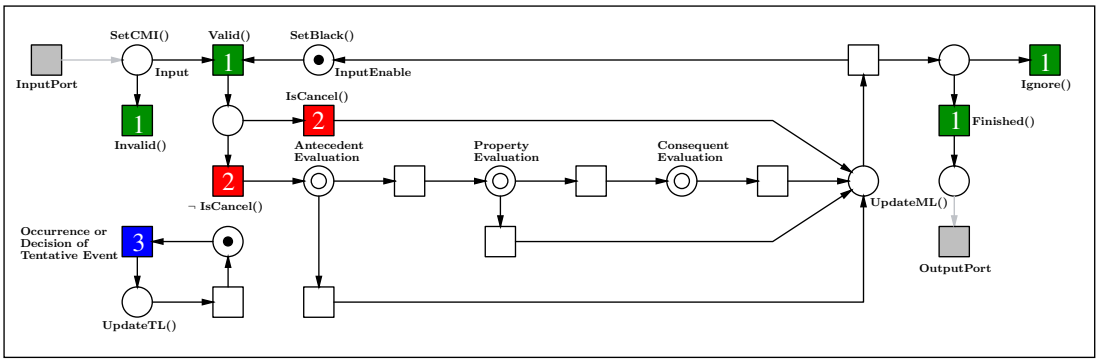

Figure 6.18: Match Filter

A small subnet of the matchfilter is responsible for keeping track of every occurrence of tentative events and for storing this information in the tentative list. In the same way all decisions concerning tentative events are also stored in the tentative list.

Incoming tokens arrive at the *Input* place first. Now it is checked whether a newly arrived token is valid or not. Tokens may be invalid for one of two reasons:

1. Every single-delay operator sends an evaluation token in three event expressions (see Section 6.2.9). Even though the first token leaving the delay operator represents the correct result, the other tokens still remain there and might leave the delay operator at a later time. If one of these remaining tokens later arrives

at the match filter, there will already be a valid entry in the match list with the exact same ID (that means thread ID and subthread ID for both antecedent and consequent match[6]). Thus, the new token has to be discarded.

2. If the thread represented by the token includes one or several tentative events and if the final decision of at least one of these events contradicts the assumption made for the token, then the token does not represent the real design behavior and is discarded.

If tokens are valid - there is no previous entry with the same ID in the match list and all decisions for tentative events (if any) are correct - then the token can be processed by the match filter.

If the token is neither valid nor invalid (this can only happen if a decision concerning a tentative event is not known yet) then the token has to remain in the *Input* place until it becomes either valid or invalid. Note that this mechanism might delay the whole evaluation by a large amount of time. On the other hand, if the remaining tokens were processed out of order, wrong results could be produced due to the dependencies between several tokens.

The black token in the *InputEnable* place ensures that only one token can be evaluated in the match filter at a time. Since token order is kept, tokens will always be evaluated in the order of their arrival. This means that the evaluation of a valid token might be delayed because an older token has not been processed yet.

Tokens entering the evaluation of the match filter are sorted in tokens of status *Cancel* and all others. The former bypass the actual evaluation, while the latter have to pass the evaluation of antecedent, property, and consequent results. Due to the complex structure of the match filter these three evaluation steps are represented by hierarchical places. The corresponding sub-nets are presented in the following.

Antecedent Evaluation

The antecedent evaluation is shown in Figure 6.19. First the mode of the antecedent sequence is checked. In case of mode *AnyMatch* all tokens of status *VacuousMatch* leave the sub-net via the *AbortPort*, bypassing the two remaining evaluation steps. All remaining tokens leave via the *ContinuePort* and are passed to the second sub-net. This behavior reflects the fact that all evaluation threads and subthreads are supposed to produce an independent result.

In all other cases (modes *FirstMatch* and *FirstMatchPipe*) every evaluation thread is only supposed to produce one result, regardless of the number of subthreads. Thus,

[6] In case of single sequence properties, only one pair of IDs exists.

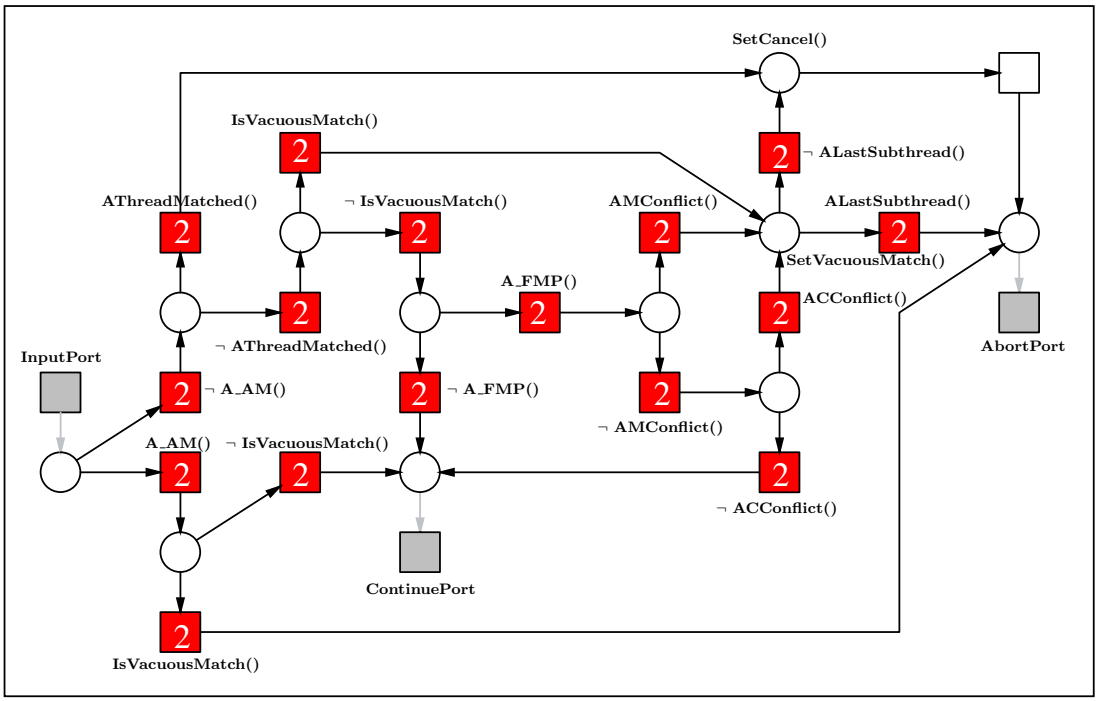

Figure 6.19: Match Filter - Antecedent Evaluation

it is first checked whether the thread represented by the token's ATID parameter has already matched. If yes, then the token is no longer needed, its status is set to *Cancel*, and it leaves the sub-net via the *AbortPort*. If no, then it is checked if the token represents a vacuous match, that means a "not match" result of the antecedent sequence. If this is the case, then it has to be determined if this is the last subthread of this evaluation thread. If yes, the token just leaves via the *AbortPort*, if no its status is changed to *Cancel* before (there might be a *Match* result in one of the later tokens after all).

If the thread has not produced a match before, the antecedent mode determines how the token is processed further. In case of mode *FirstMatch* the token has now already passed the antecedent evaluation and proceeds to the next sub-net via the *ContinuePort*. In case of mode *FirstMatchPipe* however, it has to be checked if one of two possible conflict situations did occur:

1. A match conflict occurs if several assertion threads try to match at the same trace index. In this case, only the oldest thread is allowed to match.

2. A consumption conflict occurs if several assertion threads try to "consume" the same Boolean proposition at the same trace index. Again only the oldest thread is allowed to do so.

If none of these situations arises, the token is propagated to the next sub-net via the *ContinuePort*. Otherwise the status of the antecedent is set to "not matched" - which results in the status of the whole evaluation to be set to *VacuousMatch* - and the aforementioned check on whether the token represents the last evaluation subthread is performed. The results of this check are described above.

Property Evaluation

The property evaluation is shown in Figure 6.20. At the beginning, the mode of the property is checked. In case of modes *Overlap* or *Pipe* the token immediately leaves the sub-net via the *ContinuePort*. Otherwise, it is checked whether an older thread has been evaluated that might influence the evaluation of this token. This is the case if the match list contains an entry of status *Match* or *NotMatch* (which both represent a match of the antecedent sequence) with an antecedent match prior to and a consequent match later than the antecedent match of the current token. This situation represents the fact that the current thread enabled the implication while the consequent sequence of the older thread was still being evaluated.

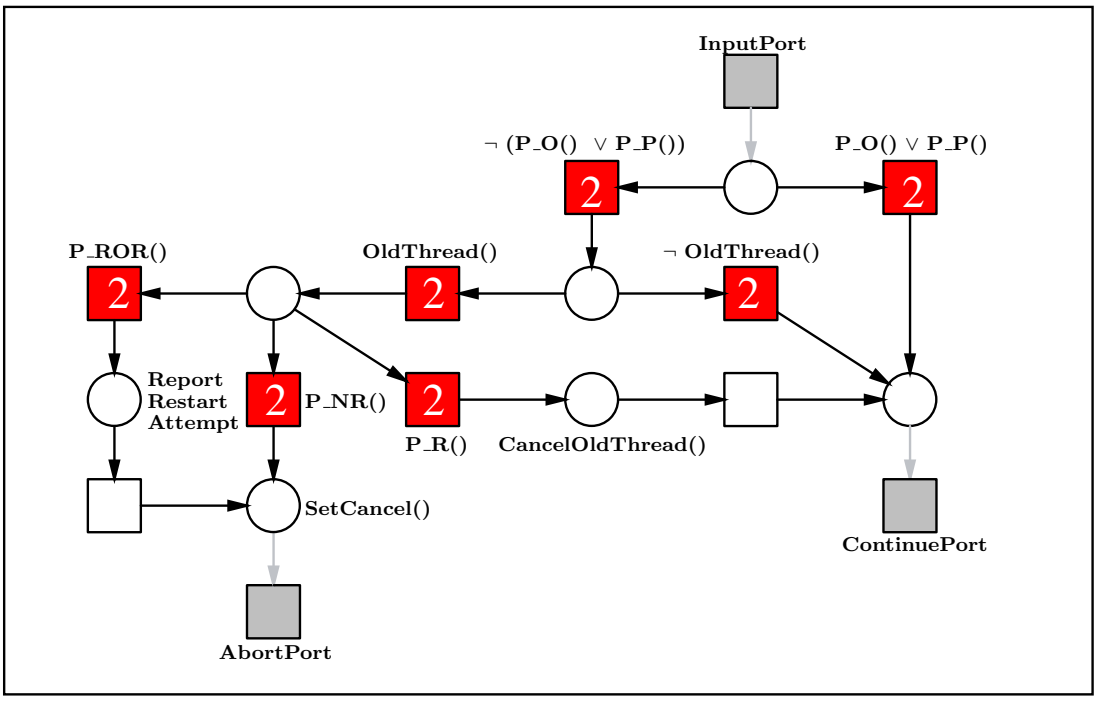

Figure 6.20: Match Filter - Property Evaluation

If there is no older thread then there is no concurrent thread evaluation and thus, the token exits the sub-net via the *ContinuePort*. Otherwise the behavior depends

on the property mode. In *NoRestart* mode, the new token has to be ignored. Thus, its status is changed to *Cancel* and it leaves via the *AbortPort*. In *ReportOnRestart* mode, an additional message is issued to inform about the restart attempt. In *Restart* mode finally, the new token is valid, but the old evaluation thread has to be discarded. This is achieved by changing the status of the old thread in the match list to *Cancel*. The new token then leaves via the *ContinuePort*.

Consequent Evaluation

The evaluation of the consequent is depicted in Figure 6.21.

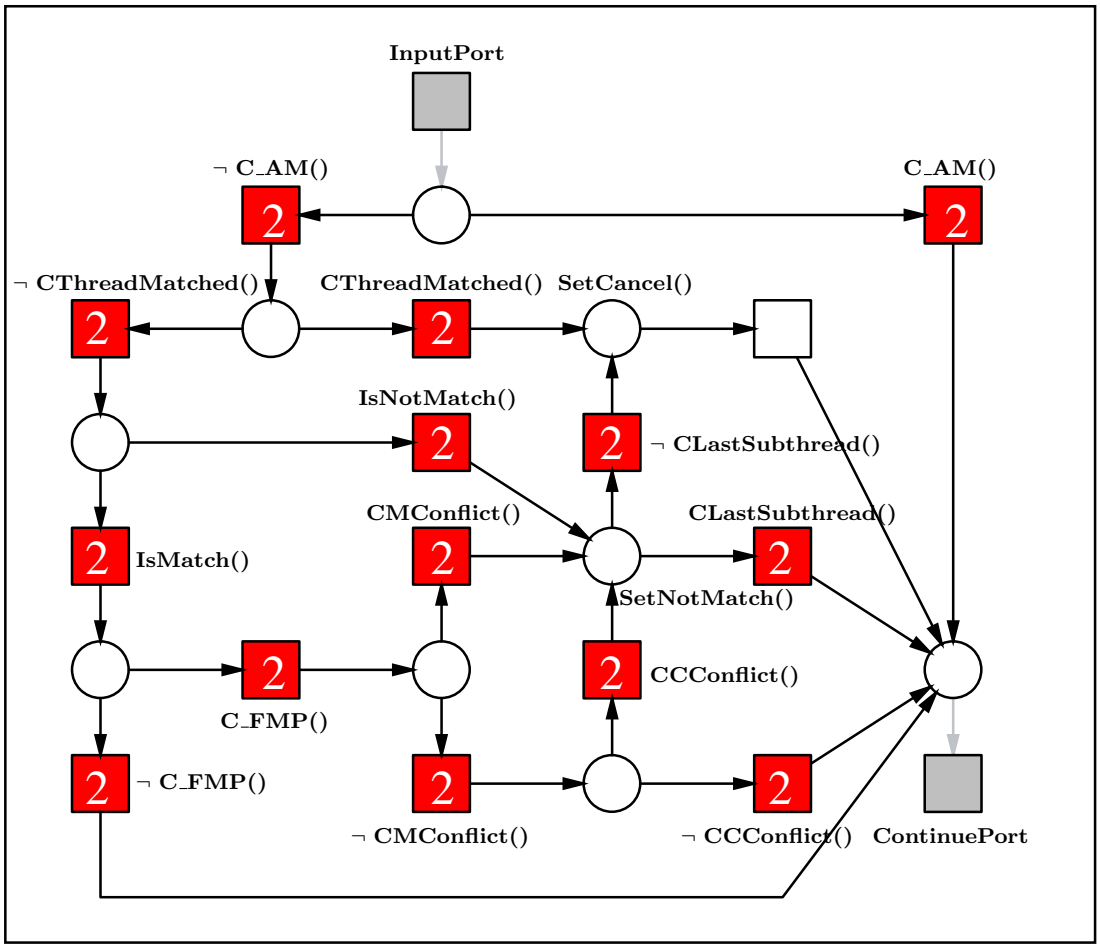

Figure 6.21: Match Filter - Consequent Evaluation

The general structure is very similar to the evaluation of the antecedent. However, three differences exist:

1. Instead of using general or token-based data referring to the antecedent, the corresponding consequent equivalents are used.
2. Since this is the last stage in the token evaluation, there is no need for two separate exit ports. Thus, all tokens leave the sub-net via the *ContinuePort*.
3. The status references are changed:
   - *Match* and *NotMatch* correspond to a match of the antecedent. On the other hand, only *Match* represents a consequent match.
   - While *VacuousMatch* represents a not matching antecedent, *NotMatch* corresponds to a not matching consequent.

