TECHNISCHE
UNIVERSITÄT
MÜNCHEN

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

PhD-Thesis

A New Assertion Language Covering Multiple Levels of Abstraction

Volkan Esen

Lehrstuhl für Entwurfsautomatisierung
der Technischen Universität München

A New Assertion Language Covering Multiple Levels of Abstraction

Volkan Esen

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.-Ing. Wolfgang Utschick

Prüfer der Dissertation:

1. Hon.-Prof. Dr.-Ing. Wolfgang Ecker
2. Univ.-Prof. Dr. rer. nat. Franz J. Rammig,
   Universität Paderborn

Die Dissertation wurde am 14.01.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 29.05.2008 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: Volkan Esen

Zusammenfassung

Im Rahmen dieser Arbeit wurde eine neue Assertionsprache und Verifikationsumgebung entwickelt, welche die Spezifizierung und Validierung von temporalen Modelleigenschaften über Abstraktionsebenen hinweg ermöglicht. Die Entwicklung der Sprache ist notwendig weil existierende Assertionsprachen die Anwendung auf nicht synthetisierbare abstrakte Modelle nur eingeschränkt ermöglichen. Die formale Semantik der Sprache wurde durch die Abbildung auf ein abstraktes gefärbtes Petrinetz definiert. Die vorteilhafte Anwendbarkeit der Sprache wurde durch einen Compiler und einen speziellen Assertionkernel in mehreren Anwendungen nachgewiesen.

Abstract

In this work, a new assertion language and verification framework has been developed. It enables the specification and validation of temporal properties accross different abstraction levels. This new language is required because existing assertion languages do only offer limited support for the verification of abstract, non-synthesizable models. The semantics of the language is defined by a mapping onto a high-level colored petri net. The advantageous applicability of this language has been shown over several applications by using a compiler-based framework and a specific assertion kernel.

Acknowledgment

This work was accomplished during my affiliation with Infineon Technologies AG - at the department IFAG COM BTS MT SD, headed by Dr. Matthias Bauer - in cooperation 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 want to thank Professor Ulf Schlichtmann for giving me the opportunity to conduct this thesis at his institute. Also I want to thank Dr. Matthias Bauer and our team for a great collaboration.

I especially want to thank my doctoral advisor Professor Wolfgang Ecker and Infineon Technologies for making this thesis possible. Wolfgang Ecker always pushed me to go beyond my limits being a great mentor. We had many fruitful discussions and brain storming sessions which were vital to the success of this work.

Furthermore, I want to thank Professor Franz Rammig for being the co-advisor of this work.

I am also grateful to Professor Manfred Glesner and Dr. Thomas Hollstein from Technische Universität Darmstadt at the institute of Microelectronic Systems for their continued support after my graduation from university.

It takes a huge amount of patience, perseverance, and most of all support to accomplish a doctoral thesis. Therefore, I want to express my deepest gratitude to Elnura, my beloved wife. She supported me in all possible ways, bearing the many hours I had to stay away to develop the ideas introduced in this work.

I also want to thank my family. They made it possible for me to come this far. Without their great support and dedication none of this would have been possible.

Besides my family, I also want to thank my dear friend Michael Velten who joined me even through nights to write most of the publications on the topic of this work and managed to keep me calm as the deadlines came closer. Also, I want to thank all my other friends whom I had to put off for so many times in the past years.

Volkan Esen
Munich, June 26, 2008

Contents

List of Tables

List of Figures

List of Listings

1 Introduction

1.1 The Ubiquity of Embedded Systems

Over the past decades, embedded systems have become an integral part of our society. This is due to the fast evolution of the semi-conductor industry, which enables more and more features integrated on a single chip for continuously decreasing prices. The application of embedded systems has a large scope. The automotive industry for example shows a trend towards integrating more and more electronic systems within a car. Whole entertainment systems are being integrated. Engine control systems are developed that allow for the most efficient use of gas in order to reduce carbon-dioxide emission. Also safety critical applications of embedded systems are being developed like break by wire or inter car communication for avoiding collisions through early warning systems. The communication business is solely based on embedded systems. The fast development and evolution of cellular phones, PDAs, DSL, and so forth, shows how much embedded systems have become a part of peoples lives. In general, information technology based on embedded systems is fundamental for keeping our industry up and running.

1.2 System Complexity

The key driver for the immense success of embedded systems are market forces that foster the development of cheaper products based on engineering genius and walking along a learning curve with an exponential slope. The empirical observation made by Gordon E. Moore in 1965 which states that the number of transistors per chip doubles every two years[1] at a minimum level of costs [6] was and is still valid. This also means that the complexity of a chip in terms of transistors shows an exponential rise over time. As indicated by the International Technology Roadmap for Semiconductors (ITRS), it can be expected that the on-chip complexity will increase further measured in terms of the number of integrated components, at least until the year 2020.

[1] In some cases an even stronger statement can be made; selected devices as for instance CPUs, double complexity within 18 months [5].

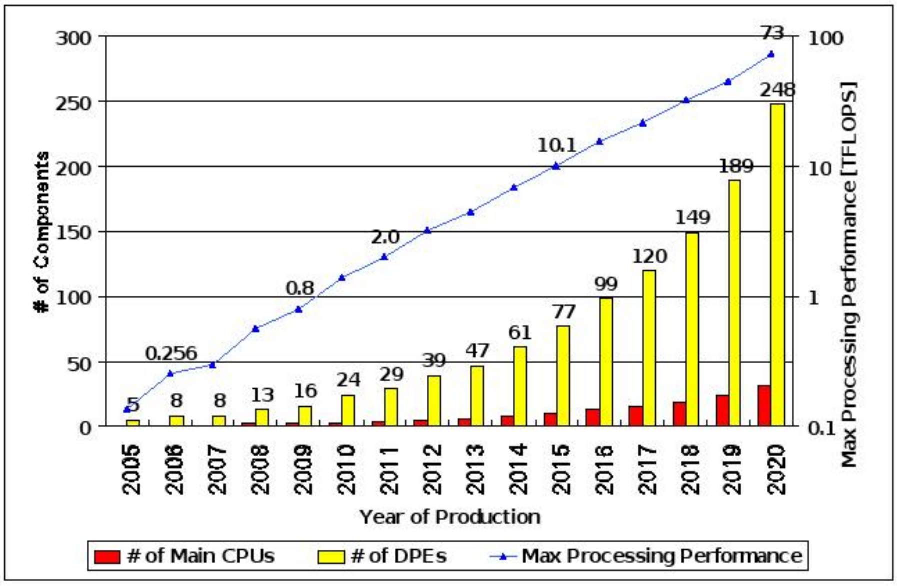


Figure 1.1: Forecast of number of components over time [1]

Figure 1.1 shows a tremendous increase of Data Processing Engine (DPE) elements over time. A DPE represents dedicated Hardware (HW) for implementing a specific task. Also the number of main Central Processing Unit (CPU) elements shows an increase, though the rate is much lower.

Given this development it can easily be seen that a company needs to put very efficient development and production processes in place in order to obtain and keep a strong competitive position in the market, because the effort involved to develop more and more complex systems keeps increasing as well.

The most common approaches taken by semi-conductor companies for tackling the problem of increasing complexity are on one hand the reuse of existing matured system components and on the other hand the early exploration of different architectures based on structurally more abstract executable descriptions of the targeted system, partially with reduced functionality. These systems include both reused and newly developed system components. Figure 1.2 shows a slightly adapted version of the V-Process-Model [2] which defines a development process which ensures the quality of a HW product. In order to speed up the development of the implementation model, early effort is spent for the architectural model. While the implementation model represents a synthesizable description at the Register Transfer Level (RTL), the architectural model is described by means offered in the Electronic System Level (ESL) domain.

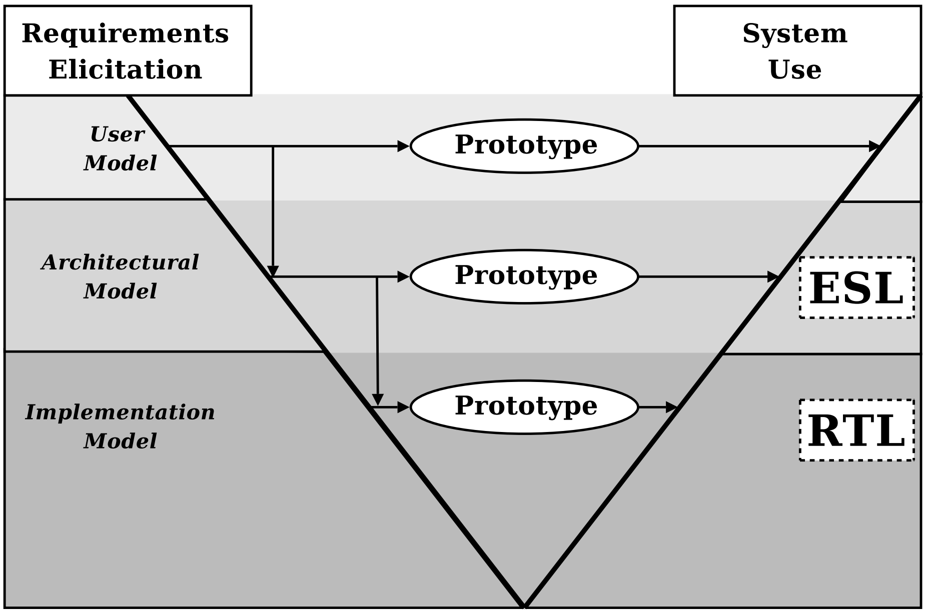


Figure 1.2: The Model of the Design Process: The V-Process-Model [2]

ESL describes the industry wide activities on modeling and analyzing systems at a higher than RTL abstraction, taking both HW and Software (SW) into account. While this term is newly evolving, it actually describes ongoing activities of the past years. However, it is more focused at present due to the increasing complexity of even abstract model descriptions. The progress is reflected by the ongoing standardization activities by Open SystemC Initiative (OSCI), such as the SystemC modeling language [7] and the TLM standard [8]. A quite accurate definition of ESL has been given in [9] which states that ESL is ”the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner, while meeting necessary constraints”.

The rationale behind investing great effort in high-level models and their analysis is straight forward. During the architectural exploration phase, many decisions have to be taken with regard to HW/SW partitioning. Therefore, it is necessary to analyze the high-level model in terms of throughput and even in terms of power in order for the best decision to take. Performing such analysis steps only after having completed the implementation model (i.e., the RTL model) usually is too late and too time consuming. Also the available analysis tool set requires too much computation time on this level to perform full system analysis. Furthermore, uncovering performance and power bottlenecks after the completion of the RTL implementation would require a full redesign in the worst case. This implies that almost all steps would have to be repeated. Such a situation is undesirable and not economic.

In addition to making architectural decisions, a behavioral model is also used for enabling the development of SW at an early stage. The SW is then executed on the behavioral model of the system. This allows for the SW to mature while the implementation model is still under way.

The whole process of developing high-level models of a target system is also described as virtual prototyping and the resulting model is called Virtual Prototype (VP).

It can be observed that the rise in chip complexity drives the need for new methods in ESL both for design and verification [10]. More and more abstract components have to be developed and integrated to an abstract representation of a System-on-a-Chip (SoC) including analog, mixed signal parts as well as the conventional digital domain. Effective methods in ESL will become challenging. The importance of these methods can be compared to the new methodologies (e.g., RTL-synthesis, linting, formal verification) which have emerged when transitioning from the gate-level towards RTL.

1.3 The Role of Verification

The rise of complexity during the past years has also brought up the issue of verification. In contrast to pure SW, a bug in a taped out HW circuit is hard to fix. Usually it is not easily possible to work around a bug by adapting the corresponding bits in the SW which is supposed to be executed on the HW. Not only that a bug in a taped out chip might scare away customers and hence, decrease a companies revenue by orders of magnitude; the fact that an undetected bug makes it to an end product in a safety critical environment can cost lives and cause other fatal consequences. To dampen such a high risk serious effort is spent in testing a taped out prototype of an SoC and upon detecting severe bugs a full re-spin of the production cycle is required, which in turn involves high cost and a bigger risk to miss the time-to-market window. Considering the fact that most hard-to-detect bugs are still introduced during the development of the RTL implementation rather than in the production phase, it was clear that functional verification had to be emphasized in the design automation activities. The validity of Moore's Law has led to very complex products, which could not be verified efficiently anymore by simulating directed tests of a system.

Figure 1.3 shows the verification technology landscape, which has evolved once verification became a hot topic. Each different technology is mapped onto the domains where it is applied best.

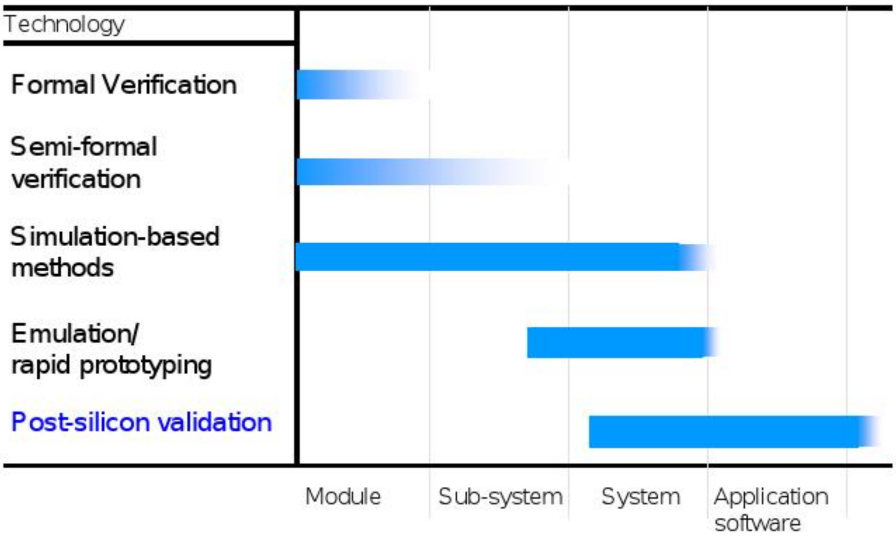


Figure 1.3: Verification Technology Landscape according to ITRS [3]

1.3.1 Formal Verification

Formal verification techniques rely on the exploration of a model on a mathematical basis. Proofs can be performed to determine, whether a design under scrutiny fulfills its specification. The most successful approach here is the application of the so called Bounded Model Checking (BMC) [11],[12], which exists in different variants. Formal verification techniques are applied mostly at the block-level. Since the underlying key idea is the mathematical analysis of transition paths through the design state space, the problem of state space explosion hinders the applicability to big complex components. Since the size of the state space grows exponentially with the number of states, an exploration of its transition paths is not computable due to insufficient computing power and memory. One of the major goals in formal verification related research is the enhancement of the applied algorithms and the development of new techniques which attempt to reduce the complexity of the original implementation model by means of abstraction. These techniques differ from abstraction techniques applied in the ESL domain.

1.3.2 Semi-Formal Verification

Semi-formal verification approaches bring together both dynamic verification (simulation) and formal verification techniques. Here, the Design Under Verification (DUV) is stimulated by a testbench. As soon as critical states (e.g., a counter has reached its maximum value) are reached, a formal analysis is started with usually a small bound. The difficulty here is that this methodology is critical with regard to verification management because this methodology does not allow proving the absence of bugs in the DUV in contrast to purely formal approaches. Applying a testbench always means that the results can not be generalized. Therefore, semi-formal techniques can be characterized as "bug-hunters".

1.3.3 Simulation Based Methods

Due to the limited power of purely formal techniques, simulation based techniques can not be replaced entirely. Dynamic verification still remains the most applicable methodology for verification. As Figure 1.3 also indicates, simulation based methods are applied at the block-level as well as at the system-level. Yet, powerful methodologies have emerged which allow to exercise a design thoroughly but still not exhaustively.

Constrained Random Testbenches

Instead of "just" simulating directed tests, techniques are applied which stimulate a DUV with randomized inputs. The randomization is constrained to provide a general direction for a simulation. In this context another technique is applied which yields measures for verification management. This technique is called "Coverage". Coverage results in general show how much a design is exercised. Different scopes for coverage exist. Coverage can be:

- **Code Coverage**: Yield how many times a certain part of code has been exercised.
- **State Coverage**: Yield how many times a certain state variable has taken on a certain value or a series of values.
- **Cross Coverage**: Yield how many times a certain state variable has taken on a certain value while a different state variable has taken on a different value.
- **Assertion Coverage**: Yield how many times a temporal behavior specified in terms of an assertion has been encountered.

Coverage results are stored in databases. The results can be used to determine when to stop a fully automated test run and furthermore, they can be used to guide the random stimulus generation such that the parts of the DUV which show less coverage are exercised more.

The effort for writing randomized testbenches is furthermore reduced by using abstraction schemes in the stimuli generation and application. Stimuli are represented in abstract data structures (e.g., an object that represents the information of a picture frame for displaying). The application of stimuli to a DUV is modeled in terms of transactions or sets of transactions. The abstract data structures are driven into a DUV by using these transactions (e.g., sending a picture frame to a display controller). This abstraction is obtained via Bus Functional Model (BFM) elements also referred to as transactors. These BFM components translate between abstraction levels and are connected between a stimulus generator and the DUV. A BFM hides the signal-level protocol details behind method calls (i.e., transactions). The responses of the DUV are again connected to BFMs in order to check them at the same abstraction as the stimuli. Such an abstraction technique also makes it possible to reuse testbenches for an RTL design with its corresponding model in the ESL domain or vice versa. However the reuse is mostly restricted to the stimuli generation part and response checking. Usually the coverage definitions need to be altered severely or even have to be written from scratch, since the corresponding implementation of a model at different abstraction levels follows different modeling paradigms (i.e., structures of lower level implementations are not available at higher levels).

Assertions

Another major development in RTL verification was introduced under the term "Assertion Based Verification (ABV)" [13]. ABV enables the application of SW development principles to RTL modeling and design, such as "defensive programming", "design-by-contract", etc. ABV is complementary to both formal and dynamic verification technologies. An assertion is an abstract statement that a certain behavioral property of a design must never be violated. Assertions can be validated using both formal and dynamic verification techniques. Conceptually, an assertion contains a formal description of a desired temporal behavior (i.e., property) and monitors the execution of the design model. Assertions can be used also internally in a design model. Any encountered violation of the desired behavior is reported.

ABV eases the development of testbenches, since assertions monitor internal behaviors of a model. Thus, if an error occurs within the scope of one assertion, this error is reported immediately. Tedious attempts to make sure that any error is propagated to the output of the model such that it can be detected by external checking mechanisms is no longer required. Furthermore, the immediate error notification spares the verification engineer from backtracking long simulation traces to find the origin of an error. Therefore, debug time can be reduced tremendously.

Furthermore, assertions have another advantage; when developing the RTL implementation a designer can specify assertions about the intent of the block which is currently developed. Also, the designer can specify which constraints to the environment of the block are assumed. In this case, the violation of an assertion would reflect the wrong usage of that specific block. As well, assertions might reveal misinterpretations of a given imprecise specification. As Leslie Lamport has stated: ”In engineering, imprecision is an invitation to error” [14]; thus, a good ABV methodology, can reveal many bugs, especially bugs which are deeply hidden in a design model. In combination with constraint random testbench techniques, assertions can reveal how often the monitored behavior has been exercised and furthermore, randomized stimuli increase the probability that an assertion detects an error which only occurs at situations which were not anticipated in advance.

Many reports have emerged which reveal that the application of ABV has lead to a boost in verification efficiency. Therefore, an ABV methodology has become a vital part of the overall verification strategy of many companies [15], [16], [17].

1.3.4 Emulation / Rapid Prototyping

A full simulation of an RTL system is too time consuming due to the high degree of details which a simulator would have to address. Therefore, emulation and rapid prototyping techniques are used to tackle this problem. These techniques refer to the utilization either of highly performing processing units to aid the verification task or of the implementation of the model on complex Field Programmable Gate Array (FPGA) boards (i.e., the model is executed rather than being simulated).

ABV is also utilized in combination with these techniques. Here, assertions are synthesized and become part of the RTL implementation. Therefore, it is possible to have assertions run checks on for instance an FPGA board [18], [19].

The use of VPs is a countermeasure, since a system at a very high-level of abstraction enables feasible simulator run-times due to the reduction of model details. Furthermore, emulation and rapid prototyping require a complete RTL implementation. Hence, SW development still would have to be started at a very late stage of implementation. Thus, it can be expected that ESL will sooner or later make emulation and rapid prototyping techniques completely redundant.

1.3.5 Post-Silicon Validation

Post silicon validation refers to plugging a silicon implementation of a system onto a tester and driving test patterns in it. On-chip-debug infrastructure allows a limited access to the internal states of the system in order to be able to test the chip for production errors. ABV is used in a similar fashion as mentioned with the previous verification techniques. Development work is currently in progress in [20] which enables debuggers to interact with on-chip assertions. Assertion failures can be used to freeze the core of a system, in order to allow a close analysis of the systems state, utilizing the on-chip scan-chains and JTAG interfaces.

In a perfect world, however, no functional bugs should exist at this stage of the development.

1.4 Motivation

As mentioned in the previous two sections, increasing complexity impacts the efficiency both of product development and verification. On the development side, the increase of complexity is tackled by component reuse and abstract modeling. On the verification side, sophisticated approaches have been used for ensuring the quality of a product. Up to now, functional verification has been mainly focused on the RTL domain. Reuse of RTL testbenches for ESL is currently the main methodology for checking the functional compliance of an RTL implementation and its corresponding ESL implementation (i.e., the ESL model is used as a golden reference for the verification of an RTL model). Keeping in mind however, that the increase of complexity which is anticipated for the upcoming years will also make even abstract ESL models highly complex, it becomes obvious that the verification of ESL models has to be much more thorough than it is today. This requires a comparable evolution of verification technology for ESL the same way it has happened for RTL. According to ITRS the following statement has been made regarding verification at higher levels of abstraction:

> As design moves to a level of abstraction above register transsfer level (RTL), verification will have to keep up. The challenges will be to adapt and develop verification methods for the higher-levels of abstraction, to cope with the increased system complexity made possible by higher-level design, and to develop means to check the equivalence between the higher-level and lower-level models. This longer-term challenge will be made much more difficult if decisions about the higher-level of abstraction are made without regard for verification (e.g., languages with ill-defined or needlessly complex semantics, or a methodology relying on simulation-only models that have no formal relationship to the RTL model) [3].

This work addresses some of the points mentioned in [3] by introducing ABV to ESL. By means of a new language, it will be shown how known RTL concepts can be adapted and extended to be applicable at ESL as well, while still allowing for a unified approach that covers RTL, too. The same benefits which ABV has introduced to RTL verification is expected for the verification of ESL. However, this work not only focuses on the sole application on ESL. It rather supports assertion specification for multiple levels of abstraction which can be present within one model. Being able to cope with multiple levels of abstraction also enables cross abstraction checks through the use of assertions. Therefore, compliance checks between ESL and RTL models can be enhanced by adding assertions which monitor both models in a co-simulation environment.

1.5 Outline

This work is organized in nine chapters addressing different aspects.

Chapter 2 gives a short description of the problems which arise when attempting to use ABV as is at higher levels of abstraction and outlines some concepts for solving these.

Chapter 3 introduces the requirements to be met by an assertion language in order to be highly applicable at higher levels of abstraction.

Chapter 4 describes and discusses the state-of-the-art and related work with regard to the tasks at hand.

Chapter 5 introduces and describes all new concepts and features of the newly designed assertion language.

Chapter 6 introduces the formal foundation and semantics of the assertion language.

Chapter 7 explains the complete application framework and highlights some aspects with regard to its implementation.

Chapter 8 describes examples for different kinds of assertions specified with the newly developed language.

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

2 Problem Statement and Targeted Approach

This chapter addresses the problems that arise when attempting to apply existing ABV approaches at higher levels of abstraction (i.e., mainly in the ESL domain). Furthermore, possible solutions are outlined.

2.1 TL Modeling and Design

The key to ESL is abstraction. Abstraction means the reduction of details within a model to the necessary level of granularity which is required to provide an executable model of a given system specification. The reduction of details in turn means that executing such a model requires less computational effort and hence, reduces both tool runtime and memory consumption. Therefore, it is possible to simulate an abstract model of a system in feasible time in contrast to RTL. The modeling paradigm in ESL is best known as Transaction Level (TL) modeling and such a model is called a Transaction Level Model (TLM). A Virtual Prototype (VP) is a TLM of a whole system. The level of details which has to be modeled within a TLM is determined by the goals of the analysis which is intended to be performed on it.

The main requirements a VP has to adhere to are the following:

- Compositional view of the system: Functionality has to be partitioned into different components which communicate. The partitioning shall reflect at least on toplevel the partition of the intended design (e.g., CPUs, bus structure).
- Register Accurate: All registers which are intended to be accessible by SW need to be modeled. The register must be accessible via the CPU bus to allow SW read and write accesses.
- Full Memory Map: All resources including blocks and registers have to follow the address map defined by the specification; if the address map is not specified, it is defined for the VP and used in later design stages.

- Communication Topology: All components need to follow the same communication topology as defined by the specification.

The fulfillment of these requirements is necessary in order to enable SW development on the VP. This SW later on also runs on the RTL and silicon implementation.

The kinds of details which are usually abstracted away are the following:

- Clocked Synchronization: Every value change of a clock signal at RTL needs to be processed by a simulator in order to execute all processes sensitive to a specific clock edge. This means that also processes need to be considered which actually do not induce any state transitions. In order to reduce this effort, clocks are usually modeled differently in TLMs or are even omitted. Synchronization is only modeled when a certain causality needs to be enforced. This is achieved by having processes emit events which other processes are sensitive to. Events may be conditional clocks or transaction state changes, to give examples.
- Timing: Timing of a system is only modeled where it is relevant. For example a purely SW-centric view of the system does not require timing to be modeled at all. In case timing is required it is modeled at that level of granularity which is needed to conduct performance analysis. Timing can either be modeled using the simulation time which comes with the simulation kernel of any popular HW description language, or it can be annotated in terms of states, that means time is calculated by the model, not by the simulation kernel.
- Signals: Communication between processes is not modeled with signal-level protocol accuracy. Here, for each signal value update events are emitted which need to be processed. Instead, complete communication protocols are abstracted and reduced to abstract message passing modeled with function calls.
- State-Machines: As soon as complete paths through an RTL state-machine can be substituted by procedural operations, it is no longer necessary to model the state-machine as such, since the state is reflected by the line of code which is executed.
- HW data types: HW data types are omitted and abstract data types (e.g., classes, structs, pointers) are used on which high-level operations are defined.

2.2 TL Modeling Impact to ABV

The better the abstractions mentioned in Section 2.1 are with regard to simulation performance and fast development of system prototypes the more difficulties they impose when applying RTL verification techniques. Most sophisticated approaches to ABV apply some form of temporal logic specification, which expresses temporal relations of signal values in terms of clock ticks. A clock tick reflects the progression in time and usually determines when to evaluate an assertion.

2.2.1 The Notion of Temporal Behavior

The first problem to be addressed for applying ABV to TLMs is the clarification of the notion of temporal behavior. The use of clocks is reduced or even avoided to increase simulation performance. In addition to that, ESL supports abstraction levels where time is not modeled at all, modeled in terms of annotations, or modeled with processes which wait for a specific time to pass prior to resuming. In connection to this issue, it also has to be taken into account that different components within a VP may be modeled at different abstractions. It is also possible that some components are completely modeled at RTL in a clock-related way. It also has to be considered when to trigger an evaluation of an assertion at all. Since assertions monitor the system behavior, it has to be solved how to keep an assertion evaluation synchronous to the monitored system. Generally, the endeavor on ESL is to reduce the number of events to be processed by a simulation kernel for the sake of performance. Therefore, using these events as a possible solution for synchronizing assertions might not suffice. Great parts of the functionality could happen in a sequential context. Hence, no interaction with the scheduling engine of a simulator is performed.

Monitoring ongoing communication within a system is also a problem. On RTL monitoring the signal-level protocol on the basis of clocks reveals the ongoing interactions of components and processes. On ESL such interactions are usually modeled with functions which are invoked by the caller and are executed in the context of the callee. This kind of function is often called a transaction. Therefore, a solution must be found that enables keeping track of ongoing transactions as well.

2.2.2 Scope of TL Assertions

Additionally, the scope an assertion has on an ESL model has to be contrasted to RTL. On RTL, assertions are usually used to monitor interface contracts within a block, timed handshake protocols, or transition paths in state machines, etc. The scope of an RTL assertion is rather on the internals of a block. Monitoring communication-centric system-level properties within an RTL system would lead to a blow up in complexity of an assertion specification. Hundreds of signals and state variables would have to be considered along with their corresponding temporal relations. On ESL however, these details are not modeled. Therefore, it can be assumed that a TL assertion covers a bigger part of system functionality than RTL assertions.

2.2.3 Communication Patterns and Pipelining

Especially to be able to monitor communication-centric behaviors adequately, assertions need to deal with for instance, ”retransmits”, pipelined bus structures, data dependent communication flows (data dependent temporal relations), and more. Pipelining as such poses a problem for RTL assertions, since the underlying formal semantics do not support real pipelining. TLMs incorporate many queue-like structures, message buffers, FIFO-based communication channels to decouple sender and receiver, and so forth. In addition to that, if components are modeled which provide pipelined services, monitoring the communication with that model would have to take the pipelining into account as well. Therefore, real pipelined evaluation semantics, at least for dynamic verification methods, must be provided by TL-assertion approaches.

It is also necessary to deal with data dependencies which have an influence on the temporal behavior.

2.3 Taken Approach

The approach presented in this work tackles these problems by introducing a framework which amongst others incorporates a new language which is referred to as Universal Assertion Language (UAL) in the remainder of this work and a compiler that generates an implementation of given UAL specifications. UAL follows an event-driven synchronization approach. However, a general concept of events is introduced which goes beyond the concept of value-change events and other simulation kernel events and allows transactions and other actions to fire events as well. Furthermore, operators on these events are introduced that can handle different abstraction levels for synchronizing assertions including self-synchronization based on time annotations. UAL also supports a general sequence mechanism, which is independent of the underlying abstraction layer and allows the specification of partial orders on events. Evaluation of sequence specifications is triggered by general events. In addition to these concepts, UAL comes with a set of different execution modes, including a real pipelined mode.

3 Requirements and Objectives for Transaction Level Assertions

This chapter gathers and explains all specific requirements which have to be met by an assertion language applicable to TLMs. These requirements were developed in an incremental process starting from a small set of key requirements for an assertion language to support transaction level assertions. Further extensions to these requirements were derived from application needs. A summary of all requirements is given in Appendix A in Section A.1. Throughout the following sections, the requirements discussed include references to the corresponding summarized requirements in Section A.1 by referring to particular requirements. The references are given in parenthesis in the form "(R X)" with "R" indicating that the referenced item is a requirement and "X" indicating the corresponding number of the requirement in Section A.1. Section A.2 in Appendix A shows a categorization of all requirements according to whether a requirement has been addressed and its importance for enabling ABV at TL.

3.1 Examples for Transaction Level Properties

To give a better impression of what properties monitored by assertions could be at TL, this section provides some informal examples.

Bus Infrastructure Checks

Checking a bus infrastructure is one possible application for TL assertions. For instance a property that states that the address decoding of the bus yields the correct address map could be:

> "*Whenever a master module initiates a transaction with address* Y*, the corresponding transaction is executed on the module where* Y *lies in the address range of that module!*"

A related check is that no other registers are illegally modified.

Another check involving timing as well could be:

"A bus response to a specific request is never issued later than 50 nanoseconds!"

Timing can also be considered more abstract:

"A request is always responded before another request is placed!"

Dataflow Checks

Dataflow properties could be checked as well:

"A write attempt to a SW visible register implies that the payload is stored into that register once the transaction has finished!"

Furthermore, tracking a data package among several stages could be monitored as well:

"If data is written to a register in an output device, it is required that this data is transported out as soon as the environment is ready!"
"If data is written to a buffer, it is required that this data flows out within a maximum amount of time!"

Controlflow

Controlflow checks would be possible as well:

"Correct occurrence of data-dependent packet requests!"
"The execution of a specific instruction implies the correct sequence of memory-fetch and IO-transactions!"

SW-Accesses

Monitoring protocols to indicate wrong SW-accesses to HW -registers could be checked:

"No write attempt to a read-only register occurs!"
"No write / read attempt to a full / empty buffer exists!"

Configurations

Configurations and their effects could be checked for correctness as well:

"*A firing interrupt implies that the interrupt was enabled!*"

As the examples indicate, TL properties reason about sequences of transactions and Boolean propositions along these. Hence, a TL assertion approach generally needs to support the specification of transaction sequences (R 1).

3.2 Characteristics of SystemC Transaction Level Modeling

TL modeling plays a major role in the success of the development of VPs. It allows breaking down a system to a set of components or blocks comprised of concurrent processes. These blocks communicate with each other via so-called transactions. The following sections give a brief overview on the main characteristics of TL modeling. The explanations are based on the semantics of SystemC[1], which is the most common language for modeling at TL. Strictly speaking however, SystemC is not a language but a class library built on top of C++. SystemC offers the neat bits for modeling communication, hierarchy, and especially concurrency in an easy fashion in C++.

Due to the relevance of SystemC for TL modeling, it is obvious that a TL assertion approach is required to support the evaluation of assertions on-the-fly during a SystemC simulation (R 2). It is also required to support all SystemC and C++ base types (R 3).

3.2.1 Hierarchy

SystemC offers a concept of hierarchy which allows encapsulation of functional units to modules. These modules can be connected via ports to enable communication. A module can also incorporate another module, thus creating levels of hierarchy. Each module is assigned a unique hierarchical name, which allows referencing a module from anywhere in the system (backdoor access). This is useful for verification purposes. Since a TL assertion can monitor actions in several modules at once, a connection mechanism is required which utilizes a backdoor access to modules and their internals (R 4).

[1] A complete introduction to SystemC is omitted. For more information on SystemC the reader is referred to [7]

3.2.2 Concurrency

Simulation Kernel

Concurrency in SystemC is handled in a very similar fashion as in VHDL [21]. Processes are used to model concurrent actions. The simulation kernel uses a delta-cycle concept for the sequential processing of concurrent statements. It incorporates a process activation list which stores handles to processes which need to be activated in the current delta-cycle. The order of process execution within one delta-cycle is random in order to ensure that no hidden dependencies on the execution order of processes exists, thus preserving the principle of concurrency. In addition to the delta-cycle concept, the kernel supports a model of simulation time which allows the scheduling of processes to specific times. In contrast to VHDL, however, the simulation kernel does not offer postponed passive processes (i.e., processes which are only activated after all regular processes have been executed in the current delta-cycle and may only read signals).

It is imperative not to alter the semantics of the SystemC simulation kernel (R 5). Changing the simulation semantics would require to prove the functional equivalence between the altered version and the OSCI reference simulation kernel. In addition to that, each new release of SystemC would require that all alterations have to be added and checked again. This is error-prone and time-consuming. To ensure that the simulation semantics remain intact, it is therefore required that a TL assertion approach works on top of SystemC, implemented as a class library of its own (R 6).

Event Concept

By leveraging the event sensitivity of processes or wait statements in conjunction with an event notification mechanism, a user can control the scheduling of processes. Processes can be made sensitive to events either statically (sensitivity lists) or dynamically (`wait`-statements or `next_trigger`-statements). A TL assertion approach has to deal with any kind of event offered by SystemC (R 7). This also requires the possibility to link assertions to any of these events (R 8). These events can be grouped as follows:

- Value-Change Events: These events are emitted by signals as soon as a value-change has occurred. Using evaluate-update mechanisms, the kernel ensures that a signal value-change can only be obtained with the last assignment to a signal within a delta-cycle.

- Custom Events: The user can declare events and add annotations in any procedural context to emit this event. The notification mechanism allows the scheduling of an event to a certain simulation time or the next delta-cycle.

  Immediate notification is supported as well. This means, that the scheduled event will be notified in the same delta-cycle where the notification has been processed. The notification of immediate events, however, does not mean that the event is emitted at once. The event is emitted immediately in the current delta-cycle but only after the process which made the notification has either suspended or terminated. Processes which react to immediate event notifications are activated in the same delta cycle.

  An event may only have one pending notification. If another scheduling request for an event is made while there is already a pending notification for this event, only the notification survives which has the earliest scheduling time. SystemC also offers event-queues which can store multiple scheduling requests. This means, that if one event is scheduled twice to the same simulation time or delta-cycle, that it will occur twice.

  In general, there is no predetermined order on the events to be emitted within one delta-cycle. This means, that the order of calls to schedule two different events to the same delta-cycle has no correlation with the actual order of occurrences of these two events in that delta-cycle.

- Implicit Events: Processes can reschedule themselves by notifying implicit events. The notification of implicit events leads to an immediate suspension of the emitting process. Since these events are not visible to any other process, a TL approach requires a mechanism that allows tracking of these events as well, however, with no change of the simulation kernel (R 9, and R 5). Two different implicit events can be emitted by a process.

  - Zero-Delay events: Through the notification of an implicit zero-delay event, a process reschedules itself to wake up at the next delta-cycle. Such a notification is accomplished by using timed wait statements, which take a time parameter. The value of this parameter equals 0 to enforce a delta-cycle delay (`wait(0,SC_NS)`) for that process.

  - Timed-Delay events: A process can also reschedule itself to a specific simulation time later than the current time. The corresponding notification is obtained through the use of timed wait statements with a non-zero time parameter (e.g., `wait(10,SC_NS)`).

3.2.3 Synchronization

Basically, SystemC offers two types of processes for modeling concurrent behavior:

- Suspendable: The execution of such a process can be partitioned into several parts by suspending it. When such a process suspends, it saves its whole context. Once the process wakes up, it restores its context and resumes from where it has stopped. Suspendable processes are modeled using the `SC_THREAD` macro offered by SystemC. A suspendable process may be put to sleep using the wait statement offered by SystemC. It can either wait until a certain amount of simulation time (implicit timed event) has passed (e.g., `wait(10,SC_NS)`) or until the next occurrence of a specific event (e.g., `wait(e1)`), or in strictly the next delta-cycle (implicit zero-delay event, `wait(0,SC_NS)`).
- Non-Suspendable: The execution of such a process may never be suspended. Once executed, the process runs until its last instruction. Hence, the complete execution of a non-suspendable process happens within one delta-cycle. Non-Suspendable processes are modeled using the `SC_METHOD` macro offered by SystemC. In order to enable an assertion based monitoring of actions within a non-suspendable process, it is required to support a more granular time resolution than delta-cycle resolution (R 10).

Both types may have a sensitivity list where all events are specified which may invoke the process. Suspendable processes are avoided as best as possible since the induced context switching is very expensive with regard to performance.