Computation of Final Results

After the token has passed or bypassed the three evaluation sub-nets, its status is stored in the match list for further token checks. Afterwards, the token passes a transition that recreates the black input token. The colored token is now checked again. *Cancel* tokens are now discarded, the same happens to tokens which are to be ignored due to a restart token (see above). On the other hand, tokens with the status *VacuousMatch* leave the match filter immediately, the same happens to tokens which cannot be affected by a restart request anymore (if there is already an entry in the match list with an antecedent match index later than the current consequent match index, the relevant time window has already passed). If neither condition is fulfilled, the token is delayed until one of them becomes `true`.

Single Sequence Properties

The match filter is also able to evaluate single sequence properties. In order to produce correct behavior, it is assumed that the antecedent mode equals *AnyMatch* and the property mode equals *Overlap*. The consequent mode is determined by the mode parameter of the single sequence.

6.2.12 Mapping of Grammar Elements to Petri Nets

The composition of a given assertion from the various petri net components presented above can easily be done by parsing the UAL+ grammar tree (all grammar rules listed here can also be found in Appendix B).

The parser starts with an assertion directive (every directive results in a separate petri net, independent from all others):

directive = *directive_kind identifier* "(" [*directive_parameter*] ")"
"=" *property_instance* ";" ;

Every assertion directive instantiates a property and additionally might specify a reset expression within its parameter list:

property_instance = *identifier* [*property_mode_list*]
param_argument_list ;

directive_parameter = *severity_level*
"," *string*
["," *reset_event_expr*] ;

The reset event expression will be used later when instantiating the various delay operators (see below).

First, a match filter (see Figure 6.18) and a token generator (see Figure 6.16) will be created. The property instance contains a list of evaluation modes that will be passed to the match filter as parameters.

Furthermore, the identifier used in the property instantiation refers to a unique property specification:

property_section = "property" *identifier property_interface*
property_declarations
property_specification
"endproperty" ;

property_specification = *implication_property*
| *single_sequence_property*

Depending on its type the property instantiates either one sequence (single sequence property) or two - antecedent and consequent - (implication property). In the latter case an implication operator (see Figure 6.17) will be created.

sequence_instance = *identifier* ["[" *sequence_mode* "]"] *param_argument_list* ;

Again, the identifier used in the sequence instantiation refers to a unique sequence specification:

sequence_section = "sequence" *identifier sequence_interface*
sequence_declarations
sequence_specification
"endsequence" ;

sequence_specification = *delay_operator* { *delay_operator* } ";" ;

Every sequence consists of one or more delay operators:

delay_operator = "#" *steps sensitivity* "{" *condition* { *action* } "}" ;

In case of a delay range, a ranged delay operator (see Figure 6.15) is created. Furthermore, every delay number of "0" results in a creation of a zero-delay operator (see Figure 6.12), while for every delay number of "1" a single-delay operator (see Figure 6.13) is created. Delay numbers greater than "1" are split into an according number of single delays and thus, result in N single-delay operators.

The outputs of every delay element is connected to the input of the following element, while every ranged delay instantiates the various fixed delay operators at the positions of the corresponding hierarchical places.

In case of single sequence properties, the output of the token generator is connected to the input of the first delay operator of the single sequence, and vice versa. The output of the last delay operator is connected to the input of the match filter.

In case of implication properties, the output of the token generator is connected to the input of the first antecedent delay operator and vice versa. The output of the last antecedent delay operator is connected to the input of the implication operator, the output of the implication operator is connected to the input of the first consequent delay operator, and finally, the output of the last consequent delay operator is connected to the input of the match filter.

Now the only elements missing are the various event operators. The reset expression mentioned above results in the instantiation of the corresponding event operators within all delay operators.

If any of the three event expressions (positive, negative, and reset) is empty, the corresponding component (see Figure 6.14) is created. In all other cases, the event expression is constructed of the various available event operators (see Figures 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 6.10, and 6.11).

In case of more complex event expressions, a sub-expression is connected at the position of the corresponding hierarchical place of the instantiating expression.

Finally, all event expressions are instantiated within their corresponding delay operators.

Figure 6.22 shows exemplarily the mapping of the following assertion:

```
assert TEST(ERROR,"Test failed!",e5) =
   #1{e1|e2}{...} |-> #{1:2}{e3;e4}{...}
```

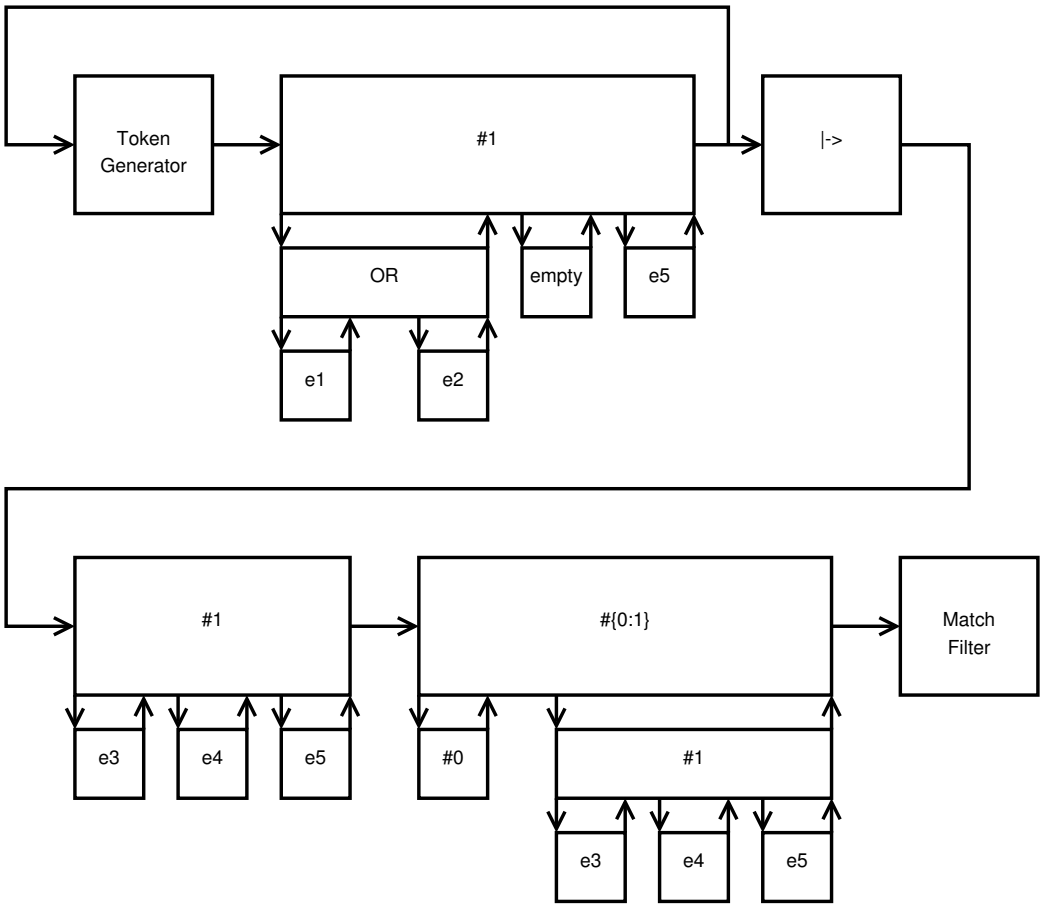

Figure 6.22: Example for Assertion Mapping

7 Assertion Transformation Framework

This chapter describes the overall application framework for UAL+ assertion transformation. Figure 7.1 gives an overview of the corresponding structure.

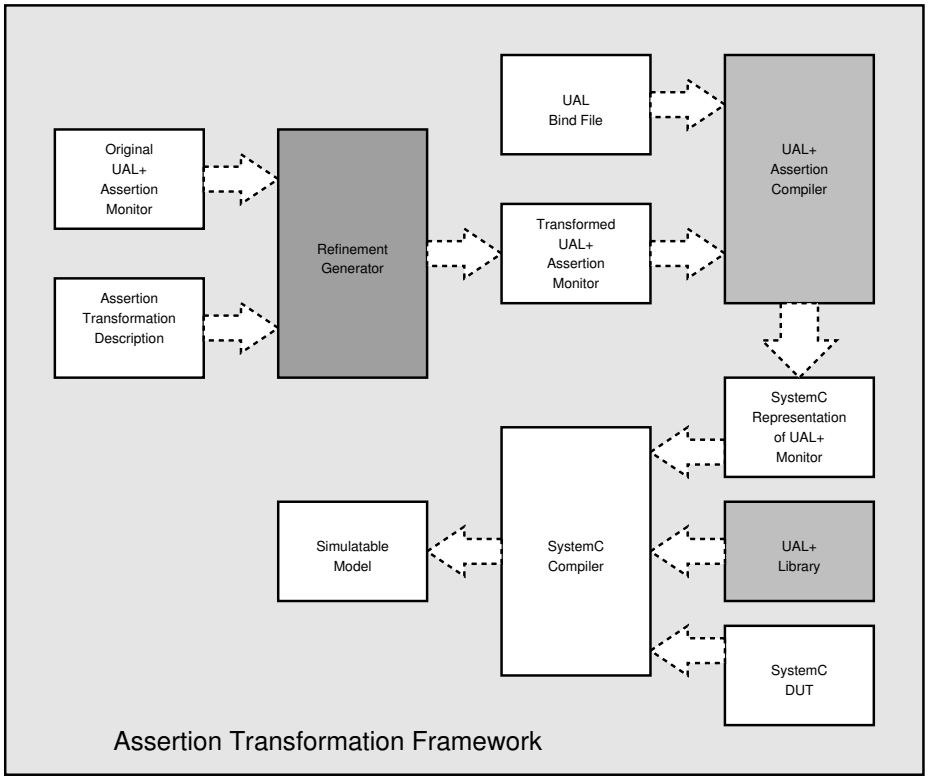

Figure 7.1: Assertion Refinement Framework

The refinement generator is responsible for transforming a UAL+ monitor file to another abstraction level. The resulting file is then read in by the UAL+ assertion compiler which produces a SystemC representation of the assertion. This representation makes use of the implementation of the various UAL+ operators which are encapsulated in the UAL+ library. The assertion together with the required library

elements and the DUT itself are then compiled with a SystemC compiler in order to produce a simulatable model.

The refinement generator was developed during this work, while both the UAL+ compiler and the UAL+ library had to be adapted and extended in order to support the features required for assertion refinement.

These three components are presented in the following sections.

7.1 Refinement Generator

The refinement generator first uses the Python [54] Gnosis library to parse both a UAL+ monitor file and a refinement file. Both are then converted into a parse tree structure.

The parser works based on a Extended Backus-Naur-Form (EBNF) grammar (see Appendix B) written in C++ which also had to be adopted to the new language extensions.

During the parsing process the files are checked for syntactical and semantical errors. The parse tree data structure keeps track of all declarations of sequences, properties, verification directives, as well as of the dependencies between these components.

Furthermore, all irrelevant information like keywords is either removed or translated in corresponding sub tree structures.

A specialized Python API was created to access and manipulate the various elements of the tree.

Afterwards, the actual refinement is done by a Python program which executes the transformation steps specified in the refinement file by directly transforming the monitor parse tree using the parse tree API.

For each operation the refinement generator first identifies the location where it has to be applied (e.g. a specific property or the negative sensitivity of the second delay operator within the antecedent sequence) and then adds, modifies, or removes nodes within the tree according to the refinement operation.

The generator also takes into account whether only one instance of a property or sequence should be affected or all (see Section 5.3.6). If only one instance is to be modified, the original declaration is copied and the modification is performed on the copy.

Similarly, if a property or sequence that is included from a library is the target of a transformation, a local copy is created and - depending on whether one or all instances should be affected - either the one corresponding reference or all references are changed to the new local copy.

Finally, the modified tree is written out using the MAKO [55] template engine. This template engine represents a flexible generation methodology by allowing to mix static content with dynamic data (that means data obtained from the parse tree) within a template.