3.2.4 Communication

A transaction represents a high-level form of a communication protocol. All protocol-specific details are encapsulated within a transaction. Hence, the actual act of initiating a transaction results in a remote function call from a process (parent). A designer focuses more or less on the data that has to be transported rather than the protocol specifics.

Transactions are modeled as functions which are defined in pure virtual interface classes and implemented in corresponding child classes which inherit the interface. The implementation details of a transaction strongly depend on the targeted abstraction level. Yet two distinctions with regard to transactions can be made:

- Blocking: A blocking transaction may suspend its parent process which means that the transaction is resumed in a later delta-cycle. This kind of transaction can be invoked in suspendable processes, only (i.e., `SC_THREAD`).

- Non-Blocking: A non-blocking transaction is atomic and may not suspend its parent process; the whole transaction is executed within the same delta-cycle it has been invoked. This kind of transaction can be called from within any process (i.e., `SC_THREAD` and `SC_METHOD`).

Invoking a transaction results in dereferencing a pointer that holds the address of the target object and in calling a member function of that object. The whole call or even several calls can happen within a single delta-cycle (e.g., with non-blocking transactions). In contrast to that, communication in RTL models is obtained via signals and hence, always consumes at least one delta-cycle due to the induced value-changes that form the protocol. Therefore, with signal based protocols it is sufficient to monitor the values of the participating signals at a granularity of delta-cycles in order to detect ongoing transactions. Since, this does not suffice at TL, it is required that a TL assertion approach is able to detect transaction calls (R 29, and R 10) in order to enable the tracking of transaction sequences (R 1). This also requires that assertions gain access to transaction return values and arguments (R 12) and that both blocking and non-blocking transactions are supported (R 11).

In order to ensure easy IP reuse and interoperability, a TL modeling standard [8] has been developed by OSCI. This standard defines different interfaces including transaction signatures and argument types. Clearly, the support of this standard by a TL assertion approach is required as well (R 13).

3.2.5 Abstraction Levels

As mentioned in the previous chapter, the key factor for the success of the ESL domain is abstraction. The objectives which determine the required abstraction level for a TLM depend on the intent of analysis: The more abstract a model is the higher the performance of a simulation becomes but the less information is available for analysis. Therefore, the chosen level of abstraction is a trade-off between performance and information. This issue has so far hindered the establishing of standards which define abstraction levels and provide guidelines on how to model at a certain abstraction. However, a common nomenclature for TL abstraction levels has been developed in conjunction with the OSCI TLM standard. Unfortunately, the definition is not exact and allows for some interpretation. The four terms that have been developed are:

- Programmer's View (PV): Access to the system does not consider timing; correct data and control flow are focused.

- Programmer's View with Timing (PVT): Additional to PV, approximate timing of system accesses is considered as well.

- Cycle Approximate (CA): System accesses are resolved in cycles.

- Cycle Callable (CC): System accesses are clocked as in RTL but the communication is still modeled with transactions in contrast to signals.

Readers should note that these definitions do only reflect a communication-centric perspective. For instance, a PV model does not have to be designed completely regardless of time. The timing of the model is just of no interest from the communication point of view. Associating a given TLM with a corresponding view cannot be accomplished easily in all cases because the borders between different abstractions blur, as with PVT and CA.

The variety of abstraction techniques in conjunction with the lack of modeling standards that clearly define the scope of each abstraction level lead to TLMs that are heterogeneous in abstraction. This gives designers a high degree of freedom easing the tasks at hand but from a verification perspective, this poses challenges with regard to formulating sequential properties about the desired behavior. A TL assertion approach thus, requires the capturing of all of these abstraction levels (R 14), even when they are mixed (R 15).

3.2.6 Design States

In addition to the aforementioned aspects on abstraction levels, the design state of a TLM is also not defined. On RTL the state of a model is the conjunction of all signal values stabilized at a specific clock tick. On TL as described earlier, clocks are usually not modeled. Signals in general are avoided as best as possible in order to reduce the number of value-change events that are emitted. Therefore, states are usually represented by variables which update their values immediately upon an assignment. This means that several state transitions can occur within a single delta-cycle. Hence, assertions must be able to access, sample, and read TLM states within one delta-cycle (R 10). From a verification point of view, it is necessary to provide access to model states (R 16) and to track assignments on these (R 17). In case a specific state variable is declared in a private context, it is also required to be able to link to a public access function if existent (R 18).

3.3 Temporal Behaviors at the Transaction Level

This section analyzes different kinds of temporal behavior inherent to TLMs.

3.3.1 Temporal Behavior of PV Models

As mentioned in Section 3.2.5, PV models provide a view regardless of timing, which does not mean however, that the model executes in zero time. Nevertheless, formulating temporal properties has to be unaware of time as well. This brings up the question of how temporal behavior can be defined for a PV model. As the examples of Section 3.1 indicate, it is necessary to formulate temporal correlations of transactions (i.e., to capture sequences of transactions). Any execution order of transactions depends either on the simulation kernel or on cause-effect chains. It is not possible to reason about transactions occurring simultaneously. The same holds for the occurrence order of events. Since the simulation kernel unrolls concurrency to a sequential algorithm, no event can occur simultaneous to another. The order is only realistic if there is a causality behind the event scheduling. If two events or two transactions are concurrent to each other, the order of their occurrence is not correlated and is set randomly by the simulation kernel.

Figure 3.1 shows all possible relations of two transactions.

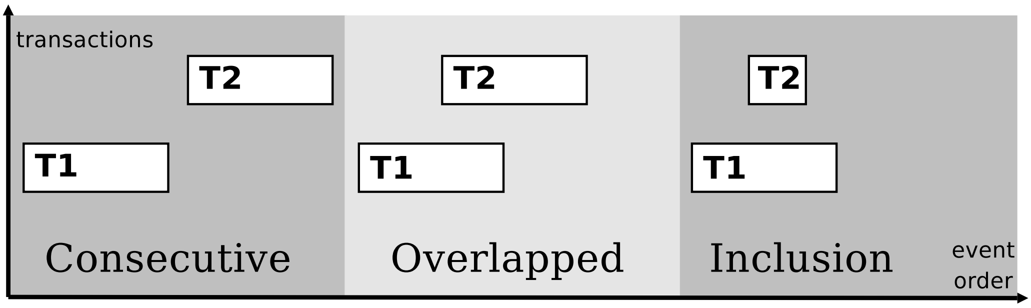


Figure 3.1: Transaction Relations on PV

As Figure 3.1 shows, no transaction can start or end simultaneous to another. Yet overlaps are possible (R 30). In order to temporally correlate events or transactions, an axis can be used which reflects the order of occurrence of any event or transaction, that means the axis reflects the unrolling, which happens within the simulation kernel. The temporal distance of two events can be determined only relative to the occurrence of further events. Figure 3.2 shows an example of a sequence of three events *E1*, *E2*, and *E3*.

The temporal distance between the first occurrence of *E1* and its second occurrence depends on the events that are taken into account for the analysis. If all events in Figure 3.2 are considered, the distance will be four, because there are three other event occurrences between the first and the second occurrence of event *E1*. If we exclude *E3* events, the distance will be three. If we only consider *E1* events, the distance will

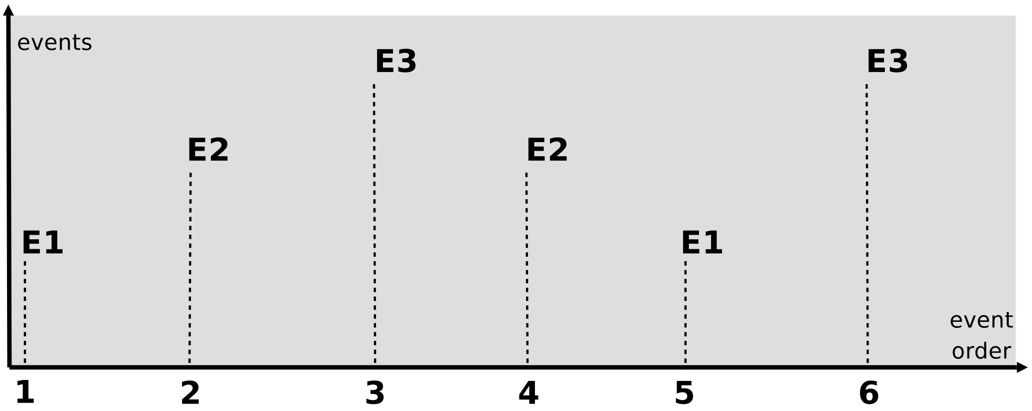


Figure 3.2: Event Sequences

be one. This means, that the second occurrence of *E1* is the next event encountered after the first occurrence of *E1*. Therefore, an assertion approach which is capable of tracking such temporal behavior on a PV model requires the specification of partial orders on events. The global order is defined by the simulation kernel and consists of all event occurrences, whereas the partial order is valid only for the considered events (R 19).

Since these events are issued by the DUV, it is possible that some event notifications are left out due to an erroneous behavior. This is not possible in RTL because a clock is an input to the design and can be assumed to tick correctly. In order to face this issue, a mechanism is also required that can specify strict partial orders on events such that a missing event occurrence can be detected relative to an occurrence of a different event (R 20).

3.3.2 Temporal Behavior of PVT Models

In contrast to PV, in a PVT model simulation time is of interest. The progression of time is modeled by timed notifications of events or timed `wait`-statements. The most interesting use-case for PVT modeling is the ability to run performance analysis. This means, that the verification has to consider that the design operates within certain time perimeters. Unlike PV, the existence of a time axis makes it possible to use time for temporal considerations as well. Here, it is possible to consider events occurring at the same simulation time to be simultaneous (R 21). Hence, a TL assertion approach needs to means for the specification of temporal behavior based on the simulation time as well (R 22). The resolution of time corresponds to the smallest time unit specified for the simulation. Figure 3.3 shows the transaction relations that can occur on PVT in addition to those in Figure 3.1. Transactions and any events can be correlated to each other over time. On this level, the specification of feasible sequences is no longer just restricted to cause-effect chains. For instance, it is possible that two devices access a bus at the same simulation time.

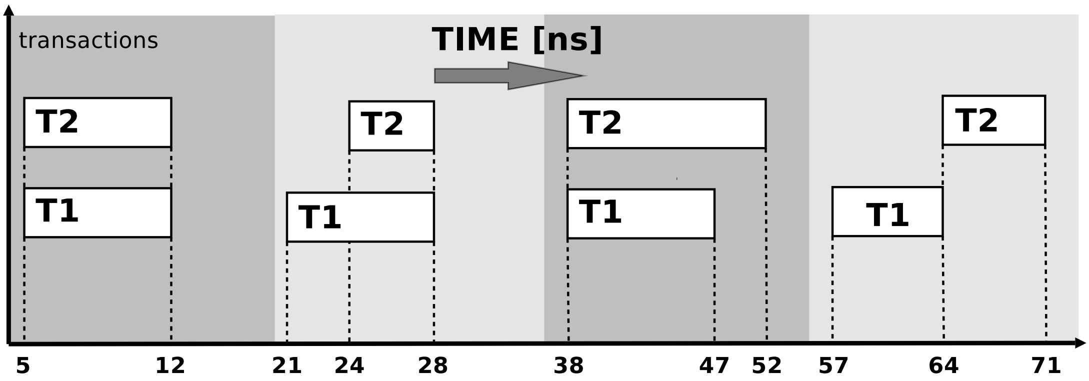


Figure 3.3: Transaction Relations on PVT

3.3.3 Temporal Behavior of CA

The CA view is close to RTL with regard to the notion of time, however, not the modeling. The model is still not clocked. Time delays are expressed in terms of multiples of a cycle period value (e.g., `wait(5*clk_period)`). Clock frequency changes are modeled through changes in the cycle period value. Actions are to consume the approximate number of cycles as they would in the corresponding RTL implementation, however, this number can be an accumulated value which sums up delays of several actions before an implicit timed event is scheduled. This abstraction is used to accomplish a more granular performance analysis of the system, including power-up and power-down phases of a device. Since time delays can change dynamically, it is necessary to provide a mechanism within a TL assertion approach to capture this timing as well (R 23).

3.3.4 Temporal Behavior of CC / RTL Models

CC models have the abstraction level closest to RTL. Hence, within CC models clocks are used to obtain synchronization of processes. However, communication is still modeled with transactions in contrast to signal based protocols. A TL assertion approach thus, needs to support the specification of temporal relations in terms of clock cycles in addition to the temporal relations of higher abstraction levels (R 14, R 15). Furthermore, it is required that a TL assertion approach supports the specification of classical RTL assertions which monitor signals (R 24). This also requires to capture resets in order to stop ongoing evaluations of assertions (R 25).

3.4 Sampling

Specifying temporal behaviors with assertions includes also a propositional logic part. Boolean propositions are formulated which have to hold at specific times. Temporal operators define when Boolean propositions are evaluated. A Boolean proposition is formulated on the model's state variables. In an RTL model, the clock defines when a state value is stabilized and also the clock indicates the progress in time for temporal relations. On TL however, as was described in Section 3.3, the progression of time can either be measured by the occurrence order of events or transactions and for lower levels by the progression of simulation time. Due to this circumstance a TL assertion approach is required to sample the model's state immediately with anything that progresses time (R 26).

All assertions need to have a read-only access to design internals (R 27). Otherwise, assertions could cause side-effects within the DUV and thus, alter its behavior (R 28).

3.5 Data-Dependent Temporal Behavior

The temporal behavior of a model may change regardless of its abstraction depending on data stored in variables or data, which is passed around with transactions. Configurations of timer modules for instance determine the exact time delay which has to pass until an interrupt is signaled. Another possibility is that the number of events or transactions may change depending on dynamically changeable configuration values. For instance, if an IP-block is configured to fetch data byte-wise and a master sends out data words, each sent word requires four fetches by the IP-block. If the configuration changes for the IP-block to fetch data in halfwords, the number of fetches is reduced to two. Furthermore, it is possible that the master module sends out bursts of dynamic size, which in turn has an influence on the number of fetches to be done by the IP-Block. Since it is neither feasible nor possible to specify assertions for each possible data dependency, it is required that dynamic temporal behavior can be captured as well (R 23).

3.6 Transaction Detection

Since transactions are modeled as function calls, it is necessary to provide a detection mechanism which enables the tracking of transaction occurrences (R 29). Since it is possible to have blocking and therefore eventually nested transactions, the detection mechanism for transactions needs to make this information accessible (R 30). This allows a more fine-grained view of the transaction activity of a model. The detection has to treat non-blocking and blocking transactions the same way.

The notification of a transaction occurrence has to be done immediately (R 31) to guarantee deterministic sampling. This means, that the state of the model may not change until the notification has been processed.

3.7 Request/Response Communication Patterns

In this section, further requirements are gathered which are derived from Request/Response communication patterns applied in most system-level models. In a Request/Response communication protocol, one communication interaction is a bundle of one request and one response. Both a request and a response can be any sequence of transactions or events in general. For the further explanations, however, request and response are treated as singularities for the sake of simplicity. Detecting a Request/Response interaction between two modules corresponds to detecting a request and the associated response. Associating a response to a request is simple as long as the underlying protocol does not support retransmissions of requests or multiple outstanding requests in parallel. Thus, one request and one response form one communication interaction and one response is always associated with the preceding occurrence of a request (there can only be one outstanding request). This can be simply expressed by formulating the following informal property:

"Every request implies a response at some arbitrary time later"

3.7.1 Retransmissions of Requests

When retransmissions of requests are allowed, associating a response to a request has to consider only the last preceding occurrence of a request. All other requests have to be neglected when attempting to detect this communication interaction. When considering the informal property formulation given in the previous section, it is easy to see that the reasoning is directed forward in time. However, the formulation does not account for retransmissions. In order to capture retransmissions, the formulation changes to the following statement:

"The last request prior to a response implies a response at some time later"

In this case, however, detecting the correct request depends on detecting the response. A response, however, might not occur at all as a result of an error. This means that the correct request might never be detected. Due to this issue and because of the popularity of such protocols at the system-level, it is required that a TL assertion approach supports the correct detection of request/response communication interactions where retransmissions of requests can be handled as well. In this context the user should be given the possibility to decide if a retransmission is to be ignored or indicated as a report to the user (R 32). A similar but more simple approach is used in the Open Verification Library (OVL) (see also Section 4.1.1).

3.7.2 Pipelined Requests

In case the protocol allows the processing of multiple outstanding requests of the same module within an arbitrary amount of time, associating a response with a specific request becomes more complex. Since the issuing of responses might not be in the same order as the issuing of the requests, this complexity even increases. Hence, if such a communication interaction is to be monitored by an assertion, it is required that pipelined behavior can be detected correctly (R 33).

3.8 General Aspects

Keeping in mind that ESL is a relatively new development, it might happen that another more sophisticated language establishes for the modeling at the transaction level. Furthermore, the possibility that a system model written in SystemC might incorporate RTL components in a co-simulation environment requires that TL assertions need to be specified with a separate declarative syntax or language (R 34). This language is also required to be aware of transactions as such in order to preserve a TL view for specifying assertions (R 34). The language also has to be a functional superset of RTL assertion languages thus, requiring the following features:

- Property Specification Language (PSL) and SystemVerilog Assertions (SVA) evaluation semantics (R 35);
- Assertion coverage for improving constrained random testbenching (R 36);
- Local variables for storing information along one assertion evaluation (R 37),
- Control mechanisms to turn on/off assertions (R 38);
- Severity levels for assertion failures (R 39);
- Customizable failure messages (R 40);
- Packaging of assertions to libraries (R 41);

4 State-of-the-Art and Related Work

This Chapter first describes the state-of-the-art of ABV as it is currently applied to industrial designs. Afterwards, related work is summarized. For both, the approaches are discussed with regard to the requirements from Chapter 3.

4.1 State-of-the-Art

4.1.1 Library Based Approaches to RTL ABV

Assertion libraries have been developed to ease the ramp up of ABV in design projects. In order to help design engineers, who are usually not familiar with specifying formal behavior using a formal language, sets of predefined checkers have been implemented. Hence, a design engineer could simply reuse a checker from a library and connect it to the DUV. A checker can be considered as a monitor module with a signal interface. The implementation of the monitor represents a finite-state automaton of a specific property. The monitor runs in parallel to the rest of the design. Its input signal values control the branching of the internal automaton. In case the represented property is violated, the automaton reaches an illegal state and fires an assertion.

The most popular representatives for a library based approach to ABV are the Open Verification Library (OVL) [22] and CheckerWare [23]. Within these libraries, a wide set of common checkers are provided, which can be customized to a certain extent. However, these library based approaches suffer from being not flexible enough. Therefore, the use of such libraries is limited to basic use cases. Nevertheless, the utilization of such libraries already has proven that ABV tremendously increased verification efficiency [13].

Several reasons exist which make the utilization of these assertion libraries at the transaction level hardly possible. The most important hindrance is that the checks express temporal relations in terms of clock ticks only. Thus, temporal behavior on high levels of abstraction can not be checked. Furthermore, the interfaces of monitors in these approaches are signal based. Hence, the application at TL would necessitate the translation of transactions and model state variables to signals. This always introduces extra delta-cycles for enforcing the corresponding value updates and requires severe annotations to the model to implement the necessary translations.

4.1.2 Language Based Approaches to RTL ABV

The most powerful approaches to ABV are based on assertion languages, which are tailored to the specification of temporal properties and thus, assertions. The most popular approaches are Property Specification Language (PSL) [24] and SystemVerilog Assertions (SVA) [25]. The e-verification language [26] contains also a support for formulating temporal expressions, referred to as temporal e. The offered features, however, are comparable to a subset of PSL.

In terms of features and expressiveness, PSL and SVA are comparable. Yet, PSL has a stronger connection to formal temporal logic, namely Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). This part of PSL is referred to as Foundation Language (FL) and Optional Branching Extensions (OBE), respectively. A good comparison of both languages regarding their expressiveness has been presented in [27].

Layers

Both PSL and SVA follow a layered approach as depicted by Figure 4.1. One layer groups the according language operators according to their functionality.

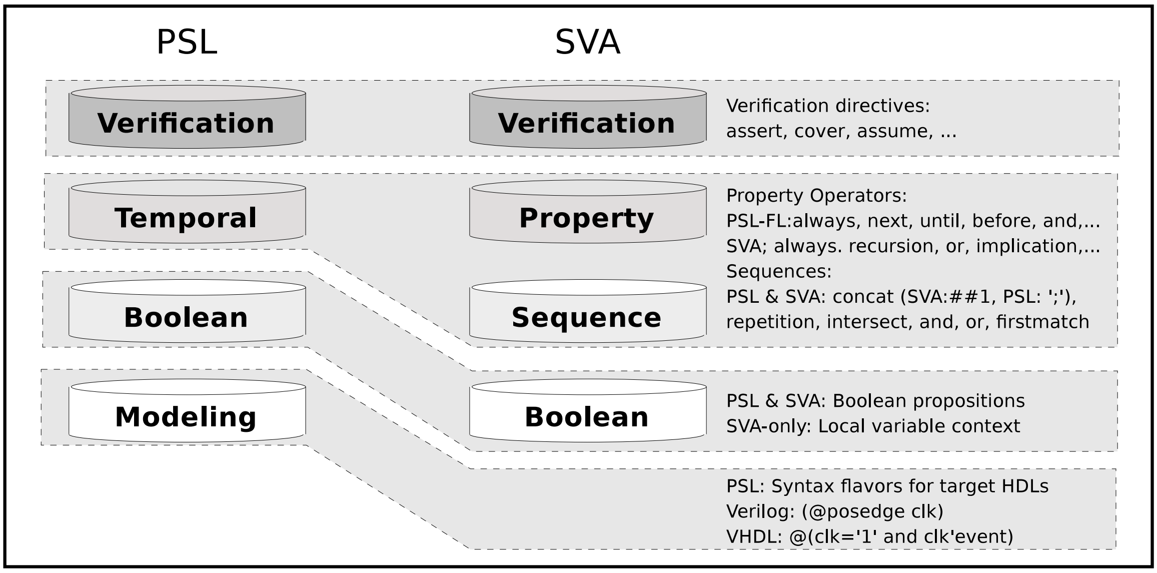

PSL
SVA
Verification
Verification
Verification directives:
assert, cover, assume, ...
Temporal
Property
Property Operators:
PSL-FL:always, next, until, before, and,...
SVA; always. recursion, or, implication,...
Sequences:
PSL & SVA: concat (SVA:##1, PSL: ';'),
repetition, intersect, and, or, firstmatch
Boolean
Sequence
Modeling
Boolean
PSL & SVA: Boolean propositions
SVA-only: Local variable context
PSL: Syntax flavors for target HDLs
Verilog: (@posedge clk)
VHDL: @(clk='1' and clk'event)

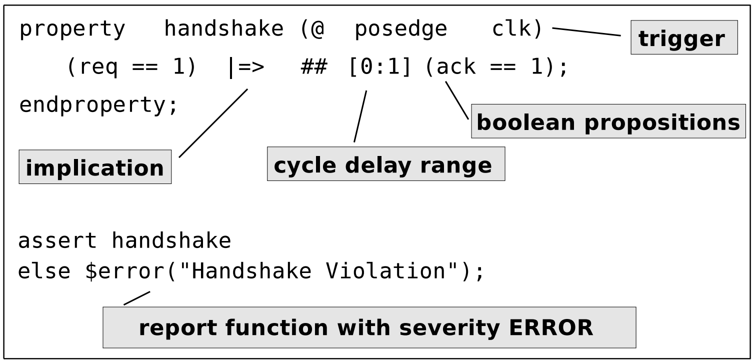
Figure 4.1: Layered Structure of PSL and SVA

As Figure 4.1 shows, both PSL and SVA have a similar layer concept. The temporal layer in PSL corresponds to the property and sequence layer from SVA. The modeling layer in PSL defines different Hardware Description Language (HDL) flavors. This is because PSL was developed to be applicable to any HDL. In contrast to that, SVA is a subset of the SystemVerilog language, which embodies both HDL and Hardware Verification Language (HVL) concepts. Hence, SVA uses SystemVerilog syntax for modeling constructs. SVA, however, offers a binding construct, which allows connecting SVAs also to other HDLs. The construct is also used for externalizing assertions from a design, since assertions are usually not meant to be synthesized to a net list.

Generally, PSL does not define any interaction with simulation engines due to its freedom of applicability. The implementations strongly differ dependent on the design language (VHDL, Verilog) and tools. In contrast to that, the evaluation of assertions written in SVA is strongly woven into the simulation kernel of the SystemVerilog language. Here, assertions are always evaluated after all other processes have finished within a simulation cycle. Furthermore, the kernel offers a sampling region, which guarantees that assertions sample only stabilized values for evaluating any Boolean propositions.

RTL Assertion Basics

The structure of a simple SVA assertion is shown in Figure 4.2.



```
property   handshake (@   posedge   clk)
    (req == 1)  |=>   ## [0:1] (ack == 1);
endproperty;


assert handshake
else $error("Handshake Violation");
```

Figure 4.2: SVA Assertion Example

The property *handshake* in Figure 4.2 checks that an asserted request signal is followed by an asserted acknowledge signal within the next or the second clock tick. A trigger expression is required which defines when the evaluation of a property is started. In this example, the positive edge of the clock signal triggers the evaluation. Furthermore, temporal delays are specified in terms of occurrences of this edge.

The example shows the use of a delay expression (`##`), which is parameterized with a range (`[0:1]`). The expressions `req==1` and `ack==1` represent Boolean propositions on the request and the acknowledge signal. The non-overlapping implication operator (`|=>`) specifies that the right hand side (RHS) expression has to be true if the left hand side (LHS) has evaluated to true. The evaluation of the RHS is started at the next clock edge after the LHS has evaluated to true. The LHS expression of an implication is called the antecedent and the RHS expression the consequent. The property is essentially turned into an assertion using the `assert` directive on the property. As soon as the property evaluates to false, a system function is called that issues a specifiable message. In this example, the `$error()` system function is used for emitting a report. This system function sets the severity to error which usually stops the simulator.

The evaluation of the property in Figure 4.2 is illustrated in Figure 4.3. An example simulation trace of the request and acknowledge signals obtained at positive edges of the clock is shown.

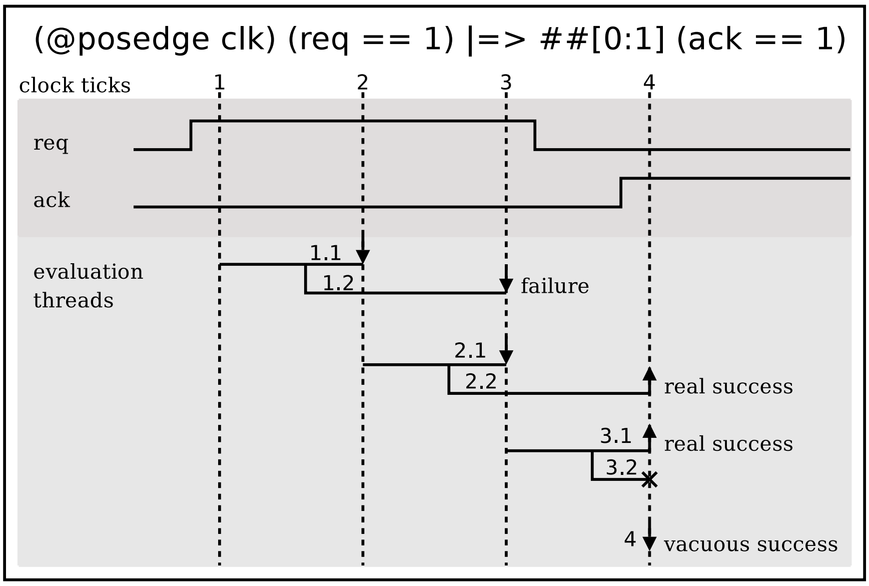


Figure 4.3: SVA Assertion Example - Evaluation

Generally, a property can either be true or false, although its evaluation can span several clock cycles. With implication properties (i.e., properties that are formed by an implication operator), a distinction is made within successes of a property evaluation, namely vacuous and real successes. This distinction is only relevant for coverage analysis of property evaluations. A vacuous success is obtained if the property succeeds because the antecedent expression does not hold. In case both the antecedent and consequent expressions hold, a real success is obtained. A property is evaluated on each occurrence of its trigger. A trigger is formulated with a so-called clocking expression. With every occurrence of the trigger, a new evaluation is started, which is referred to as evaluation attempt. One evaluation of a property is called a thread. If the specification contains alternatives such as a delay range, a thread splits into sub-threads, one for each alternative. In Figure 4.3 the threads and their sub-threads are depicted as horizontal lines below the trace. The first of two digits represents the thread id and the second the sub-thread id.

In the given example, a new evaluation thread is started on each clock tick. Due to the sampling region in the SystemVerilog kernel, the previous values of signals are sampled. Hence, in the given example, the assertion only evaluates the values shown left of a depicted clock tick.

At clock tick 1, thread 1 is started and matches due to request being asserted. Thread 1 splits immediately into two sub-threads 1.1 and 1.2.

At clock tick 2, sub-thread 1.1 fails since acknowledge is not asserted. Sub-thread 1.2 continues. Thread 2 is started and splits accordingly.

At clock tick 3, sub-thread 1.2 fails and hence, the whole evaluation for thread 1 fails. This leads to an assertion failure. Sub-thread 2.1 fails as well and sub-thread 2.2 continues. Also thread 3 is started and splits immediately.

At clock tick 4, sub-thread 2.2 succeeds and thus, the property is true. Also sub-thread 3.1 succeeds. Here, sub-thread 3.2 gets canceled since 3.1 has already succeeded. This behavior is referred to as firstmatch semantics. Furthermore, thread 4 is created which fails immediately because request is not asserted. This leads to a vacuous success of this property evaluation.

As shown in this example several threads of a property can produce a result at the same clock tick. This is especially important when a property is not asserted but covered. Here, all successes of a property are counted and reflect how often the property was observed in the DUV.

Sequences

A very useful feature offered by both PSL and SVA is the possibility to notate sequences. The behavior of a model over time can be considered as a trace of all state and signal values over time. The trace can be either produced by simulation or can be calculated based on a formal state machine representation of a model. A sequence represents a regular expression which is attempted to be matched against that trace. Such an attempt can evaluate to either a match or a not-match. This terminology is used to distinguish sequence results from property results which are Boolean. Within properties, matches of sequences map to the Boolean value *true* and not-match results map to the Boolean value *false*. Figure 4.4 depicts how a sequence is evaluated for an example trace.

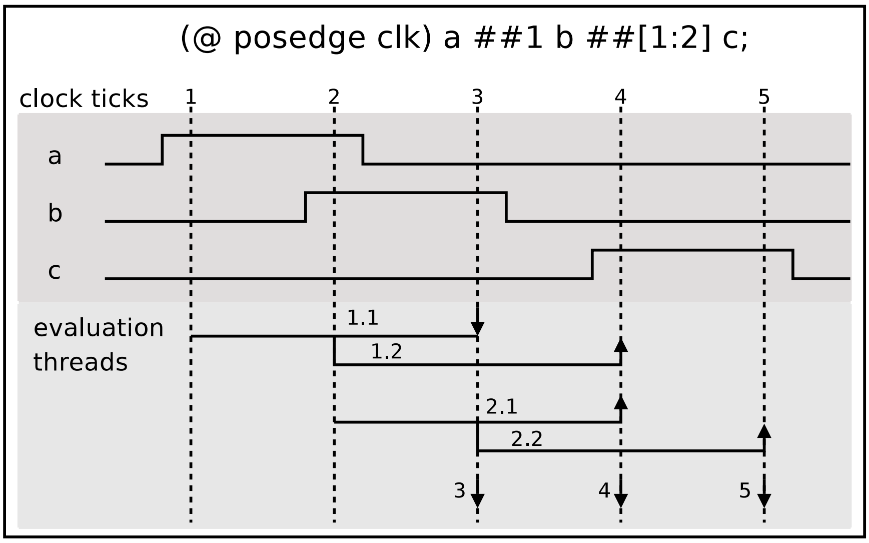


Figure 4.4: SVA Sequence Example

The evaluation of sequences is again structured in terms of evaluation attempts, threads, and sub-threads. A sequence in general, is attempted to match all specified alternatives. As the evaluation of thread 2 indicates both sub-threads match. This behavior is referred to as anymatch behavior. Many operators exist that connect sequences or build more complex sequence expressions. For instance, sequences can be build by concatenating other sequences, or sequences can be conjunctions or disjunctions of other sequences.

Like properties, sequences require a trigger as well. The depicted example again uses a clock tick as trigger and as measure for temporal relations.

4.1.3 Applicability of PSL and SVA to TL Modeling

Analysis of PSL

PSL is in its nature event based rather than cycle based. This holds true for the language presented later in this work as well. Sequence expressions using different clocking expressions can be formulated conveniently in PSL:

```
{a@e1;b@e2;c@e3}
```

The clocking expression denoted by a `@` defines when the left hand side expressions (in the example: a, b, c) is to be validated. On RTL, the events used in the clocking expression are usually clocks. Since clocking expressions define triggers which control the evaluation of properties and sequences, it is vital that the specified events do really occur. In case of clocks, this case is rather negligible due to the periodicity of clocks. At TL, however, instead of clocks, events or transactions need to be used (R 19):

```
{a@PUT;b@PUT;c@GET}
```

This sequence represents a partial order on occurrences of both a PUT and a GET transaction. It will match when a PUT transaction occurs twice followed by a GET transaction, with `a` being true at the first occurrence, `b` being true at the second occurrence of the PUT transaction, and `c` being true at the occurrence of the GET transaction. The evaluation of this sequence, however, will not terminate if for instance, no GET transaction is called due to a design error. Thus, the clocking feature for sequences does not support the specification of strict partial orders on events (R 20).

PSL also offers the FL family of operators like, `always`, `never`, `eventually`, `next`. Using these operators, it is possible to formulate temporal properties. The evaluation of these operators can be triggered by a clocking event as well. If such a trigger is omitted, time progresses at the granularity observed by a simulation tool as defined by PSL [24]. By using these operators on events issued from within a design instead of clocks edges, it would be possible to specify strict partial orders on events. However, writing longer sequences becomes complicated since the operators would have to be nested within each other[1]. The granularity observed by a simulation tool, in this case the SystemC simulation kernel, is not sufficient (R 10). Additionally, it would require to interpret SystemC events as Boolean values. With PSL it is also not possible to fulfill other important requirements mentioned in Chapter 3. Dynamic temporal behavior for instance, can not be captured since temporal relations need to be static in any PSL description. Also, capturing protocol patterns which allow for retransmits is not supported natively (R 32).

Analysis of SVA

While facing the same problems in SVA as in PSL regarding event control, an additional problem exists that needs to be resolved for applying SVA at TL. The evaluation of SVAs is strongly connected to the SystemVerilog simulation kernel, which provides various evaluation regions. Any concurrent assertion in SVA is evaluated in the Observe region [25], however, the sampling of states is done in the Sampling region [25]. This way the evaluation of SVAs can be done race free. Though SystemVerilog fully

[1] This was the motivation for developing the concept of sequences.

supports the TL modeling paradigm, SystemC has been adopted as the modeling language for high-level system modeling. The SystemC kernel, however, does not provide comparable features as the SystemVerilog kernel with regard to assertion evaluation and sampling. In fact, the SystemC standard does not define concurrent assertion support at all. Hence, applying SVA to SystemC would first of all need clarification regarding the evaluation semantics of assertions and further modeling issues (R 5).

General Considerations

Generally, both PSL and SVA lack transaction aware modeling features (R 1, R 34, R 11). Most obviously, both languages offer no interpretation of transactions. Sequences of transactions reflect the functionality of TLMs. Therefore, it is necessary to describe properties in terms of transactions to allow for an abstract view on the DUV.

The next critical issue is that both assertion languages do not support triggering assertion evaluation based on time annotations to support synchronizing with the time annotations of a design (timed `wait`-statements, R 22). Of course, this could be modeled to a certain extent with additional behavioral code around assertions, but nevertheless, it is not part of these languages. Due to this lack, it is also not possible to support different abstraction levels (R 14) and mixes of these (R 15).

Also fully pipelined evaluation semantics, which would enable the detection of resource conflicts, or allow monitoring of pipelined data flow architectures and communication patterns are not available in PSL and SVA (R 33).

The use of both PSL and SVA for TL modeling poses severe restrictions. Extending these languages would require major changes of the underlying semantics.

4.1.4 Transaction Level Verification

To foster verification consistency and efficiency, methodologies have been developed for creating testbenches systematically. The big three Electronic Design Automation (EDA) vendors have each released good and comparable testbench methodologies - Advanced Verification Methodology (AVM) (Mentor Graphics Inc., [4]), Verification Methodology Manual (VMM) (Synopsys Inc., [28]), and Universal Reuse Methodology (URM) (Cadence Inc., [29]).

The key ideas behind these methodologies are composition of commonly used functional elements to reusable blocks and abstraction. Figure 4.5 shows an example testbench structure from Mentor Graphics' AVM[4].

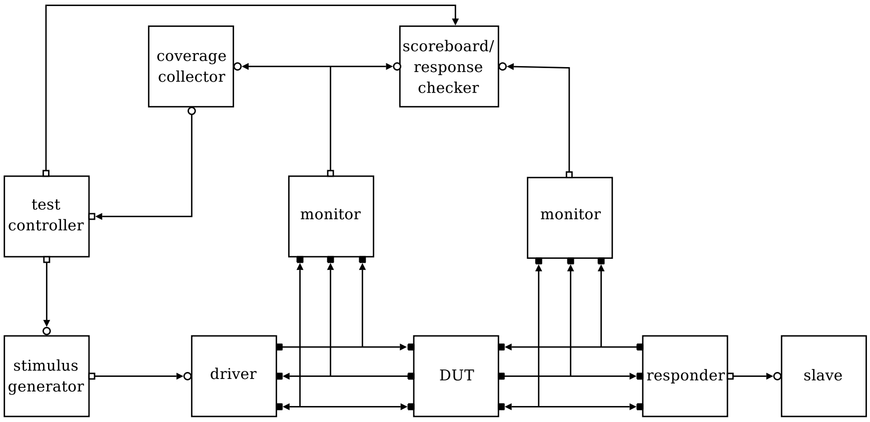


Figure 4.5: High-Level AVM Testbench Example [4]

As depicted in Figure 4.5, a testbench consists of several building blocks, including the Design Under Test (DUT)[2]:

Stimulus Generator	Generates abstract stimuli for the DUT; the generator can either produce constrained random or directed stimuli.
Driver	A BFM that translates abstract stimuli to RTL signal level accurate stimuli as input for the DUT.
Responder	A BFM that translates signal level accurate responses to abstract responses which can be sent to further blocks.
Monitor	A BFM that monitors the signal level protocol of the DUT and translates the observed patterns to abstract information.
Response Checker	Represents the golden reference and checks for compliance with the DUT; response checkers are also referred to as scoreboards
Coverage Collector	Gathers coverage information; this information is used to control the stimuli generation.
Test Controller	Utilizes coverage information to control stimuli generator.

This compositional approach enables the reuse of any block within another testbench. Hence, the development time of further testbenches for different DUTs can be reduced and can be kept consistent. Furthermore, a TLM can be used as a DUT as well, skipping the BFMs. Hence, the testbench can be reused for verifying both a TL and a RTL model.

Using assertions to monitor internal behavior of a DUT is a complementary approach. As shown in the testbench example, a checker is used in order to determine

[2] Within the testbench terminology a DUV is also called a DUT.

the input output equivalence of the DUT with the golden reference. The additional coverage information gained by assertions can be used for further refinement and control of the stimulus generation. Furthermore, assertions expand the verification scope to the internal behavior of the DUT. Hence, detecting internal bugs is not relayed to the response checker. Therefore, the complexity of a response checker can be reduced which in turn lowers the overall effort. BFMs in turn are used for bridging the gap between a high-level testbench and a low-level DUT.

4.2 Related Work

4.2.1 RTL Assertions in SystemC

Approaches have been presented which attempt to overcome SystemC's lack of temporal assertion support to at least enable RTL based ABV for RTL-style SystemC models.

Within a cooperation of IBM and the Weizmann Institute of Science a tool called FoCs [30] was developed which generates VHDL (VHDL) or Verilog implementations out of PSL property descriptions. This tool is enhanced further to generate a SystemC implementation in a similar fashion as an OVL monitor [31]. This allows a simulation based verification of PSL properties on SystemC models.

A further approach was presented by a company called Jeda Technologies Inc. [32], [33]. Here, the complete SVA subset of the SystemVerilog language has been ported to SystemC. This approach allows the notation of SVA properties natively in SystemC using a macro-style syntax. However, to the knowledge of the author, no clarification has been presented on how the differences between the SystemVerilog and SystemC simulation kernel have been overcome.

In [34], [35], [36], an approach has been presented which enables the evaluation of both PSL and SVA in conjunction with SystemC. The key idea behind the approach is the transformation of PSL and SVA assertions to abstract state machines. These in turn are translated to a C# implementation which is linked to a SystemC design. The assertions are connected to the design internals via signals and the designs clock is tapped off for triggering the assertion evaluation.

All the aforementioned approaches work under the assumption that the DUV is an RTL like implementation in SystemC. Hence, these approaches do not consider the requirements which are relevant for an application at TL.

As first steps of this work, a generator was developed which enables the generation of assertion monitors for all common HDLs, including SystemC, out of an Extensible

Markup Language (XML) description [37], [38]. One reason for doing this work, was to overcome the lack of assertion language support provided for SystemC designs in general. In addition to that, the goal was to have a consistent approach over different modeling languages. Furthermore, this work also represents a first step in the direction of using ABV at higher abstraction levels [39]. It is possible to configure the generated monitors such that the progression of time is not obtained via a clock-like trigger signal. The monitors are able to trigger themselves based on simulation time parameters specified in the XML entry. Hence, the monitors support the sampling of design states at specific simulation times (R 22). However, the approach was dropped because it was discovered that the specification of more complex properties required more and more features to be specified in XML. The XML description of one monitor which is implemented in a tree-like structure, became more and more complicated. Therefore, it was decided to engineer a declarative notation for assertions in terms of a language. This also allowed extending the language incrementally towards more TL oriented features.

4.2.2 Transaction Level Assertion Approaches

Finite Linear Temporal Logic

In [40], [41] an approach is presented which defines a bounded version of LTL called Finite Linear Temporal Logic (FLTL) and an implementation of FLTL formulas in SystemC based on Accept-Reject automata. The same logic is used for the verification of TLMs [42], [43]. Transactions are considered atomic, hence, reasoning about overlapping transactions is not possible (R 30). Furthermore, FLTL requires the reasoning on a global order of events, instead of partial orders (R 19). This requires the consideration of all possible event occurrences when specifying temporal distances in terms of event order, which is tedious work for a verification engineer. This also has the disadvantage that any property specified for a model might be falsely violated, in case the model is slightly changed. The violation does not reflect a real error because its temporal axis is altered rather than the monitored behavior. Furthermore, the approach does not account for different abstraction levels (R 14, R 15). It is not possible to reason about simulation time relations of events or Boolean propositions (R 22). In addition to these shortcomings it is not possible to express retransmission patterns (R 32) and pipelined behavior (R 33).

Structured Assertion Language for Temporal Logic

In [44], [45], [46] an approach is presented which defines a syntactical sugar layer called Structured Assertion Language for Temporal Logic (SALT) on top of LTL and

Timed Linear Temporal Logic (TLTL) formulae. A SALT formula is translated into an LTL or correspondingly into a TLTL formula which in turn is compiled into a ω-automaton representation [47]. Monitors generated this way are used to analyze behavior logs of a real-time SW-program. LTL in general, enforces a state-centric view when specifying properties. TL modeling in contrast to that, is a communication-centric view of a system. It is more natural to specify TL properties in terms of transactions, rather than states. Generally, using LTL for the specification of TL properties suffers from the same disadvantages as the FL subset of PSL. Therefore, no retransmission patterns (R 32), no pipelining (R 33) and no capturing of dynamic temporal behavior (R 23) is available among other things.

Logic of Constraints

In [48] [49] an approach is presented which combines LTL with Logic of Constraints (LOC) for performing simulation based ABV on abstract SystemC models. LTL is used for specifying temporal checks and LOC is used for specifying performance checks. Events are to be recorded with additional associated values. Both assertions and LOC formulae work on these traces either on-the-fly or stand-alone. LOC formulae correlate the associated values of events based on counters which reflect the number of occurrences of each event. For instance, the associated values of the fifth occurrence of an event *e1* are correlated with the corresponding values of the fifth occurrence of another event *e2*. This approach hence, reasons about the correlation between the order of occurrences of one event with the order of occurrences of another event. However, non-deterministic correlations can occur if these events are emitted from concurrent processes. Furthermore, if the design is updated or enhanced, all specified properties would have to be updated as well, since the order might have changed.

PSL for TL Modeling

In [50], [51] PSL is used as assertion language for expressing properties of TLMs and the reusability of assertions is claimed for lower abstraction levels. However, the approach lacks any notion of transactions (R 1, R 11, R 34). It also relies on SystemC signals to represent the state of the model. Hence, it is not possible to capture behaviors within a single delta-cycle (R 10) and to capture values of state variables (R 16, R 17). In PV models, only C-assert style assertions can be formulated. Hence, no temporal notion of a PV model is defined (R 14). It is also not possible to use the simulation time as basis for temporal reasoning (R 22).

UML Sequence Diagrams

In [52] Unified Modeling Language (UML) sequence diagrams are enhanced in order to enable an automated generation of PSL properties for TLMs. In this approach artificial clocks along with cycle durations for transactions are annotated to UML sequence diagrams. From these diagrams, PSL property skeletons are generated, which are manually refined from the class oriented description of an UML sequence diagram to an instantiation oriented description. As discussed in Chapter 3 a TLM can be modeled at abstraction levels which do not resolve the model behavior in cycles (e.g., PV). Hence, this approach is not adequate for the specification of assertions for all possible abstraction levels (R 14) and therfore, for mixes of abstraction levels (R 15).

SVA for TL Modeling

In [53], [54] an approach is taken to TL assertions with SVA as description language. In this approach, transactions are annotated with SystemC signals which are high while a transaction is ongoing and low otherwise. A clock signal is constructed which ticks every time when a transaction signal changes its value. The DUV is simulated and these signals are recorded in a Value Change Dump (VCD) file. Following that, this trace file is translated into a Verilog module. On this module, SVAs are evaluated which constitute the TL assertions. One major flaw of this approach is that only transactions which are called from within a suspendable context are supported, since extra delta-cycles for enforcing the transaction signal value-updates are required. Thus, this approach does not support non-blocking transactions called from within non-suspendable processes (R 11). Considering the fact that it is usually endeavored to prefer non-suspendable processes over suspendable ones, this restriction is not feasible. Also the use of a transaction clock as temporal reference for SVA enforces the consideration of a global order on the transactions. Here, again, the temporal relation between transactions might change when the design is updated. This leads to potential property errors which do not reflect design-errors, but changes to the global order of event occurrences (R 19). Furthermore, since SVA is used as description language, all shortcomings from SVA apply here as well, except for the difficulties with aligning the simulation kernel to SystemC. This is bypassed by using a VCD file as intermediate and Verilog as a derived design. Thus, SystemVerilog semantics apply. Since this approach is based on post-processing after a SystemC run, the detection of possible bugs is relayed to a second simulation (Verilog). This postpones the time for finding a bug. Hence, no on-the-fly assertion checking is supported within a SystemC simulation (R 5, R 2). Furthermore, this approach does not address the existence of different abstraction levels (R 14, R 15).

Native SystemC Assertions

In [55], [56], several concepts are introduced to enable TL assertions including an inlined specification in SystemC. For the PVT abstraction, a special event is introduced which preserves the order of notifications within one simulation time. Such an event is annotated in each implementation of a transaction in order to signal its occurrence. Sequences of these events can be specified and evaluated. However, it is not considered that transactions might not be called at all because no mechanism is defined to detect the absence of the corresponding events (R 20). Furthermore, it is not possible to include simulation time into the temporal reasoning. For instance, triggering a sequence at specific simulation times without using events is not possible. It is also not possible to specify a simulation time condition for an event (R 22). The use of the specially defined events anyhow does not allow for a more granular than delta cycle resolution of time for sampling (R 10) design states, for instance. This in particular is addressed in [55] by introducing a callback concept for PV models. This concept is similar in principle to the transaction detection capabilities of UAL[3]. Some temporal operators for PV models have been introduced which also allow the specification of strict partial orders on these callbacks. However, sampling of design-states is not defined (i.e., no definition is given on when to sample state values (R 26)). Furthermore, linking to transaction arguments (R 12) and generally a transaction aware description is not supported (R 34). A partial support for detecting pipelined patterns is provided, however, it requires a lot of additional code annotations to be done by a user, for each different case. Hence, a native support of detecting pipelined patterns is not provided (R 33). Additionally the pipelining support is only limited to PVT models. However, PV models may incorporate pipelined behavior as well. No support for detecting retransmission patterns is available (R 32). Mixing abstraction layers within one assertion is not supported as well because the underlying evaluation mechanisms require different kinds of events (i.e., callbacks or special events). Since, each transaction is associated with a single event, it is not possible to detect transactions which overlap partially or fully (R 30).

Temporal Logic of Actions

In [58] a formal approach to the specification of programs is introduced. The approach defines Temporal Logic of Actions (TLA). In contrast to LTL a state represents assignments to program variables. Such an assignment is called an action. Using temporal logic operators which are similar to LTL operators, actions are correlated over time. This approach does not define a notion of events. Hence, no partial ordering of events can be specified (R 19, R 20). Neither does it have a notion of

[3] The UAL concept has been published in July 2006 [57], one year before the publication of [55].

transactions. Furthermore, its close relation to temporal logics, does not allow the specification of dynamic temporal behavior (R 23), pipelining (R 33), and retransmits (R 32).

Duration Calculus with Phase-Event Automata

In [59], [60] Duration Calculus (DC) is used for specifying properties of real time systems. Formulae specified in DC allow the specification of durations of states also called phases and temporal correlations of phases. Hence, DC could be used to formulate at least simulation time based temporal relations of Boolean propositions (R 22). DC formulae, however, need to be translated into specific phase event automata. This is required to interpret a DC formula over a given behavior. The algorithms which exist for doing such a conversion, however, suffer in terms of complexity and hence, can only handle less complex formulae. Furthermore, a DUV needs to be implemented as a phase event automaton as well. This can only be handled for blocks of small complexity and contradicts the TL modeling style which is much closer to an architectural description of a system.

5 Universal Assertion Language (UAL)

This chapter introduces a newly developed assertion language UAL. UAL enables the specification of assertions for all common abstraction levels including TL and RTL. UAL is primarily extended over classical RTL assertion languages to fulfill the requirements discussed and summarized in Chapter 3. After providing a short overview of the basic concepts of UAL, a detailed description of the language is given.

5.1 Overview of UAL Concepts

This section provides a brief overview of the basic concepts of UAL. Like in common assertion languages as PSL and SVA the structure of UAL is organized in several layers. Figure 5.1 depicts the layered organization of UAL-assertions. As illustrated

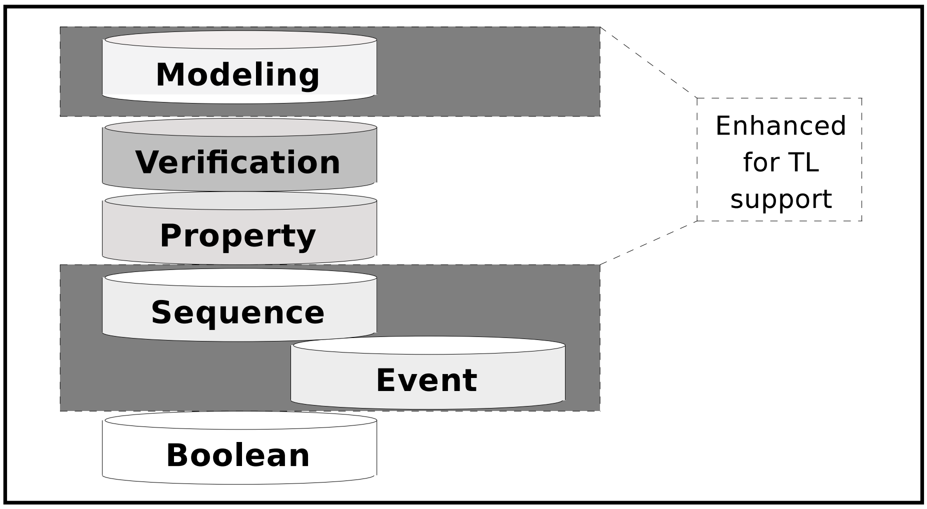


Figure 5.1: Layered Approach of UAL -assertions

UAL is organized in six layers. The modeling layer as well as the sequence layer have been enhanced to allow a transaction aware description of assertions. The event layer is a new layer when compared to PSL and SVA, and other approaches. This layer provides the key features of UAL with regard to capturing multiple abstraction layers while keeping the sequence layer as general as possible.

The basic building blocks in UAL are monitors, verification directives, properties, and sequences. One assertion is a combination of a verification directive with a property. One property in turn is build on top of sequences. These blocks can be directly mapped to the according layers in Figure 5.1, starting from top. Both the event and Boolean layer is used within sequences. Figure 5.2 depicts an example of one assertion embedded in a monitor. All constructs are mapped to the according layer. The dashed arrows indicate the instantiation hierarchy. The black arrows indicate the evaluation order of the assertion.

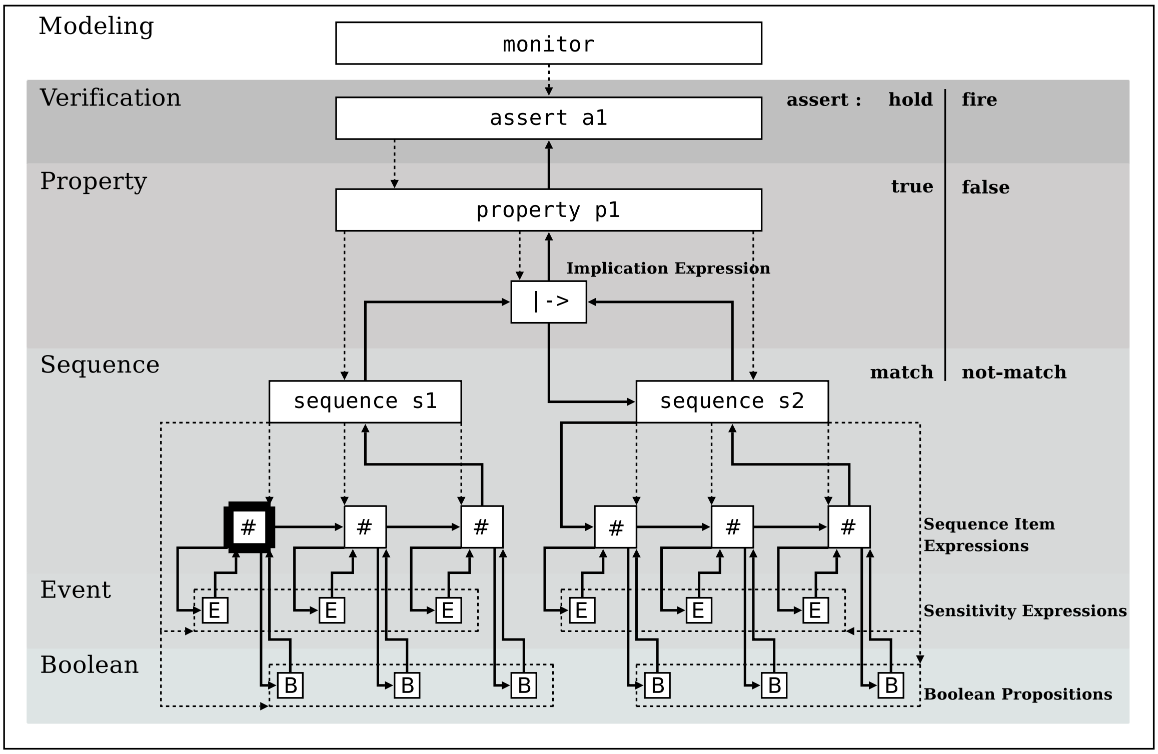


Figure 5.2: Assertion Structure

Generally, assertions in UAL are encapsulated in monitors (modeling layer). A verification directive on the one hand is necessary for enabling a property evaluation continuously. On the other hand, it expects a true / false result for each evaluation of the associated property. Depending on the result of the property a verification directive reacts differently. The example in Figure 5.2 shows an `assert`-directive. This directive fires in case of a false result returned by the property evaluation.

A property returns its corresponding evaluation results to the associated verification directive. An enabled property continuously enables the evaluation of its leftmost sequence instance. In the example in Figure 5.2, the leftmost sequence is `s1`. Two forms of properties can be specified with UAL - single sequence or implication properties. As the name indicates, a single sequence property consists of a single instance. The single sequence property expects a match / not-match result for each evaluation of the sequence. An implication property consists of two sequence instances which are connected via the UAL implication operator. This operator calculates the property evaluation result based on the results of its operand sequences. Figure 5.2 shows an example of such an implication property. Basically, the implication operator returns a result of value *true* upon either a not-match result of its LHS sequence or if the RHS sequence returns a match. The evaluation of the RHS sequence is started only for each match of the LHS sequence. If the RHS sequence returns a not-match, the implication operator returns a result of value *false*. The behavior of the implication operator can be influenced by setting a property mode. This is explained in detail in Section 5.4.3.

A sequence is the specification of a temporal behavior which is attempted to be observed when monitoring the behavior of a design under scrutiny. If the specified behavior is observed the sequence returns a match result, otherwise, it returns a not-match. A sequence returns its result to either a property directly (i.e., in single sequence properties) or to an implication operator (i.e., in an implication property). A sequence contains delay operators which are chained together. The delay operators are evaluated from left to right. A sequence that is enabled by a property, enables its leftmost delay operator continuously for evaluation. The evaluation of a delay operator returns a preliminary match / not-match result. As soon as the evaluation of one delay operator is finished the next delay operator in the chain is enabled. The preliminary result of the whole sequence corresponds to the evaluation result of the rightmost delay operator. A sequence is parameterized with an evaluation mode which determines whether the preliminary sequence result turns into the final result of the sequence. A detailed introduction and discussion of sequences is given in Section 5.5.

A delay operator can be distinguished in three categories. It can either be a zero-step delay operator, a multi-step delay operator, or a range-step delay operator. The evaluation of a delay operator is event-driven. A delay operator is configured with a sensitivity with regard to event occurrences. As soon as a delay operator is enabled it enables its sensitivity and suspends the evaluation until an event occurrence has satisfied the sensitivity as many times as specified by its step-configuration (i.e., zero-step, multi-step, step-range). If the evaluation of a delay operator is resumed the Boolean proposition is evaluated. In case of a zero-step delay operator no suspension takes place and the Boolean proposition is evaluated immediately. The leftmost delay operator (see highlighted delay operator in Figure 5.2) of an enabled sequence continuously starts new evaluations with every occurrence of events that fulfill its sensitivity. Therefore, several instances of a sequence evaluation, referred to as threads, can run in parallel. The delay operator is discussed in detail in Section 5.5.1.

The sensitivity of a delay operator is expressed with constructs offered by the UAL event layer. A detailed discussion of this layer is given in Section 5.6.

Boolean propositions are formulated using the UAL Boolean layer which is discussed in Section 5.7.

The following sections provide a detailed description of each UAL layer in descending order. The grammar is defined in an Extended Backus-Naur-Form (EBNF) form. The most important grammar rules are mentioned in the following sections and can also be found in Appendix B. A corresponding reference is provided with the rule descriptions given in the upcoming sections.

Table 5.1 provides a legend for the main EBNF syntax.

EBNF	**Description**
`"a"`	Terminal
a	Rule
$a = ...$	Assignment
$a\ b$	Concatenation
$a \mid b$	Separation
$\{\ a\ \}$	$0 \rightarrow \infty$ Repetition
$a\ \{\ a\ \}$	$1 \rightarrow \infty$ Repetition
$[\ a\ ]$	Option
$(a \mid b)\ c$	Grouping

Table 5.1: EBNF Syntax Description

5.2 Modeling Layer

Assertions are encapsulated within so-called monitors. This enables organizing related assertions to corresponding libraries (analogous to the concept of OVL) in order to leverage assertion reuse (see R 41). The specification of a monitor is defined by the following rule:

```
monitor          =  "monitor" identifier                        B.1,
                    ports_section                               p.203
                    [ constants_section ]
                    sequences_section
                    properties_section
                    verification_section
                    "endmonitor" ;
```

As the rule shows, the body of a monitor follows a hierarchical structure which is comprised of five sections. This hierarchical concept was chosen to provide a clear structure for the overall assertion specification. All items must be declared before they can be used. No forward and no recursive specification is allowed. This is the reason why the sections within a monitor are ordered as defined in Rule B.1.

These sections are described in the following.

5.2.1 Ports Section

The ports section describes the interface of a monitor according to the following rule:

```
ports_section    =  "ports"                                     B.2,
                    port_declaration { port_declaration }       p.203
                    "endports" ;
```

Within the ports section all ports of a monitor are specified. Ports always have an ingoing direction. This means, they provide a read-only access (see R 27) to the elements which are connected to them. Thus, it is ensured that no assertion may manipulate design data (see R 28). A port declaration is defined as follows:

```
port_declaration =  kind type identifier [ "[" number "]" ]     B.3,
                    [ transaction_parameters ] ";" ;            p.203
```

Besides the *kind* specifier, which is explained in the next paragraph, a port declaration consists of a *type* specifier (see Rule B.98, p.210). The type of a port has to correspond to the data type of the object to which it shall be connected. The type can be any SystemC or C++ data type. The optional number specified in anchor brackets next to the port identifier indicates that the element to which a port can be connected is an array which *number* elements. The *transaction_parameters* specifier is reserved for ports of kind *transaction*.

The first entry of a port declaration is a kind specifier which indicates the kind of design element a port can be connected to. The *kind* specifier addresses the ability to link assertions to state variables (R 16), signals (R 24), events (R 8), and transactions (R 12), as will be illustrated in the following paragraphs.

The following kinds are defined in UAL:

```
kind            = "state"                                        B.66,
                | "event"                                        p.207
                | "signal"
                | "transaction" ;
```

A port of kind *state* can be connected to any design object which stores data. Via the port, the value of the connected element can be read by referencing its identifier. No further information on the element is accessible.

A port of kind *event* can be connected to any event object in a design. The type indicates whether the event object is an annotated SystemC or special UAL event. An introduction to available event objects in UAL is given later in Section 5.6.1. The event object connected to an event port can be referenced by the port identifier.

A port of kind *signal* can be connected to any design object which stores data of the same type as the port's type. Through the port the value of the connected element can be read by referencing the port identifier but in contrast to kind *state* also the event occurrences emitted by that design element can be accessed. This can be for instance a SystemC signal which emits value-change events.

The kind *transaction* is defined in order to address the requirement for transaction aware assertion specification (R 34). A port of kind transaction can be connected to any design object which is a transaction, e.g., any function modeled in the design. The type of a port corresponds to the target functions return type. No arrays are allowed in conjunction with this port kind. The *transaction_parameters* specifier which is reserved for the kind *transaction* as mentioned earlier, defines the argument list of the target function. Via a transaction port it is possible to access events issued by the connected transaction. This is explained in detail in Section 5.6.1. Furthermore, a transaction port provides read-access to the arguments of the connected transaction and its return value. Transaction arguments and the return value can be accessed by referencing the port identifier as LHS-operand and the corresponding argument identifier as RHS-operand of a dot operator, as indicated by the last two alternatives of the following rule:

```
operand         = lastevent                                      B.41,
                | value                                          p.205
                | ( identifier [ "." identifier ] [ "[" array_index "]" ] )
                | ( identifier [ "." "RET" ] ) ;
```

Access to the return value of a transaction is obtained through the same operator, however the identifier for the return value is a reserved keyword called *RET*.

Since both kinds *signal* and *transaction* represent compound elements which offer events a corresponding event access operator is defined and a list of available events:

```
event_operand    = identifier [ "[" array_index "]" ] [ "'" event_kind ] ;  B.49,
                                                                             p.206

event_kind       = "START"                                                  B.97,
                   | "END"                                                  p.210
                   | "POS"
                   | "NEG"
                   | "CH" ;
```

The first two event kinds are available for transactions and the remaining event kinds for signals. A detailed discussion of these events is given in Section 5.6.

Listing 5.1 shows an example of a UAL ports section containing a port declaration for each UAL port kind.

```
ports
 state sc_uint<32> Reg0;
 event sc_event e2;
 signal sc_signal<sc_uint<32> > sig1;
 transaction int write (int addr, int data);
endports
```

Listing 5.1: Example: Ports Section

5.2.2 Constants Section

The constants section contains all constant declarations. Constants can be freely used within the monitor. A constants section is optional and is defined according to the following rule:

```
constants_section   = "constants"                                          B.4,
                      constant_declaration { constant_declaration }        p.203
                      "endconstants" ;
```

A constant declaration is of the following form:

```
constant_declaration= type identifier "=" value ";" ;                      B.5,
                                                                           p.203
```

The type of a constant is defined in the same way as the type of a port.

5.2.3 Sequences/Properties/Verification Sections

The sequences, properties, and verification sections provide a hierarchy to model sequences, properties, and verification directives respectively.

The sequences section encapsulates all sequence declarations and the properties section all properties respectively. A sequences section and a properties section can be specified according to the following rules:

```
sequences_section   =  "sequences"                                        B.6,
                       sequence_section { sequence_section }              p.203
                       "endsequences" ;

properties_section  =  "properties"                                       B.7,
                       property_section { property_section }              p.203
                       "endproperties" ;
```

A *sequence_section* contains the declaration of a sequence as defined by the following rule:

```
sequence_section    =  "sequence" identifier sequence_interface           B.23,
                       sequence_declarations                              p.204
                       sequence_specification
                       "endsequence" ;
```

A *property_section* contains the declaration of a property as defined by the following rule:

```
property_section    =  "property" identifier property_interface           B.14,
                       property_declarations                              p.204
                       property_specification
                       "endproperty" ;
```

The discussion of the specifiers *sequence/property_declarations* and *sequence/property_specification* is relayed to Sections 5.5 and 5.4. Within this section the corresponding interface specifiers *sequence/property_interface* shall be addressed. These specifiers are defined as follows:

```
property_interface  =  [ property_mode_list ] formal_argument_list ;      B.15,
                                                                          p.204
sequence_interface  =  [ "[" sequence_mode "]" ] formal_argument_list ;   B.24,
                                                                          p.205
```

The meaning of specifiers *property_mode_list* and *sequence_mode* is explained later in Sections 5.4 and 5.5. It shall suffice to say that these specifiers configure the behavior of property and sequence evaluations. The specifier *formal_argument_list* describes the arguments that are to be passed to a property or a sequence instance.

A *formal_argument_list* is defined as follows:

```
formal_argument_list    =  "(" [ formal_argument_decl ]                   B.60,
                           { "," formal_argument_decl } ")" ;             p.207
```

A formal argument declaration is defined according to the following rule:

formal_argument_decl = ["ref"] [*kind*] [*type*] *identifier* B.61,
[*transaction_parameters*] ; p.207

The reserved keyword *ref* indicates that an argument is passed by reference into the according body. It may only be used to pass local variables into a sequence. As the declaration rule indicates, arguments are declared the same way as ports but a comma is defined as separator. When mapping local arguments to formal arguments the principle of positional mapping is applied. Listing 5.2 shows an example of a formal argument list for a sequence declaration.

```
sequence s1 (
 ref sc_uint<32> local_var ,
 event sc_event e2 ,
 signal sc_signal<sc_uint<32> > sig1 ,
 transaction int write (int addr , int data))
...
endsequence
```

Listing 5.2: Example: Formal Argument Lists

The verification section contains all verification directive declarations and can be specified according to the following rule:

verification_section = "verification" B.8,
directive { *directive* } p.203
"endverification" ;

A detailed explanation of the specifier *directive* is given in Section 5.3.

5.3 Verification Layer

The verification layer is comprised of all verification directives available in UAL. A verification directive is associated with a property and specifies that this property is to be evaluated and how the results have to be treated. A declaration of a verification directive consists of a directive kind followed by an identifier and parameters. Such a directive is assigned an instance of a property:

directive = *directive_kind* *identifier* "(" [*directive_parameter*] ")" B.9,
"=" *property_instance* ";" ; p.203

By assigning a property instance to a directive the property instance is enabled for continuous evaluation.

The following kinds of directives are defined in UAL:

```
directive_kind      = "assert"                                B.10,
                      | "cover"                               p.204
                      | "assert_cover"
                      | "assume" ;
```

The parameters to a directive are specified as follows:

```
directive_parameter      = severity_level                     B.11,
                           "," string                         p.204
                           [ "," reset_event_expr ] ;
```

Since not all directive parameters are applicable with all directives, they are discussed subsequently with their directives.

An *assert*-directive asserts that the associated property always succeeds. Whenever the property evaluates to false an interaction with a simulator has to be invoked according to the specified severity level which is set using a parameter. In all cases the interaction includes the displaying of the specified report string. This parameter fulfills the requirement for supporting customizable report messages in order to ease debugging of falsified properties (see R 40). To address the requirement for characterizing falsified properties with a severity (see R 39) UAL defines the following severity levels:

```
severity_level           = "INFO"                             B.12,
                           | "WARNING"                        p.204
                           | "ERROR"
                           | "FAILURE" ;
```

By default severity levels *INFO* and *WARNING* may not stop a simulation, but are to display the report string if the property assigned evaluates to false. Severity levels *ERROR* and *FAILURE* are to halt the simulation in addition to displaying the report string. Through a simulator a user shall be given the possibility to override these settings.

An example *assert*-directive statement is shown in Listing 5.3.

```
assert A0(ERROR,"ReportMsg",e1) = p1(a,b,c);
```

Listing 5.3: Example: Assert Directive

This statement describes that a property *p1* which is mapped to ports *a,b,c* is asserted with a severity level set to *ERROR*, a report message *"ReportMsg"* which is displayed if the property returns a result of value false. The evaluation of property *p1* is reset with any occurrence of the event *e1*.

A *cover*-directive counts all property evaluation results. If the property evaluates to false no interaction takes place with a simulator. While counting the property evaluation results a *cover*-directive distinguishes in property successes (result is true) and failures (result is false) in order to fulfill the requirement to offer the same coverage features as in PSL and SVA (see R 36). Property successes are distinguished further into real and vacuous successes. The meaning of this success characterization is explained in Section 5.4. The specification of a severity level and a report string bears no meaning in the context of a *cover*-directive. Such directives shall not have directive parameters other than a reset expression.

An example *cover*-directive statement is shown in Listing 5.4

```
cover C1(e1) = p1(a,b,c);
```

Listing 5.4: Example: Cover Directive

This statement describes that the evaluation of property *p1* is covered. The evaluation of the property is reset with any occurrence of event *e1*.

An *assert_cover*-directive is a combination of the directives above. It acts as an *assert*-directive which also counts the property evaluation results according to a *cover*-directive. It shall have the same set of directive parameters as an *assert*-directive.

An *assume*-directive is meant for later formal analysis. In formal analysis the associated property has to be assumed to be true, hence restricting the state space for the analysis. In simulation it acts the same way as *assert_cover*. Further on, it is recommended that the *assume*-directive is only applied in conjunction with constraints on external interfaces.

The parameter *reset_event_expr* specifies a condition based on event expressions under which the associated property has to be reset. All ongoing evaluations in the associated property are stopped and canceled immediately. Resetting the property may not reset the collected coverage. The structure of the parameter *reset_event_expr* and thus event expressions is explained in more detail in Section 5.6.

5.4 Property Layer

The property layer of UAL is used to give sequence evaluation results a propositional meaning. Within the property layer, properties are specified which express an intended behavior. A property can thus either be true or false. A property is not evaluated unless it is instantiated in a context which is associated with a verification directive from the verification layer.

A property declaration has to be specified as indicated by the following rule:

property_section = `"property"` *identifier property_interface* B.14,
property_declarations p.204
property_specification
`"endproperty"` ;

Property declarations have an optional default property evaluation mode setting which can be used to override the default settings. Generally, a formal argument list has to be specified which defines the interface of a property. Furthermore, local variables can be declared within properties. A property itself may not manipulate a local variable, but pass it either by reference or by copy to its underlying sequence instances.

A property is instantiated according to the following rule:

property_instance = *identifier* [*property_mode_list*] B.20,
param_argument_list ; p.204

According to this rule, an instance is obtained by referencing the identifier of a property, by optionally setting the mode parameter, and by passing arguments to the property interface (see also Rule B.63, p.207).

Generally, UAL properties can be categorized into two classes - implication and single sequence properties - as indicated by the following rule:

property_specification = *implication_property* B.17,
| *single_sequence_property* p.204

These two property classes are explained in the upcoming sections. The meaning of property modes and their default settings for both implication and single sequence properties is explained in connection to that.

5.4.1 Implication Properties

Implication properties are used to specify a desired behavior which has to be observed only after a precondition has been fulfilled. Hence, the fulfillment of a precondition implies the validity of a subsequent behavior. An implication property is constructed through the use of the UAL implication operator (`|->`). The following rule shows the definition of an implication property:

implication_property = *sequence_instance* `"|->"` *sequence_instance* `";"` ; B.18,
p.204

This operator is a property operator since it may only be used within a property. However, its operands are sequences. The LHS-sequence is called the antecedent and the RHS-sequence is called the consequent. The evaluation of the consequent is only enabled upon a match result of the antecedent. The antecedent is always ready for evaluation if the surrounding property is enabled. An implication operator can produce three possible results:

- Vacuous Success
- Real Success
- Failure

A vacuous success is produced for all cases where the antecedent produces a not-match result. Since, the overall property evaluation result is Boolean, a vacuous success of an implication leads to a result of value true for the property.

When the antecedent produces a match the evaluation of the consequent is started. If this evaluation produces a match as well the whole implication produces a real success. A real success maps also to a result of value true for the property.

A failure is produced for all cases where the antecedent produces a match result and the consequent does not. Hence, a failure maps to a result of value false for the property.

If an implication property is associated with a verification directive which collects coverage, property results of value true are distinguished according to the categorization of the implication operator results.

The formal semantics of the implication operator are defined in Section 6.6.

Listing 5.5 shows an example implication property declaration. In the example, two sequences are instantiated which take a local variable and a transaction as argument.

```
property p_implication (transaction void PUT(int x))
    int local_var;
    s1(local_var,PUT) |-> s2(local_var,PUT);
endproperty
```

Listing 5.5: Example: Implication Property

5.4.2 Single Sequence Properties

A single sequence property declaration contains only one sequence instance in the body:

single_sequence_property = *sequence_instance* ";" ; B.19, p.204

The sequence instance of a single sequence property is continuously enabled if the enclosing property is enabled, analogously to the antecedent of an implication property.

The result of the sequence evaluation, is directly transformed to a Boolean result of the enclosing property. If the sequence instance produces a not-match result the enclosing property evaluates to false. A match hence, evaluates to true. In terms of coverage, a property with result true is counted as real success, a property with result false as failure. A vacuous success is not possible with single sequence properties.

Listing 5.6 shows an example of a single sequence property declaration.

```
property single_sequence_property(
        event E1, state int D1)
    s1(E1,D1);
endproperty
```

Listing 5.6: Example: Single Sequence Property

5.4.3 Property Evaluation Modes

As already mentioned, a property can be parameterized with a mode setting in order to influence its evaluation. If a mode setting is provided, the default evaluation mode is overridden and the new setting becomes new default for this property. If a property instantiation is configured with a mode the default value is overridden.

The according parameter list for setting the property evaluation mode is defined in the following form:

property_mode_list = `"["` [*sequence_mode* `","`] *property_mode* `"]"` ; B.21, p.204

In case of implication properties the specifier *property_mode_list* always consists of two parameters. The parameter *sequence_mode* sets the evaluation mode of the antecedent, whereas the parameter *property_mode* determines the evaluation mode of an implication operator and its consequent sequence. The default mode setting for an implication property is *AnyMatch* mode for the antecedent sequence and *Overlap* mode for the implication operator. As an example, the specifier *property_mode_list* for explicitly describing this setting is formulated as follows: [**AnyMatch**,**Overlap**]

The available sequence modes are explained in Section 5.5.3.

In case of single sequence properties the specifier *property_mode_list* only consists of parameter *property_mode*. The default mode is *Overlap* mode.

The UAL property layer provides seven possible property modes. The modes offered are defined as follows:

```
property_mode  =  "Restart"                                B.22,
                | "NoRestart"                              p.204
                | "ReportOnRestart"
                | "Overlap"
                | "Pipe"
                | "PipeOrdered"
                | "Cover" ;
```

The property mode setting determines the sequence evaluation mode for the consequent of an implication property or the sequence instance in a single sequence property. Table 5.2 shows how the corresponding sequence mode is derived from the property mode.

Property Mode	**Consequent**	**Single-Sequence**
Restart	*FirstMatch*	*N/A*
NoRestart	*FirstMatch*	*N/A*
ReportOnRestart	*FirstMatch*	*N/A*
Overlap	*FirstMatch*	*FirstMatch*
Pipe	*FirstMatchPipe*	*FirstMatchPipe*
PipeOrdered	*FirstMatchPipeOrdered*	*FirstMatchPipeOrdered*
Cover	*N/A*	*AnyMatch*

Table 5.2: Property Mode Derivation

Evaluation Modes for Retransmission Patterns

The property modes which address the requirement for dealing with retransmission behaviors (see Sec. 3.7) are *Restart*, *NoRestart* , and *ReportOnRestart* (see R 32). These three modes may not be used in conjunction with single sequence properties for they affect the evaluation of an implication operator. Each mode handles the occurrence of an antecedent match differently if the consequent is still under evaluation because of a prior match of the antecedent.