MAKO offers conditional branches, loop constructs, access to data structures like the parse trees, access to the complete Python functionality, and packaging of frequently used template parts in so called sub templates. It is possible to pass parameters to these sub templates which further increases their reuse potential. Additionally, it is possible to use Python functions for often used calculation tasks.

The templates place the information stored in the transformed tree into a framework of UAL+ keywords and syntax, basically reversing the reduction process that happened during the parsing.

An example for a simple MAKO template is shown in Listing 7.1.

```
1 % for monitor in api.getMonitor(section):
2     % if monitor != None:
3 monitor ${api.getIdentifier(monitor)}
4     % endif
5 % endfor
```

Listing 7.1: Example of MAKO template

The leading `%` in a line distinguishes between a MAKO command and a normal line of text that should be printed. Using `for` and `if` commands it is possible to generate several monitor declarations which all follow the same structure while the monitor name is different.

Using a `$` it is possible to refer to local variable objects like the monitor loop object or to call functions or subtemplates that were defined previously.

A possible result of this generation process is shown in Listing 7.2.

```
1 monitor timer_checker
2 monitor cpu_test
3 monitor fft_example
```

Listing 7.2: Sample output of MAKO template

The generator can be parameterized with several options that allow for a more flexible way of dealing with problematic situations. It is possible for instance to specify how the generator should deal with ambiguous situations. Details concerning these ambiguity issues were discussed in Chapter 5.4.1.

7.2 UAL Library

The UAL+ library contains a SystemC implementation of all UAL+ operators presented in Chapter 6, including those that were developed during this work. Due to the heavy influence of the tentative matching concept even most of the existing operators had to be adapted.

7.2.1 Event Handler

The event handler is the central component of every UAL+ assertion. It is responsible for gathering all event data from a design and / or the corresponding assertion itself and then distributes these events to all delay operators that need to process it for their triggering. Since every simulation may only contain one instance of the event handler, it is implemented as a singleton object (see [56] for details).

The event handler does not distinguish between the various kinds of UAL+ events. On the other hand, the different event kinds differ in their origin, behavior, and the way they can be accessed. In order to overcome these discrepancies, all event kinds are converted to special callback events by proxy modules that are included within the design and capture signal changes as well as transactions.

Furthermore, the event handler holds a list specifying which event is required by which delay operator. Whenever an event occurrs, it is distributed to only those event operators that are sensitive to it.

Due to the introduction of triggers (and thus tentative events) and derived events, the event handler had to be enhanced to take these extensions into account. For this purpose an additional step is performed: All primary events are first distributed to all corresponding trigger sequences in order to evaluate which additional derived events occur. These derived events are then also propagated to all registered event operators.

The event handler also continuously monitors the status of the various tentative evaluation attempts. As soon as a tentative event is confirmed, the corresponding entry in all token tentative lists is removed, converting the token to a non-tentative one (unless it depends on other tentative events as well); all tokens assuming the

tentative event to be false are removed. If the tentative event is detected as invalid, validating and discarding of tokens happens just in the opposite way.

7.2.2 New Event Based Operators

In addition to the existing `timer` operator and the `$delta_t` function, corresponding versions have been introduced which work on the basis of events instead of simulation time. Since various instances of these event based operators might look for different events, it is not sufficient to implement a global counter as in the case of the time based versions. Instead, the UAL+ kernel has to provide one counter object per event which is used in this way. In order to keep the overhead small, the information which counters have to be implemented has to be passed to the kernel. This is done via the classes generated by the UAL+ compiler.

7.2.3 Token Handling

Due to the handling of tentative information and the connected fact that antecedent and consequent sequence within an implication property cannot be handled separately anymore, the amount of data stored within the various tokens has increased. In order to keep the resulting impact on performance as low as possible, the token handling has been changed so that not the tokens themselves are passed through the various operators, but instead only pointers to the tokens. Tokens are created and destroyed as necessary and are stored at a centralized place.

7.2.4 Further Enhancements

In order to ensure that no delay operator triggers more than once per primary event, an evaluate update mechanism is included. In this case, if a primary event is issued, the corresponding derived events are computed and all of them are passed to the assertion structure. The results of all delay operators are stored in temporary lists. As soon as all events have been processed, the results are written back to the assertion and the next primary event is processed.

In addition to the existing Boolean function *last_event(*event*)* that indicates whether the last occurred primary event equals "event", a second function is introduced: *has_occurred()* can only be used in combination with derived events but otherwise works in the same way as *last_event()*.

7.3 UAL Compiler

The UAL+ compiler has to generate a SystemC representation of the UAL+ assertion. This means that all language elements have to be mapped on a specific SystemC component - which is contained in the library. The compiler is responsible for building up the necessary structures for sequences, properties, etc. by instantiating the corresponding library elements. Furthermore, the compiler uses an additional bind file that specifies how the assertion monitor is connected to the DUT. Both the monitor file and the bind file are checked for correct syntax and semantics.

Due to the fact that the UAL+ library had to be reworked quite extensively, the existing generator was no longer usable. Instead of adapting the previous C++ based generator, a new template based generator based on the MAKO template engine was created.

In combination to adapting the compiler to the new library structure the various new features introduced to the UAL+ language had to be supported as well.

One UAL+ monitor can instantiate an arbitrary number of properties which in turn can instantiate one or two (in case of an implication property) sequences. Trigger specifications result in additional properties and sequences.

Using MAKO's support for hierarchical templates the generation process is structured and the assertion functionality is distributed to different output files: one SystemC header file is created for each monitor, one for each property, and one for each sequence.

- The monitor file implements all directives and trigger statements. It contains one class that instantiates all properties used within the directives, all properties and sequences used as triggers, and takes care of the connectivity between these elements.

- Each property file implements two classes: the first one represents the actual property and thus instantiates the one or two sequences used within the property while the second one is responsible for passing local variables from the antecedent to the consequent. Thus, the second class is only needed for implication properties and only if information has to be passed between the corresponding sequences.

- Each sequence file implements one class for the actual sequence and one class for the handling of local variables. Since the Boolean expressions occurring as part of the Boolean expression of the delay operator, of the constraint operator, of the accumulator operator, or the timer / event timer operator are not mapped

to petri net blocks (and thus UAL+ operators) but instead to basic SystemC expressions, these are collected within four additional classes.

In order to structure the compiler further, those structures that are needed in various contexts, like generating lists of include files, class declarations, class constructors, or frequently used class functions have been placed into sub templates.

After parsing the monitor file, the compiler assembles the various monitor structures form library elements and connects them together in the specified way.

Note that the binding of monitors to designs has not been influenced by the UAL+ extensions and thus, the existing generator for the bind functionality has been reused.

8 Application

This chapter presents a detailed example applying the transformation process to a given design. It focuses especially on the peculiarities of the various abstraction levels and the corresponding influence on the transformation. Following that, an analysis of the efficiency of the transformation process with regard to lines of code and execution penalty is done. Finally, a performance analysis concerning several of the new features introduced into UAL is done.

8.1 Transformation Example

The transformation process is explained using an Advanced High-performance Bus (AHB) timer component, which is modeled on all but one of the abstraction levels involved, combined with assertions on that abstraction level. Although creating a PV design is not sensible due to the nature of the component, the PVT implementation was combined with PV assertions.

8.1.1 Design Structure

The Design

The structure of the TL version of the AHB timer is shown in Figure 8.1. The design implements 32 countdown timer components that can be used independently from each other. It contains a TL specific transaction port[1] shared by all timers, as well as one interrupt output port, one enable register, and one counter register per timer.

A timer can be used by first setting the counter value of a specific timer and then enabling the timer. Both are done via the transaction call *WRITE* that carries data and address information. The address selects a specific enable or counter register while the data value specifies either the counter value or information about enabling or disabling the timer. The corresponding interrupt is issued counter value times 10 time steps after the enabling of a counter. The only legal way of reconfiguring a running timer is by disabling it first.

[1] In the RTL model this transaction port is replaced by several signal ports.

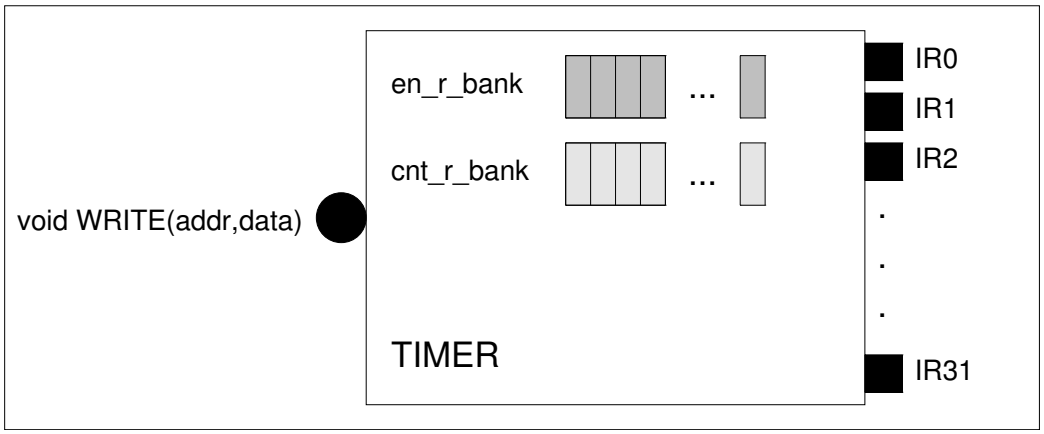

Figure 8.1: AHB Timer

Figure 8.2 shows the temporal behavior of one timer including its configuration. The first call of *WRITE* sets the counter register of the timer, the second call addresses the timer's enable register and sets its value to active. The interrupt is issued after counter value times 10 time steps.

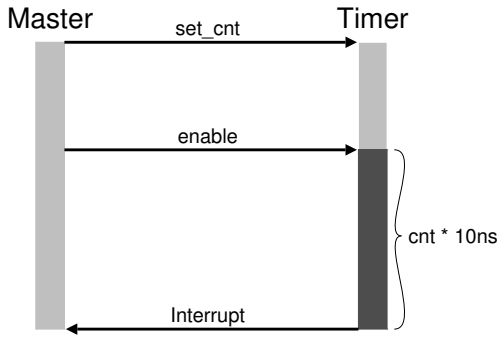

Figure 8.2: Sequence of the Timer Interrupt Procedure

The PVT Assertion

The corresponding assertion (see Listing 8.1) specifies that after setting a counter value and enabling a timer, either the interrupt is issued after the given time or the timer is disabled before. Any external change of the counter value while the timer is active causes the assertion to fire. The same happens if the interrupt is not issued in time.

On all TL sublevels the transaction call to the timer module is detected by the assertions via a proxy module that is inserted in between the timer module and any possible master. This is shown in Figure 8.3. Listing 8.1 shows the sequences section of the PVT assertion monitor.

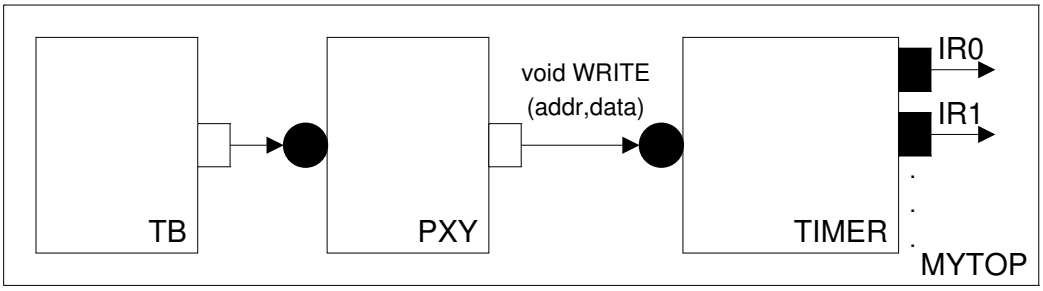

Figure 8.3: Transaction detection via Proxy Modules

```
1  sequence ir1_ante(transaction void WR(
2            sc_uint<12>addr,sc_uint<12>data),
3         ref sc_uint<12> cnt)
4    #1{WR'END@(WR.addr == 0)}{true, cnt = WR.data}
5    #1{WR'END@(WR.addr == 1)}{WR.data == 1};
6  endsequence
7  sequence ir1_cons(transaction void WR(
8            sc_uint<12>addr,sc_uint<12>data),
9         signal sc_signal<bool> IR1,ref sc_uint<12> cnt)
10   #1{IR1'POS@($delta_t == (cnt*10)) |
11      WR'END@(((WR.addr == 1) && (WR.data == 0)) &&
12      (($delta_t >= 0) && ($delta_t <= (cnt*10))));
13      WR'START@((WR.addr == 0) &&
14      (($delta_t >= 0) && ($delta_t <= (cnt*10)))),
15      timer((cnt*10)+1)}{true};
16 endsequence
```

Listing 8.1: PVT Timer Assertion Monitor

The antecedent sequence checks for the end of a first write transaction that has to write to the counter register of the first timer. The data parameter of the transaction is stored in the local variable *cnt*. At the next step a second write transaction has to occur that targets the enable register of the first timer and writes the value "1" in it in order to enable the counting.

The positive sensitivity of the consequent consists of two alternative conditions:

1. A positive edge of the first interrupt signal is detected with a temporal distance to the enabling of exactly *cnt* times 10 time steps after the last triggering;

this means the interrupt has occurred at exactly the right time (see line 10 in Listing 8.1)

2. Another write transaction is completed that writes the value "0" in the enable register of the first timer; this means the timer has been disabled before it could run out (see lines 11 and 12 in Listing 8.1)

Additionally, the consequent includes two conditions in its negative sensitivity:

1. A start of any write transaction to the counter register of the first timer before the timer runs out which represents an illegal reconfiguration attempt of a running timer (see lines 13 and 14 in Listing 8.1)

2. A timeout mechanism that causes the assertion to fire if the specified time for the interrupt has expired (see line 15 in Listing 8.1)

Transformation File Basics

The various transformation files used in the examples use the language features introduced in Chapter 5.3. Additionally to making use of various declarations (e.g., for constants, ports), each transformation file uses one or several rules (see Section 5.3.3) for handling the complex transformations between different abstraction levels, which in turn consist of several basic transformation directives (see Section 5.3.5). For transformation between RTL and the various TL sublevels transactors (see Section 5.3.4) are required as well.