As shown in Table 5.2, for property modes *Restart*, *NoRestart*, and *ReportOnRestart*, the sequence evaluation mode for the consequent of an implication property is derived to the mode *FirstMatch* (Sec. 5.5.3).

In *Restart* mode, an antecedent match forces any ongoing evaluation of the consequent to be canceled and restarts the consequent evaluation. A canceled evaluation bears no meaning in terms of a property result. This mode is appropriate for instance when monitoring request/response communication patterns, where the retransmission of a request is allowed. The match of the antecedent sequence indicates an issued request and the match of the consequent indicates an issued response.

In *NoRestart* mode, an antecedent match is ignored if there is at least one ongoing evaluation of the consequent. In contrast to that, in *ReportOnRestart* mode such a match is reported to the user. In case a report is issued it is ignored in terms of coverage because it does not yield a property result. The severity for such a report corresponds to level *INFO* and is fixed. Similarly to mode *Restart*, these two modes are appropriate for monitoring request/response communication patterns. However, mode *NoRestart* is useful for checking timing relations between the first request and its response. Mode *ReportOnRestart* is useful for detecting that retransmissions occur.

All these modes have in common that no parallel evaluations of the consequent can exist.

Overlapped Evaluation Mode

The *Overlap* mode corresponds to the semantics of PSL and SVA and hence, addresses the requirement for support of evaluation modes of current assertion language standards (see R 35). This mode allows an overlapped evaluation of a property. In implication properties, each match of the antecedent starts an additional evaluation in the consequent. For single sequence properties the sequence instance may start further evaluations while other evaluations are ongoing. Each evaluation is computed individually. Hence, in contrast to the property modes mentioned in the last section, several evaluations of the consequent can exist in parallel. No evaluation may have a side-effect to another ongoing evaluation of the same property in the same instance. This mode makes it possible that a consequent sequence instance or single sequence instance produces a match for each evaluation simultaneously at the same occurrence of one event. Which means that several overlapped evaluations of a property in *Overlap* mode may terminate successfully at the occurrence of one event. This can happen if the degree of overlap in the evaluation is big enough such that an evaluation of the consequent catches up with an earlier started and still ongoing evaluation. This is explained in more detail in Section 5.5.3. As shown in Table 5.2, in both implication and single sequence properties the *FirstMatch* mode is derived for the consequent sequence or single sequence instance.

Pipelined Evaluation Modes

The pipelined evaluation modes *Pipe* and *PipeOrdered* address the requirement to be able to observe pipelined behavior (see R 33). In both modes, the evaluation of properties may overlap. However, in contrast to *Overlap* mode, two parallel, overlapping evaluations may not succeed simultaneously with the same occurrence of one event. Furthermore, a design behavior which is observed by one running evaluation may not be considered by another evaluation which has been started later. This way it is ensured that two parallel evaluations may not succeed by observing the same design behavior in parallel.

The actual pipelined evaluation is solely done in the consequent or in case of single sequence properties, in the sequence instance. Therefore, a corresponding sequence mode is derived from these property modes, as shown in Table 5.2. In *Pipe* mode the sequence evaluation mode of a consequent or single sequence instance is set to *FirstMatchPipe*. The *Pipe* mode allows the observation of pipelined behavior not considering the order of the pipelining stages.

In *PipeOrdered* mode the corresponding sequence evaluation mode is set to *FirstMatchPipeOrdered* respectively. This mode allows the observation of pipelined behavior like the *Pipe* mode. However, this mode is best fit for monitoring ordered pipeline behavior. This means the order of how pipeline stages are filled has to correspond to the order observed at the output of the pipeline.

A detailed explanation of these sequence modes is given in Section 5.5.3.

Coverage-Oriented Evaluation Mode

The *Cover* mode may only be used in conjunction with a single sequence property which in turn is only associated with a *cover* directive. The *Cover* mode allows that the single sequence instance may produce several results, both match and not-match, for one evaluation. This can be the case if the sequence declaration of the corresponding instance contains specifications of alternatives. In this case, all alternatives are evaluated until they produce a result. This mode corresponds to the coverage semantics of SVA with regard to sequences and thus, also addresses the requirement for support of evaluation modes of current assertion languages (R 35). As Table 5.2 shows, the sequence evaluation mode derived for the single sequence in case of mode *Cover* is *AnyMatch* which is discussed in Section 5.5.3.

The formal semantics of all UAL property modes are defined in Sections 6.6 and 6.7.3.

5.5 Sequence Layer

A key feature of any assertion language is the ability to specify sequences. A sequence can be best explained as a temporal pattern of Boolean propositions formulated on the states of a model. These propositions have to hold in a specific temporal order which is defined by a required sequence of event occurrences. A sequence evaluation result can be either a match or a not-match. Sequences can be instantiated in properties, such that a property reasons about sequence results. The syntax for a UAL sequence declaration is defined as follows:

```
sequence_section  =  "sequence" identifier sequence_interface       B.23,
                     sequence_declarations                          p.204
                     sequence_specification
                     "endsequence" ;
```

A sequence declaration may contain declarations of local variables:

```
sequence_declarations   =  { localvar_declaration } ;               B.25,
                                                                    p.205
```

The meaning of local variables and how they are applied is explained in Section 5.5.2.

A sequence specification is comprised of delay operators which express the temporal order of Boolean propositions:

```
sequence_specification  =  delay_operator { delay_operator } ";" ;   B.26,
                                                                    p.205
```

How temporal patterns are specified with delay operators is explained in detail in Section 5.5.1.

The evaluation of a sequence instance is continuously enabled if it is the leftmost sequence instance in a property, which in turn, is enabled by a verification directive. A sequence instantiation is obtained by referencing the sequence name, by optionally setting a default sequence mode (see Sec. 5.5.3), and by setting the argument list:

```
sequence_instance  =  identifier  [ "[" sequence_mode "]" ]          B.39,
                      param_argument_list ;                          p.205
```

In the remainder of this section, the structure and the evaluation of a sequence specification is described.

5.5.1 Sequence Specification

As mentioned earlier, a sequence specification, can be formulated by chaining delay operators from left to right (see Rule B.26, p.205). The evaluation of a sequence specification progresses from left to right. Hence, the temporal progress is specified from left to right. The start of an evaluation of an enabled sequence is defined by its leftmost delay operator instance. Within sequences one evaluation is referred to as thread. The result of a thread as it proceeds through a sequence specification can either be a preliminary match or a preliminary not-match. After a thread has proceeded through the whole sequence specification it terminates and its final result is calculated by the sequence evaluation mode.

The syntax for the specification of a delay operator is defined as follows:

delay_operator = "#" *steps sensitivity* "{" *condition* { *action* } "}" ; B.27, p.205

As Rule B.27 shows, it is required to specify *steps* and *sensitivity* for a delay operator. Furthermore, a delay operator includes a Boolean proposition as indicated by the specifier *condition* and optionally an action as indicated by the specifier *action*. The meaning of the latter is explained in Section 5.5.2. Section 5.7 describes how Boolean propositions are specified. In general, an action is only executed after the evaluation of the Boolean proposition (i.e., no Boolean proposition may be specified at the RHS of an action).

A thread that reaches a delay operator, immediately enables the sensitivity of this operator and suspends. The specifier *steps* determines for how long a thread has to be suspended. The specifier *sensitivity* determines when to evaluate whether a thread has to be resumed again. This evaluation has to happen for *steps* times in order for the thread to resume. When a thread is resumed in a delay operator, the corresponding Boolean proposition is evaluated, followed by the execution of the corresponding action, if present. If the proposition evaluates to true the preliminary result of this thread is a match, otherwise, a not-match. In case of a match the thread proceeds to the next delay operator in a sequence specification. In case of a not-match the thread is terminated and the sequence evaluation mode determines the final result for this thread.

The syntax rules for the specifier *sensitivity* are defined as follows:

sensitivity = "{" [*pos_sensitivity*]
[";" *neg_sensitivity*] "}" ; B.32, p.205

pos_sensitivity = *trigger_expression* ; B.45, p.206

neg_sensitivity = *trigger_expression* ; B.46, p.206

As the first rule shows, the sensitivity of a delay operator can consist of two parts - *pos_sensitivity* and *neg_sensitivity*. Both specifiers are mapped to the specifier *trigger_expression*. A trigger expression consists of constructs offered by the event layer, which is described in Section 5.6. A trigger expression formulates a condition on event occurrences. If the condition is satisfied, the trigger expression emits a trigger. Upon such a trigger occurrence the delay operator evaluates whether to resume a thread. The result of a trigger expression used for *pos_sensitivity* from now on is referred to as the occurrence of a positive trigger and, if used for *neg_sensitivity*, it is referred to as the occurrence of a negative trigger. Generally, the setting for specifier *steps* denotes how many occurrences of a positive trigger are required in order for a thread to be resumed. As soon as a thread is resumed the Boolean proposition of the delay operator is evaluated. Occurrences of a negative trigger terminate threads immediately and set the preliminary result to not-match.

The syntax rule for the specifier *steps* is defined as follows:

```
steps           =  zero_step                                B.28,
                   | multi_step                             p.205
                   | range_step  ;
```

As the rule shows three possible settings for the specifier *steps* are defined - *zero_step*, *multi_step*, or *range_step*.

The syntax rule for specifier *zero_step* is defined as follows:

```
zero_step       =  "0"                                      B.29,
                   | ( "{" "0" "}" )  ;                     p.205
```

Setting the specifier *steps* to a *zero_step* denotes that a thread that reaches such a delay operator is not suspended at all. Hence, the Boolean proposition is evaluated immediately. Since, specifying a sensitivity in combination with a *zero_step* setting, does not make sense, it is defined that the setting for specifier *sensitivity* must be empty. An example of a delay operator with a *zero_step* configuration can be specified as follows:

$$\#0\{\}\{\mathbf{true}\}$$

The syntax rule for the specifier *multi_step* is defined by Rule B.30:

```
multi_step      =  non_zero_number                          B.30,
                   | ( "{" non_zero_number "}" )  ;         p.205
```

Setting the specifier *steps* to a *multi_step* denotes that a thread that reaches such a delay operator has to be suspended for as many occurrences as indicated by the setting of specifier *multi_step*. The specifier *multi_step* can be set to any natural number not equal to zero. An example delay operator with a *multi_step* configuration can be specified as follows:

$$\#5\{\mathrm{e1}\}\{\mathbf{true}\}$$

This example denotes that a thread that reaches the delay operator is suspended until five occurrences of an event *e1* are encountered. A reference to an event in a *trigger_expression* turns an event occurrence into a trigger. The following example also shows the use of a delay operator configured with a *multi_step* and a *neg_sensitivity*:

#5{e1;e2}{**true**}

This example denotes in addition to the previous example, that a thread which is suspended is terminated if one occurrence of an event *e2* is encountered before the thread is resumed again. If a thread is terminated, its result is set to a preliminary not-match. The final decision is relayed to the evaluation mode of a sequence.

The syntax rule for the specifier *range_step* is defined as follows:

range_step = "{" *number* ":" *number* "}" ; B.31, p.205

Setting the specifier *steps* to a *range_step* denotes that a thread is suspended for at least as many positive trigger occurrences as specified by the left number and at most as many occurrences as specified by the right number. Since, the setting of specifier *steps* corresponds to an interval it can be considered as a specification of alternative delay operators with a *multi_step* configuration. Each alternative for the *multi_step* configuration corresponds to a value from within the interval of the *range_step* configuration. Hence, a thread that reaches a delay operator with a *range_step* configuration is split into so called sub-threads for each alternative. These sub-threads are evaluated in parallel. An example delay operator with a *range_step* configuration can be specified as follows:

#[5:7]{e1}{**true**}

This example denotes that a thread is suspended for at least five and at most seven occurrences of the event *e1*. Hence, a thread that reaches this delay operator is split into as many alternatives as allowed by the interval. In this example, a thread is split into three sub-threads $(7 - 5 + 1)$. Each sub-thread suspends for a specific number of event occurrences that lies within the interval (i.e., one sub-thread for five, one sub-thread for six, and one sub-thread for seven event occurrences).

As mentioned earlier, the leftmost delay operator in a sequence which is enabled by a property, defines when threads have to be created, in order to allow for a continuous monitoring of design behavior. Initially one thread is created when a sequence is enabled by a property. This thread enables the first delay operator and suspends immediately. As soon as this delay operator receives a positive or a negative trigger, a new thread is created. Hence, threads start with every occurrence of either a positive or a negative trigger of the leftmost delay operator. Due to the dependency on triggers the leftmost delay operator of an enabled sequence shall not be configured with a *zero_step*. The semantics for the creation of new threads is formally defined in Section 6.7.1.

The general use of the delay operator shall be explained in the upcoming sections. For simplicity positive and negative sensitivity is expressed by event references only.

A detailed discussion of trigger expressions in general, and how they interact with a delay operator is given in Section 5.6. The formal definition of the semantics of a delay operator is given in Section 6.7.2. In the upcoming sequence specification examples it is assumed that the corresponding sequence is continuously enabled. Furthermore, it is assumed that the sequence evaluation mode always turns preliminary evaluation results at the last delay operator of a sequence directly to final results.

Evaluation of Partial Orders

As with SVA and PSL a sequence requires a trigger which reflects the increment in temporal order. In RTL assertion languages this is usually the edge of the clock of the DUV. Since clock signals are avoided in TL modeling, it has to be clarified how the evaluation of sequences can be triggered. As stated in R 19 and R 20 it is necessary to specify partial orders of event occurrences. Hence, some way must be found to express temporal relations between Boolean propositions based on the occurrence order of events in a simulation. It also has to be achieved that the evaluation of such a temporal relation must not depend on the global order of event occurrences. This can be accomplished by phasing out or by specifying only relevant event occurrences for the checking of a Boolean proposition. The following example shall illustrate how the delay operator from the UAL sequence layer can be used to formulate partial orders.

Given is a synchronizer block that handles the synchronization of a master and a display controller. Figure 5.3 shows a waveform of the communication between master and controller via the synchronizer. The number of frames a master sends to the controller is indicated by a state variable called `FRAMES` within the synchronizer. The master indicates how many frames are to be sent to the controller by writing the corresponding number to the `FRAMES` variable. The synchronizer indicates that data is to be fetched by the controller by emitting an event called `START`. Each time the controller fetches a frame it decrements the value stored in `FRAMES` and emits an event called `FETCH`. When no frame is left the synchronizer emits an event called `STOP` to indicate this to the master. The whole system is modeled without timing.

Now, the task is to specify a sequence which matches whenever the frames sent by the master are received by the controller. The start of communication is indicated by the emission of event `START` with the value in variable `FRAMES` being greater than zero. The end of the communication is indicated by the emission of event `STOP` with the value in variable `FRAMES` being zero.

The question is how to specify a sequence that matches this behavior and how to trigger its evaluation. Most approaches to TL-assertions construct a global clock event which corresponds to the disjunction of all events available in the DUV. The example shown here contains only three distinct events, assuming that master and

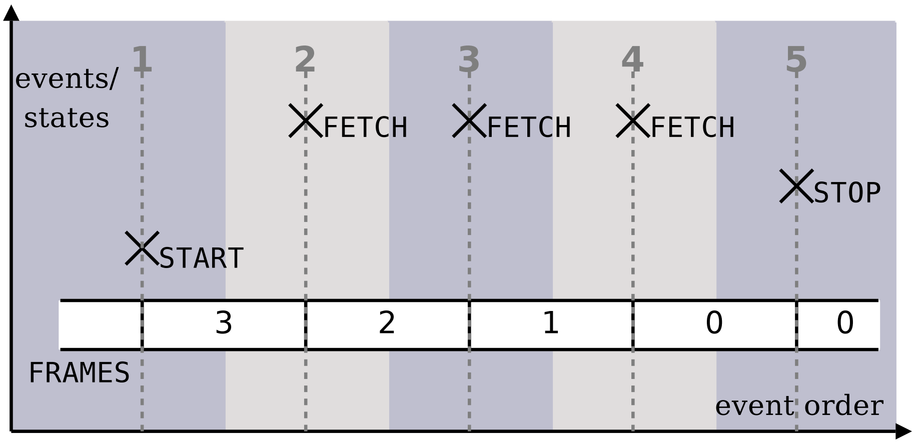


Figure 5.3: Synchronizer Block Example

controller do not emit other events and that no other block that emits events is in the system. In total five event occurrences are depicted in the example waveform in Figure 5.3. Hence, when taking all events into account the occurrence of the event `STOP` is the fourth occurrence after the occurrence of event `START`. However, the behavior of this model shows that the distance between events `START` and `STOP` depends on the number of fetches done by the controller which in turn depends on the value stored in variable `FRAMES`. Therefore, it is necessary to phase out all occurrences of the `FETCH` event when considering the relation between `START` and `STOP`. Thus, a sequence evaluation needs to be triggered only with the events of interest in order to obtain an independent description of the temporal relation of `START` and `STOP` events. The following UAL sequence specification shows how this task can be accomplished by using the UAL delay operator introduced earlier:

$$\#1\{\text{START}\}\{\text{FRAMES} > 0\}\ \#1\{\text{STOP}\}\{\text{FRAMES} == 0\};$$

The first delay operator is only triggered with the occurrence of event `START`. Since it is configures as a single-step delay operator and since it is the leftmost delay operator, the Boolean proposition is evaluated on each occurrence of event `START`. The Boolean proposition states that the value of variable `FRAMES` is greater than zero at the occurrence of event `START`. For the given example, the proposition matches with the depicted behavior in Figure 5.3. Hence, the evaluation shifts to the next delay operator where one occurrence of the `STOP` event is expected, upon which the variable `FRAMES` is required to be equal to zero.

As this example shows, the sensitivity of the delay operator ensures, that no event occurrences are considered for the evaluation except the ones specified in the sensitivity.

Evaluation of Strict Partial Orders

Letting the evaluation be triggered by events abolishes an assumption underlying the triggering concept of RTL assertion languages. Here, usually clock ticks are used for triggering assertions and thus, sequence evaluations. A clock can usually be considered as an input to a DUV. Hence, the precondition for a design to work is that its clock ticks. This assumption is also made when letting an assertion be triggered by clock ticks. In contrast to that, in a TLM events do not generally represent inputs which can be assumed. The emission of events in a TLM usually relies on the correct implementation of the model. For instance, the emission of `START` and `STOP` events in the previous example, depends on the correct implementation of the synchronizer module. It is not legal anymore to assume that these events are really emitted, at least not always emitted when they have to be. Thus, the sequence specification from the example above is risky. If for instance the behavior which leads to the emission of the `STOP` event is not implemented correctly, the evaluation of the sequence might starve for cases where the `STOP` event fails to occur. Hence, the problem to solve is to enhance a sequence specification such that it can also detect the absence of events. However, determining the absence of an event can only happen relatively to a point of reference.

A solution to this problem is to make the partial order more strict (see R 20). Hence, the absence of an event can be determined by the occurrence of another event. This means that an occurrence of another event serves as a point of reference. For defining such a point of reference the UAL delay operator offers a negative sensitivity as further parameter. In contrast to a positive trigger, a negative trigger calculated by a corresponding trigger expression can stop an evaluation of the delay operator forcing the sequence result to a not-match. Hence, the sequence specification example from above can be refined such that a not-match is produced if the `START` event occurs twice without a `STOP` event in between:

$$\#1\{\mathrm{START}\}\{\mathrm{FRAMES} > 0\}\ \#1\{\mathrm{STOP;START}\}\{\mathrm{FRAMES} == 0\};$$

The `START` event stops an evaluation which has reached the second delay operator and is waiting for the occurrence of the `STOP` event. In this case, the occurrence of the `START` event serves as the point of reference for detecting the absence of the `STOP` event.

In general, the use of a negative sensitivity is optional. In case a negative trigger leads to the termination of a thread, the result of this thread is a preliminary not-match. The Boolean expression of the corresponding delay operator hence, is not evaluated.

5.5.2 Local Variables

As indicated by the declaration syntax for sequences (see Rule B.23, p.204), UAL supports the concept of local variables, a powerful feature included in SVA but yet missing in PSL. This feature increases the expressiveness of sequence descriptions by allowing a sequence evaluation to store data along with the evaluation. The declaration syntax of local variables is defined as follows:

localvar_declaration = *type identifier* ";" ; B.62, p.207

A local variable can be either declared within a sequence or passed into a sequence either by copy or by reference. Assigning a value to a local variable is defined with the specifier *action* which was used in Rule B.27, p.205. The syntax rule for an action is defined as follows:

action = "," *identifier* "=" *localvar_expression* ; B.34, p.205

As indicated by this rule, a local variable assignment is added with a preceding comma. If more than one local variable needs to be assigned some value, the corresponding assignments are specified from left to right. This order also reflects the order of execution of a local variable assignment. A local variable updates its value immediately upon assignment and hence, the new value can be used in local variable assignments to the right.

For each evaluation thread of a sequence a new instance of the same local variable declaration is created and is valid through the lifetime of that thread within the sequence. If passed by reference the local variable can flow out of a sequence.

In general, a local variable is used for storing data within an evaluation thread in order to analyze it at a later stage of the same evaluation thread. For instance, the declaration of a sequence which captures that the FRAME variable is decremented from one FETCH event to the next, could be formulated as shown in Listing 5.7.

```
1 sequence s1(event FETCH, state int FRAMES)
2  int L1;
3  #1{FETCH}{FRAMES > 0,L1 = FRAMES}
4  #1{FETCH}{FRAMES == (L1 - 1)};
5 endsequence
```

Listing 5.7: Sequence with Local Variables

First, a local variable L1 is declared (line 2). The first delay operator (line 3) expresses that the FRAMES variable is greater zero at the occurrence of a FETCH event.

This prevents the sequence to start matching with the last occurrence of the `FETCH` event. Furthermore, the first delay operator contains a local variable assignment which samples the value of variable `FRAMES` into the local variable `L1`. This value is used at the next occurrence of the `FETCH` event in order to check that the new value of variable `FRAMES` equals the decremented value of `L1` (line 4).

5.5.3 Sequence Evaluation Modes

This section clarifies in detail how sequences are evaluated. In conjunction with that, four modes are introduced which influence the way a sequence is evaluated. These modes are the following:

```
sequence_mode  =  "AnyMatch"                          B.40,
                | "FirstMatch"                        p.205
                | "FirstMatchPipe"
                | "FirstMatchPipeOrdered" ;
```

The formal semantics of these modes are defined in Section 6.7.3.

AnyMatch and FirstMatch

Modes *AnyMatch* and *FirstMatch* are the most common matching strategies for sequences in assertion languages like SVA and PSL. The former mode is primarily used for obtaining coverage, the latter is used for obtaining a proposition on the behavior of a system. Since, the remaining modes defined in this work are enhancements to these modes a short description of these modes is given here. In general, both modes work the same way unless the sequence contains delay operators configured with a *step_range* setting (i.e., threads are split into sub-threads).

The *AnyMatch* mode does not influence preliminary results of evaluation threads of sequence specifications. Thus, each preliminary result of a thread at its termination represents a final result of a sequence evaluation. If a thread is split to sub-threads, the preliminary results of these sub-threads also represent final results of a sequence evaluation. Hence, one thread can produce several results, one per sub-thread.

In contrast to *AnyMatch* mode, in *FirstMatch* mode the first sub-thread that terminates with a preliminary match result represents the final result of the thread it belongs to. In such a case, all remaining sub-threads of the same thread are canceled. A sequence does not match if all sub-threads of a thread terminate with a preliminary not-match result.

Figure 5.4 illustrates the evaluation of a sequence both in *AnyMatch* and *FirstMatch* mode using an example sequence.

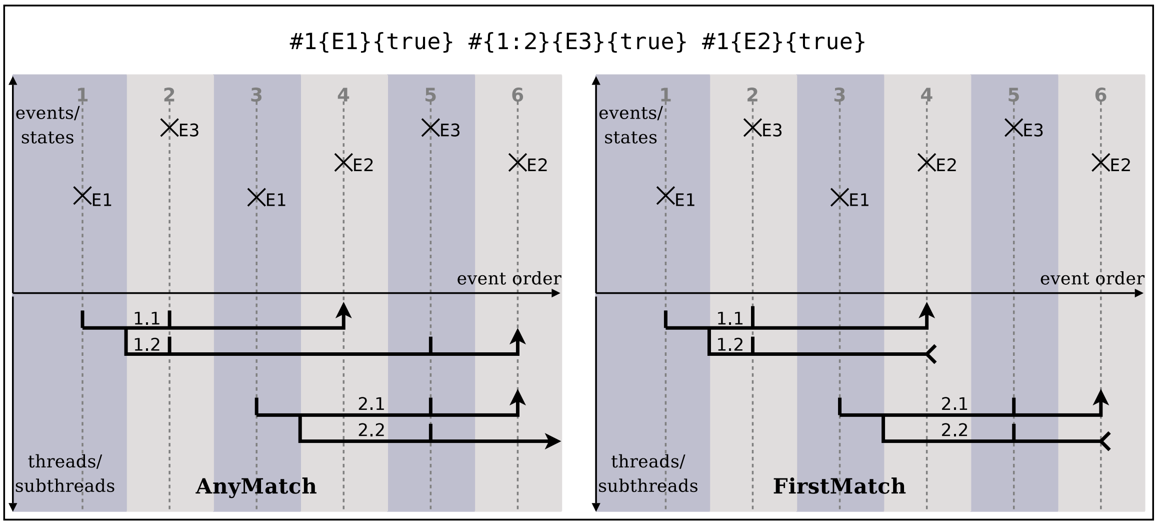


Figure 5.4: UAL Modes *AnyMatch* and *FirstMatch* for Sequences

A thread is depicted as an horizontal arrow in Figure 5.4. The corresponding identification number of a thread is noted to the left of the dot in Figure 5.4. A thread is split into sub-threads as soon as it reaches a delay operator configured with a *step_range*. The number of a sub-thread is noted to the right of the dot in Figure 5.4. In Figure 5.4 it is assumed that the given sequence is continuously enabled.

Analyzing the left part of Figure 5.4 shows that in *AnyMatch* mode all alternatives specified by the example sequence are evaluated. When sub-thread 1.1 matches it has no influence on the evaluation of the remaining sub-thread as well as the thread which is evaluated in parallel. Hence, all sub-threads are evaluated until they produce a result. Thus, one thread can have multiple results. Therefore, this mode is not suitable for obtaining a proposition on the correct behavior of a model. It is rather meant for obtaining coverage results.

The right part of Figure 5.4 shows the *FirstMatch* mode. Here, the match of sub-thread 1.1 cancels the remaining sub-thread 1.2. It however bears no effect on the evaluation of thread 2. A thread in *FirstMatch* mode hence, can only produce one result for a thread. Therefore, this mode is suitable for obtaining a proposition on the correct behavior of a model.

By supporting these evaluation modes, present in SVA and PSL, UAL fulfills requirement R 35.

FirstMatchPipe

One precondition for understanding the *FirstMatchPipe* mode is that each event occurrence increments a global counter. This counter reflects the global order of event occurrences and is referred to as the event index. Another precondition is that, each Boolean proposition in a sequence specification is assigned a unique identification value. Two propositions that are equivalent have the same identification value and can not be distinguished.

The *FirstMatchPipe* mode is defined by the following rules:

1. First Match Principle: A finally matching sub-thread of a thread cancels the evaluation of any other sub-thread of this particular thread.
2. Match Conflict: Threads/sub-threads that terminate at the same event index and have a preliminary match result are not allowed; this is called a match conflict. In such a case, only the oldest thread/sub-thread is allowed to finally match and the final result of the remaining threads/sub-threads is not-match.
3. Consumption Attempts: A thread/sub-thread that proceeds through a sequence specification attempts to consume the Boolean propositions at the event index at which they have been evaluated for this thread/sub-thread. A consumption attempt consists of the identification value of a Boolean proposition and the event index value at which it is evaluated for a specific thread/sub-thread. A trivially true ({**true**}) expression can not be consumed by any thread/sub-thread.
4. Consumption: A thread/sub-thread that finally matches, consumes all consumption attempts.
5. Consumption Conflict: A preliminarily matching thread/sub-thread can only match finally, if it does not attempt to consume Boolean propositions at event index values for which these propositions have already been consumed by another thread/sub-thread. A consumption conflict occurs if a thread/sub-thread attempts to consume an already consumed proposition. The final result for such a thread/sub-thread is not-match.

The motivation behind the *FirstMatchPipe* mode and how its rules are applied is explained in the following.

A closer analysis of the *FirstMatch* mode introduced in the last section shows that it lacks the ability to match pipelined behavior. An example shall illustrate this issue.

Given is a model which contains two blocks, a sender and a receiver, which communicate. Each time the sender sends a data value an event `SENT` is emitted.

The receiver is supposed to send a response which is an increment of the sent data[1] back to the sender. However, the receiver is able to receive up to two values from the sender prior to the response being ready. When a response is ready a DONE event is emitted. Specifying a sequence which matches the pair of one data being sent and its corresponding response could be done as shown in Listing 5.8.

```
sequence sent_done(
      event SENT, event DONE,
      state int data_s , state int data_r)
    int L1;
    #1{SENT}{true,L1=data_s} #{1:2}{DONE}{data_r == (L1 + 1)};
endsequence
```

Listing 5.8: Data Transport Sequence

This sequence is supposed to match if a SENT event is followed by one or two occurrences of a DONE event where the response data (data_r) is the increment of the sent data (data_s). The range is necessary since the receiver may process two data values in parallel.

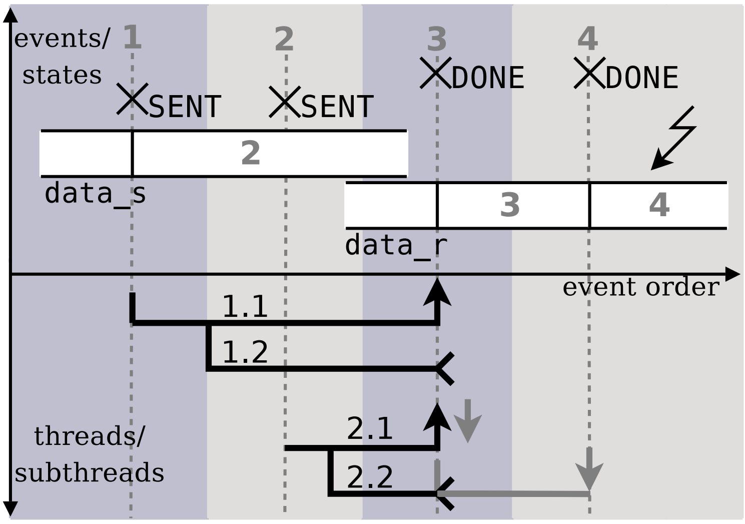


Figure 5.5: Insufficiency of *FirstMatch* Mode for Pipelined Behavior

Figure 5.5 depicts an example waveform of the described system. At the fourth event the system behaves wrong. Instead of value four, value three is expected in variable data_r at the second occurrence of the DONE event. However, when applying the *FirstMatch* mode for obtaining a proposition on the correctness of this model

[1] The receiver could perform any complicated operation on the received data value. The increment operation is only used for the sake of simplicity.

this error remains undetected. This is because the *FirstMatch* mode does not cope with matching pipelined behavior. As shown in Figure 5.5 the second thread matches with the occurrence of the first `DONE` event. However, this `DONE` event is the response to the first `SENT` event. Pipelining leads to a blurring of temporal behaviors since actions can overlap when pipelining is involved. Hence, a way must be found to cope with pipelining in a sequence evaluation as well. The first rule specific for a pipelined sequence evaluation is that no two threads may match at the same event occurrence (see above, Rule 2). If two threads match at the same event occurrence, it indicates that the younger has caught up with the older thread. Sub-thread 2.1 hence, is not allowed to match due to the match of sub-thread 1.1. Hence, the result of sub-thread 2.1 has to be considered as a not-match. Note that the principle of *FirstMatch* still has to hold, which means that within a thread the first matching sub-thread cancels the evaluation of the remaining sub-threads (see above, Rule 1). If sub-thread 2.1 is treated as a not-match, the evaluation of sub-thread 2.2 may proceed (indicated with a gray line). This sub-thread hence, triggers with the first occurrence of the `DONE` event and waits for the next occurrence. The sub-thread 2.2 is triggered further with the second occurrence of the `DONE` event. Here, by evaluating the Boolean proposition of the second delay operator the error in the variable `data_r` can be detected.

Disallowing threads that match at the same event index, however is not enough for capturing pipelining. At TL a sequence of event occurrences can reflect an abstract transaction. If a sequence is specified which shall capture this abstract transaction it has to produce unique matches.

Given is the following example of a three stage communication. The system contains a master which sends data to a target via a channel, indicated through an event called `SENT`. The channel is pipelined. The channel breaks one send request of the master into two consecutive requests for the target, indicated by the emission of an event called `TRANS`. The target responds as soon as two requests have been detected by emitting an event called `DONE`.

In order to capture the communication between master and target via the channel a sequence could be specified as shown in Listing 5.9.

```
sequence abstract_trans(
      event SENT, event TRANS, event DONE)
   #1{SENT}{true}
   #{1:2}{TRANS}{$l_event(TRANS)}
   #1{TRANS}{$l_event(TRANS)}
   #1{DONE}{true};
endsequence
```

Listing 5.9: Pipelined Communication Protocol Sequence

This description introduces the UAL Boolean Layer function `$l_event(event)`. This function returns true if the event occurrence that has triggered the delay operator is equal to the event specified as argument to the function. At first sight, this operator seems redundant, since the delay operators are triggered by one event each. However, by calling this function in the Boolean proposition part of a delay operator, the information of the occurrence of an event can be transformed into a Boolean proposition. How useful this is, is described within the next paragraphs.

Figure 5.6 illustrates how this sequence is evaluated with mode *FirstMatchPipe* if only the first two rules are considered.

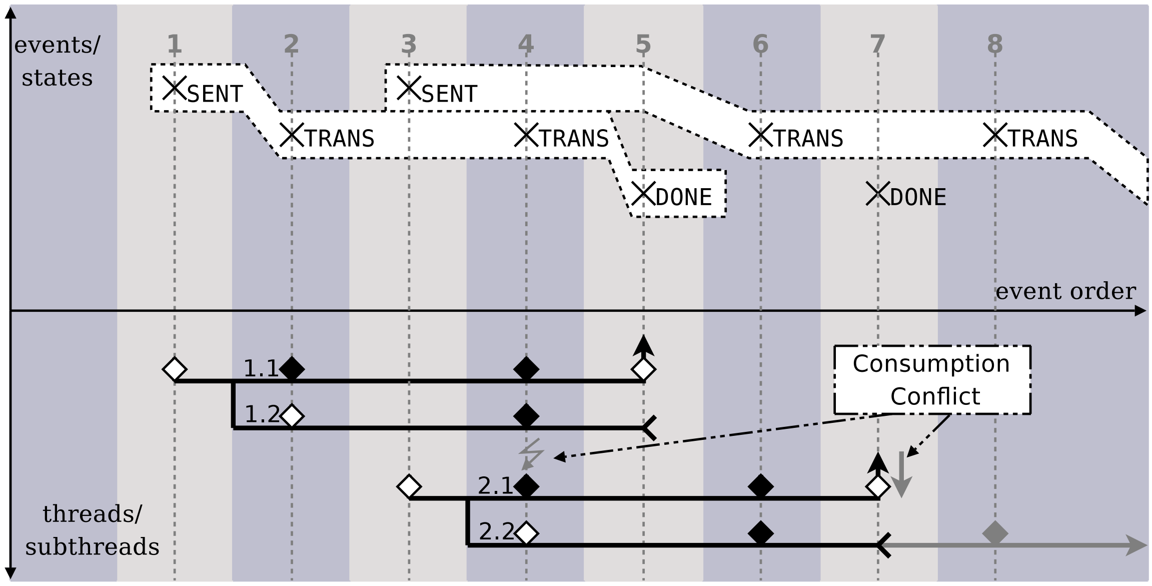


Figure 5.6: Illegal Overlapping of Threads

The black diamonds indicate that a thread has been triggered and that a Boolean proposition which is not a constant true expression (`{true}`) evaluates to true. The white diamonds indicate that a thread has been triggered and that a constant true expression is evaluated. The polygons surrounding particular event sequences indicate distinct communication interactions between the master and the target. One complete interaction and one ongoing interaction is shown. The figure also shows, that the target module emits a `DONE` event (at index 7) already after having received the first request (`TRANS` at index 6) from the channel. This request belongs to the second communication interaction. The event actually is supposed to be issued after the second request (`TRANS` at index 8).

The lower part of Figure 5.6 indicates the evaluation of the sequence by depicting each thread. As shown, thread 2 matches at event index 7. The thread hence, matches although the `DONE` event has been emitted too early. This is due to the fact that, thread 2 interprets the occurrence of event `TRANS` at index 4 as the first channel request belonging to the second communication interaction. However, this particular request is the last request of the first communication interaction. Actually, sub-thread 2.2 is still synchronous to the second communication interaction. However, this sub-thread is canceled due to the *FirstMatch* principle.

This example shows, that pipelined behavior is hard to track with common sequence evaluation semantics. Due to the pipelining, threads start to overlap. Due to such an overlap, information which belongs to one action is shared among threads and thus is interpreted as if it belonged to several distinct actions. The question is, how to deal with such ambiguities.

One possible solution to this could be the definition of a rule that permits only one thread to be triggered, while other threads wait for the next occurrence of the trigger. This way it is enforced that threads can not overlap in critical regions. Such a concept has been introduced by JEDA Technologies [55]. This approach however, is dangerous and can lead to extremely non-deterministic sequence evaluation. When assertions are to be evaluated on-the-fly during a simulation, this would mean, that one older thread which is not yet decided can block a younger thread from continuing until the older thread has terminated with a result. If the older thread however evaluates to a not-match at a later stage, i.e., at some time later it is found that this thread was not supposed to block another, the younger thread might not be able to match anymore. This is because all events that could have triggered its evaluation if it was not blocked have already passed at the time when it is reactivated again. Debugging of such an assertion can become a nightmare because the user would have to reconstruct when a thread was blocked and why. This can become very time-consuming and contradicts the endeavor to reduce debug time with ABV.