The transformation language supports the concept of removing frequently used parts (especially declarations) from the transformation file and grouping them in library files which are then imported in the transformation file (see Section 5.3.2). In order to increase readability of the transformation files this feature has been used for the examples presented here as well; all declarations, transactors, and rules have been placed in several library files, each corresponding to a certain purpose. This also eases reuse of existing rules as will be shown later.

The first library file (see Listing 8.2) `declarations_lib.tll` contains various declarations that are required for the transformation process.

The constant *CLOCK_CYCLE* is used in combination with CA models. The various ports are needed, whenever the interface has to be changed between signal based and transaction based; the signal port *clk* is also used in combination with CC models. Finally, the variable *cnt* is used for the transformation of PV assertions to other abstraction levels.

```
1  library declaration_lib
2    declarations
3      ports
4        transaction void WRITE(sc_uint<12> addr,sc_uint<12> data);
5        signal sc_signal<bool> clk;
6        signal sc_signal<bool> rst;
7        signal sc_signal<bool> en;
8        signal sc_signal<bool> wr;
9        signal sc_signal<sc_uint<12> > addr;
10       signal sc_signal<sc_uint<12> > data;
11     endports
12
13     constants
14       int CLOCK_CYCLE = 10;
15     endconstants
16
17     variables
18       sc_uint<12> cnt;
19     endvariables
20   enddeclarations
21 endlibrary
```

Listing 8.2: Library File containing Declarations

The library `transactor_lib.tll` contains the necessary transactor for the conversion between TL and RTL. It is shown later (see Listing 8.15) when it is used.

All conversion rules have been split up thematically into the libraries `timing_rule_lib.tll` for all timing related transformations, `reset_rule_lib.tll` for all transformation related to the addition or removal of a reset expression, and `misc_rule_lib.tll` for all remaining rules. Excerpts from these libraries are shown when needed later on.

8.1.2 Transformation Categories

Although during the application all abstraction levels have been transformed into all other abstraction levels, several of these transformations repeat certain features. For the transformation the abstraction levels can be grouped into three categories:

- PVT, CA, and CC only differ concerning the way timing is noted. The only affected items within an assertion are timer operators and time constraints. The only additional changes deal with adding or removing constants specifying clock periods (CA) and clock signals (CC).
- PV models pose slightly bigger problems due to the complete lack of timing information. On the one hand, adding time to an untimed assertion requires

detailed information about where and which timing has to be added. On the other hand, removing timing information when transforming an assertion to PV might not be sufficient; if additional constructs (like local variables) refer to the timing of the design, they have to be removed / changed as well.

- RTL designs show the biggest differences when compared to the remaining levels. The notion of transactions is dropped in favor of signal interfaces. The inclusion of reset information is only a minor issue here, though.

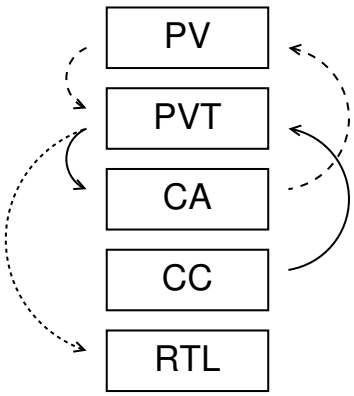

Figure 8.4: Overview of the presented transformations

Figure 8.4 presents the examples for the transformation between these categories, which will be explained in the following sections including a detailed discussion of the features involved. The transformations denoted by the solid arrows will be covered in Section 8.1.3, those with dashed arrows in Section 8.1.4, and those with dotted arrows in Section 8.1.5. Not all possible transformations are covered in examples, since some of these transformations consist of features discussed before or can even be completely expressed by combining several of the presented transformations.

8.1.3 Transformation between PVT, CA, and CC

As mentioned above, a transformation between these TL sublevels only affects the way timing is modeled. Thus, the corresponding rules have a relatively simple structure.

Refinement

Listing 8.3 shows the refinement file for a transformation from PVT to CA.

```
1 refinement timer_pvt2ca
2   import declaration_lib;
3   import timing_rule_lib;
4
5   apply
6     timing_pvt2ca<timer_checker_pvt.TIMER_IR1.ir1_prop.ir1_cons.
          *.*>(CLOCK_CYCLE);
7   endapply
8 endrefinement
```

Listing 8.3: Transformation from PVT to CA

This transformation consists of only the rule *timing_pvt2ca*. The rule gets parameterized by both a reference to the constant declaration *CLOCK_CYCLE* and the location the rule should be applied at. The location in the example specifies all delay operators (the second '*') within the positive and negative trigger expressions (the first '*') of the sequence *ir1_cons* within the property *ir1_prop* of the assertion *TIMER_IR1* within the monitor *timer_checker_pvt*. The implementation of this rule can be seen in Listing 8.4.

```
1 rule timing_pvt2ca<location>(declaration const)
2   add.constant<location>
3   {
4     additive const;
5   }
6   modify.time_constraint<location>
7   {
8     refines const.value();
9     modification const.name();
10  }
11  modify.timer<location>
12  {
13    refines const.value();
14    modification const.name();
15  }
16 endrule
```

Listing 8.4: Changing from Absolute Timing to Cyclic Timing (from timing_rule_lib.tll)

The first directive *add.constant* adds a constant declaration at the given location. The declaration is named *const* within the rule and refers to the declaration

```
1  sequence ir1_ante(transaction void WR(
2            sc_uint<12>addr,sc_uint<12>data),
3         ref sc_uint<12> cnt)
4    #1{WR'END@(WR.addr == 0)}{true, cnt = WR.data}
5    #1{WR'END@(WR.addr == 1)}{WR.data == 1};
6  endsequence
7  sequence ir1_cons(transaction void WR(
8            sc_uint<12>addr,sc_uint<12>data),
9         signal sc_signal<bool> IR1,ref sc_uint<12> cnt)
10   #1{IR1'POS@($delta_t == (cnt*CLOCK_CYCLE)) |
11      WR'END@(((WR.addr == 1) && (WR.data == 0)) &&
12      (($delta_t >= 0) && ($delta_t <= (cnt*CLOCK_CYCLE))));
13      WR'START@((WR.addr == 0) &&
14      (($delta_t >= 0) && ($delta_t <= (cnt*CLOCK_CYCLE)))),
15      timer((cnt*CLOCK_CYCLE)+1)}{true};
16 endsequence
```

Listing 8.5: CA Timer Assertion Monitor

CLOCK_CYCLE which is passed to the rule as a parameter as explained above. While the location points to both sensitivity lists of all consequent delay operators, this directive expects only a monitor. Thus, the remaining part of the location is ignored and the constant is added to the *constants* section of the monitor.

The second and third directive are similar in nature. They modify all time constraints and timer operators within the consequent sequence by replacing the value of the constant *CLOCK_CYCLE* (referenced by *const.value()*) by its name. Since the value of this constant is "10" (see Listing 8.2), all absolute time values of 10 time steps are replaced by one occurrence of *CLOCK_CYCLE*, in other words one clock cycle. The resulting UAL file is shown in Listing 8.5.

The refinement file for other transformations from one of these levels to another one looks almost exactly like Listing 8.3, just with another name for the rule and potentially other parameters. Thus, these other files are not further discussed.

Abstraction

Listing 8.6 shows the same monitor modeled as CC, while Listing 8.7 depicts the rule for transforming this CC assertion to PVT.

The directive *remove.parameter* removes the parameter *clock*, which references the *clk* signal (not shown in the listing), from the interface of the specified sequence. Since this parameter has to be passed down through the hierarchy, this means that the signal has to be removed from the interfaces of property, verification directive, and monitor as well (as long as it is not used by other sequences, properties, etc.).

```
1  sequence ir1_ante(transaction void WR(
2            sc_uint<12>addr,sc_uint<12>data),
3         ref sc_uint<12> cnt)
4    #1{WR'END@(WR.addr == 0)}{true, cnt = WR.data}
5    #1{WR'END@(WR.addr == 1)}{WR.data == 1};
6  endsequence
7  sequence ir1_cons(transaction void WR(
8            sc_uint<12>addr,sc_uint<12>data),
9         signal sc_signal<bool> IR1,signal sc_signal<bool> clk,
10        ref sc_uint<12> cnt)
11   #1{IR1'POS@($delta_t[clk'POS] == (cnt * 1)) |
12      WR'END@(((WR.addr == 1) && (WR.data == 0)) &&
13      (($delta_t[clk'POS] >= 0) && ($delta_t <= (cnt * 1))));
14      WR'START@((WR.addr == 0) &&
15      (($delta_t[clk'POS] >= 0) && ($delta_t[clk'POS] <= (cnt *
            1)))),
16      timer[clk'POS]((cnt * 1)+1)}{true};
17 endsequence
```

Listing 8.6: CC Timer Assertion Monitor

The other two directives are responsible for transforming the time constraints and timer operators. In this case the value "1" is replaced by the value "10" and additionally the term "[clk'POS]" is removed from all time constraints and timer operators. As a result both counted occurrences of positive clock edges before and now consider time in multitudes of ten time steps for triggering or Boolean results.

```
1  rule timing_cc2pvt<location>(declaration clock, name clock_event
       )
2    remove.parameter<location>
3    {
4      removal clock;
5    }
6    modify.time_constraint<location>
7    {
8      refines [clock_event] "1";
9      modification "10";
10   }
11   modify.timer<location>
12   {
13     refines [clock_event] "1";
14     modification "10";
15   }
16 endrule
```

Listing 8.7: Changing from Clocked Timing to Absolute Timing (from timing_rule_lib.tll)

8.1.4 Transformation between PV and other TL sublevels

Abstraction

In case of a transformation from PVT, CA, or CC to PV, all timing information has to be removed from a given assertion. Listing 8.8 shows a sample refinement file transforming from CA to PV.

```
1  refinement timer_ca2pv
2    import declaration_lib;
3    import timing_rule_lib;
4    import misc_rule_lib;
5
6    apply
7      remove_variable<timer_checker_ca.TIMER_IR1.ir1_prop>(cnt);
8      remove_variable_assignment<
           timer_checker_ca.TIMER_IR1.ir1_prop.ir1_ante.*>(cnt);
9      timing_ca2pv<timer_checker_ca.TIMER_IR1.ir1_prop.ir1_cons.*.
           *>(CLOCK_CYCLE);
10   endapply
11 endrefinement
```

Listing 8.8: Transformation from CA to PV

In this particular example, it is also necessary to remove the local variable used for checking the timing condition.

The first two transformation rules called in the *apply* section can be found in Listing 8.9.

```
1  rule remove_variable<location>(declaration var)
2    remove.variable<location>
3    {
4      removal var;
5    }
6    remove.pass_reference<location.*>
7    {
8      removal var;
9    }
10 endrule
11 rule remove_variable_assignment<location>(declaration var)
12   remove.variable_assignment<location>
13   {
14     removal var.name();
15   }
16 endrule
```

Listing 8.9: Removing Local Variable References (from misc_rule_lib.tll)

The location passed to *remove_variable* specifies the implication property of the assertion. The rule consists of two parts. The first removes the declaration of a local variable from the specified location, the second removes the local variables from the parameter list of the interfaces of the instantiated sequences and also from the argument list of the sequence instantiations. This is done by adding a wildcard to the location before passing it to the *remove.pass_reference* directive.

The third rule concerns the change of the assertion timing. The corresponding rule can be seen in Listing 8.10.

```
1 rule timing_ca2pv<location>(declaration const)
2   remove.constant<location>
3   {
4     removal const;
5   }
6   remove.time_constraint<location>
7   remove.timer<location>
8 endrule
```

Listing 8.10: Changing from Cyclic Timing to Untimed (from timing_rule_lib.tll)

This rule removes all time constraints and timer operators from a given location, in this case all sensitivity lists of all delay operators within the consequent sequence. Additionally, the declaration of the constant *CLOCK_CYCLE* is removed, since it is only needed on CA. The resulting UAL monitor file can be seen in Listing 8.11

```
1 sequence ir1_ante(transaction void WR(
2           sc_uint<12>addr,sc_uint<12>data))
3   #1{WR'END@(WR.addr == 0)}{true}
4   #1{WR'END@(WR.addr == 1)}{WR.data == 1};
5 endsequence
6 sequence ir1_cons(transaction void WR(
7           sc_uint<12>addr,sc_uint<12>data),
8       signal sc_signal<bool> IR1)
9   #1{IR1'POS | WR'END@((WR.addr == 1) && (WR.data == 0));
10      WR'START@(WR.addr == 0)}{true};
11 endsequence
```

Listing 8.11: PV Timer Assertion Monitor

Refinement

When reversing this process, the situation gets a bit more complicated. The transformation has to specify where timing information has to be added to the assertion. Similarly, different time constraints might have to be added to the various places

within the assertion. As a result the directives for adding timing information have to allow the exact specification of where and what information should be added to the assertion. An example for the transformation from PV to PVT can be seen in Listing 8.12.

```
1  rule timing_pv2pvt<location>(name var_name)
2    add.time_constraint<location.*>
3    {
4      refines "IR1'POS";
5      additive "$delta_t==(",var_name,"*10);";
6    }
7    {
8      refines "WR'END";
9      additive "($delta_t >= 0) && ($delta_t <= (",var_name,"*10))
           ;";
10   }
11   add.timer<location.negative>
12   {
13     additive "(",var_name,"*10)+1;";
14   }
15 endrule
```

Listing 8.12: Changing from Untimed to Absolute Timing (from timing_rule_lib.tll)

First, this rule adds time constraints to two specific expressions. The first half of the *add.time_constraint* directive looks for a positive edge of the *IR1* signal and attaches a time constraint of exactly 10 time steps to every occurrence of this expression. The second half adds another constraint to all occurrences of the end of a write transaction. Finally, the second directive adds a timer operator to the negative sensitivity list. Since no *refines* keyword is used with this directive, the timer operator is just added to the whole expression.

Additionally, a local variable declaration and a corresponding assignment have to be added (see Listing 8.13).

8.1.5 Transformation between RTL and timed TL sublevels

The transformation from RTL includes not only a change of the timing scheme (except for the transformation to CC), but also modifies the module interface by changing from a transaction based variant to a signal based one. Listing 8.14 shows the corresponding transformation file.

In addition to the timing change that happens similarly to the examples shown in the previous sections, a transactor is used for the interface transformation. Further-

```
1  rule add_variable<location>(declaration var)
2    add.variable<location>
3    {
4      additive var;
5    }
6    add.pass_reference<location.*>
7    {
8      additive var;
9    }
10 endrule
11 rule add_variable_assignment<location>(name var_name, value
       var_value)
12   add.variable_assignment<location>
13   {
14     additive var_name,"=",var_value;
15   }
16 endrule
```

Listing 8.13: Adding Local Variable References (from misc_rule_lib.tll)

```
1  refinement timer_pvt2rtl
2    import declaration_lib;
3    import transactor_lib;
4    import timing_rule_lib;
5    import reset_rule_lib;
6
7    apply
8      timer_write<timer_checker_pvt.TIMER_IR1.ir1_prop.*>[down](
           WRITE, write_trans, clk, rst, en, wr, addr, data);
9      timing_pvt2cc<timer_checker_pvt.TIMER_IR1.ir1_prop.ir1_cons.
           *>(clk, "clk 'POS");
10     add_reset<timer_checker_pvt.TIMER_IR1>("rst 'NEG");
11   endapply
12 endrefinement
```

Listing 8.14: Transformation from PVT to RTL

more, a reset statement is included in the assertion. Listing 8.15 shows the code of the assertion transactor.

The assertion transactor instanciation then maps the various declaration objects to the transactor interface while the keyword "down" specifies which direction the transformation should take. The transactor then proceeds to insert the given sequence into the sequences section of the generated monitor file, to declare a corresponding trigger, and finally to replace all references to events of the translated transaction by derived events bound to the trigger. Furthermore, the various transaction parameters are mapped to the signals mentioned in the sensitivity list.

```
1  library transactor_lib
2    transactors
3      transactor timer_write<location>(transaction void WRITE(
           sc_uint<12> addr, sc_uint<12> data),
4                  trigger write_trigger,
5                  signal sc_signal<bool> clk,
6                  signal sc_signal<bool> rst,
7                  signal sc_signal<bool> en,
8                  signal sc_signal<bool> wr,
9                  signal sc_signal<sc_uint<12> > addr,
10                 signal sc_signal<sc_uint<12> > data)
11       map
12         WRITE.addr => addr;
13         WRITE.data => data;
14       endmap
15       sequence write_trigger[FirstMatch](signal sc_signal<bool>
             clk,signal sc_signal<bool> rst,signal sc_signal<bool>
             en,signal sc_signal<bool> wr)
16         #1{clk 'POS;rst 'NEG}{en && wr} #1{clk 'POS;rst 'NEG}{true};
17       endsequence
18     endtransactor
19   endtransactors
20 endlibrary
```

Listing 8.15: Library File containing Transactor

Finally, the last rule adds a reset expression to the assertion. The corresponding implementation is shown in Listing 8.16 while Listing 8.17 shows the RTL implementation of the assertion.

```
1 rule add_reset<location>(name reset_event)
2   add.reset<location>
3   {
4     additive reset_event;
5   }
6 endrule
```

Listing 8.16: Adding a Reset Statement to an Assertion (from reset_rule_lib.tll)

8.1.6 Transformation between RTL and PV

As was shown in the previous sections, the transformation of the assertion interface and the change of the corresponding timing are independent from each other. Since there is no dependency between these two transformation steps, every transformation between RTL and PV can be regarded as a combination of a RTL-CC and a CC-PV transformation. The former takes care of the interface transformation without

```
1  sequence write_trans(signal sc_signal<bool> clk,
2        signal sc_signal<bool> rst,signal sc_signal<bool> en,
3        signal sc_signal<bool> wr)
4    #1{clk'POS;rst'NEG}{en && wr} #1{clk'POS;rst'NEG}{true};
5  endsequence
6  sequence ir1_ante(trigger WR,
7        signal sc_signal<sc_uint<12> > addr,
8        signal sc_signal<sc_uint<12> > data,
9        ref sc_uint<12> cnt)
10   #1{WR'MATCH@(addr == 0)}{true, cnt = data}
11   #1{WR'MATCH@(addr == 1)}{data == 1};
12 endsequence
13 sequence ir1_cons(trigger WR,
14       signal sc_signal<sc_uint<12> > addr,
15       signal sc_signal<sc_uint<12> > data,
16       signal sc_signal<bool> IR1, signal sc_signal<bool> clk,
17       ref sc_uint<12> cnt)
18   #1{IR1'POS@($delta_t[clk'POS] == (cnt * 1)) |
19      WR'MATCH@(((addr == 1) && (data == 0)) &&
20      (($delta_t[clk'POS] >= 0) && ($delta_t[clk'POS] <= (cnt *
          1))));
21      WR'START@((addr == 0) &&
22      (($delta_t[clk'POS] >= 0) && ($delta_t[clk'POS] <= (cnt *
          1)))),
23      timer[clk'POS]((cnt * 1)+1)}{true};
24 endsequence
```

Listing 8.17: RTL Timer Assertion Monitor

influencing the timing (RTL and CC are both based on clocks), while the second does not touch the interface, but changes the timing.

8.1.7 Summary

It can be summarized that the two main topics for refinement - interfaces and timing - can be treated independently from each other:

The transformation between RTL signal interfaces and TL transaction interfaces can be easily handled using transactors specified in the refinement file. The only required information is the signal sequence representing the transaction on the one hand, and the mapping of signals and transaction parameters on the other hand.

Timing transformation between PVT, CA, and CC does not require much effort. The parameters of time constraints and timer operators have to be adjusted according to simple modification rules, CA designs require an additional constant that specifies the clock period, while CC models need a clock signal. Transformation to PV is also

quite easy, since the removal of all timing information is sufficient[2]. Transformation from PV to one of the other levels requires more effort, since the exact nature of the timing behavior has to be specified. The complexity does not increase overly much, nevertheless the reuse potential of these transformation rules is limited due to their specific nature.

Adding and removing reset information is very easy and can be considered independent from both categories mentioned above.

The only area not covered by these categories is assertion specific information, like the declaration of local variables for instance. Since this information is so specific, these parts of an assertion are probably not reusable at all in most cases.

One advantage of this step wise approach as presented here is that the various transformation steps can be combined to cover other transformations as well. For instance, if the transformation rules for PV to PVT and those for PVT to CA already exist, then both sets of transformations can be applied after each other to cover the transformation from PV to CA. This allows easy reuse of existing transformations and avoids the necessity to specify transformation rules from each level to every other one.

8.1.8 Transformation Analysis

It has to be noted that the generated assertions are almost equivalent to hand written assertions as long as only the levels PVT, CA, and CC are included, since the assertion is only slightly modified. When refining towards PV this statement also holds true. Refining from PV to another level however requires more sophisticated transformation rules. In some cases these complicated rules might lead to less than optimal results.

The performance of the transformation itself on the other hand has no impact on the simulation speed, since the assertion refinement happens before and independent from the simulation.

While it is possible to compare the number of lines of code between the refinement file on the one hand and the generated UAL file on the other, this comparison leads to very different results, depending on the circumstances.

When using a refinement file for transforming exactly one assertion exactly once, the lines of code in the refinement file might very well exceed those in the generated assertion, especially if a transformation from PV to another level or to / from RTL is done. This effect becomes even more noticable when the original assertion file is small.

[2]Sometimes, additional information specific to the assertion in question has to be removed as well.

If, on the other hand, the assertion file is bigger, this effect tends to get reversed at some point. Similarly, writing flexible transformation descriptions and reusing them for the transformation of several assertions also shows a gain in efficiency.

A big gain in efficiency can also be achieved by using several transformation rules in order to generate all other abstraction levels from one given assertion. This gain is caused by the reuse of declarations, transactors, and even several rules that are applied at several different levels.

It has to be noted that an efficient transformation with regards to performance and lines of code is only secondary. The primary goal of assertion refinement lies in the verification of models written on different levels of abstraction using consistent assertions. Thus, using transformed assertions instead of hand written ones employs additional consistency checks and thus improves quality of the assertions. Additionally, this automated refinement process saves time, since the corresponding assertions have to be specified only on one level and can be transformed to the other ones.

8.2 Simulation Performance Analysis

Other designs - for instance a CPU subsystem - have been tested in combination with this assertion refinement approach as well. Due to the higher complexity of these designs, and thus also of the assertions, presenting the corresponding code examples would exceed the scope of this work, though.

However, several general observations could be made based on these tests as to how the various features introduced in this work impact the simulation performance.

This section will present an overview of the general performance drawback of assertions, followed by a closer examination of the concrete drawback of the features *evaluate-update*, *tentative matching*, and *trigger sequences*.

8.2.1 Assertion Drawback

The designs tested with this assertion approach were also used to determine the impact of assertions on the overall simulation performance. For this purpose, every design was simulated both without any assertions and with a typical set of assertions attached to the designs.

The designs tested in this context include a systolic array for hardware sorting for both up to 16 and up to 32 data words, a CPU subsystem modeled on TLM and RTL, an FFT algorithm, and a switch device modeled on TLM and as a TLM / RTL mix.

The simulation was performed using an automated regression suite which repeated each simulation several times and calculated an average result.

Design	Assertion Coverage	Time wo Assertions [s]	Time w Assertions [s]	Drawback Factor
Sort (16)	398193	5.8	15.9	2.7
Sort (32)	724181	16.9	45.5	2.7
CPU (TLM)	508000	0.3	12.9	43
CPU (RTL)	508000	11.4	25.9	2.3
FFT	767918	18.9	80.2	3.2
Switch (TLM)	359991	25.2	57.2	2.7
Switch (mixed)	755970	52.7	262.2	5.0

Table 8.1: Assertion Performance Impact

The assertion drawback is shown in Table 8.1. As can be seen, first the general impact of adding assertions to a design in terms of simulation time was measured. The coverage information describes the sum of all assertion threads which produced a result during the simulation. Thus, this can be used as a rough estimation about the assertion activity.

For most examples the drawback factor ranges between 2 and 3. In case of the FFT algorithm and the mixed level switch application (which are both algorithms handling a high number of data words) the slightly higher drawback factor correlates with a higher coverage as well.

The big drawback in case of the TLM CPU in contrast to the RTL CPU (while the coverage is identical) can be explained with the difference in simulation performance of the model itself. The absolute drawback due to the assertions is almost identical in both cases.

8.2.2 Impact of Evaluate-Update and Tentive Mechanisms

Two of the mechanisms introduced in this work were the evaluate-update mechanism, which is needed for the correct handling and distinction of primary and derived events, and the tentative matching concept. Since both have quite extensive impact on the implementation of the assertion engine, several dedicated tests for checking the impact of these features were done.

The RTL CPU was chosen as the basis for this test, since the RTL abstraction allows a natural application of the tentative concepts while the Central Processing Unit (CPU) system is complex enough to allow for more and non-trivial assertions.

Six different version were compared for this test:

1. The CPU system without any assertions
2. The assertions are directly triggered by primary events
3. The primary event sequences are transformed to trigger sequences; the assertions are triggered by the corresponding derived events (non-tentative)
4. Same as above, but instead of non-tentative events, tentative events with a life time[3] of one delay step are used
5. Same as above, but with a life time of two delay steps
6. Same as above, but with a life time of four delay steps

Table 8.2 shows the corresponding results.

Design	**Assertion Coverage**	**Simulation Time [s]**
No Assertions	n/a	0.7
No EvalUpdate	4096	1.1
No Tentative	4096	2.4
Tentative (1)	4096	100.1
Tentative (2)	4096	134.8
Tentative (4)	4096	195.8

Table 8.2: Impact of Evaluate-Update and Tentative Matching

As can be seen, the impact of the assertions without evaluate-update shows a performance drawback of less than factor 2. The drawback of assertions with evaluate-update are within the range of the results shown in the previous section.

As soon as tentative events are used, the drawback becomes much more pronounced. However, the test results seem to indicate a fixed initial drawback combined with a dynamic drawback that increases directly proportionally to the life time of the used tentative events.

Due to the complexity of the assertions used within the test example, the drawback in a typical design might be lower. In general it can be said that the shorter the trigger sequence in question, the lower the impact on the simulation performance.

[3]The life time of a tentative event denotes the number of delay steps of the corresponding trigger sequence and thus the number of delays between the emission of the tentative token and its confirmation.

8.2.3 Impact of Trigger Mechanism

Since the tests presented in the last section did not show a big difference concerning the drawbacks of the evaluate-update mechanism, another test was performed, geared to get more concrete results in this regard. For this test the CPU system from before was reused, in combination with a complete set of assertions for every single instruction supported by the CPU. Each assertion checks the complete instruction execution (which covers several clock cycles on RTL and is partially pipelined). All in all the complexity of the complete set of assertions exceeds the complexity of the design by far.

In this context the following design variants were compared:

1. The TLM CPU system without any assertions
2. The TLM CPU combined with the corresponding assertions
3. The corresponding RTL implementation of the CPU system without assertions
4. The RTL CPU with manually written assertions (which do not use trigger sequences, and thus also no evaluate-update)
5. The RTL CPU with assertions that have been refined from the TLM versions and thus use trigger sequences and evaluate-update

The corresponding results are presented in table 8.3.