The *FirstMatchPipe* mode defined in UAL, considers Boolean propositions as resources. Hence, if a thread evaluates a Boolean proposition and obtains a true result, it attempts to consume this proposition / resource (see above, Rule 3). Once, a thread has evaluated to a match all its consumption attempts turn into granted consumptions (see above, Rule 4). Figure 5.6 indicates the consumption attempt of a non constant true expression by black diamonds. A consumption attempt of a constant true expression is marked with a white diamond. When analyzing the consumption attempts of the threads depicted in Figure 5.6 it can be found that at the fourth event occurrence sub-thread 2.1 attempts to consume the same proposition as sub-thread 1.1 (**$l_event**(TRANS)). This means two threads attempt to use one piece of information at the same event index. However, since these consumptions are still attempts at the fourth event occurrence, the evaluation has to proceed until either of these threads matches. As soon as one thread matches preliminarily, it is checked whether its consumption attempts overlap with the consumption attempts of an earlier matching thread. If this is the case the final result of the thread is forced to a not-match (see above, Rule 5). If not, all its consumption attempts are finally consumed and the thread matches. In the given example, when sub-thread 2.1. matches preliminarily at the seventh event occurrence, it is checked whether its consumption attempts overlap with an earlier matching thread. In this case, there is an overlap at event occurrence four with a consumption attempt of sub-thread 1.1 which has already matched and thus, consumed. Hence, sub-thread 2.1. is forced to a not-match result, leaving sub-thread 2.2 active for the remaining evaluation, as indicated by the gray colored arrows in Figure 5.6. As the example shows, the use of the Boolean function `$l_event(event)` transforms the information of an event occurrence to a Boolean proposition which can be consumed by threads/sub-threads. However, note that not the event itself is consumed.

FirstMatchPipeOrdered

The *FirstMatchPipe* mode, described in the previous section, handles pipelining in general. Hence, this mode does not distinguish between in-order and out-of-order pipelining. As a complement to the *FirstMatchPipe* mode, the *FirstMatchPipeOrdered* mode considers also the pipelining order. The *FirstMatchPipeOrdered* mode is defined by an additional rule with regard to the *FirstMatchPipe* mode:

1. Ordered Matching of Threads: No thread/sub-thread may match as long as the next older thread has not terminated.

The motivation behind the *FirstMatchPipeOrdered* mode is best explained with a FIFO example.

Given is a FIFO module with three FIFO stages. When data is put into the FIFO it is indicated by the emission of a `PUT` event. Fetching data from a FIFO is indicated by a `GET` event. Specifying a sequence that matches when a data word is put into the FIFO and fetched later in a guaranteed amount of fetch accesses can be formulated as shown in Listing 5.10 as follows:

```
sequence fifo (
      event PUT, event GET,
      state int p_data, state int g_data)
    int L1;
    #1{PUT}{true,L1 = p_data} #{1:3}{GET}{g_data == L1};
endsequence
```

Listing 5.10: FIFO-Pipeline

This sequence matches when a data word which is put into a FIFO propagates out within the next three occurrences of a `GET` event. The left part of Figure 5.7 shows

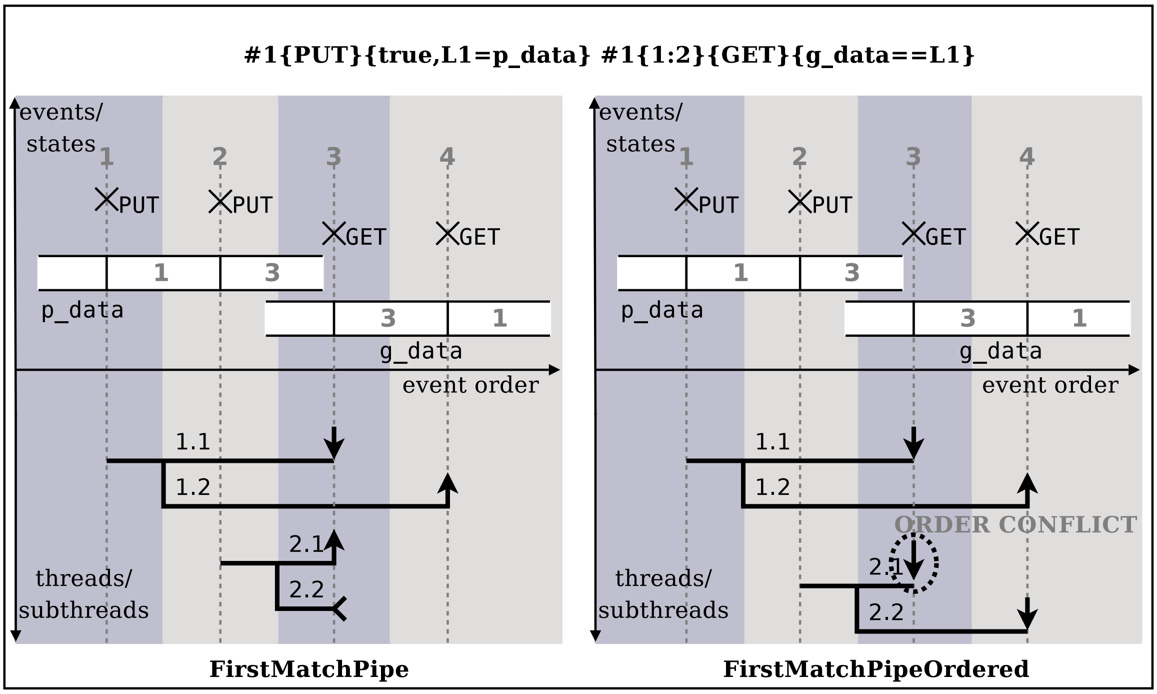


Figure 5.7: In-Order / Out-Of-Order Pipelining

the evaluation of the given sequence, using the *FirstMatchPipe* mode. Since, this example does not produce consumption conflicts diamonds are omitted in the figure. As the left part shows, the evaluation of the sequence still leads to matches although the underlying FIFO acts as a FILO, due to a wrong implementation. Since, the *FirstMatchPipe* mode does not include the pipelining order it is impossible to check that the data propagates out in the correct order. The right part of Figure 5.7 shows how the *FirstMatchPipe* mode can be enhanced to include order preservation as well. The occurrence order of PUT events determines the creation order of threads for the evaluation. Hence, for order preservation it is required that these threads match in the same order as they are created. Hence, sub-thread 2.1 depicted in Figure 5.7 is forced to a not-match result, since thread 1 has not finished its evaluation. Thus, the overall evaluation of thread 2 fails.

5.6 Event Layer

In the previous section the sequence layer of UAL was introduced. It was shown how sequences can be specified. In the given examples, the evaluation of sequences was triggered through simple events. This however, is not sufficient for a TL assertion approach, since the assertion specification also needs to cope with multiple abstraction levels and sequences of transactions. This section introduces the UAL event layer. A general concept of events is introduced which goes beyond events that are available for instance in SystemC. Afterwards, a powerful set of event operators is introduced. These operators allow the formulation of complex trigger expressions. These trigger expressions in turn can be used in conjunction with the sequence layer to control the triggering of a sequence specification while capturing different abstraction levels.

5.6.1 Categorization of Events

The event concept of UAL defines additional events to the ones available in languages such as SystemC. Figure 5.8 shows a tree diagram that represents how events in UAL can be grouped into different categories.

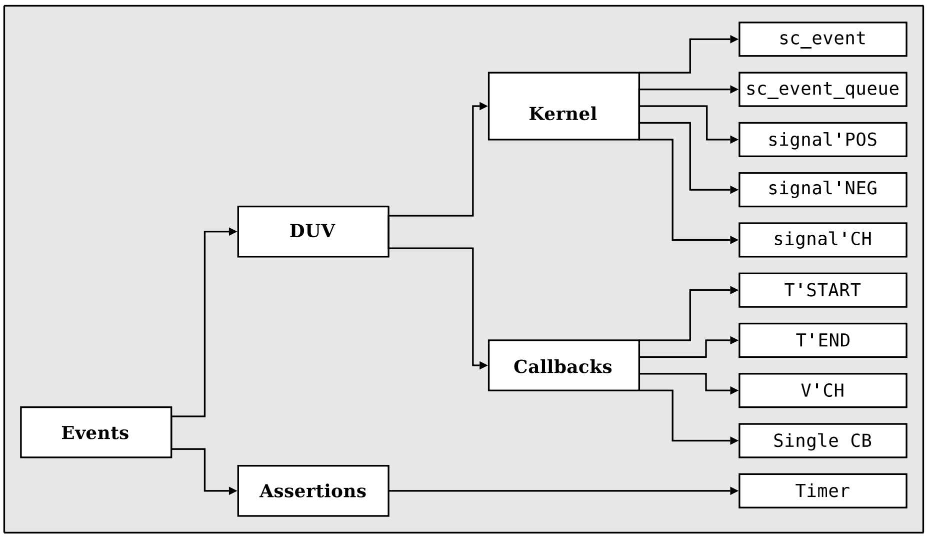


Figure 5.8: Categorization of Events

Generally, UAL differentiates between events originating from a DUV and from assertions.

Within a DUV, events can be categorized further into kernel and callback events. The former can be any event which is offered by SystemC and the latter is a special category that belongs to the UAL approach. This category enables a more granular than delta-cycle resolution of a model behavior (see R 10), because the notification of these events happens with no interaction with a simulation kernel. Hence, a callback event notification does not introduce new delta-cycles in the SystemC simulation cycle. The notification order equals to the scheduling order. Using these callback events hence, allows for tracking behavior within a delta-cycle as well, making them most suitable for the application with Programmer's View (PV) models. However, the use of these events is not restricted to a particular abstraction layer within TL. Callback events generally are to be annotated in the design by a user. However, the overall framework of UAL offers means to keep this additional effort at a minimum level.

Transaction Events

As it was discussed in Chapter 3, a TL assertion approach needs to support the specification of transaction sequences (see R 1). Furthermore, it is necessary to be able to detect overlapping transactions as well (see R 30). Hence, in UAL a transaction is defined to emit an event upon its start and an event as soon as it has completed. By using these events for triggering sequences, it is possible to specify sequences which detect whether transactions overlap or not regardless of the abstraction level. Furthermore, a start event of a transaction is defined to be issued always before the end event of the same transaction with regard to the global event ordering. In addition to that a transaction event may not be modeled as a regular event. A notification of a transaction event has to happen immediately, while blocking the emitting process. This way, the pre- and postconditions of a transaction remain fixed while the corresponding event is being processed by an assertion. As these events can be used as triggers in delay operators which in turn sample state variables for the evaluation of Boolean propositions, it is important that all states remain stable until the event has been processed by all assertions.

State Variable Assignment Events

Since states in TL modeling are rather implemented with variables instead of signals, which usually emit an event upon a value-change, it is necessary to have value-change events available for variables as well. Therefore, UAL supports treating variable assignments as if the variables were signals. The corresponding value-change event of a variable is required to be a callback, since a variable can change its value more than once within a single delta-cycle.

Single Callback Events

One transaction or variable assignment event can be considered a single callback event. However, users can specify single callback events in all parts of the model for instance in order to provide a temporal view of a complex algorithm. Assertions could be used to check the control-flow of such an algorithm externally.

Assertion Timer Events

The UAL event layer offers a special timer operator for allowing an assertion to trigger itself. This operator schedules a reserved SystemC event to a specific time.

5.6.2 Operators

The event layer of UAL comes with a smart set of event operators which enable the specification of powerful event expressions which can be used as triggers for delay operators of the sequence layer. Both the positive and the negative sensitivity of a delay operator is defined to be a trigger expression (see Rule B.45, B.46, p.206). The syntax of a trigger expression is defined as follows:

trigger_expression = (*event_expression* ["," *trigger_timer*])
| *trigger_timer* ; B.47, p.206

The result of a trigger expression is a trigger. A delay operator is hence, sensitive to such a trigger. A delay operator may only have one instance of a timer trigger in total. Hence, either the positive or the negative sensitivity may instantiate a timer trigger. The formal syntax for a timer trigger can be found in Rule B.48 on page 206.

Event expressions in general formulate conditions and relations on event occurrences. The formal syntax of event expressions corresponds to a common expression syntax based on factors, terms, and operands. The corresponding rules are B.55, B.54, B.53, B.49, and B.52 on page 206. Event expressions hence, can be considered as expression trees where each node represents an event operator and the leafs represent references to events or further operator nodes. With the exception of a single event operator, which is explained later, each event operator can have a trigger as operand. Each event operator returns a trigger if its operands satisfy a corresponding condition.

Table 5.3 shows a brief informal overview on the available event operators. Each operator is explained in detail in the next sections.

In addition to the event operators shown in Table 5.3 UAL's event layer offers a function `$delta_t` which returns a relative simulation time value. This function is only allowed to be used in the event layer.

An event operator is enabled for evaluation as soon as a delay operator is enabled which utilizes the corresponding operator in its sensitivity. For most event operators the enabling time influences the corresponding behavior. Furthermore, an event operator works for each evaluation thread individually. The operators are explained in detail in the following sections followed by a short discussion of reset event expressions as parameters of verification directives, mentioned in Section 5.3.

Name	Symbol	Definition
		returns a trigger . . .
Single Event	*e1*	. . . if event *e1* occurs
OR	*ev_expr* | *ev_expr*	. . . if either of the operands occurs
AND	*ev_expr* **&** *ev_expr*	. . . if both operands occur at the same simulation time
CONSTRAINT	*ev_expr***@(bool_expr)**	. . . if `bool_expr` is true on occurrence of *ev_expr*
TIMER	**timer**(int_expr)	. . . `int_expr` time units later than the current evaluation time
ACCUMULATOR	*ev_expr***%(int_expr)**	. . . after `int_expr` occurrences of *ev_expr*

Table 5.3: Event Operators

Single Event Operator

The single event operator is an artificial event operator and has already been used in the examples given in Section 5.5. The single event operator is a unary operator with a reference to a specific event of the DUV. It transforms an occurrence of this event into a trigger.

OR Operator

The OR operator returns a trigger as soon as one of its operand expressions return a trigger. The OR operator can be used to specify alternative triggers for a delay operator. By using the Boolean function **$l_event**(**event**) it is hence, possible to make the Boolean proposition of a delay operator more expressive, since it is possible to decode which event has led to the triggering of the delay operator. The following sequence specification example illustrates the use of the OR operator in conjunction with the function **$l_event**(**event**):

#1{e1 | e2} {(**$l_event**(e1) && a == 1)|| (**$l_event**(e2) && b == 1)};

The operands of the OR operator are single event operators. The formal semantics of the OR operator are defined in Section 6.8.3.

AND Operator

The AND operator is the first event operator that also addresses the requirement of taking the simulation time as a temporal reference as well (see R 22). The AND operator is a binary operator which returns a trigger only if its two operand event expressions return a trigger at the same simulation time. The operand event expressions can return a trigger at any time equal or after the time the AND operator has been enabled. The restriction is only for both operands to return a trigger at the same simulation time and that both operands return a trigger strictly after the event index where the operator has been enabled. Using the AND operator, it is for example possible to specify a sequence that triggers only if two transactions `T1` and `T2` start at the same time:

$$\#1\{\mathrm{T1'}\mathbf{START}\ \&\ \mathrm{T2'}\mathbf{START}\}\{\mathbf{true}\};$$

Note that the start events of both transactions are emitted in some order, since no two events can be processed simultaneously corresponding to the global order of event occurrences. However, the simulation time can be used as a grid in which two events can occur simultaneously.

Clear semantics are needed to define the operation of an AND operator if both operand expressions can emit several triggers within one simulation time grid, especially if the associated delay operator has to process several threads within one simulation time. Figure 5.9 shows some examples for the AND operator which shall illustrate the behavior of the operator. The figure shows the evaluation of two sequences.

All examples take place within the same simulation time slot. First the evaluation of the upper half of Figure 5.9 is discussed. From event index 1 to 3 the simple case for an evaluation is given. The AND operator in the second delay returns a trigger upon the detection of event `E2` and `E3` at index 2 and 3. A double ended arrow indicates which pair of event occurrences has lead to a trigger returned by the AND operator. The number assigned to an arrow represents the thread which is affected.

The second example from index 4 to 7 shows that the AND operator considers only the first trigger returned of one of its operands to pair it with a later trigger returned by the other operand within the same simulation time.

The third example from index 8 to 11 shows the behavior when the second delay operator is enabled twice before any operand of the AND operator has returned a trigger. Hence, the pair at index 10 and 11 triggers both threads 3 and 4.

The fourth example from index 12 to 16 shows the behavior when the second delay operator is enabled twice in between the evaluation of the AND operator for the

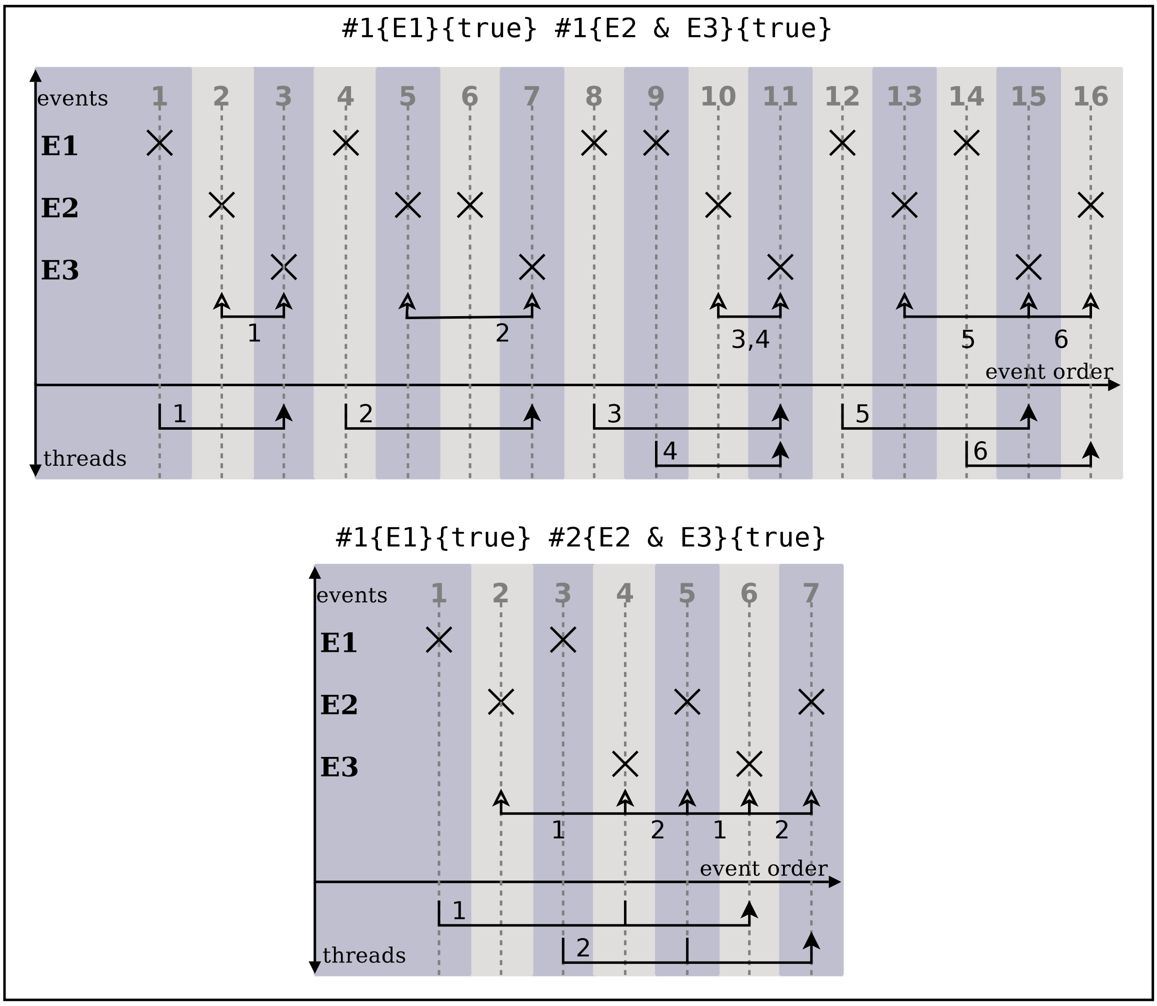


Figure 5.9: AND Operator Examples

first thread with number 5. Thread 5 is triggered by the pair at 13 and 15 whereas thread 6 is triggered with the pair at 15 and 16. This is due to the requirement that both operands of the AND operator need to return a trigger strictly after the operator is enabled for a specific thread. The AND operator is enabled for thread number 6 after the first operand occurrence at index 13. Hence, this occurrence is ignored for thread 6.

The lower half of Figure 5.9 shows another sequence evaluation example where the second delay operator is supposed to delay the evaluation for two occurrences of its trigger expression. The figure shows an example where the second delay operator has to evaluate two threads, 1 and 2, in parallel. However the second thread starts again in between the evaluation of the first thread. As indicated the AND operator triggers the first thread at index 4 and 6. If a delay operator has a delay value greater than 1 it re-enables its sensitivity as soon as it has triggered. Hence, at index 4 the second delay operator is triggered the first time which leads to a re-enabling of the AND operator for this thread. Since the re-enabling is caused by the trigger at index 4, event `E2` at index 5 is considered as the next trigger returned by an operand of the AND operator for thread 1, instead of event `E3` at index 4. Hence, the AND operator returns a trigger only for non-overlapping pairs of trigger occurrences of its operands for one thread.

The definition of the formal semantics of the AND operator is given in Section 6.8.4.

CONSTRAINT Operator

The CONSTRAINT operator is a unary operator with an event expression as operand and a configuration expression. Through the configuration expression it is possible to specify a Boolean condition which has to hold when the operand returns a trigger. If this condition is not fulfilled the CONSTRAINT operator does not return a trigger. Hence, the CONSTRAINT operator works as an event filter. For instance if a transaction event is assigned to the operand expression via a single event operator, it is possible to consider only the transaction occurrences where the address argument of the transaction equals to a desired value. Doing so, a sequence can consider only transactions to a certain address, making the formulation of a sequence much easier.

Another interesting application of the CONSTRAINT operator is to specify also a time constraint on a trigger returned by the operand, addressing requirement R 22. For doing this the UAL function `$delta_t` can be used within the configuration expression. The function `$delta_t` returns the simulation time that has passed since the simulation time the sensitivity of the corresponding delay operator has been enabled for one thread. Hence, if several threads have enabled the sensitivity at different simulation times and are still waiting for the constraint to be fulfilled the result of function `$delta_t` can be different for each individual thread and so can be the result of the constraint.

Using a time constraint allows filtering of event occurrences which do not fit in the window. Given is the following example:

$$\#1\{E1\}\{\textbf{true}\}\ \#1\{E2@(\textbf{\$delta\_t} == 50)\}\ \{\textbf{true}\};$$

This sequence matches only if an `E2` event occurs exactly 50 simulation time steps[2] after the occurrence of an `E1` event. When specifying time constraints on events the user has to take care that the evaluation does not starve. In this example the

[2] The resolution of simulation in UAL depends on the resolution chosen for a simulation.

evaluation would starve if no E2 event occurs exactly 50 time steps later than an E1 occurrence for instance by adding a negative trigger expression to the second delay operator. One neat feature for doing this is described in the next paragraph.

The definition of the formal semantics of the CONSTRAINT operator is given in Section 6.8.5.

TIMER Operator

A TIMER operator does not have an explicit operand event expression. Implicitly though, a TIMER operator can be considered as a reference to the reserved timer event depicted in Figure 5.8. However, a TIMER operator is parameterized with a simulation time value which is relative to the simulation time at which the operator has been enabled for one thread. The operator returns a trigger as soon as the specified time has passed. The TIMER operator schedules the reserved timer event and reacts to it.

The most intuitive use of a TIMER operator is as a timeout expression. The previous example which illustrates the time constraint feature can hence be enhanced by adding a timeout condition as follows:

$$\#1\{\mathrm{E1}\}\{\mathbf{true}\}\ \#1\{\mathrm{E2}@(\mathbf{\$delta\_t} == 50)\ ;\ \mathbf{timer}(51)\}\{\mathbf{true}\};$$

Hence, if no event occurrence of E2 fulfills the time constraint, the TIMER operator returns a trigger at the 51^{st} time step after the occurrence of the E1 event. The TIMER operator can also be used as a positive trigger of a delay operator. Here, it can be considered as a time value that the delay operator has to wait prior to triggering once. If the delay operator is the first delay operator in an enabled sequence, a TIMER operator in the positive sensitivity leads to a periodic start of the sequence evaluation. Hence, the TIMER operator allows the specification of sequences which do not need any event issued by the DUV in order to trigger.

While addressing the requirement to take simulation time as a temporal reference (see R 22), the TIMER operator also enables the tracking of dynamic temporal behavior (see R 23). The configuration parameter of the TIMER operator need not be static. Any arithmetic expression can be used for the calculation of the time value. The expression may be based on design variables as well as local variables. The time value is calculated upon the enabling of the timer. The time value does not change for one thread once it has been calculated.

The following example illustrates how a TIMER can be used to adapt a sequence to the timing of a DUV. A system contains a sender and a receiver. The sender can transmit data bursts of a variable size. The receiver issues an event called DONE as soon as the burst has been received. The sender transmits the burst via a transaction SEND which carries an argument BS in the payload that indicates the burst size and a pointer to the start address of the data to be transmitted. The receiver has an internal variable called CT which holds the duration in terms of simulation time for one data fetch phase. Hence, the sender consumes one cycle for each data item of a burst. A sequence that matches a complete burst transmission can be specified as shown in Listing 5.11.

```
sequence adapt(
      transaction void SEND(int BS,int* data),
      event DONE,
      state double CT)
    int S;
    #1{SEND'START}{true,S=SEND.BS}
    #1{DONE;timer(S*CT+1)}{true};
endsequence
```

Listing 5.11: Adaptive Timing in Sequences

As the example shows, it is possible to calculate the required timeout value based on a local variable which stores the size of the burst and the CT variable in the receiver which holds the cycle duration of the receiver. Hence, the timeout value can be calculated for any burst size.

The definition of the formal semantics of the TIMER operator is given in Section 6.8.5.

ACCUMULATOR Operator

The ACCUMULATOR operator is a unary operator. The ACCUMULATOR operator can be configured with an arithmetic expression which has to evaluate to a natural number on the enabling of the operator. The calculated number indicates how many triggers of the operand event expression have to be returned for one thread before the operator returns a trigger. This operator can be used to deal with data dependent temporal behavior (see R 23). The configuration expression can contain any variable, constant, and also local variables. The operator can be used for dealing with dynamic occurrences of events or transactions respectively. Figure 5.10 picks up the synchronizer example used in Section 5.5.

As Figure 5.10 shows, the number of FETCH events depends on the value of variable FRAMES at the occurrence of the START event. So far it was not possible to take the FETCH event occurrences into account in a sequence specification. Using the ACCUMULATOR operator it is possible to specify a sequence which matches if the

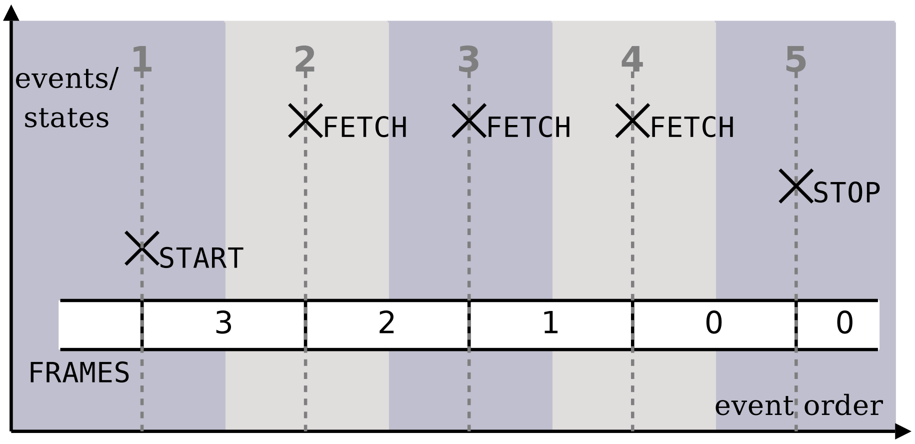


Figure 5.10: Synchronizer Block Example Revisited

expected number of FETCH event occurrences is encountered in between the occurrence of event START and event STOP:

```
sequence dynamic_temporal(
        event START, event STOP,
        event FETCH, state int FRAMES)
    int L1;
    #1{START}{true, L1 = FRAMES}
    #1{FETCH%(L1) ; STOP}{FRAMES == 0}
    #1{STOP ; FETCH}{FRAMES == 0};
endsequence
```

Listing 5.12: Adaptive Triggering of Sequences

This sequence example matches if a START event is followed by as many FETCH events as indicated by the variable FRAMES. These events have to be followed by an occurrence of the STOP event. Neglecting the Boolean propositions the sequence does not match if the STOP event is emitted before all required fetches have occurred. Furthermore, the sequence does also not match in case more fetches occur than indicated by the initial value of variable FRAMES before the STOP event occurs.

It is also allowed that the configuration expression evaluates to zero. In this case the ACCUMULATOR operator emits an event immediately upon being enabled.

The formal semantics of the ACCUMULATOR operator are defined in Section 6.8.6.

Reset Event Expressions

As mentioned in Section 5.3, a verification directive can be parameterized with a reset event expression which determines when the associated property has to be reset. Since most event operators consider their enabling time (e.g., simulation time, current event index) for a thread as reference point for the further evaluation, it is not sensible to use these operators for resetting a property. Hence, a reset expression may only contain operators which do not require a relative reference. Hence, the following operators may be used within a reset expression:

- Single Event
- OR
- CONSTRAINT

Within the constraint expression of the CONSTRAINT operator it is furthermore not allowed to use function **$delta_t**.

5.6.3 Multi-Abstraction Example

Using the event operators from the event layer in conjunction with the delay operator of the sequence layer enables to specify sequences which can capture behavior across different abstraction levels. Figure 5.11 shows an example behavior obtained by simulating a TLM with blocks modeled at different abstractions.

The example sequence is given in Listing 5.13.

```
1 sequence multi_abstraction(...)
2   #1{T1'START}{true}                                  //PV
3   #1{T2'START;T1'END}{true}                           //PV
4   #1{T1'END & T2'END;T3'START}{true}                  //PVT
5   #1{T3'START@($delta_t == 10) ; timer(11)}{true}//PVT
6   #2{clk'POS}{active}                                 //CC/RTL
7   #1{clk'POS}{!active} ;                              //CC/RTL
8 endsequence
```

Listing 5.13: Multi-Abstraction Sequence

The dashed boxes indicate which line in the sequence matches for the given behavior in Figure 5.11.

The lines 2 and 3 match if transaction `T1` starts and is followed by the start of transaction `T2` before transaction `T1` ends. This corresponds to a PV abstraction.

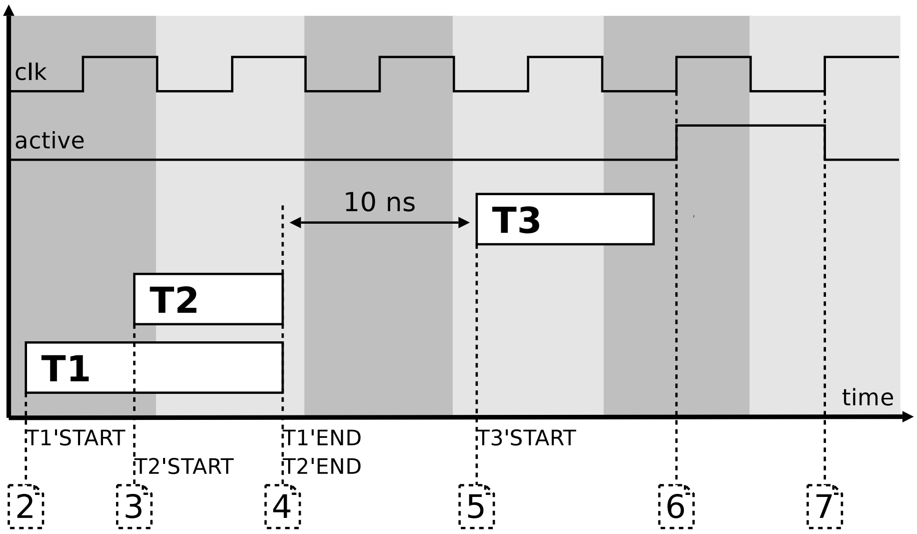


Figure 5.11: Multi-Abstraction Example

Line 4 matches if both transactions `T1` and `T2` finish at the same simulation time but before the start of transaction `T3`. Line 5 matches if transaction `T3` starts exactly 10 time units (the time unit of the simulation is assumed to be set to *nanoseconds*) after transactions `T1` and `T2` have ended. This corresponds to a Programmer's View with Timing (PVT) abstraction.

Line 6 matches after two clock ticks later than the start of transaction `T3` and when the signal `active` is true at the second clock tick. Line 7 matches at the third clock tick when signal `active` is false. This corresponds to a Cycle Callable (CC) or RTL abstraction.

As this example shows, the event layer enables the specification of sequences across different abstraction levels. The operators provided by the event layer allow bundling of several event occurrences to abstract triggers. The sequences hence, work with this abstract triggers and can be adapted to different abstraction levels through the use of event operators.

5.7 Boolean Layer

The syntax for Boolean propositions in a delay operator is defined as follows:

condition = *boolean_expression* B.33,
[`"?"` *boolean_expression* `":"` *boolean_expression*] ; p.205

The Boolean layer incorporates all Boolean operators. Boolean operators have their usual meanings. The syntax for Boolean operators is the same as defined for C++.

UAL defines several helper functions which can be used in Boolean expressions:

Function	**Definition**
$l_event(event)	Returns true if the last encountered event within a sequence corresponds to its argument
$time	Returns the current simulation time
$thread_id	Returns the identification number of the thread which invokes the function

Table 5.4: Boolean Layer Helper Functions

The Boolean layer allows the use of arithmetic expressions as well. However, these may only be used to formulate Boolean equations.

6 Formal Semantics

This chapter introduces the formal semantics of UAL in order to have a clear and unambiguous specification of the execution semantics. First, a formal representation of a trace which represents the behavior of a TLM is defined. After discussing the major shortcomings of LTL based assertion languages (e.g., PSL and SVA), a High-Level Colored Petri Net (HLCPN) is defined as the basis for the formal description of the UAL semantics. Following that, the semantics are defined by mapping all language layers to the HLCPN.

6.1 Trace Semantics

This section discusses the required information to be included in a temporal trace of a TLM. The definitions presented in this section have been aligned with the definitions presented in the dissertation of Thomas Steininger [61], due to the similarities in the formal basics of this work and the work described in [61].

In addition to that, the shortcomings of classical LTL based trace semantics are discussed.

6.1.1 Traces

In classical temporal logics checking temporal logic formulae against the behavior of a formal model is inductively defined over an infinite path of this formal model. A formal model is usually expressed in terms of a Kripke structure [62] also referred to as temporal structure [63]. According to [63], a temporal structure is a tuple $M := (S, R, L)$ with:

1. $S :=$ finite set of states
2. $R :=$ transition relation with $R \subseteq S \times S$ and $\forall s \in S \exists s' \in S, (s, s') \in R$
3. $L :=$ labeling function with $L : S \mapsto 2^{AP}$ with
   $AP :=$ set of atomic propositions

The structure is called linear if the relation R defines exactly one successor state $s' \in S$ for every state $s \in S$. The labeling function L denotes which propositions are true for which state. The atomic propositions in AP are formulated on internal variables, constants, and signals of the model. The values of these variables include output values of the model as well as the state s of the model. The model of time defined by such a temporal structure is linear[1] and discrete. Time is abstract and is defined by the transition of one state to its successor.

According to [63], a state path of such a temporal structure is the infinite succession of states starting from an initial state s^0. The succession of states is determined by R. The path as a whole can be considered as a vector, where each element reflects a state $s \in S$ and the strictly adjacent element the successor $s' \in S$ of s.

A temporal logic formula describes a temporal relation of properties which in turn can be further temporal expressions or Boolean propositions. Such a formula is evaluated against the path of a model. A temporal logic formula is defined to hold for a specific state at one element i in the vector if there exists a path suffix starting from i, which satisfies the formula. A temporal logic formula is supposed to hold for any state in the path. This means that for all elements in the path there must be a suffix that satisfies the formula starting from that element.

In formal verification approaches, the temporal structure is extracted from a given implementation model of a design. The computation effort for such an extraction is very expensive in terms of resources, since the complete state space needs to be constructed. Due to this fact, such an approach is limited to designs of a medium complexity. Its application for HW models is feasible, since the state space of an RTL model is small in comparison to the state space of an industrial SW model. An additional problem occurring in SW is that it is possible that new objects can be constructed dynamically. This means that the state space of the model can change when the SW is executed. TLMs represent a mixture of both HW and SW modeling paradigms and hence, inherit these difficulties.

Due to the complexity limitation, dynamic verification plays still a major role for the evaluation of temporal design properties. This is also the reason why UAL is mainly targeted in this domain. In dynamic verification, the implementation model of a design is simulated. A simulation automatically yields one possible path through the state space of a design. Hence, the evaluation of temporal properties is not general compared to a formal analysis. In simulation, the path through the design state space is an observation of all valuations of constants, variables, and signals of a design over time and is usually called a trace of a design model. To the knowledge of the author

[1] Also branching time models exists, where R defines more than one successor state for a specific state. Such models however, can be used only in formal verification. UAL semantics are based on dynamic verification which enforces a linear time model. Hence, branching time models are not considered in this work.

no general definition of the time granularity of a trace and its content exists. For instance, in PSL, the granularity is defined to be dependent on the "granularity of time as seen by the verification tool" [24]; its content is the valuation of design signals or variables at specific points in time. Hence, the trace is updated with a step increment of the verification tool. In contrast to that, in [64], [46], and [49] the granularity of a trace is configured by a user. It is the users responsibility to define what is to be traced and when it is to be traced. In SVA in turn, the trace is defined over the simulation time. More specifically the Language Reference Manual (LRM) [25] states that multiple occurrences of an event at the same simulation time slot shall not be used for sampling design states. As will be described in the next section, UAL follows a hybrid approach starting from a default granularity which can be refined further by a user. The next sections also provide a formal definition of a UAL trace and a discussion of UAL semantics with regard to the semantics of RTL assertion languages.

6.1.2 UAL Trace

The following two sections describe informal characteristics of the trace underlying the evaluation semantics of UAL. Subsequent to that, the UAL trace is described based on formal definitions.

Granularity of time

Occurrences of events in a simulation of a TLM are always ordered. The order is determined by the scheduling algorithm of SystemC and may be non-deterministic. The order of event occurrences is the same for repeated simulation runs of the same model with the same stimuli. The functionality of a TLM is modeled through SystemC processes which perform actions on any variables and signals and invoke transactions. Since these processes are activated based on occurrences of either value-change events, custom events, or implicit events (Sec. 3.2.2), the order of value transitions of variables and signals is strongly related to the occurrence order of events. Therefore, an intuitive way for defining the granularity of time for a trace is to control the progression of a trace by observing event occurrences. Each event occurrence leads to a new entry in a trace. This trace contains the sampled values of all variables and signals employed in a TLM. However, since occurrences of implicit events can not be observed directly, these events do not lead to new entries in the trace. These considerations yield the default granularity of time of the UAL trace. Such a trace can be obtained without additional annotations of a design, because only events offered by SystemC are considered.