Design	**Assertion Coverage**	**Simulation Time [s]**
TLM (no assertions)	n/a	0.3
TLM (assertions)	917504	79.7
RTL (no assertions)	n/a	11.4
RTL (manual assertions)	917504	4728
RTL (refined assertions)	917504	2728

Table 8.3: Impact of Trigger Mechanism

As can be seen, the performance drawback for both TLM and RTL assertions is quite big due to the high complexity of the assertions being used. Another result is the obvious advantage of the trigger mechanism (which uses the evaluate-update mechanism) over the version without this feature.

This seems to contradict the results of the previous test where the drawback of assertions using the evaluate-update mechanism was about three times as big as that of assertions without this feature.

On second thought however, it becomes clear why this test shows the opposite result. The assertion refinement process detects whether several assertions use the same trigger sequence and instatiates only one version of this sequence. Only this one sequence will then have to be evaluated during the simulation and produce evaluation threads. If, on the other hand, all assertions merge the trigger sequence delay conditions with their own, then all operators and all logical computations have to be duplicated many times with each duplicate producing its own evaluation threads.

Additionally, if trigger sequences are used, the main sequences are triggered less often in most cases. As a result the computation effort per assertion decreases (assuming that assertions share their trigger sequences).

8.2.4 Results

As shown in the various tests above, the performance impact of most assertions is within an acceptable range. The use of tentative matching has a higher impact on the simulation performance, but on the other hand even this impact might be acceptable for smaller designs, and additionally this feature allows checking of TLM-like correlations also for RTL designs which was not directly possible before. Thus, there is no other solution which this approach could be compared against.

Finally, it was shown that refined assertions might result in a big performance gain when compared to manually written RTL assertions. Thus, using the refinement process not only allows to specify assertions for a given design only once (and additionally on a higher level of abstraction that is easier to understand), but also leads to better simulation performance in case of complex systems.

9 Conclusion and Outlook

This thesis presented a novel approach for freely transforming assertions between several abstraction levels, including register transfer level and the various transaction levels above. After gathering the necessary requirements for the transformation process, the existing assertion language UAL that is capable of modeling multi level assertions was enhanced by additional functionality especially targeted at supporting the transformation process. A refinement language was introduced that allows the specification of transformation rules in order to refine an existing UAL assertion monitor from one level of abstraction to another and the reuse of existing transformation rules for other assertions. A formal model for the enhanced assertion language in the shape of a high-level colored petri net was introduced in order to detail the underlying semantics. The existing framework of the UAL language was extended to support both the new functionality included in the language and the direct transformation of existing assertions by using the new transformation descriptions. A detailed application example was given in order to show how to categorize several classes of transformations. Using this information, transformation rules can be written specifically for certain reocurring tasks, which eases reuse of these rules.

Parts of the results of this work were already prepublished at several conferences, namely the following papers [27], [29], [30], [28], [31], [26], [57], [32], [58], [59], [60].

Of these publications, the paper "Requirements and Concepts for Transaction Level Assertion Refinement" [60] was awarded the "Best Paper Award Bronze Medal"

Students who contributed to this work are listed as co-authors of the publications mentioned above.

The scientific contribution of this work can be summarized as follows:

- A declarative language for the specification of assertion transformations covering all TL sublevels as well as RTL
- Definition of the concept of structure preservation for additional consistency between original and generated assertion
- Definition of an RTL representation of transactions useable for assertion refinement

- Introduction of the concept of derived events for the handling of RTL transactions
- Definition of the tentative matching concept that allows the correct handling of RTL transactions while handling the uncertainty of transaction occurrence
- Specification of the correct interaction between tentative matching and existing assertion evaluation modes
- Definition of a method for refining constrained TL transactions to RTL and of several possible interpretations of the resulting RTL transactions
- Definition of the implicit timing concept that allows the transformation of untimed assertions to timed assertions and vice versa
- Specification of transactor notations that allow the easy transformation between transaction based and signal based interfaces
- Categorization of transformation tasks that allows a flexible modeling of transformation descriptions between the covered abstraction levels that can be easily reused for different assertions
- A formal representation of assertion behavior including tentative matching based on high level colored petri nets

The concepts were validated based on a framework consisting of the following components:

- Enhanced UAL Library written in SystemC
- Enhanced UAL Parser written in C++ (used for the following two parts)
- Enhanced UAL Compiler written in Mako and Python
- UAL Refinement Generator written in Mako and Python

The validation was done using several designs, including the timer component of the LEON2 processor, a complete CPU subsystem, and an FFT algorithm.

The current work is able to transform given UAL assertions to every other (supported) abstraction level. However, the binding between assertion monitor and design is not transformed. Further work has to provide a way of automatically adapting the assertion binding as well, if necessary based on additional specifications about the design.

In addition to this, further work includes the following aspects:

1. It has been shown that several parts of the transformation process follow very regular patterns. Examples for this are the timing transformation from one timed TL sublevel to another. By providing one or several library files that capture the required functionality (if necessary making use of parameterization for the transformation rules), a complete package of standard transformation tasks could be provided which only requires slight adaptation to the specific properties of the assertion in question.

2. The current transformation approach is not yet capable of correctly handling drastic changes in the structure of the design monitored by the assertions. Examples for these changes include the use of a pipelined or superscalar CPU architecture on RTL while the TL model uses a sequential evaluation. Additional research is necessary of how to take these changes into account and incorporate this into the assertion transformation.

3. This transformation approach only covers the various TL sublevels and RTL. It might be sensible to include higher and / or lower abstraction levels as well (specification level and gate level respectively). Higher level assertions would be able to enhance the verification process even more, since errors can be detected even earlier, while gate level assertions can be included in the synthesized chip itself later, which allows the check of assertion violations even after the product has been delivered to the customer. Including additional abstraction levels would require additional research though, since these levels include very different requirements for an assertion language and, as a consequence, also for the transformation process.

4. At the beginning of this thesis no assertion language provided all features required for a complete transformation approach. Even the language this work is based on needed several feature extensions. On the other hand, there are ongoing attempts to enhance established RTL assertion languages towards TL support. Since this process is not complete yet, it might be sensible to influence the development of these languages so that they include the necessary features for assertion transformation as well. In this case, it is possible to also apply the approach presented in this work to these other assertion languages.

Bibliography

[1] R. Hodgson, "The X-Model: A Process Model for Object-Oriented Software Development," *Fourth International Conference "Software Engineering and its Application"*, 1991.

[2] Mentor Graphics, *Advanced Verification Methodology Cookbook*. Mentor Graphics, 2006. [Online]. Available: http://www.mentor.com/products/fv/_3b715c/

[3] SPiRiT Consortium, "SPiRiT." [Online]. Available: http://www.spiritconsortium.org/

[4] C. Karfa, D. Sarkar, C. Mandal, and C. Reade, "Hand-in-hand verification of high-level synthesis," in *GLSVLSI '07: Proceedings of the 17th great lakes symposium on Great lakes symposium on VLSI*. New York, NY, USA: ACM Press, 2007, pp. 429–434.

[5] W. Ecker, "A Classification of Design Steps and their Verification," in *Proceedings of the Conference on European Design Automation*, Brighton, England, 1995, pp. 536–541.

[6] D. Parker, *Transactor generation using the CY Language*. [Online]. Available: http://www10.edacafe.com/link/display_detail.php?link_id=16239

[7] IEEE Computer Society, *SystemVerilog LRM P1800*. [Online]. Available: http://www.ieee.org

[8] Accellera, *SystemVerilog LRM*. [Online]. Available: http://www.systemverilog.org/SystemVerilog_3.1a.pdf

[9] ——, *Accellera PSL v1.1 LRM*. [Online]. Available: http://www.eda.org/vfv/docs/PSL-v1.1.pdf

[10] J. Havlicek and Y. Wolfsthal, "PSL AND SVA: TWO STANDARD ASSERTION LANGUAGES ADDRESSING COMPLEMENTARY ENGINEERING NEEDS," *Design and Verification Conference (DVCON)*, 2005.

[11] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel, "Simulation-Guided Property Checking Based on Multi-Valued AR-Automata," 2001.

[12] R. J. Weiss, J. Ruf, T. Kropf, and W. Rosenstiel, "Efficient and Customizable Integration of Temporal Properties into SystemC," Lausanne, Switzerland, September 2005.

[13] D. Lettnin, R. J. Weiss, A. Braun, J. Ruf, and W. Rosenstiel, "Temporal Properties Verification of System Level Design," Erfurt, Germany, September 2005.

[14] P. M. Peranandam, R. J. Weiss, J. Ruf, and T. Kropf, "Transactional Level Verification and Coverage Metrics by Means of Symbolic Simulation," February 2004.

[15] A. Kasuya and T. Tesfaye, "Verification Methodologies in a TLM-to-RTL Design Flow," *DAC*, 2007.

[16] A. Kasuya, T. Tesfaye, and E. Zhang, "Native SystemC Assertion mechanism with transaction and temporal assertion support," *EDA Tech Forum*, 2006.

[17] B. Niemann and C. Haubelt, "Assertion Based Verification of Transaction Level Models," in *ITG/GI/GMM Workshop*, vol. 9, Dresden, Germany, February 2006, pp. 232–236.

[18] ——, "Assertion Based Verification of Transaction Level Models," in *MBMV*, 2006.

[19] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, May 1994.

[20] R. Meyer, J. Faber, and A. Rybalchenko, "Model checking duration calculus: A practical approach," in *Theoretical Aspects of Computing - ICTAC 2006*, ser. LNCS, K. Barkaoui, A. Cavalcanti, and A. Cerone, Eds., vol. 4281, 2006, pp. 332–346. [Online]. Available: http://csd.informatik.uni-oldenburg.de/~jfaber/dl/MeyerFaberRybalchenko2006.pdf

[21] J. Faber and R. Meyer, "Model checking data-dependent real-time properties of the european train control system," in *Formal Methods in Computer Aided Design, 2006. FMCAD '06.* IEEE Computer Society Press, Nov. 2006, pp. 76–77.

[22] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, "Logic of Constraints: A Quantitative Performance and Functional Constraint Formalism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2004.

[23] ——, "Automatic trace analysis for logic of constraints," in *DAC '03: Proceedings of the 40th conference on Design automation.* New York, NY, USA: ACM Press, 2003, pp. 460–465.

[24] T. Peng and B. Baruah, "Using Assertion-based Verification Classes with SystemC Verification Library," *Synopsys Users Group, Boston*, 2003.

[25] J. T. Inc., "Native SystemC Assertion (NSCa)," 2005. [Online]. Available: http://www.jedatechnologies.net

[26] W. Ecker, V. Esen, M. Hull, T. Steininger, and M. Velten, "XML-Based Assertion Generation," in *15th IP-Based SoC Design Conference & Exhibition (IP/SOC)*, Grenoble, France, December 2006, pp. 359–364.

[27] W. Ecker, V. Esen, J. Smit, T. Steininger, and M. Velten, "Implementation of a SystemC Assertion Library," in *14th IP-Based SoC Design Conference & Exhibition (IP/SOC)*, Grenoble, France, December 2005, pp. 9–13.

[28] ——, "IP Library For Temporal SystemC Assertions," in *Forum on Specification & Design Languages (FDL)*, Darmstadt, Germany, September 2006, pp. 301–308.

[29] W. Ecker, V. Esen, M. Hull, T. Steininger, and M. Velten, "Execution Semantics and Formalisms for Multi-Abstraction TLM Assertions," in *4th International Conference on Formal Methods and Models for Co-Design (MEMOCode)*, Napa, California, July 2006, pp. 93–102.

[30] ——, "Requirements and Concepts for Transaction Level Assertions," in *24th International Conference on Computer Design (ICCD)*, California, USA, October 2006.

[31] ——, "Specification Language for Transaction Level Assertions," in *11th IEEE International High Level Design Validation and Test Workshop (HLDVT)*, Monterey, California, November 2006, pp. 77–84.

[32] ——, "Implementation of a Transaction Level Assertion Framework in SystemC," in *Design, Automation and Test in Europe (DATE)*, Nice, France, April 2007.

[33] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, *Verification Methodology Manual for SystemVerilog.* New York, USA: Springer Science+Business Media, 2006.

[34] Cadence, *Unified Verification Methodology.* [Online]. Available: http://www.cadence.com/whitepapers/4442_Unified_VerificationMethodology_WP1.pdf

[35] H. Shim, S.-H. Lee, Y.-S. Woo, M.-K. Chung, J.-G. Lee, and C.-M. Kyung, "Cycle-accurate Verification of AHB-based RTL IP with Transaction-level System Environment," in *International Symposium on VLSI Design, Automation and Test*, 2006, pp. 1–4.

[36] R. Jindal and K. Jain, "Verification of transaction-level systemc models using rtl testbenches," in *First ACM and IEEE International Conference on Fromal Methods and Models for Co-Design (MEMOCODE)*, Mont Saint-Michel, France, June 2003, pp. 199–203.

[37] N. Bombieri and F. Fummi, "On the Automatic Transactor Generation for TLM-based Design Flows," in *11th IEEE International High Level Design Validation and Test Workshop (HLDVT)*, Monterey, California, November 2006, pp. 85–92.

[38] F. Balarin and R. Passerone, "Functional Verification Methodology Based on Formal Interface Specification and Transactor Generation," in *9th International Conference on Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2006, pp. 1013–1018.

[39] N. Bombieri, A. Fedeli, and F. Fummi, "On PSL Properties Re-use in SoC Design Flow Based on Transaction Level Modeling," in *6th International Workshop on Microprocessor Test and Verification (MTV)*, November 2005.

[40] N. Bombieri, F. Fummi, and G. Pravadelli, "On the Evaluation of Transactor-based Verification for Reusing TLM Assertions and Testbenches at RTL," in *9th International Conference on Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2006.

[41] OMG, *Model Driven Architecture*. [Online]. Available: http://www.omg.org/mda/

[42] T. Mens, K. Czarnecki, and P. V. Gorp, "A taxonomy of Model Transformations," in *International Workshop on Graph and Model Transformation (GraMoT)*, 2005.

[43] N. Bombieri, F. Fummi, and G. Pravadelli, "A Methodology for Abstracting RTL Designs into TL Descriptions," in *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, July 2006.

[44] Mentor Graphics, *Catapult C*. [Online]. Available: http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/index.cfm

[45] W. Ecker, V. Esen, T. Steininger, and M. Lis, "A Case Study in Rule-Based Modeling," in *14th IP-Based SoC Design Conference & Exhibition (IP/SOC)*, Grenoble, France, December 2005.

[46] ——, "A False Dichotomy? High-Level Synthesis and Control: A CPU Case Study," in *Design & Verification Conference & Exhibition (DVCon)*, San Jose, USA, February 2006.

[47] F. Oquendo, "Arl: an architecture refinement language for formally modelling the stepwise refinement of software architectures," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–20, September 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1022494.1022517

[48] T. Mossakowski, D. Sannella, and A. Tarlecki, "A simple refinement language for casl." in *WADT*, 2004, pp. 162–185.

[49] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Berlin, Heidelberg, and New York: Springer-Verlag, 1997, vol. 1241, pp. 220–242.

[50] H. Foster, "Techniques for higher-performance Boolean equivalence verification," *The Hewlett-Packard Journal*, pp. 30–38, August 1998.

[51] V. Esen, *A New Assertion Language Covering Multiple Levels of Abstraction*, to be published 2008.

[52] V. Stolz and F. Huch, "Runtime verification of concurrent haskell programms," 2004. [Online]. Available: citeseer.ist.psu.edu/stolz05runtime.html

[53] A. Bauer, M. Leucker, and C. Schallhart, "Monitoring of real-time properties," in *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, ser. Lecture Notes in Computer Science, S. Arun-Kumar and N. Garg, Eds., vol. 4337. Berlin, Heidelberg: Springer-Verlag, Dec. 2006.

[54] Python Software Foundation (PSF), *Python Programming Language*. [Online]. Available: http://www.python.org

[55] Mako Templates for Python, *Mako Templates for Python*. [Online]. Available: http://www.makotemplates.org

[56] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Professional Computing Series, 2005.

[57] W. Ecker, V. Esen, M. Hull, T. Steininger, and M. Velten, "A Prototypic Language for Transaction Level Assertions," in *Design & Verification Conference & Exhibition (DVCon)*, San Jose, California, February 2007.

[58] W. Ecker, V. Esen, T. Steininger, and M. Velten, "On the Application of Transaction Level Assertions," in *Ekompass-Workshop (edaWorkshop)*, Hannover, Germany, June 2007.

[59] ——, "Requirements and Concepts for Transaction Level Assertion Refinement," *IFIP International Federation for Information Processing*, vol. 231, pp. 1–14, May 2007.

[60] ——, "Requirements and Concepts for Transaction Level Assertion Refinement," in *International Embedded Systems Symposium (IESS)*, Irvine, California, USA, May 2007.

List of Acronyms

ABV

Assertion **B**ased **V**erification

AHB

Advanced **H**igh-performance **B**us

AOP

Aspect **O**riented **P**rogramming

AVM

Advanced **V**erification **M**ethodology

BFM

Bus-**F**unctional **M**odel

CA

Cycle **A**pproximate

CASL

Common **A**lgebraic **S**pecification **L**anguage

CBS

Cycle **B**ased **S**imultaneity

CC

Cycle **C**allable

CPU

Central **P**rocessing **U**nit

CTL

Computation **T**ree **L**ogic

DC

Duration **C**alculus

DUT

Design **U**nder **T**est

DUV

Design **U**nder **V**erification

EBNF

Extended **B**ackus-**N**aur **F**orm

EBS

Event **B**ased **S**imultaneity

EDA

Electronic **D**esign **A**utomation

EFSM

Extended **F**inite **S**tate **M**achine

ESL

Electronic **S**ystem-**L**evel

FLTL

Finite **L**inear **T**emporal **L**ogic

FSM

Finite **S**tate **M**achine

HDL

Hardware **D**escription **L**anguage

HDVL

Hardware **D**escription and **V**erification **L**anguage

HLCPN

High-**L**evel **C**olored **P**etri **N**et

HW

Hard**W**are

IP

Intellectual **P**roperty

LOC

Logic **O**f **C**onstraints

LTL

Linear **T**emporal **L**ogic

MDA

Model **D**riven **A**rchitecture

OSCI

Open **S**ystem**C** **I**nitiative

PSL

Property **S**pecification **L**anguage

PV

Programmer's **V**iew

PVT

Programmer's **V**iew with **T**iming

RTL

Register **T**ransfer **L**evel

SERE

Sequential **E**xtended **R**egular **E**xpression

SoC

System **o**n **C**hip

SVA

System**V**erilog **A**ssertions

SW

Soft**W**are

TBS

Time **B**ased **S**imultaneity

TL

Transaction **L**evel

TLM

Transaction **L**evel **M**odel

UAL

Universal **A**ssertion **L**anguage

UVM

Unified **V**erification **M**ethodology

VCD

Value **C**hange **D**ump

VHDL

Very High Speed Integrated Circuit **H**ardware **D**escription **L**anguage

VMM

Verification **M**ethodology **M**anual

VP

Virtual **P**rototype

XML

E**X**tensible **M**arkup **L**anguage

XSLT

E**X**tensible **S**tylesheet **L**anguage **T**ransformations

Glossary

Assertion Based Verification

Dynamic or formal checking of design properties

Bus Functional Model

A component that translates between an abstract model of a communication protocol and its implementation model

Electronic System Level

A term which describes the industry wide activities for modeling and analyzing systems at a higher level of abstraction while taking both HW and SW into account

Hardware

A silicon implementation

Linear Temporal Logic

A logic which enables the specification of temporal relations between Boolean propositions

Register Transfer Level

Synthesizable, pin and cycle accurate hardware description

Software

A program which is executed on a HW platform

System-on-a-Chip

A system which is fully integrated onto a single chip, with one to four cores, a high-speed bus, a peripheral bus, and several dedicated HW blocks like a display controller, a USB interface, etc.

SystemC

A class library on top of C++ which supports HDL concepts for modeling concurrency and communication

SystemVerilog

HDVL based on and extending the Verilog HDL which includes design features, testbench features, and assertion features

Transactor

A component that translates between an abstract model of a communication protocol and its implementation model (see BFM)

VHDL

The most common hardware description language in European semi-conductor industry

Virtual Prototype

A fully virtual executable model of a System-on-a-Chip (SoC)

A List of Requirements for Assertion Transformation

This appendix provides a short summary of the requirements discussed in Chapter 3. If necessary, a short explanation is given below.

R 1 : *Independence of design behavior from assertions*

R 2 : *Independence of assertion behavior from unrelated design changes*
No global event sensitivity

R 3 : *Independence of assertion behavior from other assertions*
No global event sensitivity

R 4 : *Assertion evaluation during simulation*

R 5 : *Providing of different kinds of coverage data*
Supported coverage information:

- *Vacuous Success: Antecedent sequence not matched*
- *Real Success: Antecedent sequence and consequence sequence both matched*
- *Success: Sum of Vacuous Success and Real Success*
- *Failure: Antecedent sequence matched, consequent sequence not matched*

R 6 : *Support of multiple abstraction levels*
Support of models written on PV, PVT, CA, CC, and RTL

R 7 : *Support of abstraction level mix*
Support of mixed level models with parts modeled on the abstraction levels listed in R 6

R 8 : *Single assertion language for original and transformed assertion*

R 9 : *Ability to detect transactions*

R 10 : *Ability to link to transaction return values and arguments*

R 11 : *Ability to link to design signals*

R 12 : *Support of a transaction reprensentation on RTL*

R 13 : *Support of synchronization based on events*

R 14 : *Support of synchronization based on simulation time*

R 15 : *Support of synchronization based on clock signals*

R 16 : *Ability to detect transaction relations*
Supported relations:

- *Consecutive: Second transaction starts after the end of the first one*
- *Overlapped: Second transaction starts before and ends after the end of the first one*
- *Inclusion: Second transaction starts and ends before the end of the first one*

R 17 : *Ability to specify a partial order on events*

R 18 : *Ability to specify a strict partial order on events*
Strict partial order is used to detect missing event occurrences

R 19 : *Ability to specify temporal relations based on simulation time*

R 20 : *Ability to capture of dynamic temporal behavior*
Supported dynamic behavior:

- *Dynamic time delays*
- *Dynamic amount of transaction calls*

- *Dynamic amount of event occurrences*

R 21 : *Ability to specify temporal relations based on clock signals*

R 22 : *Ability to detect pipelining behavior*

R 23 : *Assertion evaluation of SystemC designs*

R 24 : *Support of all SystemC and C++ base types*

R 25 : *Compatibility to OSCI SystemC reference simulation kernel*

R 26 : *Support of OSCI TLM standard*

R 27 : *Support of seamless access of assertions to modules and their internals*

R 28 : *Support of all event types available in SystemC*
Events in question:

- *Signal change events: Issued by a signal after a successful signal assignment*
- *Explicit events: Issued by the programmer (either with a timed delay, at the next delta cycle, or immediately)*
- *Implicit events: Issued by a suspending process to schedule its own wake up (either with a timed delay or at the beginning of the next delta cycle)*

R 29 : *Support of blocking and non-blocking transactions*

R 30 : *Support of more granular time resolution than delta-time*

R 31 : *Support of immediate notification of occurring transactions*
Notification has to happen before any of the variables or signals in the design change their value

R 32 : *Ability to link to design state variables*

R 33 : *Ability to detect assignments of design variables*

R 34 : *Support of flexible sampling of design states*
Support of sampling with the occurrence of any transaction, any event (including clock events), or at any simulation time

R 35 : *Ability to transform between lower and higher levels of abstraction*

R 36 : *Support of multiple abstraction levels*
Transformation to and from abstraction levels listed in R 6

R 37 : *Support of abstraction level mix*
Transformation to and from mixed level assertions with parts modeled on the abstraction levels listed in R 6

R 38 : *Consistency of assertion behavior before and after refinement*

R 39 : *Ability to transform between TL transactions and RTL signal protocols*

R 40 : *Ability to transform between TL events and RTL signal protocols*

R 41 : *Ability to transform between TL transaction arguments and return values on one side and RTL signals on the other side*

R 42 : *Ability to transform between absolute timing, cyclic timing, and clocked timing*

R 43 : *Support of adding / removing timing information*

R 44 : *Support of adding / removing reset behavior*

R 45 : *Ability to handle ambiguous transformations*
Supported Solutions:

- *Warning: Issue a warning message that target assertion might produce wrong results*
- *Strict transformation: Target assertion avoids false positves at the cost of false negatives*

- *Safe transformation: Target assertion avoids false negatives at the cost of false positives*

R 46 : *Support of complex transformation rules*
Rules can be comprised of many basic transformation directives and even other rules

R 47 : *Support of parameterizable transformation rules*

R 48 : *Support of transformation libraries*
Transformation rules can be encapsulated in transformation libraries

B Language Grammar

B.1 Refinement Grammar

refinement_file = *refinement_definition* { *refinement_definition* } ;
(B.1)

refinement_definition =
`"refinement"` *identifier*
[*import_section*]
[*declarations_section*]
[*transactors_section*]
[*rules_section*]
apply_section
`"endrefinement"` ;
(B.2)

library_file = *library_definition* { *library_definition* } ;
(B.3)

library_definition =
`"library"` *identifier*
[*declarations_section*]
[*sequences_section*]
[*properties_section*]
[*transactors_section*]
[*rules_section*]
`"endlibrary"` ;
(B.4)

import_section = *import_declaration* { *import_declaration* } ;
(B.5)

import_declaration = `"import"` *identifier* `";"` ;
(B.6)

declarations_section =
`"declarations"`
[*ports_section*]
[*constants_section*]
[*variables_section*]
`"enddeclarations"` ;
(B.7)

variables_section =
`"variables"`
localvar_declaration { *localvar_declaration* }
`"endvariables"` ;
(B.8)

transactors_section = `"transactors"`
transactor_section { *transactor_section* }
`"endtransactors"` ;
(B.9)

transactor_section = `"transactor"` *identifier transactor_interface*
[*map_section*]
sequence_section
`"endtransactor"` ;
(B.10)

transactor_interface = `"<"` *identifier* `">"` *formal_argument_list* ;
(B.11)

map_section = `"map"`
map_declaration { *map_declaration* }
`"endmap"` ;
(B.12)

map_declaration = *identifier* `"."` (*identifier* | `"RET"`) `"=>"` *identifier* `";"` ;
(B.13)

transactor_instance = *identifier* `"<"` *target_locations_list* `">"`
`"["` *refinement_mode* `"]"`
param_argument_list `";"` ;
(B.14)

rules_section = `"rules"`
rule_section { *rule_section* }
`"endrules"` ;
(B.15)

rule_section = `"rule"` *identifier rule_interface*
rule_specification
`"endrule"` ;
(B.16)

rule_interface = `"<"` *identifier* `">"`
`"("` [*rule_parameters*] `")"` ;
(B.17)

rule_parameters = *rule_parameter* { `","` *rule_parameter* } ;
(B.18)

rule_parameter = *rule_parameter_type identifier* ;
(B.19)

rule_parameter_type = `"declaration"`
| `"name"`
| `"value"`
| `"type"`
| `"expression"` ;
(B.20)

apply_section = `"apply"`
rule_specification
`"endapply"` ;
(B.21)

rule_specification = *rule_item* { *rule_item* } ;
(B.22)

rule_item = *directive_instance*
| *transactor_instance*
| *rule_instance* ;
(B.23)