In between two adjacent SystemC event occurrences, it is possible that many actions take place which can repeatedly alter the values of certain variables. These value changes however, are not visible in a trace if the sampling is done with the occurrences of SystemC events only. In order to make such value changes visible, it is necessary to increase the granularity of the trace. For accomplishing this, a user is given the possibility to annotate UAL callback events in a design, which for instance mark the beginning and the termination of a call to a specific transaction or the assignment to a variable. Hence, in addition to each SystemC event occurrence, the occurrence of callback events leads to a new entry in the UAL trace and thus, a new sampling as well.

As previously mentioned, implicit events can not be observed in order to control the progression of the trace. Implicit events in SystemC are used for modeling synchronization of processes on a simulation time basis (Sec. 3.2.2). UAL offers special timer events which can be emitted from within assertions. These timer events enable active rather than reactive monitoring of a design at specific simulation times. Hence, also the occurrence of timer events leads to a new entry in the UAL trace.

According to these considerations, the progression of the UAL trace is controlled by the following classes of events:

- SystemC events: These include value-change events of signals as well as custom events
- UAL callback events
- UAL timer events

Content

Every entry of the UAL trace has to yield the sampled valuation of any variable or signal employed in a TLM at the event occurrence which has lead to the creation of such an entry in the trace. In addition to that, such an entry also has to yield the current valuations of all inputs and outputs of the TLM. Inputs and outputs in turn can be signals or variables, as well as transaction arguments and return values. Since, UAL sequences are evaluated on an event-driven basis, an entry in the trace also has to reveal which event has led to its creation in order to enable a clear identification of an event occurrence. Since UAL also provides the possibility to express propositions based on the simulation time, a UAL trace entry also reveals a time stamp which yields the simulation time at which a particular entry has been created.

Formal Definition of the UAL Trace

Since all events are treated the same way within UAL, a common definition for events suffices.

Definition 1 *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.*

This includes that an event object value *Fire* showing up in two consecutive elements of a trace for one particular event object represents two occurrences of that event object.

Definition 2 *The event object tuple* E *consists of all existing event objects. 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 of a set of non-deterministic choices. Any two events are defined to never occur concurrently. Hence, occurrences of event objects are disjunct and only one event object in tuple E *may have the value* Fire *at a time.*

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

Definition 3 *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 4 *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 1 and 3 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 1 and 3:

Definition 5 *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 6 *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 7 *The simulation time* $\boldsymbol{T}$ *is represented by a natural number.*

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

The current time $t_s \in \boldsymbol{T}$ *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 8 *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.

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

The current value of E^i 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^i in τ.*

Definition 9 *The UAL trace τ can be considered as a compound of three sub-traces formed by the tuple items of s over the index i:*

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

By these definitions, a trace is obtained which reveals the global order of event occurrences including their time stamps and the sampled values of all data objects of a design.

Furthermore, the meaning of simultaneity is defined as follows:

Definition 10 *For the transaction level two definitions of simultaneity are possible:*

1. *Event-Simultaneity: The elements of sub-traces χ, and ω at an index $i \in \mathbb{N}$ are defined to be event-simultaneous to the event occurrence of an event object which has the value* Fire *in $\epsilon(i)$.*

2. *Time-Simultaneity: At all indices i and $j \in \mathbb{N}$ where $\chi(i) = \chi(j)$ the elements of $\epsilon(i)$ and $\omega(i)$ are considered time-simultaneous to elements $\epsilon(j)$ and $\omega(i)$.*

The trace and these concepts of simultaneity shall serve as the basis for the evaluation of temporal UAL specifications.

6.1.3 UAL Semantics with Regard to PSL and SVA

This section discusses the semantics of UAL with regard to the formal foundation of PSL and SVA.

Sequences

For RTL models, a state is defined to be the stabilized value of all signals at either the positive or negative edge of the models clock signal. The definition of a trace as for instance, given for PSL[2] [24] includes the value of a clock signal. Hence, elements can exist in the trace, where the clock does not change or makes an opposite transition (i.e., the edge is opposite to the one which triggers sequential processes). Due to the strong relation of the definition of an RTL model state to a specific clock edge, it suffices to consider temporal behavior only in terms of occurrences of the according clock edge. The definition of a trace, as for instance, used in PSL however, is more granular. Therefore, a reduction of this general trace to an RTL trace is necessary. This is achieved by defining clocked temporal operators [65]. These operators ensure that all transitions which are not simultaneous to the active edge of a corresponding clock are excluded from the evaluation. Hence, an RTL trace can be considered as a projection of the original unclocked trace to a clocked trace and the projection is obtained through clocked temporal operators. In order to define how the trace should be reduced (i.e., according to which edge), both PSL and SVA require clocking expressions for property and sequence descriptions. In multi-clocked sequence and property descriptions the projection of the general trace to a reduced trace can change depending on which clocking region a sequence or property evaluation has reached.

Sequences in PSL and SVA define regular expressions which are attempted to be matched against the reduced trace. The characters of regular expressions are propositions about the current state of the RTL model. The reduced state trace is "searched" incrementally for words which fulfill the specified regular expression. In case of multi-clocked sequences, the pattern is divided into consecutive sub-patterns which are defined over different clocking expressions. The sub-pattern which is temporally the first in the compound pattern is evaluated first, based on a state trace reduction defined by the clocking expression of this sub-pattern. As soon as a word is found which fulfills this sub-pattern, the next sub-pattern is evaluated. Here, the state trace is reduced based on the clocking expression of this sub-pattern and so forth. Hence, when evaluating multi-clocked sequences the reduction of the state trace changes when crossing from one clock to the next. Multi-clocked sequences in RTL-ABV are used for verifying clock domain crossing in an RTL model which incorporates differently

[2] A similar definition exists for SVA however the granularity of the trace is determined by the SystemVerilog simulation kernel.

clocked communicating processes. However, the majority of assertion specifications for RTL are formulated using singly clocked sequences.

In contrast to that, UAL sequences are multi-clocked in nature. A sequence is built on top of delay operators which are concatenated in order to express a temporal order on both events and Boolean propositions. The temporal delay in such an operator is expressed in terms of occurrences of abstract triggers. An abstract trigger is the outcome of arbitrarily complex trigger expressions. These trigger expressions reason about event occurrence and are used for expressing the sensitivity of a delay operator. Hence, within a UAL sequence, the TL trace τ is reduced dynamically as soon as the evaluation shifts from one instance of a delay operator to the next. The reduction is defined by the positive and negative sensitivity of a delay operator. The trigger expressions which define the sensitivity are formulated on the basis of the UAL event layer operators. Hence, the trigger expressions alone represent patterns specified on the ϵ sub-trace defined in Definition 1. Furthermore, the reduction also occurs on the basis of the time trace χ defined in Definition 3 if event operators are used which incorporate timing as well (e.g., the event layer operator *AND*). Time based reductions can not be specified with the semantics of PSL and SVA, since the underlying state trace does not incorporate timing information. The reduction is based solely on value changes of clock signals.

Evaluation Modes

In addition to the semantical difference of clocking expressions in RTL assertions on the one hand and trigger expressions in UAL on the other, further differences exist with regard to the UAL evaluation modes. The formal foundation of both PSL and SVA is defined by the semantics of linear temporal logic (LTL), which in turn is inductively defined over traces. As also mentioned earlier, a temporal logic formula has to hold for each element of the state path. The state progression starting from any state in the trace has to fulfill a given temporal logic formula. This definition allows to evaluate a temporal logic formula individually for all elements of the state transition trace. One evaluation beginning at one element with index i in the trace does not have to consider the history reflected by the elements in the trace with an index lower than i. Hence, different evaluations of a temporal logic formula overlap in terms of the considered elements of the trace. However, no evaluation has an effect on another overlapping evaluation. Therefore, the evaluation of a temporal logic formula can be mapped to deterministic finite state automata [47]. Due to these characteristics, it is not possible to use LTL formulas which specify pipelined behavior. With pipelined behavior it is necessary to consider the history of a trace, because it reflects the overall state of a pipeline. Hence, it is necessary that different evaluations of the same temporal logic formula can influence the results of each other.

This is defined in UAL through different evaluation modes tailored to the capturing of pipelined behaviors as well as retransmission patterns.

Due to the fundamental differences of UAL regarding the formulation of temporal sequences as well as the consideration of behaviors depending on past behaviors, a formalism is developed in the next sections of this chapter based on a HLCPN, since the classical approach of a state machine does no longer fit in the general case.

6.2 Concept

This section discusses the concepts for providing a formal semantics for UAL based on a HLCPN. Furthermore, all basic elements of the HLCPN are defined.

The evaluation of a UAL assertion is defined by colored tokens that propagate through the structures of the HLCPN representation of that assertion. Differently colored tokens represent different evaluations of the same assertion.

Basically, all UAL operators map to HLCPN components which define the function of the corresponding operator. UAL assertions are built by connecting the different HLCPN components together. In order to do this, a hierarchical representation of the HLCPN is chosen.

For the definitions provided in the next sections, it is assumed that the HLCPN is executed on-the-fly since assertions can influence the number of elements of the UAL trace τ by emitting the special assertion timer event t_x in E for obtaining a new sample of all design objects. Hence, an a priori existence of the UAL trace τ can not be assumed. However, it is assumed that the HLCPN always stabilizes before a new entry is created in the trace.

6.3 Global Definitions

This section provides vital definitions which are relevant for the comprehension of the further sections. At first, the interface of the HLCPN to the UAL trace τ is defined followed by the definition of the HLCPN.

6.3.1 Interfacing the Trace

It is necessary to interface the HLCPN to the UAL trace τ. In order to accomplish this, the following definitions are given:

Definition 11 *The variable C_IDX represents the current index in τ and as such, yields the current count of event occurrences.*

Definition 12 *The function C_EV returns the current event object with value $Fire$ stored in $\epsilon(C\_IDX)$.*

Definition 13 *The function C_TIME returns the current time value $\chi(C\_IDX)$.*

In order to guarantee that the UAL trace will have an element with a desired time stamp, a function is required which schedules the emission of a timer event t_x in E. The occurrence of the timer event leads to adding a new symbol to the UAL trace τ.

Definition 14 *The function $TX(t_s \in \boldsymbol{T})$ enforces the emission of one event of event object t_x in* E *at $t_s + C\_TIME$.*

6.3.2 High-Level Colored Petri-Net

The High-Level Colored Petri Net for representing the operational semantics of UAL is a tuple:

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

P := Finite set of nodes called Places
TR := Finite set of nodes called Transitions with $P \cap TR = \emptyset$
A := Finite set of arcs $A \subseteq (P \times TR) \cup (TR \times P)$
connecting places with transitions and vice versa
M_0 := Initial marking of the net
C := Set of colors

The particular elements of the HLCPN are defined in the following sections.

6.3.3 Token Structure

One evaluation thread of an assertion is represented by one colored token -with the exception of a black token - which propagates through the HLCPN structures. The token is a high-level data structure, which is used to store information to be processed by various HLCPN components. The definition of a token is as follows:

Definition 15 (Token) *A colored token - with the exception of a black token - represents one evaluation attempt (also referred to as 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.*

$$C_t := (TID, STID) \tag{6.11}$$

$$I_t := (STS, S, TS, ACC\_LST, CA\_LST, ELID, IDX) \tag{6.12}$$

The different structure items are defined as follows:

$TID \in \mathbb{N}_0$:= *ThreadID as unique identifier of an evaluation thread*

$STID \in \mathbb{N}_0$:= *SubThreadID as unique identifier of an alternative of one evaluation thread also referred to as sub-thread*

$STS \in \mathbb{N}$:= *SubThreadSpace as indicator for the number of available sub-thread IDs (i.e., the number of sub-threads that can be created)*

$S \in Res$:= *The current state of one thread, initially set to* $Match$ *with* $Res := \{Match, NotMatch, Vacuous, Report\}$

$TS \in \mathbb{N}_0$:= *Holds a simulation time stamp of a token for comparisons with the current simulation time*

ACC_LST := *List of variables* $ACC\_LST := \{acc|acc_0, acc_1, \dots\}$ *with accumulation values* $acc_i \in \mathbb{N}$*;*
$acc_i :=$ *accumulation value of the* $i^{th}ACCUMULATOR$ *operator in a sequence, counted from left to right*

CA_LST := *List of consumption attempts;* $CA\_LST := \{ca_1, ca_2, \dots ca_i\}$ *with* i *: Number of delay operators in a sequence; each consumption attempt* ca_i *is represented by a tupel* $(Index, Bool\_ID)$*, with* $Index, Bool\_ID \in \mathbb{N}_0$ *where* $Index$ *holds the trace index at the time of the consumption attempt while Bool_ID holds the unique identifier (see Def. 34 in Sec. 6.7) of the Boolean proposition to be consumed*

$ELID \in EL$:= *Tag for identifying the origin of one token processed in the event expressions of a delay operator with* $EL := \{POS\_SENS, NEG\_SENS, NONE\}$

$IDX \in \mathbb{N}$:= *Field for storing the current index of the UAL trace* τ

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 sets of* TID, STID *are the natural numbers the set of possible colors* C *is infinite:*

$$C := {\mathbb{N}_0}^2 \tag{6.13}$$

Definition 16 (Structure Item Reference) *References to a structure item in* TK *is 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 TID for instance is written in the form $TK.TID$.*

In order to control the propagation of tokens, which represent threads, through a HLCPN representation of an assertion, sub-structures similar to semaphores are modeled. Hence, a special token is required which interacts with tokens that represent threads. The special token is a black token. A black token is defined as follows:

Definition 17 (Black Token) *Black tokens do not carry any information. A black token is identified by its valuation:*

$$TK^{Black} = ((0,0),(0, Match, 0, \emptyset, \emptyset, NONE, 0)) \tag{6.14}$$

In order to model the first thread of an assertion, an initial token is required which is defined as follows:

Definition 18 (Initial Token) *The initial token does not carry any information and is only needed for starting the assertion evaluation by starting the evaluation of the leftmost sequence in a property. Otherwise, it is treated as any other colored token. The valuation of the initial token is defined as follows:*

$$TK^{Gray} = ((1,1),(MSTS, Match, 0, \emptyset, \emptyset, NONE, 0)) \qquad \text{with } MSTS \in \mathbb{N} \tag{6.15}$$

The value MSTS *represents the maximum number of sub-threads that can be created in a sequence. Therefore, it depends on the structure of a sequence. It is calculated as follows:*

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

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

The color equality of two different tokens depends on the valuation of the structured color type C_t in a token:

Definition 19 (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.

Several HLCPN components need to store auxiliary information which is independent from the data in the tokens. While it is possible to store this information in additional tokens of appropriate types and provide arcs from the storage places of these tokens to all transitions where they are needed and back again, this would make the graphical representation of the petri net less intuitive. Due to this reason, all of this additional information is stored in variables which are visible to all transitions and places of a corresponding component.

6.3.4 Places

One key element of any petri net is a place. A place is the only item of a petri net where tokens may reside. In this work, places may hold an infinite number of tokens. The following figure defines the graphical notation of places in the HLCPN introduced in this work.

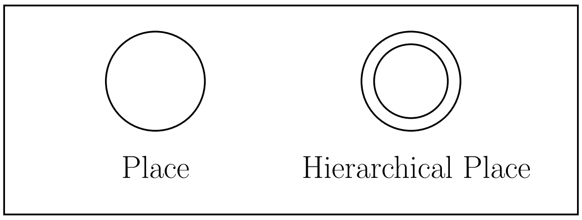


Figure 6.1: Types of Petri Net Places

Definition 20 (Place) *A place p $\in$ P can hold an unconstrained number of tokens of any color. A place can include an action to be performed uniformly on tokens arriving in p. An action may change the color and the information represented in a token. If no action is specified, the NULL action, which does not modify a token, is performed. Tokens which are added to one place event-simultaneously (i.e., caused by the same event occurrence), are sorted within a place from lowest to highest TID value of tokens TK. Tokens with the same TID value are sorted from lowest to highest according to the $STID$ value of these tokens. This order is preserved when tokens propagate out of a place.*

In order to enable a hierarchical construction of the HLCPN, it is necessary to define hierarchical places:

Definition 21 (Hierarchical Place) *A hierarchical place* p *is an encapsulation of a petri sub-net. Hierarchical places allow a concise description of modular structures.*

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.

Definition 22 (Marking) *The set of tokens residing in a 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.*

$$m'(p) = m(p) \pm k\ TK_x \tag{6.17}$$

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

6.3.5 Transitions

Transitions are required in order to propagate a token from one place to another. A transition controls which tokens may propagate from its set of input places to its set of output places. Furthermore, it controls when these tokens propagate. In order to be able to map all UAL constructs to a corresponding HLCPN, it is required to define several transition types.

Figure 6.2 depicts the graphical notation of the different transition types used within this HLCPN.

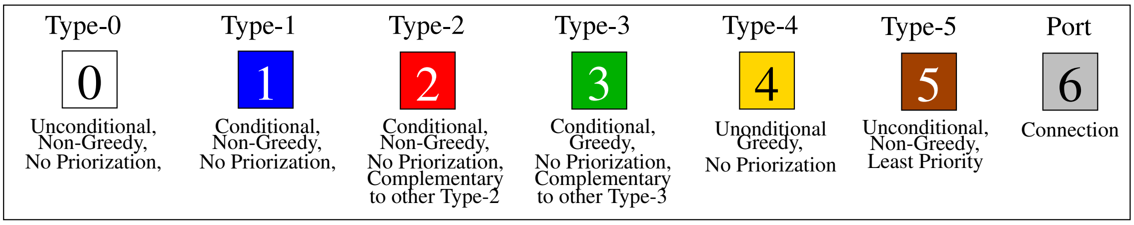

Type-0
0
Unconditional, Non-Greedy, No Priorization,
Type-1
1
Conditional, Non-Greedy, No Priorization,
Type-2
2
Conditional, Non-Greedy, No Priorization, Complementary to other Type-2
Type-3
3
Conditional, Greedy, No Priorization, Complementary to other Type-3
Type-4
4
Unconditional Greedy, No Priorization
Type-5
5
Unconditional, Non-Greedy, Least Priority
Port
6
Connection

Figure 6.2: Types of Petri Net Transitions

The general behavior of a transition is defined as follows:

Definition 23 (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 is described 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* $p \in tr_i{}^\bullet$

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 is determined by the non-black input tokens.

If n tokens $TK^c_{1...n}$ *of the same color c but possible different valuations in the structured information type* I *are removed, the following transformation rules are applied in the Action Phase for determining the valuation of structured information type* I *for a token* TK_+ *which is then added to the set of output places:*

$$\begin{array}{lcl}
TK_+.STS & = & \max(TK^c_{1...n}.STS) \\
TK_+.S & = & \begin{cases} \mathit{Report} & \mathit{if} \quad \exists TK_i^c : TK^c.S = \mathit{Report} \\ \mathit{Vacuous} & \mathit{if} \quad (\nexists TK_i^c : TK^c.S = \mathit{Report}) \wedge \\ & \qquad (\exists TK_i^c : TK^c.S = \mathit{Vacuous}) \\ \mathit{NotMatch} & \mathit{if} \quad (\nexists TK_i^c : TK^c.S = \mathit{Report}) \wedge \\ & \qquad (\nexists TK_i^c : TK^c.S = \mathit{Vacuous}) \wedge \\ & \qquad (\exists TK_i^c : TK^c.S = \mathit{NotMatch}) \\ \mathit{Match} & \mathit{else} \end{cases} \\
TK_+.TS & = & \max(TK^c_{1...n}.TS) \\
TK_+.ACC\_LST[i] & = & \max(TK^c_{1...n}.ACC\_LST[i]) \\
TK_+.CA\_LST & = & \bigcup_{i=1}^{n} TK_i^c.CA\_LST \\
TK_+.ELID & = & \begin{cases} NEG\_SENS & \mathit{if} \quad \exists TK_i^c : TK^c.ELID = NEG\_SENS \\ POS\_SENS & \mathit{if} \quad (\nexists TK_i^c : TK^c.ELID = NEG\_SENS) \wedge \\ & \qquad (\exists TK_i^c : TK^c.ELID = POS\_SENS) \\ NONE & \mathit{else} \end{cases} \\
TK_+.IDX & = & \max(TK^c_{1...n}.IDX)
\end{array} \tag{6.18}$$

The transformation rules for the *Action* phase are defined such that no information is lost. The rules which are defined using the *max* function are defined for consistency reasons. Within the HLCPN representation of UAL assertions the corresponding valuations of the removed tokens are always equal.

The most general transition of this HLCPN is a Type-0 transition which is defined as follows:

Definition 24 (Type-0 Transitions) *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:*

$$\begin{aligned} &tr_i^e : M \rightarrow \mathbb{B} \\ &\exists c \in C, \forall p \in {}^\bullet tr_i : |m_c(p)| \geq 1 \end{aligned} \tag{6.19}$$

The Removing Phase extracts one token from all input places for the color c:

$$\begin{aligned} &tr_i^- : M \rightarrow M,\ m_c \rightarrow m'_c \\ &\forall p \in P, c \in C : m'_c(p) = \begin{cases} m_c(p) - 1\,\mathrm{TK}^c & \textit{if } p \in {}^\bullet tr_i \\ m_c(p) & \textit{else} \end{cases} \end{aligned} \tag{6.20}$$

The Adding Phase then sends one token TK_+ (see Def. 23) to all output places.

$$\begin{aligned} &tr_i^+ : M \rightarrow M,\ m'_c \rightarrow m''_c \\ &\forall p \in P, c \in C : m''_c(p) = \begin{cases} m'_c(p) + 1\,\mathrm{TK}_+ & \textit{if } p \in tr_i{}^\bullet \\ m'_c(p) & \textit{else} \end{cases} \end{aligned} \tag{6.21}$$

A Type-1 transition is used for interfacing the HLCPN to the ϵ and χ sub-traces of the general TL trace τ.

Definition 25 (Type-1 Transitions) *A Type-1 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, 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

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

The Removing and Adding Phase have the same behavior as defined in 6.20 and 6.21.

The Boolean condition G may only represent propositions on the current trace element indicated by $C\_IDX \in \mathbb{N}$ of the ϵ and χ -sub-trace of the TL trace τ and the color $c \in C$ of a token TK. Condition D guarantees that a token is delayed for at least one event occurrence.

Type-2 transitions are used for modeling alternative ways for token propagation through the petri net via Boolean conditions. The definition is as follows:

Definition 26 (Type-2 Transitions) *A Type-2 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:*

$$\begin{array}{l} tr_i^e : M \mapsto \mathbb{B} \\ \exists c \in C, \forall p \in {}^\bullet tr_i : (|m_c(p)| \geq 1) \wedge (G = true) \end{array} \tag{6.24}$$

The Removing and Adding Phase have the same behavior as defined in 6.20 and 6.21. Type-2 transitions must be specified in groups. The Boolean conditions within a group are disjunctive and complementary. Hence, one transition of a group fires immediately.

Type-2 transitions can only occur in at least pairs per place. The corresponding enabling conditions are disjunctive, but at any given time exactly one will evaluate to true. The Boolean condition G may only represent propositions formulated on the current element of the ω sub-trace of the TL trace τ or on the marking of internal variables of the surrounding HLCPN structure which are part of the overall marking M of the HLCPN. Furthermore, G may also represent propositions formulated on internal token data.

A Type-3 transition allows that one token in one input place leads to the removal of all tokens in the remaining input places if the assigned condition evaluates to true. The definition is as follows:

Definition 27 (Type-3 Transitions) *A Type-3 transition tr_i is enabled according to the same definition as given in 6.22. In the Removing Phase however, the transition extracts all tokens of color c from the associated input places (greedy behavior):*

$$\begin{array}{l} tr_i^- : M \rightarrow M,\, m_c \rightarrow m'_c \\ \forall p \in P, c \in C : m'_c(p) = \begin{cases} \emptyset & if\, p \in {}^\bullet tr_i \\ m_c(p) & else \end{cases} \end{array} \tag{6.25}$$

The Adding Phase has the same behavior as defined in 6.21. Type-3 transitions, like Type-2 transitions, can only occur in groups per place. The corresponding enabling conditions are disjunctive and complementary. Hence, one transition of a group fires immediately.

A Type-4 transition is an unconditional transition which shows greedy behavior.

Definition 28 (Type-4 Transitions) *A Type-4 transition tr_i is defined like a Type-3 transition (Def. 27), however no condition is assigned.*

A Type-5 transition is used for ensuring that tokens are removed from the transitions set of input places if and only if no other transition fires for these tokens. The definition is as follows:

Definition 29 (Type-5 Transitions) *A Type-5 Transition tr_i is similar to Type-0 transitions, but is to be enabled for a color $c \in C$ if and only if no other transition tr_j with ${}^{\bullet}tr_i \cap {}^{\bullet}tr_j \neq \emptyset$ is enabled:*

$$\begin{aligned} & tr_i^e : M \to \mathbb{B} \\ & \exists c \in C, \forall p \in {}^{\bullet}tr_i : (|m_c(p)| \geq 1) \wedge (\nexists tr_j \in \{p^{\bullet} \setminus tr_i\} : tr_j^e = true) \end{aligned} \tag{6.26}$$

The Removing and Adding Phases have the same behavior as defined in 6.20 and 6.21

In order to enable a modular description of the HLCPN transition ports are defined as follows:

Definition 30 (Port Transitions) *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, no specific behavior is associated with it.*

6.4 Hierarchical Overview

Figure 6.3 shows the overall structure of the HLCPN representation of UAL assertions. The gray boxes indicate the basic components of the HLCPN mapped to the formal UAL grammar listed in Appendix B. The dashed boxes are only compound components which can be constructed through the basic components following the grammar rules. The solid arrows indicate the information flow between two components. The dashed arrows are only a visual aid. All basic components are introduced in the next sections following the layered approach of UAL. Both Boolean and Modeling layer are skipped because they do not influence the temporal semantics of UAL.

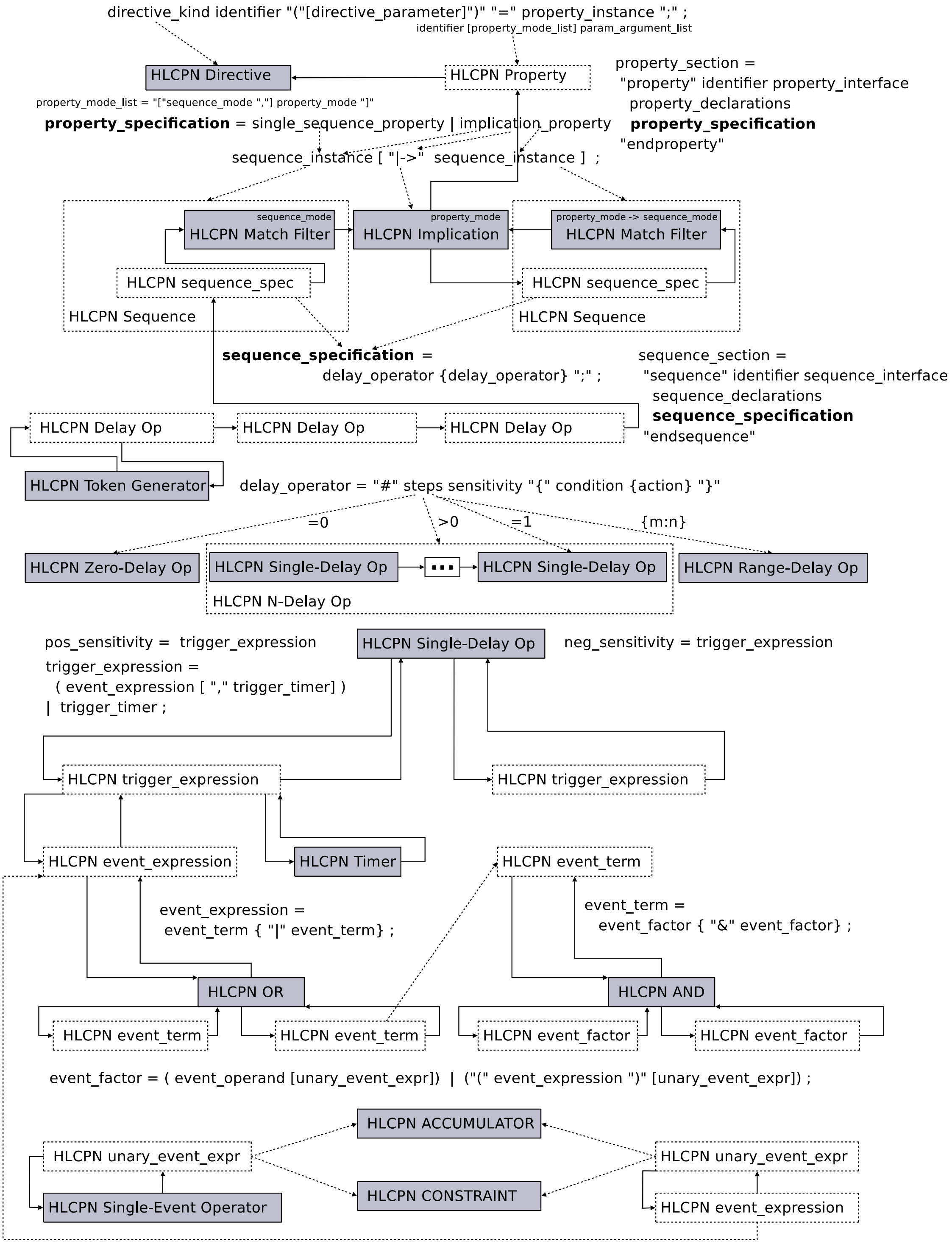


Figure 6.3: HLCPN Mapping of UAL

Table 6.1 lists some methods which are used in various of the components to be introduced in the next sections.

Symbol	**Definition**
Decr(x)	$x' = x - 1$
Incr(x)	$x' = x + 1$
SetBlack()	$TK' = TK^{Black}$
SetNotMatch()	$TK'.S = NotMatch$
SetTokenIDX()	$TK'.IDX = C\_IDX$

Table 6.1: Common Methods

6.5 Verification Layer

The main operators available at the verification layer are the various verification directives. These are represented by the HLCPN Directive component in Figure 6.3. If a token reaches a verification directive the state $TK.S$ is decoded upon which the corresponding action defined by the directive is executed. A violation of the associated property is indicated by $TK.S = NotMatch$ or $TK.S = Report$. A real success is indicated by $TK.S = Match$ and a vacuous success by $TK.S = Vacuous$. For directives which collect coverage data the HLCPN representation of a directive contains variables for counting the various property evaluation results.

The association of a directive with a particular property instance enables the evaluation of this instance. The only impact a verification directive may have further on a property evaluation is caused through a reset event expression which may have been specified for a directive. The following constraints are defined for a reset event expression:

Definition 31 *The reset event expression of a verification directive has to obey several rules.*

1. *In addition to Single Event Operators, only operators may be used which do not involve timing and no accumulation formulated on events. These operators are:*

   - *OR*
   - *CONSTRAINT*

   Furthermore, the use of function $\$delta\_t$ is forbidden in conjunction with a constraint operator.

2. *A reset expression of a verification directive may never be fulfilled event-simultaneously to any positive or negative trigger expression in a sequence which lies in the scope of this particular directive. In this case the behavior of the property evaluation is non-deterministic.*

The HLCPN representation of the reset detection is part of the HLCPN Directive component. Figure 6.4 depicts how a reset is handled based on the evaluation of a reset event expression which is constructed out of event-layer HLCPN components. Event layer expressions in general are discussed later in Section 6.8.

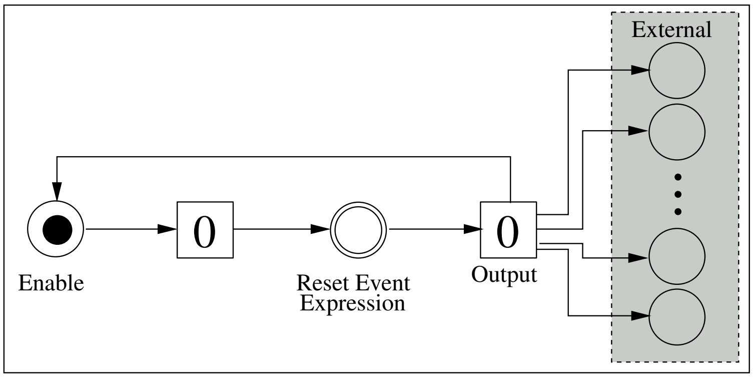


Figure 6.4: HLCPN Reset Representation

The place *Enable* contains a black token in its initial marking. The hierarchical place represents a reset event expression. When the black token propagates to the reset event expression it will propagate out again if a reset occurs. The token is copied through the transition called *Output*. One copy propagates back to the place *Enable*. The number of remaining arcs is determined by the number of places which need to be reset. The graphical notation of the other components in the remaining layers shows which places are reset by having an arc marked with identifier *Reset* attached to it as indicated by Figure 6.5.

Figure 6.5: HLCPN Reset Representation

The coverage variables of a verification directive are not affected by a reset. The further structure of the HLCPN Directive component is skipped to simplify the graphical notation.

6.6 Property Layer

Two possible kinds of properties are supported in UAL. With single sequence properties, the only effect of a property on the overall evaluation is the derivation of the sequence mode for its sequence instance. Any received token is passed on to the directive to which it is connected.

Implication properties consist of a sequence instance as antecedent for an implication operator and a sequence instance for the consequent of that same operator. Through the mode settings the evaluation mode for the antecedent is derived as well as the evaluation mode for the implication operator and its consequent. The mode setting for the consequent ensures that only one token can be produced by the consequent sequence for one token arriving from the antecedent sequence.

Figure 6.6 depicts the internal structure of the HLCPN Implication component from Figure 6.3. The Boolean conditions are listed beside the corresponding Type-2 transitions. Methods performed on the internal variables and tokens are marked by identifiers. The corresponding definitions can be found in Table 6.2. All methods are

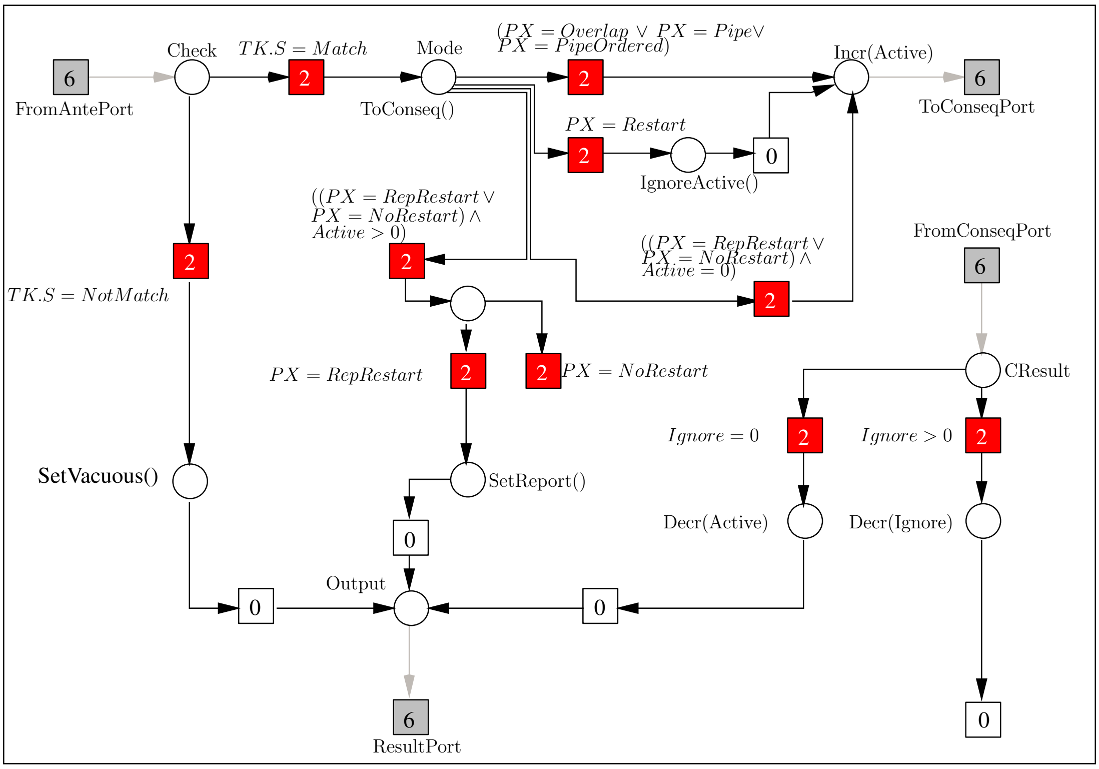


Figure 6.6: HLCPN Implication Component

atomic. Procedural assignments are separated by ';'. Each procedural assignment could be represented by an additional place with a Type-0 transition in between. In order to keep the graphical notation of the petri net model concise these are subsumed to one place.

Symbol	**Definition**
SetVacuous()	$TK'.S = Vacuous$
ToConseq()	$TK' = TK^{Black}$; $TK' = TK^0; TK'.TID = CSID; TK'.STS = CSTS$; $CSID' = CSID + 1$
SetReport()	$TK'.S = \mathit{Report}$
IgnoreActive()	$\mathit{Ignore}' = \mathit{Active}; \mathit{Active}' = 1$

Table 6.2: HLCPN Implication Component: Internal Methods

The implication operator has four port transitions. One for receiving tokens from the antecedent sequence (*FromAntePort*), one each for sending tokens to and receiving tokens from the consequent sequence (*ToConseqPort*, *FromConseqPort*). The fourth port transition represents the output of an implication property (*ResultPort*). The implication operator has a parameter PX which represents the property mode and a parameter $CSTS$ which holds the maximum number of sub-threads that can be created in the consequent. How this number is calculated is defined in Section 6.3.3, Equation 6.16. Furthermore, the implication operator has three variables:

- $CSID \in \mathbb{N}$: This variable stores the consequent thread id which is used for setting the color of a token which is routed to the consequent sequence. The initial and reset value $CSID^0$ is defined to be equal to one.
- $\mathit{Active} \in \mathbb{N}$: This variable reflects the number of tokens which are sent to the consequent and have not yet arrived back at the implication operator. The initial and reset value Active^0 is defined to be equal to zero.
- $Ignore \in \mathbb{N}$: This variable reflects the number of tokens to be ignored when received from the consequent. The initial and reset value $Ignore^0$ is defined to be zero.

These variables are also modeled through HLCPN s. However, due to the simplicity a graphical representation is skipped in order to avoid bloating.

The transition *FromAntePort* transports a token from the antecedent sequence to the place called *Check*. Here, the token is checked whether it represents an antecedent match or not. In case it represents a not-match, it has to be interpreted as a vacuous success of the implication and propagates out as such via transition *ResultPort*.

In case it represents a match, the token propagates to the place called *Mode* where it is recolored to a consequent token by using method *ToConseq*. This method first recolors the token to a black token. Following that, it sets a new color c which represents a new token for the consequent.

In case the property mode is set to either mode *Overlap* or one of the pipelining modes, the token propagates to the input place of transition *ToConseqPort*. Before a token propagates for further evaluation to the consequent via transition *ToConseqPort*, the value of variable *Active* is incremented. In mode *Restart* the token represents a retransmission which leads to setting variable *Ignore* to the current value of variable *Active* which is then set to 1 by calling method *IgnoreActive*. If either mode *NoRestart* or *ReportOnRestart* is set, it is checked whether there are active evaluations in the consequent. If there are no active evaluations, the token proceeds towards the consequent. If there are active evaluations the token may not proceed towards the consequent. Hence, in case of mode *ReportOnRestart* the token is recolored by method *SetReport* and propagates to the transition *ResultPort*. In case of mode *NoRestart* the token is discarded.

When a token arrives from the consequent, which means that the consequent has completed the evaluation for the thread represented by the arriving token, it is checked whether it has to be ignored. This is indicated by variable *Ignore*, which may only be set to a value not equal zero in case of mode *Restart*. If the token is to be ignored the variable *Ignore* is decremented and the token discarded. If the token may not be ignored the variable *Active* is decremented. The state of the token represents the result of the property evaluation and the token propagates towards the output of the implication operator.

6.7 Sequence Layer

The HLCPN contains the following components from Figure 6.3 which when combined represent the UAL sequence layer:

- HLCPN Token Generator: This component generates tokens. It represents the creation of evaluation threads for UAL sequences.
- HLCPN Zero-Delay Operator: This component represent the UAL delay operator configured with a zero-step setting (see Rule B.28, p.205).
- HLCPN Single-Delay Operator: This component represent the UAL delay operator configured with a multi-step setting (see Rule B.28, p.205) equal to value 1.

- HLCPN Range-Delay Operator: This component represent the splitting of threads to subthreads, and thus is a representation of a delay operator configured with a range-step setting (see Rule B.28, p.205).
- HLCPN Match Filter: This component represents the evaluation mode of a UAL sequence.

A sequence can be built by these elementary HLCPN components. The following rewriting rules for a delay expression within a sequence are used:

Definition 32 *The minimum delay of a delay range always has to equal zero:*
`#{m:n}{...}{BE} =>`
 `#m{...}{true} #{0:(n-m)}{...}{BE}`

Definition 33 *A delay operator configured with a multi-step setting consists of several delays configured with a multi-step setting equal to one:*
`#N{...}{BE} =>`
 $\texttt{\#1\{...\}\{true\}}_0\ \texttt{\#1\{...\}\{true\}}_1\texttt{...\#1\{...\}\{true\}}_{N-1}\texttt{\#1\{...\}\{BE\}}_N$

In order to be able to define consumption attempt conflicts on Boolean propositions, it is necessary to define a measure which provides a distinction of Boolean propositions. Therefore, it is necessary to provide a unique identifier for Boolean propositions present in a sequence description.

Definition 34 *The function ID is defined over the set B of Boolean proposition present in a sequence. The set B contains all Boolean proposition bp_i of a sequence description except for constant true propositions tp.*

$$\begin{array}{l} B := \{bp|bp \not\equiv tp\} \\ ID : bp \in B \mapsto N \\ \forall bp_x, bp_y \in B \wedge bp_x \not\equiv bp_y : ID(bp_x) \neq ID(bp_y) \\ \forall bp_x, bp_y \in B \wedge bp_x \equiv bp_y : ID(bp_x) = ID(bp_y) \end{array} \tag{6.27}$$

Any arbitrary function for associating a Boolean proposition $bp \in B$ with a number $i \in \mathbb{N}$ can be used which fulfills the above mentioned restrictions.

Within the next sections each mentioned HLCPN component is introduced.

6.7.1 HLCPN Token Generator

The token generator is connected to the left-most delay operator in the left-most sequence of a property. Hence, in case of a single sequence property it is the first delay operator of the sequence instance and within an implication property respectively the first delay operator of the antecedent. The first operator is determined after having applied the rewriting rule, mentioned above, where applicable.

The task of the token generator is to produce evaluation threads by generating new tokens. The graphical representation of the HLCPN Token Generator from Figure 6.3 is given in Figure 6.7.

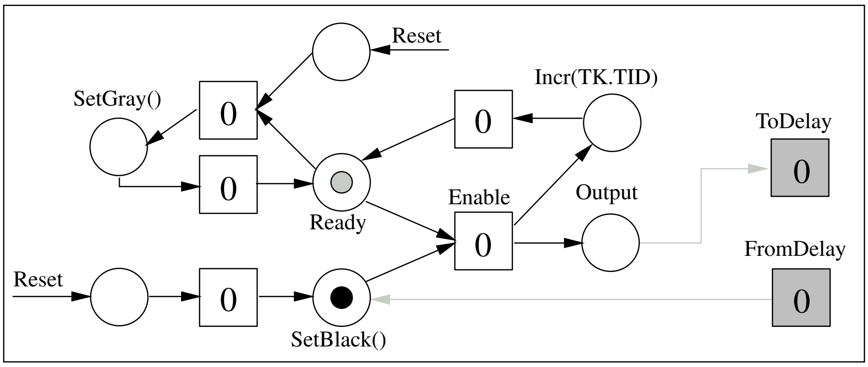


Figure 6.7: Token Generator

The initial marking of the token generator has two tokens. The black colored token in Figure 6.7 represents also the defined color TK^{Black}. The gray colored token in Figure 6.7 represents a token with the initial color TK^{Gray} as defined in Definition 18 in Section 6.3.3.

Method *SetGray()* performs the following operation on a token:
$SetGray : TK' = TK^{Gray}$

The maximum number of sub-threads that can be created is determined by the sequence to which the token generator is connected. The calculation of the maximum amount of sub-threads that can be produced in a sequence is defined in Definition 6.16, Section 6.3.3.

Initially, the transition *Enable* is enabled due to the existence of both the gray and the black token. The transition *Enable* fires immediately and adds the gray token to the place *Output* from where it is sent immediately to the first delay operator of a sequence. This represents the enabling of the sequence in which the token generator is located. The black token is consumed at transition *Enable* according to Definition 23.

At transition *Enable*, the gray token is also copied and propagated back to the place *Ready* after incrementing its TID structure item. Hence, the color of the token is changed to represent a new thread by changing its color c. This token however, can not enable the transition *Enable* since the black token does no longer reside in the original place. As soon as the connected delay operator has been triggered once, a copy of its output token propagates back to the token generator, where it is recolored to black using the method *SetBlack()*. Hence, there must be at least one event occurrence between a token propagating out via port *ToDelay* and a token propagating in via port *FromDelay*. The arrival of a black token back to the token generator via port *FromDelay* leads to an enabling of the transition *Enable* which immediately fires and sends a new token to the connected delay operator. Thus, a new evaluation thread is started.

If a reset occurs, the token residing in place *Ready* is removed from that place. Following that, it is set back to the initial gray color and propagated back to place *Ready*. A reset also leads to an adding of a black token to the initial place. Hence, a reset leads to the initial marking of this HLCPN component. Since a reset may not occur event-simultaneously to the triggering of a delay operator, it is guaranteed that no black token resides in the token generator on the arrival of a reset token.

6.7.2 HLCPN Sequence Item

Two HLCPN components are defined to represent the different sequence items of UAL. A sequence item can either be a Zero-Delay or Single-Delay operator or a Range-Delay operator (see Fig. 6.3). The latter works on top of the former two Delay operators.

Table 6.3 gives a definition for methods and functions used for the representation of the delay operator as a HLCPN.

Symbol	Definition
ELID(X)	$TK'.ELID = X$
AddCA()	$TK'.CA\_LST = TK.CA\_LST \cup \{(C\_IDX, ID(BE))\}$ with $BE \not\equiv tp$ $TK'.CA\_LST = TK.CA\_LST$ with $BE \equiv tp$
G(Pos)	$\exists TK^c : TK^c.ELID = POS\_SENS \quad \wedge$ $\nexists TK^c : TK^c.ELID = NEG\_SENS$
G(Neg)	$\exists TK^c : TK^c.ELID = NEG\_SENS$

Table 6.3: Delay Operator Methods and Functions

HLCPN Zero-Delay Operator

Since a zero-delay operator is not sensitive to events, a distinct HLCPN component is defined to represent its operational semantics.

Figure 6.8 shows the graphical representation of the HLCPN Zero-Delay operator.

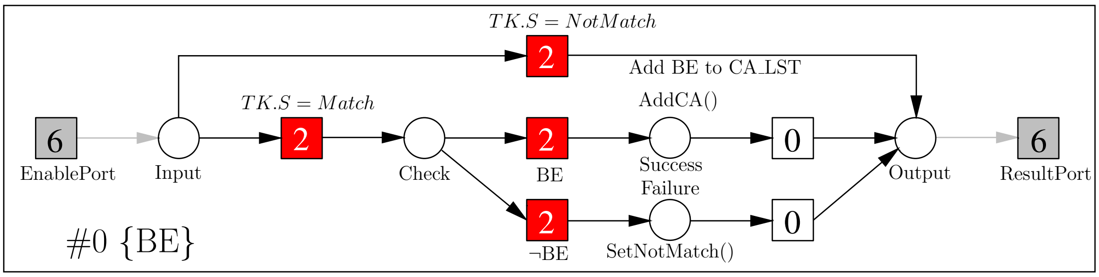


Figure 6.8: HLCPN Zero-Delay Operator

A zero-delay operator may not delay the evaluation. Therefore, the Boolean expression assigned with it has to be evaluated immediately. Tokens which arrive via transition *EnablePort* reside in place *Input*. Here, tokens are checked whether they carry a match or a not-match result. Tokens which represent a not-match are directly routed to the output of the operator, since the evaluation of the Boolean expression is redundant for a token that already represents a not-match. Tokens which are not marked as not-matches propagate to place *Check*.

Here, a token propagates immediately to place *Success* if the Boolean expression of the delay operator results to true when being evaluated against the ω trace at the current index of the UAL trace τ (see Def. 26 in Sec. 6.3.5). In place *Success* a consumption attempt has to be stored in the token for later evaluation of possible consumption conflicts in the match-filter. The consumption attempt is stored in the token by using method *AddCA()* defined in Table 6.3. Here, a tokens structure item *CA_LST* is used, which stores the unique ID of a Boolean proposition represented by the Boolean expression (see Def. 34 in Sec. 6.7) of the delay operator and the current index in τ. As the token propagates through the delay operators of a sequence, this list grows and reflects the consumption attempts of the thread represented by the token. Note that if the Boolean expression is a constant true expression *tp* nothing is added to this list.

At place *Check*, if the Boolean expression evaluates to false, a token immediately transitions to the place *Failure* where it is marked as not-match. The token is immediately propagated to the operators output.

Single-Delay Operator

The HLCPN component for the UAL delay operator with a delay value set to one is more complex.

Figure 6.9 represents the graphical notation of the HLCPN Single-Delay Operator component.

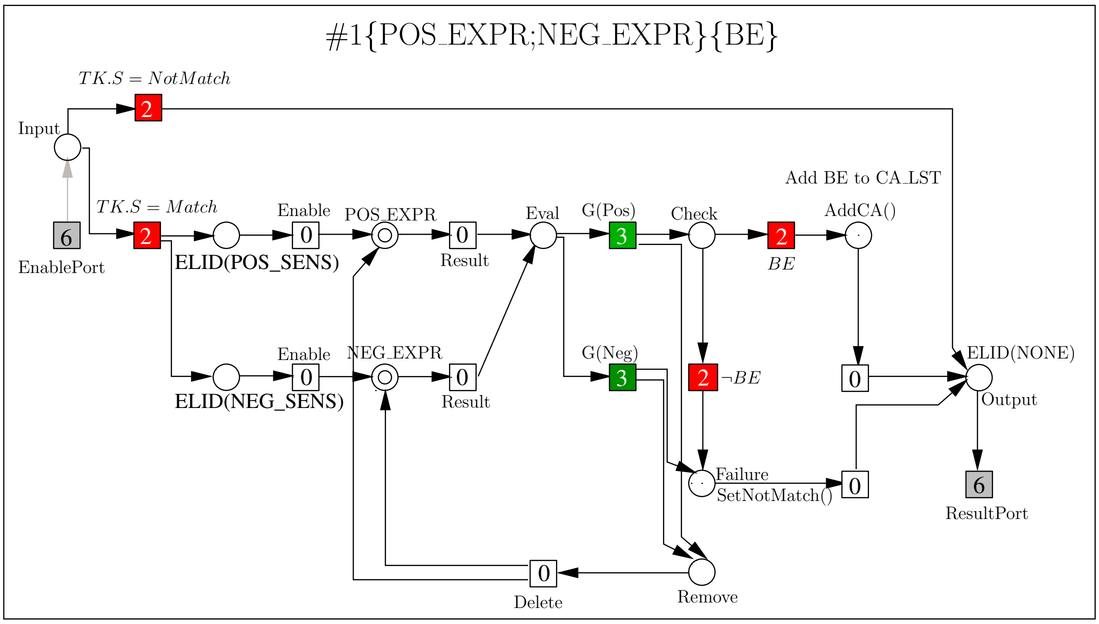


Figure 6.9: HLCPN Single-Delay Operator

Arriving tokens which represent a not-match result are directly routed to the place called *Output*. Tokens which reflect a match result are copied and propagated to the hierarchical places *POS_EXPR* and *NEG_EXPR*. These hierarchical places contain the HLCPN representation of the according positive and negative sensitivity of the delay operator. The positive and negative sensitivity are modeled with HLCPN representations of event layer operators. If either sensitivity is not specified, the petri net shown in Figure 6.10 is inserted in order to preserve petri-net semantics.

A token returns from these hierarchical places if the corresponding trigger has occurred. It is possible that both hierarchical places return a token event-simultaneously. In this case the tokens coming from place *NEG_EXPR* have to have a higher priority. Therefore, the tokens are gathered in the place called *Eval* where the prioritization is applied. However, in order to be able to distinguish two tokens that represent the same thread ($TK_x.TID = TK_y.TID$) it is necessary that the tokens

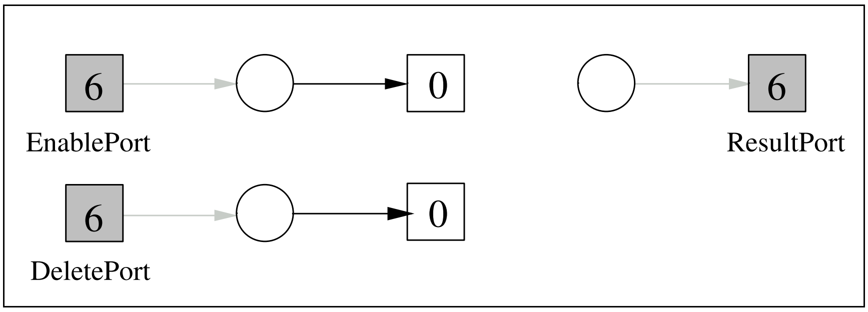


Figure 6.10: HLCPN Empty Sensitivity

have a different valuation in their information type *I*. This is done by calling method $ELID(X)$ to set the corresponding value for the structure item $ELID$ of a token.

The upper Type-3 transition is conditioned with $G(Pos)$ which is defined in Table 6.3. This transition fires if only a token from the hierarchical place *POS_EXPR* has arrived and no token from the hierarchical place *NEG_EXPR* has arrived event-simultaneously. This reflects that the delay operator has been triggered by a positive trigger only. The token propagates further to the place called *Check*. From place *Check* the evaluation proceeds the same way as in the HLCPN Zero-Delay Operator component. Only at the place called *Output* the $ELID$ item of the token is set to value $NONE$.

The lower Type-3 transition marked with condition $G(Neg)$ fires as soon as a token from the place *NEG_EXPR* has arrived. It ensures that in case a token of the same color *c* has arrived from the place *POS_EXPR* that it is removed as well from place *Eval*. Since the firing of this transition represents a negative trigger of the delay operator, the corresponding token propagates to the place called *Failure* where it is recolored to represent a not-match result. The token propagates out through place *Output*.

Both Type-3 transitions send a copy of a token to the place called *Remove*. The copy needs to propagate back to the hierarchical places in order to ensure that tokens which have the same color *c* and possibly are still under evaluation in the hierarchical places are deleted. For instance, on the occurrence of a positive trigger, the token which propagates into the hierarchical place *NEG_EXPR* is still under evaluation and needs to be deleted for the overall trigger evaluation has already completed.

Range-Delay Operator

A range delay operator has to accomplish the splitting of one thread into as many alternative sub-threads as required by its range expression. The HLCPN Range-Delay

Operator component is depicted in Figure 6.11. The methods used in Figure 6.11 are defined in Table 6.4.

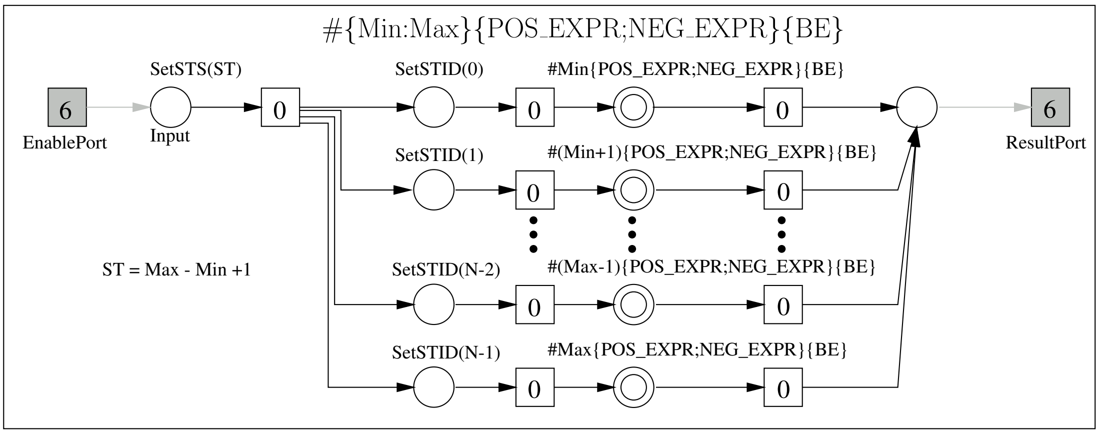

#{Min:Max}{POS_EXPR;NEG_EXPR}{BE}
SetSTS(ST)
SetSTID(0)
#Min{POS_EXPR;NEG_EXPR}{BE}
6
EnablePort
Input
0
0
0
6
ResultPort
SetSTID(1)
#(Min+1){POS_EXPR;NEG_EXPR}{BE}
0
0
ST = Max - Min +1
SetSTID(N-2)
#(Max-1){POS_EXPR;NEG_EXPR}{BE}
0
0
SetSTID(N-1)
#Max{POS_EXPR;NEG_EXPR}{BE}
0
0

Figure 6.11: HLCPN Range-Delay Operator

Symbol	Definition
SetSTS(x)	$TK'.STS = TK.STS \div x$
SetSTID(x)	$TK'.STID = TK.STID + x \cdot TK.STS$

Table 6.4: Methods for HLCPN Range-Delay Operator

On each token that arrives at place *Input*, the method *SetSTS* is used. The method *SetSTS* calculates a new sub-thread space $TK.STS$ for a token. This item expresses how many further sub-threads can be created for a token in other range-delay operators. Through the division by the number of sub-threads *ST* to be created in the range delay operator the calculation which yields the overall number of sub-threads for a sequence is performed backwards (see Eq. 6.16 in Sec. 6.3.3). Hence, the sub-thread space is reduced with every range delay operator in a sequence and should equal to one after the last range delay has been traversed.

When a token is removed from place *Input* it is copied to all following branches. Each branch represents one alternative evaluation of the range-delay operator. In each branch single-delay operators are used. The notation of the hierarchical places in Figure 6.11 is a shorthand notation of the rewriting rule mentioned earlier.

Right after a token is copied in all branches, a new value for the token structure item $TK.STID$ representing the sub-thread identification number of a token has to be calculated and assigned. This is accomplished by the method *SetSTID*. In order to avoid collisions within the numbering of sub-threads, it is necessary to add an offset between two sub-thread $STID$ values. Since the already reduced sub-thread space describes in how many sub-threads a thread is split further in later range delay operators, this value is used as an offset.

All alternatives evaluate in parallel. Tokens returning from the hierarchical places all propagate out via port *ResultPort*.

6.7.3 HLCPN Match Filter

A key HLCPN component within the sequence layer is a match filter which computes the final decision for a sequence result by applying the specified evaluation mode. The HLCPN Match Filter component is attached to the result port of the right-most delay operator in a sequence. Tokens arriving at the match filter represent preliminary results for the sequence evaluation. The sequence mode setting determines which preliminary results are confirmed and which are discarded. The match filter has two parameters listed in Table 6.5.

Definition	**Description**
SX	Sequence Modes
$SSTS \in \mathbb{N}$	Maximum sub-thread space of sequence

Table 6.5: HLCPN Match Filter: Parameters

The parameter SX can have the following values reflecting the according sequence mode:

- *AnyMatch*
- *FirstMatch*
- *FirstMatchPipe*
- *FirstMatchPipeOrdered*

The parameter $SSTS$ holds the maximum number of sub-threads possible in the sequence to which the match filter is connected.

The match filter component includes also list variables which store the necessary information for making the relevant decisions within one evaluation mode. Table 6.6

provides an overview on the structure of these lists and the methods which are defined for updating the lists. Also, the initial values are given. All lists take on their initial values on a reset. These lists also represent HLCPN structures . A net representation of these lists is skipped to allow for a well arranged graphical representation of the match filter component.

List	Definition	Update Method
$RL:$	$\{x_1, x_2, \dots\}$,	$UpdRL():$
	$x_i := (TID \in \mathbb{N}, STID \in \mathbb{N})$	$RL' = RL \cup \{(TK.TID, TK.STID)\}$
$RL^0:$	$\emptyset$	
$ML:$	$\{x_1, x_2, \dots\}, x_i \in \mathbb{N}$	$UpdML():$
		$ML' = ML \cup \{TK.TID\}$
$ML^0:$	$\emptyset$	
$PL:$	$\{x_1, x_2, \dots\}$,	$UpdPL():$
	$x_i = (y \in \mathbb{N}, z \in \mathbb{N})$	$PL' = PL \cup TK.CA\_LST$
$PL^0:$	$\emptyset$	
$MIL:$	$\{x_1, x_2, \dots\}, x_i \in \mathbb{N}$	$UpdMIL():$
		$MIL' = MIL \cup \{C\_IDX\}$
MIL^0	$\emptyset$	

Table 6.6: HLCPN Match Filter: Internal Lists and Update Methods

The list variable RL (Received List) is a set which holds tuples of thread and sub-thread identification numbers $(TK.TID, TK.STID)$. The variable is used to determine whether all possible tokens representing sub-threads have been received. This list is used with all sequence modes except *AnyMatch*.

The list variable ML (MatchedList) is a set which holds only the identification numbers $TK.TID$ of threads for which a match result has already been computed. This list is used for deciding whether a token has to be discarded due to the first-match principle. This list is used for all sequence modes except *AnyMatch*.

The list variable PL (PipeList) is a set which holds all consumption attempts which have already been granted. A consumption attempt is characterized by a tuple (i, b_id). This tuple corresponds to the same type as the elements in the consumption attempt list $TK.CA\_LST$ of a token (see Def. 15 in Sec. 6.3.3). The existence of such a tuple in the list PL indicates that the corresponding Boolean proposition has been attempted to be consumed at a specific index of the UAL trace τ. If a token and respectively the represented thread is decided to match, the content of its consumption attempt list $TK.CA\_LST$ is stored in list PL. This way all consumption attempts of the token are turned to granted consumptions. This list is used only for sequence modes $FirstMatchPipe$ and $FirstMatchPipeOrdered$.

The list variable MIL (MatchIndexList) contains the indices of the UAL trace τ at which threads have matched. This list is used in order to detect whether a match conflict exists and hence, is used only for sequence modes *FirstMatchPipe* and *FirstMatchPipeOrdered*.

Table 6.7 lists the major functions representing Boolean conditions which are queried while processing a token in the match filter component.

Symbol	**Definition**
AM:	$SX = AnyMatch$
FM:	$SX = FirstMatch$
FMP:	$SX = FirstMatchPipe$
FMPO:	$SX = FirstMatchPipeOrdered$
LSThread():	$(\lvert\{x \lvert x \in RL \wedge (x.TID = TK.TID)\}\rvert) = SSTS$
InOrder():	$((\lvert\{x \lvert x \in RL \wedge (x.TID = (TK.TID - 1))\}\rvert) = SSTS) \vee (ML \cap \{(TK.TID - 1)\} \neq \emptyset) \vee (TK.TID = 1)$
HMatched():	$ML \cap \{TK.TID\} \neq \emptyset$
CConfl():	$PL \cap \{TK.CA\_LST\} \neq \emptyset$
MConfl():	$MIL \cap \{C\_IDX\} \neq \emptyset$

Table 6.7: HLCPN Match Filter: Conditions

Figure 6.12 shows the graphical representation of the HLCPN Match Filter component.

Tokens arrive via the transition *InputPort* and propagate to the place *Input*. The black token in place *InputEnable* ensures that only one token is processed in the match filter at a time. However, all tokens in place *Input* are processed still event-simultaneous. As soon as the processing of one token is complete a black token propagates back to place *InputEnable*.

A token propagates from place *Input* to place *Mode1*. Here, through the condition *AM* in Table 6.7 it is checked whether the mode parameter SX is set to mode *AnyMatch*. If so the token propagates directly to place *Output*. Hence, in mode *AnyMatch* any token that arrives at the match filter propagates out of the match filter again. Each token represents a result of the sequence.

For all other modes the token propagates to place *Upd1* where the variable RL is updated. Afterwards, in place *FMCheck* the first-match condition is checked through

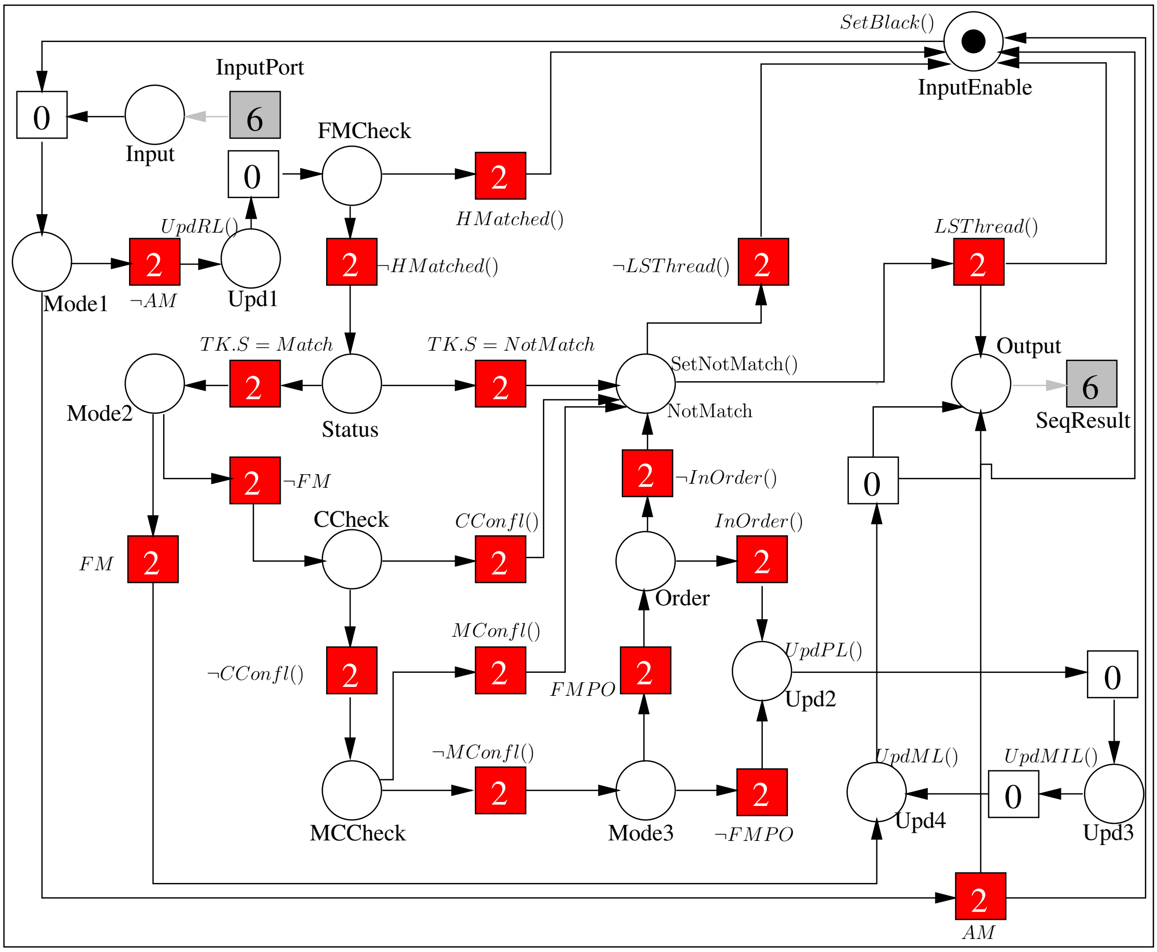


Figure 6.12: HLCPN Match Filter

determining whether a match has already been computed for the thread identification number $TK.TID$. The condition is evaluated through the function *HMatched()* listed in Table 6.7. The condition evaluates to true if the identification number $TK.TID$ is already in the list ML which only holds identification numbers of threads that have matched. If there was a prior match, the token propagates to place *InputEnable*, where it is recolored to black.

If there was no prior match the token continues to place *Status*, where it is checked whether the token represents a match or a not-match. If it is a not-match it propagates to place *NotMatch*. Here, through the use of condition *LSTThread()* it is decided whether this token is the last possible token for the thread it represents. The condition *LSTThread()* evaluates to true if the number of tuple elements in RL, which have the same thread identification number $TK.TID$ as the token in place

NotMatch, corresponds to the maximum number of sub-threads $SSTS$. If the evaluation is true the token represents a not-match result for the sequence and propagates to place *Output*. If not, the token is discarded by moving it to place *InputEnable*.

If a token in place *Status* does represent a match it propagates to place *Mode2*. Here, the evaluation mode setting determines where the token has to proceed to. If the mode is set to *FirstMatch*, the evaluation for that token is done and the token represents a match of the sequence. Hence, it propagates to place *Upd4* where the token identification number $TK.TID$ is added to the list variable ML which indicates which threads have matched. The token propagates further to place *Output* and a copy of it to place *InputEnable* to enable the next token in place *Input*.

If the sequence mode is set to any of the pipelined modes, a token propagates from place *Mode2* to place *CCheck*. Here, it is checked whether the thread represented by the token attempts to consume Boolean propositions which already have been consumed by earlier matching threads. If so, a consumption conflict exists and the token has to be propagated to place *NotMatch*, where it is recolored to represent a not-match. From there on, it is again checked whether the token is the final result for the thread it represents. The consumption conflict detection is checked through condition *CConfl()* listed in Table 6.7. The condition results to true if the consumption attempt list of the token $TK.CA\_LST$ is a subset of the list PL which holds all consumption attempts that have been granted already.

If no consumption conflict exists the token is propagated to place *MCCheck* where it is checked whether a match conflict exists. In case of a match conflict the token propagates to place *NotMatch* and is processed as if it was a not-match result. The match conflict condition *MConfl()* listed in Table 6.7 evaluates to true if the current state of the event counter returned by function *C_IDX* is already part of the list variable MIL.

If no match conflict exists the token propagates to place *Mode3*. Here, the continuation depends on which of the pipelined modes is selected. In case SX is set to mode *FirstMatchPipe* the evaluation of the token is complete. The token propagates to place *Upd2* where the consumption attempt list CA_LST is added to the list PL. Finally, the token propagates through places *Upd3* and *Upd4* where the list variables MIL and ML are updated. The token represents a final result of the sequence and propagates to place *Output* and a copy of it propagates to place *InputEnable*.

In case SX is set to mode *FirstMatchPipeOrdered*, the token propagates from place *Mode3* to place *Order*. Here, it is checked whether a final result has already been computed for the next thread identification number which is lower than the thread identification number of the token. If not it would mean that the token has overtaken an older token which is a violation of the in-order condition, posed by mode *FirstMatchPipeOrdered*. Whether an order violation exists is checked through condition *InOrder()*. The order condition *InOrder()* evaluates to true if either all tokens for the next smaller token identification number $TID_{low} = TK.TID - 1$ have passed the match filter, or if a match has already been computed for TID_{low}, or if the token represents the first thread of the sequence evaluation. An order conflict is treated the same way as a consumption or match conflict. If no order conflict exists the token propagates to the various places for updating the corresponding lists and finally propagates to place *Output* representing a match result of the sequence.

6.8 Event Layer

The HLCPN representation of the UAL event layer offers a component for each event operator. Furthermore, both a Single Event Operator and a TIMER are represented as a HLCPN component each. Based on the hierarchy concept, these components are connected to represent trigger expressions according to the UAL syntax tree for the event layer. These trigger expressions represent the positive and negative sensitivity of delay operators as well as reset event expressions of verification directives.

The following components are introduced in the next sections:

- HLCPN Single Event Operator
- HLCPN TIMER
- HLCPN OR Operator
- HLCPN AND Operator
- HLCPN CONSTRAINT Operator
- HLCPN ACCUMULATOR Operator

Each of these components and all combinations have three port transitions:

- *EnablePort*
- *DeletePort*
- *ResultPort*

These port transitions reflect the interface between event layer components as well as between a delay operator, where trigger expressions are represented as hierarchical places *POS_EXPR* and *NEG_EXPR*. A token arriving from the *EnablePort* enables the corresponding component. A token arriving from the *DeletePort* invokes the deletion of all tokens in the event layer operators which have exactly the same color. Tokens are passed to other components through *ResultPort*, including the delay operator.

The common methods and functions used in the event layer are listed in Table 6.8.

Symbol	**Definition**
SetTokenTime()	$TK'.TS = C\_TIME; TK'.IDX = C\_IDX$
ClearTokenTime()	$TK'.TS = 0; TK'.IDX = 0$
\$delta_t()	$C\_TIME - TK.TS$
TimeOut(X)	$(\$delta\_t = X \wedge C\_EV = t_x)$

Table 6.8: Event Layer Methods and Functions

6.8.1 HLCPN Single Event Operator

The graphical notation of the HLCPN Single Event Operator from Figure 6.3 is depicted in Figure 6.13.

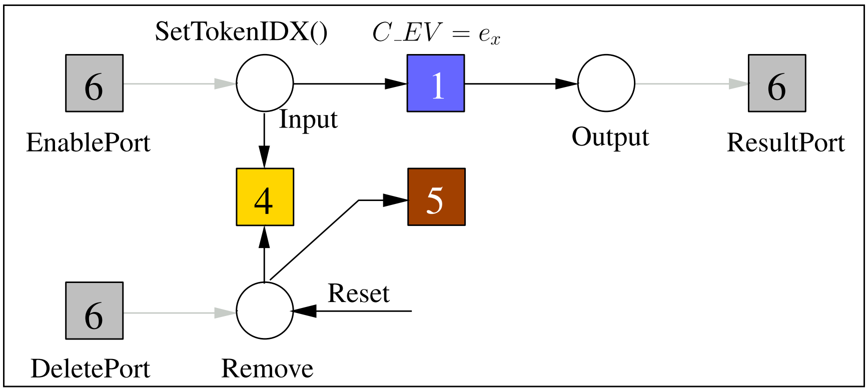


Figure 6.13: HLCPN Single Event Operator

The HLCPN Single Event Operator represents the link of an assertion to the event occurrences of the UAL trace τ, by permanently evaluating the function C_EV (see Def. 12, Sec. 6.3.1). The Single Event Operator has a parameter e_x which is a reference to an event object. The Single Event Operator thus, reacts on the occurrences of the referenced event object. Tokens arriving from transition *EnablePort* reside in the place called *Input* where the current trace index is stored in the structure item

$TK.IDX$ through the method $SetTokenIDX()$. Note that tokens may reside in the place *Input* for more than one progression of the UAL trace τ. The Type-1 transition transports a token to the place *Output* unless it is deleted, via the Type-4 transition. The condition of the Type-1 transition evaluates to true whenever the current event returned by function C_EV equals to the value of parameter e_x and if the value of the structure item $TK.IDX$ is less than the trace index (see Def. 25 in Section 6.3.5).

When a token arrives from the transition *DeletePort* it propagates to place *Remove*. If the tokens originate from a delay operator, they carry the color of the tokens to be deleted. Hence, if one token arrives in the place *Remove* and has the same color as a token in place *Input* it enables the Type-4 transition which removes both tokens. Since no output arc is attached to the Type-4 transition, the removed tokens are deleted. Note that if tokens arrive via transition *DeletePort* the Type-4 transition acts as a regular Type-0 transition, because it is not possible that place *Input* holds more than one token of a specific color.

If a black token arrives in place *Remove* through the reset arc, all tokens in place *Input* are removed along with the black token. This is due to the greediness of a Type-4 transition and due to the fact that the black color is considered to be equal to any other color.

In case a token arrives in place *Remove* while no appropriate tokens reside in place *Input*, the enabling condition of the Type-4 transition is not met. Therefore, the Type-5 transition fires, because it has the lowest priority. The Type-5 transition extracts the token from place *Remove* and deletes it.

6.8.2 HLCPN Timer

The internal structure of the HLCPN Timer component from Figure 6.3 is depicted in Figure 6.14.

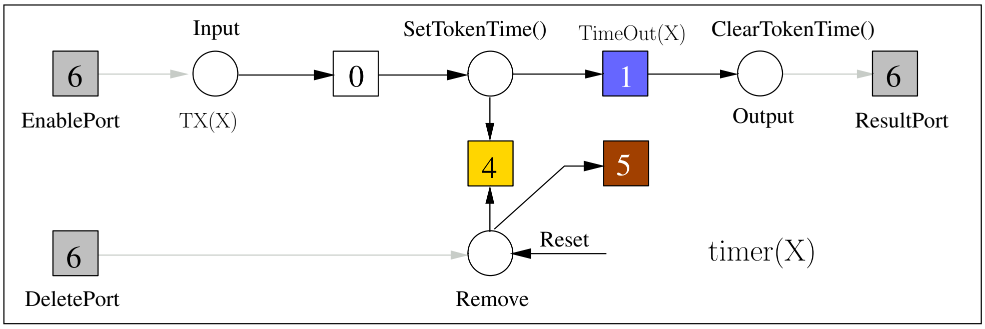


Figure 6.14: HLCPN Timer

When tokens arrive at place *Input* the emission of an assertion event t_x is scheduled to X time steps later than the current time by calling $TX()$ (see Def. 14, Sec. 6.3.1). This is done in order to ensure that the UAL trace will contain an entry at the desired time later. Tokens propagate from place *Input* to the next place where the time stamp $TK.TS$ is set to the current time using method *SetTokenTime()* listed in Table 6.8. Within this method also the trace index is stored in a token. The Type-1 transition is configured with a condition *TimeOut(X)* this condition evaluates to true only if an assertion event t_x occurs and if the time stamp of event t_x in the χ sub-trace of trace τ yields a difference of exactly X when compared to the time stamp value $TK.TS$ of a token. Only the tokens which have a corresponding time stamp value transition to place *Output*. Deletion and reset are the same as in the HLCPN Single Event Operator.