rule_instance = *identifier* "<" *target_locations_list* ">"
"(" [*rule_arguments*] ")" ";" ;
(B.24)

rule_arguments = *rule_argument* { "," *rule_argument* } ;
(B.25)

rule_argument = *string*
| *expression*
| *event_expression*
| (*identifier* "." *declaration_access*)
| *value* ;
(B.26)

directive_instance = *add_directive*
| *modify_directive*
| *remove_directive* ;
(B.27)

target_locations_list = *target_locations* { "," *target_locations* } ;
(B.28)

target_locations = *identifier*
{ "." (*identifier*
| *number*
| "*") } ;
(B.29)

add_directive = "add" "." *directive_target*
"<" *target_locations_list* ">"
add_specification { *add_specification* } ;
(B.30)

add_specification = "{" [*refines*] *additives* "}" ;
(B.31)

additives = *additive* { *additive* } ;
(B.32)

additive = "additive" [*position_marker*] *directive_string* ";" ;
(B.33)

refines = "refines" ["[" *event_operand* "]"] *directive_string*
{ "," *position_marker* "," *directive_string* } ";" ;
(B.34)

modify_directive = "modify" "." *directive_target*
"<" *target_locations_list* ">"
modify_specification { *modify_specification* } ;
(B.35)

modify_specification = "{" *refines modification* "}" ;
(B.36)

modification = "modification" ["[" *event_operand* "]"]
directive_string ";" ;
(B.37)

remove_directive = "remove" "." *directive_target*
"<" *target_locations_list* ">"
{ *remove_specification* } ; (B.38)

remove_specification = "{" *removal* { *removal* } "}" ; (B.39)

removal = "removal" *directive_string* ";" ; (B.40)

directive_string = (*string* | (*identifier* ["." *declaration_access*]))
{ "," (*string* | (*identifier* ["." *declaration_access*])) } ; (B.41)

position_marker = "(" *identifier* ")" ; (B.42)

directive_target = "boolean_expression"
| "constant"
| "parameter"
| "pass_copy"
| "pass_reference"
| "reset"
| "sensitivity"
| "time_constraint"
| "timer"
| "variable"
| "variable_assignment" ; (B.43)

declaration_access = "name()"
| "type()"
| "value()" ; (B.44)

refinement_mode = "up"
| "down" ; (B.45)

B.2 Basic Monitor Grammar

monitor = "monitor" *identifier*
ports_section
[*constants_section*]
sequences_section
properties_section
verification_section
"endmonitor" ; (B.46)

ports_section = "ports"
port_declaration { *port_declaration* }
"endports" ; (B.47)

port_declaration = *kind type identifier* ["[" *number* "]"]
[*transaction_parameters*] ";" ; (B.48)

constants_section = "constants"
constant_declaration { *constant_declaration* }
"endconstants" ; (B.49)

constant_declaration = *type identifier* "=" *value* ";" ; (B.50)

sequences_section = "sequences"
sequence_section { *sequence_section* }
"endsequences" ; (B.51)

properties_section = "properties"
property_section { *property_section* }
"endproperties" ; (B.52)

verification_section = "verification"
directive { *directive* }
"endverification" ; (B.53)

directive = *directive_kind identifier* "(" [*directive_parameter*] ")"
"=" *property_instance* ";" ; (B.54)

directive_kind = "assert"
| "cover"
| "assert_cover"
| "assume" ; (B.55)

directive_parameter = *severity_level*
"," *string*
["," *reset_event_expr*] ; (B.56)

severity_level = "INFO"
| "WARNING"
| "ERROR"
| "FAILURE" ; (B.57)

reset_event_expr = *trigger_expression* ; (B.58)

property_section = "property" *identifier property_interface*
property_declarations
property_specification
"endproperty" ; (B.59)

property_interface = [*property_mode_list*] *formal_argument_list* ; (B.60)

property_declarations = { *localvar_declaration* } ; (B.61)

property_specification = *implication_property*
| *single_sequence_property*
(B.62)

implication_property = *sequence_instance* "|->" *sequence_instance* ";" ;
(B.63)

single_sequence_property = *sequence_instance* ";" ;
(B.64)

property_instance = *identifier* [*property_mode_list*]
param_argument_list ;
(B.65)

property_mode_list = "[" (([*sequence_mode* ","] *property_mode*)
| *sequence_mode*) "]" ;
(B.66)

property_mode = "Restart"
| "NoRestart"
| "ReportOnRestart"
| "Overlap"
| "Pipe"
| "PipeOrdered"
| "Cover" ;
(B.67)

sequence_section = "sequence" *identifier sequence_interface*
sequence_declarations
sequence_specification
"endsequence" ;
(B.68)

sequence_interface = ["[" *sequence_mode* "]"] *formal_argument_list* ;
(B.69)

sequence_declarations = { *localvar_declaration* } ;
(B.70)

sequence_specification = *delay_operator* { *delay_operator* } ";" ;
(B.71)

delay_operator = "#" *steps sensitivity* "{" *condition* { *action* } "}" ;
(B.72)

steps = *zero_step*
| *multi_step*
| *range_step* ;
(B.73)

zero_step = "0"
| ("{" "0" "}") ;
(B.74)

multi_step = *non_zero_number*
| ("{" *non_zero_number* "}") ;
(B.75)

range_step = "{" *number* ":" *number* "}" ;
(B.76)

sensitivity = "{" [*pos_sensitivity*]
[";" *neg_sensitivity*] "}" ;
(B.77)

condition = *boolean_expression*
["?" *boolean_expression* ":" *boolean_expression*] ;
(B.78)

action = "," *identifier* "=" *localvar_expression* ;
(B.79)

boolean_expression = *expression* ;
(B.80)

accumulator_expression = *expression* ;
(B.81)

timer_expression = *expression* ;
(B.82)

localvar_expression = *expression* ;
(B.83)

sequence_instance = *identifier* ["[" *sequence_mode* "]"] *param_argument_list* ;
(B.84)

sequence_mode = "AnyMatch"
| "FirstMatch"
| "FirstMatchPipe"
| "FirstMatchPipeOrdered" ;
(B.85)

operand = *lastevent*
| *value*
| (*identifier* ["." *identifier*] ["[" *array_index* "]"])
| (*identifier* ["." "RET"]) ;
(B.86)

factor = *operand*
| ("(" *expression* ")")
| (*unary_operator factor*) ;
(B.87)

term = *factor* { *arith_operator factor* } ;
(B.88)

expression = *term* { *boolean_operator term* } ;
(B.89)

pos_sensitivity = *trigger_expression* ;
(B.90)

neg_sensitivity = *trigger_expression* ;
(B.91)

trigger_expression = (*event_expression* ["," *trigger_timer*])
| *trigger_timer* ;
(B.92)

trigger_timer = "timer(" *timer_expression* ")" ;
(B.93)

event_operand = *identifier* ["[" *array_index* "]"] ["'" *event_kind*] ;
(B.94)

event_constraint = "@(" *boolean_expression* ")" ;
(B.95)

event_accumulator = "%(" *accumulator_expression* ")" ;
(B.96)

unary_event_expr = *event_accumulator*
| *event_constraint* ; (B.97)

event_factor = (*event_operand* [*unary_event_expr*])
| ("(" *event_expression* ")" [*unary_event_expr*]) ; (B.98)

event_term = *event_factor* { "&" *event_factor* } ; (B.99)

event_expression = *event_term* { "|" *event_term* } ; (B.100)

lastevent = "$l_event(" *event_operand* ")" ; (B.101)

arith_operator = "&"
| "|"
| "+"
| "-"
| "/"
| "%"
| "*" ; (B.102)

boolean_operator = "&&"
| "||"
| "<="
| ">="
| "!="
| "=="
| "<"
| ">" ; (B.103)

unary_operator = "!"
| "-"
| "~" ; (B.104)

formal_argument_list = "(" [*formal_argument_decl*]
{ "," *formal_argument_decl* } ")" ; (B.105)

formal_argument_decl = ["ref"] [*kind*] [*type*] *identifier*
[*transaction_parameters*] ; (B.106)

localvar_declaration = *type identifier* ";" ; (B.107)

param_argument_list = "(" [*parameter_argument*]
{ "," *parameter_argument* } ")" ; (B.108)

parameter_argument = *identifier* [("'" *event_kind*) | ("." *identifier*)] ; (B.109)

transaction_parameters = "(" *type identifier*
{ "," *type identifier* } ")" ; (B.110)

kind = "state"
| "event"
| "signal"
| "transaction" ;
(B.111)

B.3 Changes/Enhancements of Monitor Grammar

monitor = "monitor" *identifier*
[*import_section*]
[*ports_section*]
[*constants_section*]
[*triggers_section*]
[*sequences_section*]
[*properties_section*]
verification_section
"endmonitor" ;
(B.112)

triggers_section = "triggers"
trigger_declaration { *trigger_declaration* }
"endtriggers" ;
(B.113)

trigger_declaration = "trigger" *identifier* "=" *trigger_kind* ";" ;
(B.114)

trigger_kind = *sequence_trigger*
| *property_trigger* ;
(B.115)

sequence_trigger = "sequence" *sequence_instance* ;
(B.116)

property_trigger = "property" *property_instance* ;
(B.117)

trigger_timer = "timer" ["[" *event_operand* "]"]
"(" *timer_expression* ")" ;
(B.118)

event_kind = "START"
| "END"
| "POS"
| "NEG"
| "CH"
| "ATTEMPT"
| "MATCH"
| "NOTMATCH"
| "SUCCESS"
| "FAIL" ;
(B.119)

action = "," ((*identifier* "=" *localvar_expression*) | *ual_action*) ;
(B.120)

ual_action = "$confirm" ; (B.121)

ual_value = "$time"
| "$thread_id"
| "$sthread_id"
| ("$delta_t" ["[" *event_operand* "]"]) ; (B.122)

kind = "state"
| "event"
| "signal"
| "transaction"
| "trigger" ; (B.123)

B.4 Common Grammar

string = """ *string_characters* """ ; (B.124)

filename = (*non_digit* | *digit* | "-" | ".")
{ *non_digit* | *digit* | "-" | "." } ; (B.125)

event_kind = "START"
| "END"
| "POS"
| "NEG"
| "CH" ; (B.126)

type = *object_type* [*template*] ; (B.127)

template = "<" ((*object_type* [*template*]) | *value*)
{ "," ((*object_type* [*template*]) | *value*) } ">" ; (B.128)

object_type = *cpp_type*
| *sc_type*
| *ual_type* ; (B.129)

ual_type = "callback" ; (B.130)

cpp_type = *int_type*
| "double"
| ("long" "double")
| "float"
| "bool"
| "void" ; (B.131)

timeunit_definition = "timeunit" "=" *time_number time_unit* ; (B.132)

int_type = [*sign_type*] *cpp_integer* ; (B.133)

cpp_integer = ("long" "int")
| ("long" "long" "int")
| ("long" "long")
| "long"
| ("short" "int")
| "short"
| "int" ; (B.134)

sign_type = "signed"
| "unsigned" ; (B.135)

sc_type = "sc_signal"
| "sc_int"
| "sc_uint"
| "sc_event"
| "sc_event_queue"
| "sc_time"
| "sc_bit"
| "sc_bv" ; (B.136)

value = *integer_value*
| *real_value*
| ("'" *bit_value* "'")
| (""" *bv_value* """)
| *boolean_value*
| *ual_value*
| ("(" *array_value* ")") ; (B.137)

array_value = (*integer_value*
| *real_value*
| ("'" *bit_value* "'")
| (""" *bv_value* """)
| *boolean_value*
| *ual_value*)
{ "," (*integer_value*
| *real_value*
| ("'" *bit_value* "'")
| (""" *bv_value* """)
| *ual_value*
| *boolean_value*) } ; (B.138)

ual_value = "$time"
| "$thread_id"
| "$sthread_id"
| "$delta_t" ; (B.139)

integer_value = ["-"] *number* ; (B.140)

real_value = ["-"] *number* "." *number* ; (B.141)

bv_value = *bit_value* { *bit_value* } ; (B.142)

bit_value = "1"
| "0" ; (B.143)

boolean_value = "true"
| "false" ; (B.144)

time_number = "100"
| "10"
| "1" ; (B.145)

time_unit = "s"
| "ms"
| "us"
| "ns"
| "ps"
| "fs" ; (B.146)

array_index = (*identifier* ["." *identifier*])
| *number* ; (B.147)

number = "0"
| *non_zero_number* ; (B.148)

non_zero_number = *non_zero_digit* { *digit* } ; (B.149)

identifier = *non_digit* { *digit* | *non_digit* } ; (B.150)

string_characters = { *digit*
| *non_digit*
| *special_character* } ; (B.151)

digit = "0"
| *non_zero_digit* ; (B.152)

non_zero_digit = "1"
| "2"
| "3"
| "4"
| "5"
| "6"
| "7"
| "8"
| "9" ; (B.153)

```
non_digit = "_"
          | letter ;
```
(B.154)

```
letter = uppercase_letter
       | lowercase_letter ;
```
(B.155)

```
uppercase_letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                 | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                 | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                 | "V" | "W" | "X" | "Y" | "Z" ;
```
(B.156)

```
lowercase_letter = "a" | "b" | "c" | "d" | "e" | "f" | "g"
                 | "h" | "i" | "j" | "k" | "l" | "m" | "n"
                 | "o" | "p" | "q" | "r" | "s" | "t" | "u"
                 | "v" | "w" | "x" | "y" | "z" ;
```
(B.157)

```
special_character = "-" | "/"
                  | "+" | "*" | "<"
                  | ">" | "=" | "$"
                  | "!" | "^" | "~"
                  | "{" | "}" | "["
                  | "]" | "(" | ")"
                  | "?" | "." | "'"
                  | "|" | "&" | ","
                  | ";" | ":" | "%" ;
```
(B.158)