6.8.3 HLCPN OR Operator

The internal structure of the HLCPN OR operator from Figure 6.3 is depicted in Figure 6.15.

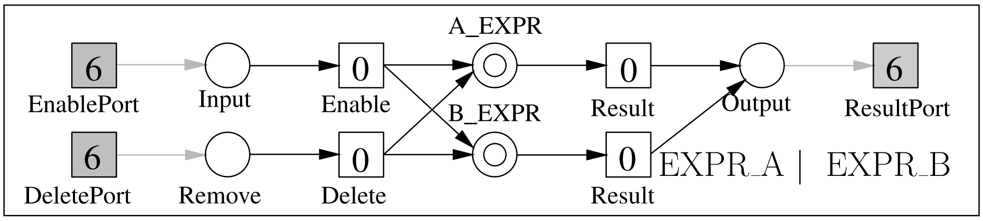


Figure 6.15: HLCPN OR Operator

Operand expressions are depicted as hierarchical places. As shown, tokens are broadcast via transition *Enable* to the place *Input* which is part of the operand expressions. Tokens for initiating a deletion are broadcast via transition *Delete* to the place *Remove* which in turn is part of the operand expression. Hence, the HLCPN OR operator ensures that both operand expressions are enabled and that any tokens returning from these operands reflect the result of the evaluation of this operator.

6.8.4 HLCPN AND Operator

The internal structure of the HLCPN AND operator from Figure 6.3 is depicted in Figure 6.16. The represented net has a symmetrical structure for both factor expressions. Tokens arriving via transition *EnablePort* propagate to the corresponding

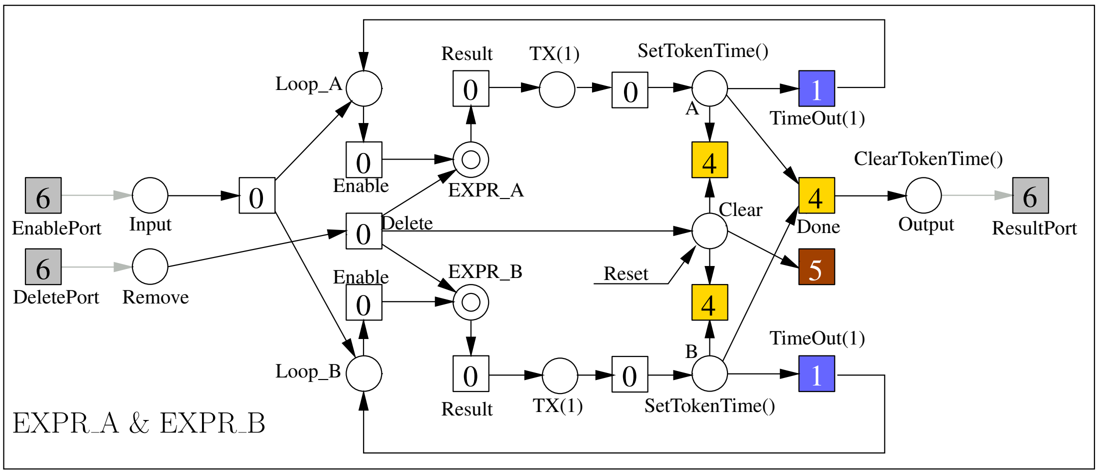


Figure 6.16: HLCPN And Operator

hierarchical places which represent the factor expressions. If a token propagates out of a hierarchical place it means that the corresponding factor expression is fulfilled. For such a token an assertion event t_x is scheduled to the next time increment because the definition of the AND operator requires that both factor expressions are fulfilled time-simultaneous. Hence, if a token propagates from one hierarchical place, a token of the same color c is required to propagate from the other hierarchical place time-simultaneously in order for the AND operator to be fulfilled. If a token resides in place A or B respectively the maximum duration for such a marking is one increment of time. If transition *Done* does not get enabled, the corresponding Type-1 transition will fire as soon as one increment of time has elapsed. In this case the token is sent back to the according hierarchical place for re-evaluation of the factor expression. Transition *Done* is a Type-4 transition, since it is possible that one operand expression can be fulfilled more than once within one simulation time slot for the same color. For instance, this could be the case when the hierarchical place *EXPR_A* represents an OR operator. If both operands of the OR operator are fulfilled in the same simulation time slot two tokens of the same color would propagate to place A of the AND operator. As soon as a token propagates out from *EXPR_B* at the same simulation time slot, all tokens of the same color are extracted from places A and B and are merged to a single token due to the greedy behavior of transition *Done*.

Tokens arriving from the *DeletePort* transition are propagated to both hierarchical places to delete tokens of the same color which are still under evaluation within the hierarchy. Furthermore, delete tokens are propagated to the place *Clear*, because it is possible that tokens of the same color are still waiting in places A or B. Note that for the color c of the delete token it is not possible that a token of the same color can reside both in place A and B. Therefore, both Type-4 transitions can not be enabled at the same time.

A Reset token is propagated to place *Clear*, because tokens can remain either in place *A* or *B* for several event occurrences. In this case the greedy behavior of the Type-4 transition ensures that one black token leads to the removal of all tokens in either place *A* or *B*. The Type-5 transition ensures that both delete and reset tokens are discarded in case the Type-4 transitions do not fire.

6.8.5 HLCPN CONSTRAINT Operator

The internal structure of the HLCPN CONSTRAINT operator from Figure 6.3 is depicted in Figure 6.17. The operand of the CONSTRAINT operator is included through a hierarchical place. As soon as a token propagates out of this place, it is ensured that all tokens of the same color are deleted from the operand evaluation in order to avoid duplicating the same evaluation. When a token arrives at place *Check*, the constraint expression is evaluated. If it is not fulfilled, the token is sent back to the hierarchical place in order to restart the evaluation of the operand for that token. In case the condition does not fail, the token propagates to the transition *ResultPort*. Since no token can reside for several event occurrences in any of the places except for the hierarchical place, delete tokens are sent there directly.

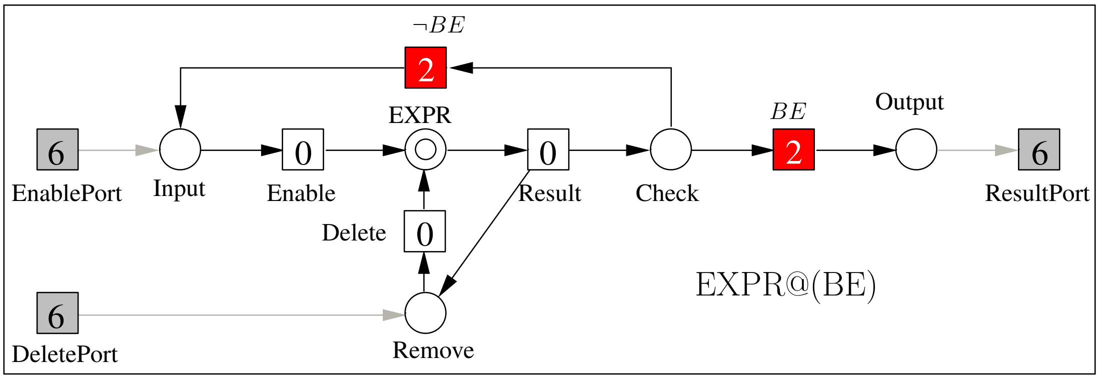


Figure 6.17: HLCPN CONSTRAINT Operator

6.8.6 HLCPN ACCUMULATOR Operator

The internal structure of the HLCPN ACCUMULATOR operator from Figure 6.3 is depicted in Figure 6.18. Table 6.9 lists the according methods and functions referred to in Figure 6.18.

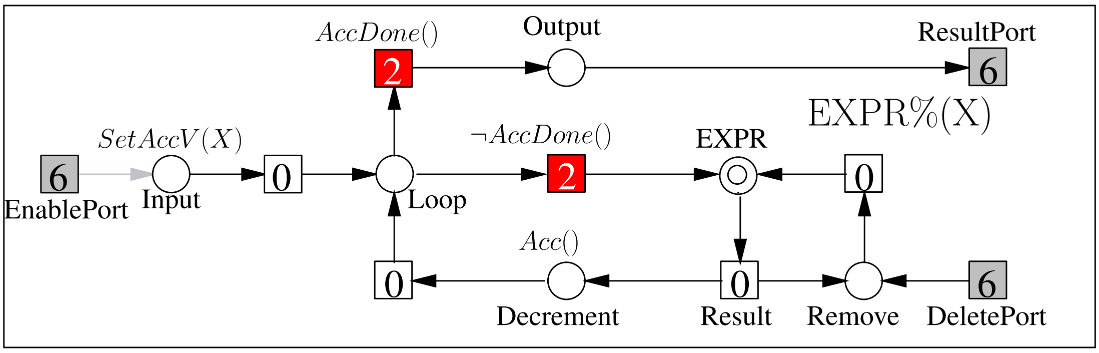


Figure 6.18: HLCPN ACCUMULATOR Operator

Symbol	**Definition**
SetAccValue(X)	$TK'.ACC\_LST[i] = X$
Acc()	$TK'.ACC\_LST[i] = TK.ACC\_LST[i] - 1$
AccDone()	$TK.ACC\_LST[i] = 0$

Table 6.9: HLCPN ACCUMULATOR: Methods and Functions

It is assumed that every HLCPN ACCUMULATOR operator is associated with a unique index i in the *ACC_LST* item of a token.

The HLCPN ACCUMULATOR operator is fulfilled for a token if that token has propagated through the hierarchical place *EXPR* for as many times as indicated by the accumulation value (X in Figure 6.18), with which the operator is parameterized. Since the accumulation value in turn can be the result of an expression, it needs to be evaluated once for every token arriving at this component. The expression is evaluated once and its result is stored to the *ACC_LST* item in the corresponding token. This is accomplished in the place *Input* by the method *SetAccValue(X)* listed in Table 6.9.

Tokens proceed to the hierarchical place if the accumulation value is not equal zero. Hence, if the accumulation value for a token is equal to zero the evaluation for that particular token is done. The corresponding comparison is evaluated with function $AccDone()$ listed in Table 6.9. A token propagating out of the hierarchical place means that the operand evaluation has been fulfilled for that token once. Therefore, its accumulation value needs to be decremented by one. This is accomplished in place *Decrement* by method *Acc()*. A copy of the token is also sent back to the hierarchical place as a delete token in order to remove any token of the same color. By doing so, duplication of tokens of the same color is avoided. From place *Decrement*, a token is looped back into the hierarchical place until its accumulation value equals to zero. Delete tokens from the transition *DeletePort* are sent directly to the hierarchical place.

7 UAL Application Framework

This chapter describes the key components of the UAL application framework. After providing a general overview, an auxiliary language for specifying how UAL monitors are bound to a DUV is introduced. Following that, a further language is introduced which allows testing UAL assertions prior to applying them with a DUV. Afterwards, the key concepts of the compiler-based UAL implementation are explained.

7.1 Overview

The heart of the framework consists of a UAL base library implemented in SystemC and C++, and a UAL compiler. The base library provides an implementation of all UAL operators which are introduced in Chapter 5 and formally defined in Chapter 6. Among other things, it also includes an event handler which controls the triggering of assertions based on the general event concept of UAL, a proxy interface class which provides means to access the event handler for issuing callback events, and a tracer which creates an event based waveform for later debugging.

The compiler generates the whole assertion infrastructure in SystemC. This includes code that instantiates and links library elements in order to perform checks as specified in the assertion language. Generally, the entities of a UAL description (i.e., monitors, properties, and sequences) are represented as SystemC modules in the generated code.

The UAL framework offers an additional binding language (Sec. 7.2) for specifying how monitors are mapped to a DUV. The compiler interprets such a binding specification and generates a corresponding SystemC module. This module has to be instantiated within the DUV in order to finalize the binding of the monitors. The mechanisms applied in such a module are explained in Section 7.5.

The UAL framework also offers a language for specifying assertion tests (Sec. 7.3). This allows testing of assertions prior to their application with a DUV and hence, offers means for assertion quality assurance. Based on a selftest specification, the compiler generates a whole automated test framework.

Figure 7.1 shows the general structure of the UAL application framework and indicates the work flow.

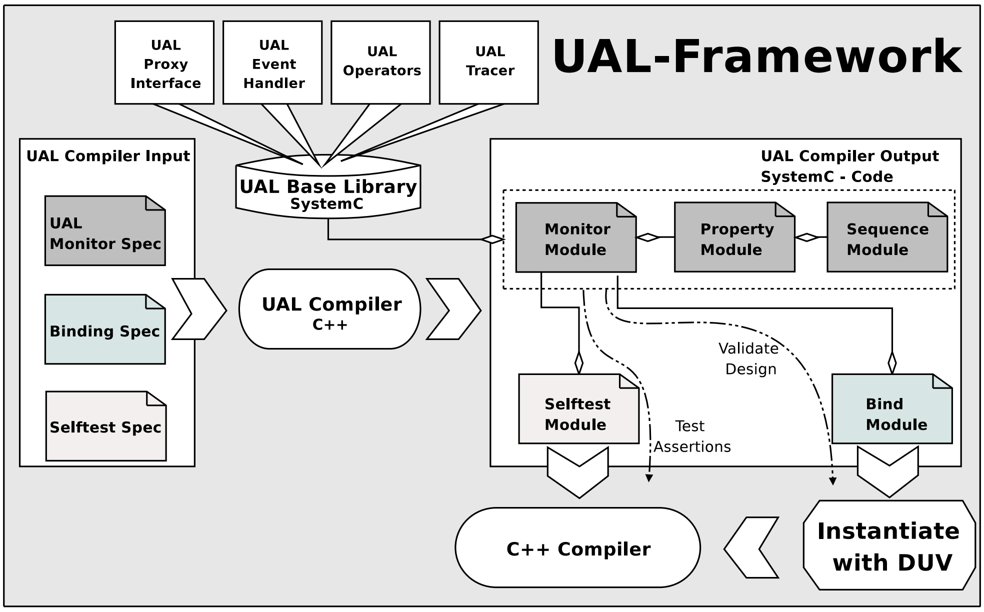


Figure 7.1: UAL Application Framework Overview

An arc with a diamond at the end denotes an aggregation. The module attached to the diamond contains at least one instance of the module connected to the other end of the arc.

As Figure 7.1 shows, two work flows exist in the framework - testing assertions and validating a design. Both flows require a UAL monitor specification. For testing assertions, a corresponding selftest specification is required. For validating a design, a binding specification is required. In both work flows the UAL compiler is used for generating the according SystemC implementation. When testing assertions, the generated selftest module including the monitor implementation is fed into a C++ compiler to generate an executable. For validating a design, it is also necessary to instantiate the generated bind module, also including the monitor implementation, in the DUV prior to starting the C++ compiler.

7.2 Binding Language

The UAL framework provides a language for binding UAL assertions to a design while preserving the same modeling abstraction of a UAL monitor interface. The binding language was developed to ease the integration of assertions in a DUV, since the manual mapping of the SystemC monitor interface to the corresponding targets would require a non-feasible effort and on top of that, would pose a high probability for error.

The UAL compiler generates a SystemC module out of the binding specification. This module has to be instantiated in the DUV as indicated in Figure 7.1. No further SystemC coding has to be done by the user to accomplish the connection (see R 4, p. 197). The generated file is self-contained and handles the connection automatically.

A binding specification contains the mapping of monitors to a design (i.e., the connection of the monitor ports to the corresponding parts in a design). The binding works on instances as well as classes[1]. The following paragraphs introduce the binding concept of UAL. The complete grammar can be found in the Appendix in Section B.2.

The declaration of a binding specification is defined as follows:

```
bind_definition    =  "bind" identifier                                    B.68,
                      targets_section                                      p.207
                      mappings_section
                      "endbind" ;
```

A binding specification contains two sections - *targets* and *mappings* - to describe all necessary information.

7.2.1 Targets Section

The *targets* section is used for declaring scopes which contain the actual objects to which a monitor can be bound. Furthermore, it contains declarations of the monitors to be bound. The targets section can be specified according to the following rule:

```
targets_section    =  "targets" [ "(" "class" identifier ")" ]             B.69,
                      target_declaration { target_declaration }            p.207
                      "endtargets" ;
```

With the declaration of the targets section, it is possible to specify whether the whole binding has to be done to a specific class or to instances in general. The former means that the bind module generated by the UAL compiler has to be instantiated in the specified class. Thus, anytime an object of this class is created, new instances of the monitors bound to it are created. Binding to class is indicated by the syntax option in the first line. If nothing is specified here, binding to instance is in effect. This means, that monitors are bound only to specific objects of a class rather than all objects of it. The generated bind module has to be instantiated in the *sc_main* routine which instantiates the DUV. The bind module has to be instantiated after the DUV. Binding to class and binding to instance are mutually exclusive concepts and therefore, may not be combined within a single binding specification.

[1] This concept is comparable to the binding features of SVA [25]

A target declaration has the following form:

target_declaration = *monitor_target* B.70,
| *design_target* ; p.208

A target declaration can either be a monitor target or a design target. A monitor target is defined as follows:

monitor_target = `"monitor"` *identifier* `"="` *identifier* `";"` ; B.71,
p.208

The LHS identifier of the assignment represents the local name of the monitor referenced by the RHS identifier. The declaration of a monitor target represents an instantiation of the corresponding monitor in the generated bind module. The declaration syntax allows, that several instances of a monitor can be bound within the binding specification.

A design target is defined as follows:

design_target = `"module"` *identifier* B.72,
`"="` *identifier* { `"."` *identifier* } p.208
`"("` *identifier* [template] `","` `"""` *filename* `"""` `")"`
`";"` ;

The LHS identifier of the assignment again represents the local name of the target. The RHS expression represents the instantiation path of the module which contains the member objects to which a monitor is to be bound later on in the mappings section. The local name of a design target can be compared to an alias of the path information. Following the path specification, the class name of the corresponding module shall be specified, as well as the header file name which contains the class declaration of the module. In case the target module is a class template, it is possible to give a corresponding template specification with the class name.

If binding to class is used, the specified instantiation path is considered relative to the location of an instance of the generated bind module. The local names of targets shall be unique.

The following example shall illustrate the specification of both a design and a monitor target. The example assumes the existence of a class *fifo<typename T>* located in a header file *"fifo.h"*, and the existence of a corresponding monitor *fifo_mon* which is to be bound to an instance *rx_fifo* located within an object named *top*. The instance *rx_fifo* is an object of class *fifo<typename T>* with *int* as the specialization for *T*. Listing 7.1 shows the corresponding target declaration *myMod* which refers to object *rx_fifo*. Since the corresponding class is a template, the correct template specialization has to be given. Furthermore, the corresponding header file *"fifo.h"* has to be specified. The target declaration *myMon* is an instantiation of monitor

```
targets
  module myMod = top.rx_fifo(fifo<int>, "fifo.h");
  monitor myMon = fifo_mon;
endtargets
```

Listing 7.1: Example Target Section

fifo_mon. Since it is possible to include several instances of monitor *fifo_mon*, it is necessary to specify it as a target with a unique identifier which is the local name of the target.

7.2.2 Mappings Section

The *mappings* section specifies the actual connections of the ports of monitor targets to objects located in the design targets. It supports the mapping to transactions, events, signals, variables, and public access functions (see R 8, p.197, R 12, p.198, R 16, p.198, R 18, p.198, R 24, p.198). The declaration of the *mappings* section is defined according to the following rule:

```
mappings_section    =  "mappings"                                          B.74,
                       mapping_declaration { mapping_declaration }         p.208
                       "endmappings" ;
```

A mapping declaration is of the following form:

```
mapping_declaration    =  identifier "." identifier                        B.75,
                          "=>" identifier "." design_object ";" ;          p.208
```

The mapping is obtained through the use of the map operator (=>) and works unidirectional. This means that the monitor has a read-only access to the objects in the targets (see R 27, p.199, R 28, p.199). The LHS operand of the map operator refers to a port (see Rule B.3, p.203) of a monitor target. The LHS identifier of the dot operator is the local name of a monitor target while the RHS identifier is the name of its corresponding port.

The RHS operand of the map operator consist of a reference to a design target identifier and the corresponding member object. Here, the LHS identifier of the dot operator refers to the local name of the corresponding design target while the RHS identifier refers to the corresponding member object (*design_object*).

It is possible to map to any member variable of the target, to public member access functions, and especially to transactions, as indicated by the following rule:

```
design_object        =  transaction_object                               B.76,
                          | array_object                                 p.208
                          | function_object
                          | variable_object  ;
```

Mapping to a transaction is attempted if the corresponding monitor port is of kind *transaction*. The according rules for a member object which is a transaction are defined as follows:

```
transaction_object   =  variable_object parameter_mapping  ;             B.77,
                                                                         p.208

variable_object      =  identifier { "." identifier }  ;                 B.80,
                                                                         p.208

parameter_mapping    =  "(" ( identifier | "RET" ) "=>" identifier       B.81,
                        { "," identifier "=>" identifier } ")"  ;        p.208
```

The rule *variable_object* refers to the identifier of the corresponding member object. As such, it can also consist of an instantiation path. For instance if the member object of the target is a structure it is possible to reference members of the structure using the dot operator. If the member object to map to is either a public member variable, signal, or event, this rule alone shall be used to accomplish the mapping.

If the member object to map to is a private member variable or signal, but a public access function is provided, the rule *function_object* can be applied. The according rule is defined as follows:

```
function_object      =  variable_object "(" ")"  ;                       B.79,
                                                                         p.208
```

If the member object to map to is an array the rule *array_object* applies, which is defined as follows:

```
array_object         =  variable_object "[" number "]"  ;                B.78,
                                                                         p.208
```

The following example shall illustrate how these rules are applied in particular. It is assumed that the monitor from the previous example in Listing 7.1 contains a transaction port named *mPUT* with an integer argument named *mPUT_data* and two state ports named *mfifo_value*, which is an array, and *mfifo_index*. Furthermore, it is assumed that object *rx_fifo* includes a transaction named *PUT* with an integer argument named *data*, an array member variable named *fifo_value*, and a private member variable named *fifo_index* with a public access function named *get_fifo_index* which returns the value of *fifo_index*.

The corresponding mappings can be specified as shown in Listing 7.2.

```
mappings
  myMon.mPUT          =>  myMod.PUT(mPUT_data => data);
  myMon.mfifo_value  =>  myMod.fifo_value[];
  myMon.mfifo_index  =>  myMod.get_fifo_index();
endmappings
```

Listing 7.2: Example Mappings Section

The binding language hence, allows the instrumentation of a design with monitors which contain assertions in an easy fashion by preserving the same abstraction as provided by the UAL modeling layer.

7.3 Selftest Language

Generally, the powerful expressiveness of RTL assertion languages allows a verification engineer to formulate quite complex assertions. However, debugging a design often results in debugging an ill specified assertion. Given that UAL adds more degrees of freedom to this expressiveness, it is clear that a quality assurance methodology for these assertions has to be established.

The UAL application framework hence, provides the possibility to write test-cases for any assertion specification, offering the same level of abstraction an assertion is written at. Hence, users have the possibility to test their assertions before instrumenting the target design with them. This can shorten the overall effort for debugging assertion failures in the context of a DUV by eliminating errors in assertions first, prior to their application.

This section introduces a selftest language for testing the soundness of UAL assertions. The language offers constructs for specifying stimuli and for checking the assertion evaluation results. The formal syntax can be found in the appendix in Section B.3.

Tests are written within the *testbenches* section which is declared according to the following rule:

test_definition	=	"testbenches" *identifier*	B.83,
		testbench_section { *testbench_section* }	p.208
		"endtestbenches" ;	

A testbenches section contains a set of testbenches, each for testing a particular monitor. Within the *testbenches* section, *testbench* sections are declared following this rule:

```
testbench_section      = "testbench" identifier                        B.84,
                         testcase_section { testcase_section }        p.208
                         "endtestbench" ;
```

The identifier of a *testbench* section has to match the name of the UAL monitor to be tested. The identifier hence, represents an instantiation of the corresponding monitor and serves as a reference for accessing the ports of the monitor. Each monitor to be tested requires a *testbench* section declaration of its own. A *testbench* section contains one or more *testcase* sections which perform the actual testing. Each testcase is executed in descending order, as declared within a testbench.

The following rule shows the syntax for declaring testcases:

```
testcase_section       = "testcase" identifier testcase_parameters     B.85,
                         test_stimulus { test_stimulus }              p.209
                         "endtestcase" ;
```

The identifier specifies the name of a testcase and is used for reporting. A *testcase* section is configured through the use of parameters. Within the body of a *testcase* section the stimuli can be specified.

7.3.1 Testcase Parameterization

The parameters of a testcase are specified according to the following rule:

```
testcase_parameters = "(" "loop" "=" number ","                         B.90,
                      "reset" "=" reset_type                           p.209
                      [ "," "trace" "=" ( "ENABLE" | "DISABLE" ) ] ","
                      expect_statement ")" ;
```

The parameter *loop* determines how many times the testcase has to be reiterated. The parameter *reset* specifies whether and how a reset has to be performed upon the start of the testcase. The following reset types are provided:

```
reset_type          = "MONITOR"                                         B.94,
                       | "COVERAGE"                                    p.209
                       | "NONE"
                       | "ALL" ;
```

The value *"MONITOR"* denotes that all assertions in the monitor are reset, in order to ensure that no threads are still running when switching from one testcase to another. The value *"COVERAGE"* denotes that the collected coverage values of all assertions in the monitor are reset. This way, a user does not have to keep track of coverage which was obtained in previously running testcases when formulating expectations on coverage values. The value *"NONE"* denotes that no reset is performed.

The value *"ALL"* denotes that all assertions and all coverage data is reset. Hence, it represents a combination of values *"MONITOR"* and *"COVERAGE"*.

The parameter *trace* is used for enabling or disabling tracing for the testcase.

The parameter *expect* configures the checking by formulating expectations on the coverage results (i.e., success, real and vacuous success, and failure) of specific assertions in the monitor. Note that all verification directives in the monitor are interpreted as *cover*-directives when self-testing is used. This is ensured by the UAL-compiler. An expect statement has the following form (see also Rule B.91):

```
expect_statement    =  "expect" "=" "[" identifier cover_assignment        B.91,
                       { "," identifier cover_assignment } "]"  ;          p.209
```

Within the `expect`-statement it is possible to reference several verification directives by their name and to formulate the expected coverage results of the corresponding property which is associated with the directive. The expected coverage is checked with the collected coverage at the end of the testcase execution. In case of a mismatch, an error log is written. A coverage expectation is defined as follows:

```
cover_assignment    =  "(" cover_type "=" number                           B.92,
                       { "," cover_type "=" number } ")"  ;                p.209
```

The available cover types are the same for UAL coverage directives, which was discussed in Section 5.3.

7.3.2 Stimuli Specification

The test language allows the specification of stimuli at the same level of abstraction an UAL assertion is specified at. The body of a testcase is executed sequentially.

The syntax for stimuli generation is defined as follows:

```
test_stimulus       =  ( assign_stimulus                                   B.86,
                       | event_stimulus                                    p.209
                       | wait_statement  ) ";"  ;

assign_stimulus     =  identifier [ "." identifier ] [ "[" number "]" ] "="B.87,
                       value  ;                                            p.209

event_stimulus      =  identifier [ "[" number "]" ] [ "'" event_kind ] ;  B.88,
                                                                           p.209

wait_statement      =  "wait" "(" number ")"  ;                            B.89,
                                                                           p.209
```

According to these rules, it is possible to directly assign values to ports of kind *state* and *signal* by referencing the corresponding port. Events are emitted by solely referencing the corresponding port of the monitor. It is also possible to emit transaction events. In this case, the same event syntax for referencing a transaction event is used, as within a monitor specification. It is also possible to specify timed and zero-delay `wait` statements in order to enforce the simulation time as a stimulus to the monitor as well.

The following example shall illustrate how a testbench with one testcase is specified. The example is based on the monitor *fifo_mon*, mentioned in the previous examples (see Listings 7.1 and 7.2). Here, it is also assumed that the monitor has a transaction port named *mGET* which has an argument named *mGET_data* and a directive which asserts the correct data flow through the FIFO. Listing 7.3 shows an example testcase which shall check whether directive *A1* works properly for the case that two values are written into the FIFO and afterwards, read from it. With this test the directive

```
testbench fifo_mon
  testcase fifo_df(
     loop=1,reset=ALL,
     expect=[A1(SUCCESS=2,REAL=2,VACUOUS=0,FAILURE=0)])
    mPUT.mPUT_data = 1;
    mPUT'END;
    mPUT.mPUT_data = 2;
    mPUT'END;
    mGET.mGET_data = 1;
    mGET'END;
    mGET.mGET_data = 2;
    mGET'END;
  endtestcase
endtestbench
```

Listing 7.3: Example Testbench Section

A1 is expected to produce two successes which are categorized as real successes and that neither a vacuous success nor a failure is produced.

7.4 UAL Base Library

As mentioned earlier, a key component of the UAL application framework is the base library. This library contains SystemC modules and C++ classes which implement all UAL-operators as well as the general event handling, tracing, and runtime API. The following sections describe the key concepts of the implementation.

7.4.1 Token Network

The implementation of an assertion is strongly related with the formal petri net model described in Chapter 6. Each petri net block is implemented as either a SystemC module or C++ class. Hence, the base library is structured according to the layer concept of UAL. The whole petri network that represents one assertion is built by generating sequence and property modules which instantiate the library operators. The generated code implements the arcs of the petri net. A structural overview of an assertion is shown in Figure 7.2.

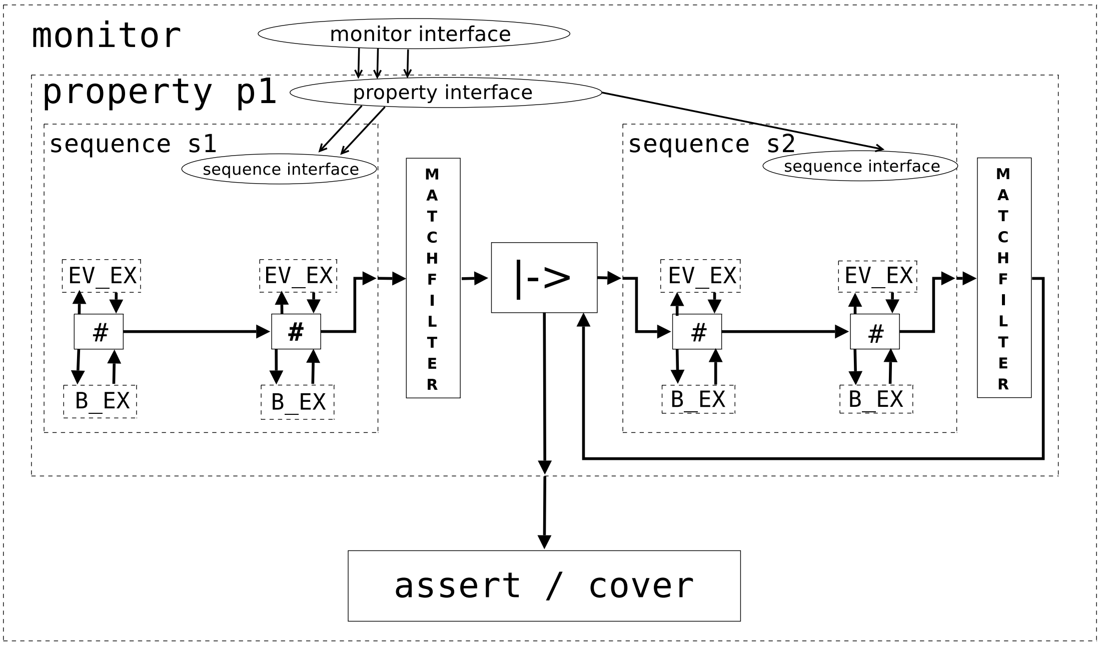


Figure 7.2: Implementation Structure

The boxes surrounded by a straight line represent elements provided by the library. The boxes surrounded by a dashed line represent assertion dependent code which is generated. The Boolean layer expressions, as well as event layer expressions are generated as well. The solid arrows indicate the direction of the token propagation.

When generating property and sequence blocks it is also necessary to implement their interfaces as well as the mapping. Furthermore, the library elements are templated and therefore need to be configured when used. For instance, the left-most delay operator in Figure 7.2 is configured such that it also generates a token each time a new thread has to be created.

7.4.2 Event Handling

This section describes how the general event system of UAL is implemented. The implementation of UAL monitors does not distinguish between event types. Events are implemented as callbacks to the UAL Event Handler, which is the central unit of the UAL event system. Only one instance of the event handler can exist in a simulation. This is accomplished by applying the Singleton pattern [66] for the implementation of the event handler.

Figure 7.3 depicts the UAL event propagation infrastructure based on the UAL event handler.

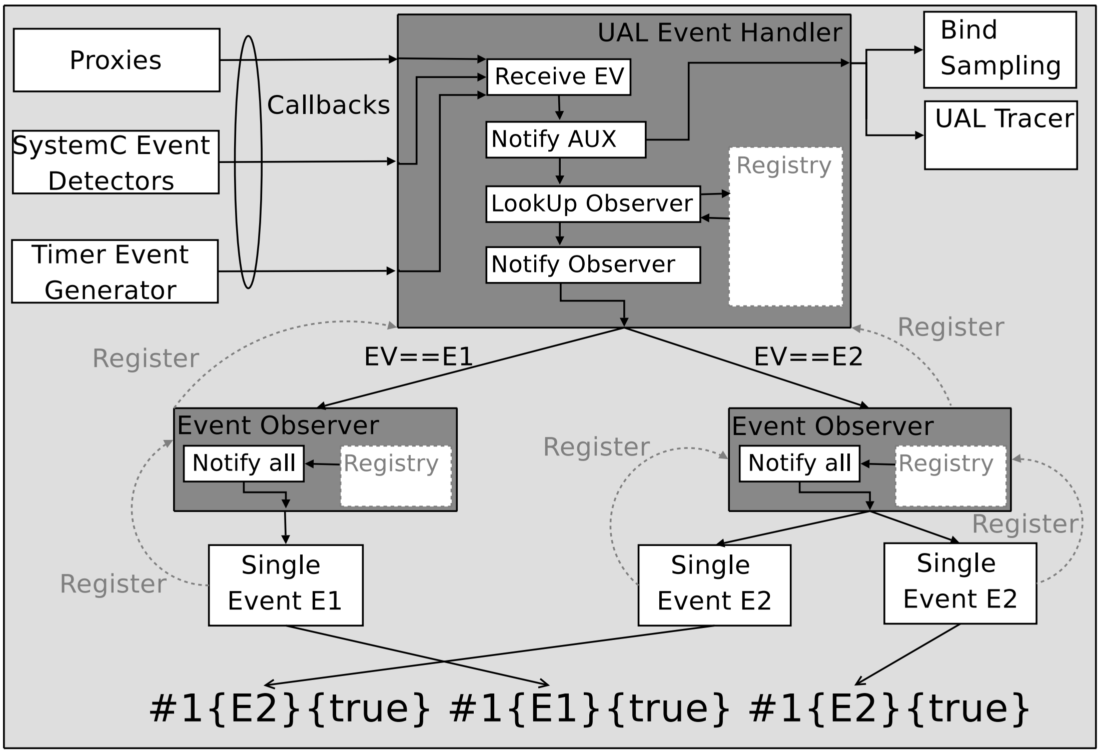


Figure 7.3: Event Propagation Infrastructure

UAL transaction events and value-change events of SystemC variables are implemented as callbacks which are invoked by proxy modules. Proxy modules have to be modeled by the user. A detailed explanation of its general structure is given in Section 7.4.3. Signal value-change events as well as annotated SystemC events which are issued by the DUV are translated to callbacks by so called SystemC event detectors. These are provided in the UAL base library. A SystemC event detector holds a pointer to the corresponding signal or event instance and implements a `SC_METHOD` process which is sensitive to the event. Every time the process is called by the SystemC simulation kernel, a callback to the UAL event handler is issued. The UAL base library also offers a Singleton timer event generator which corresponds to the special timer event object in Definition 2 (Sec. 6.1.2, p.97). The implementation of the TIMER operator from the event layer requests a timer event from this generator. The generator calculates the necessary target time values for each request and merges all requests which have the same target simulation time. Once, the target simulation time has been reached a timer event callback is invoked in the event handler.

Any callback carries a unique identifier for the event it represents. These identifiers are generated by the UAL compiler.

Upon receiving a callback the event handler first activates the *Bind Sampling* which calls all public access functions to which monitors are bound. This is explained in Section 7.5. After the activation of the *Bind Sampling*, the event handler notifies the UAL *Tracer*.

The counter part to the event handler is a so called event observer. Each sequence implementation contains event observers - one for each distinct event. An event observer is parameterized with the unique identifier of the event to be observed. On its construction, the event observer registers itself at the event handler for that particular event. The registry in the event handler mainly is a map of event identifiers to lists of references to the corresponding event observer instances. An event observer in turn contains a registry of references to the single event operators which need to be notified with the event an observer has registered for.

After the notification of the UAL *Tracer* and the *Bind Sampling* the event handler performs a lookup in its event observer registry in order to determine which event observers have to be notified. Once an event observer is notified, it propagates the notification to the associated single event operators, as already indicated by Figure 7.2. The order in which event observers are notified corresponds to the order the particular event observers have registered with the event handler. The same holds for the single event operators registered with an event observer.

The notification of a particular single event operator leads to an evaluation of the event expression where this operator is used. If a trigger is calculated the delay operator is triggered. The notification of the single event operator returns to the event observer as soon as the event has been processed completely by the underlying elements. The notification returns from an event observer to the event handler after all corresponding single event operators have been notified. The notification returns from the event handler to the source of the notification after all corresponding event observers have been notified. In this case the simulation of the design proceeds.

As mentioned in the paragraphs above, the order of registration determines the order in which event observers as well as single event operators are notified. The structure imposed by the operators of the UAL base library require that the notification of one event proceeds from the right most sequence in a property and within that sequence from the rightmost delay operator backwards. This right to left order has to be obeyed in the event propagation infrastructure. This order is the opposite direction of the token propagation which is from left to right, as explained in Chapter 6. Propagating an event in the opposite order of the token movement ensures that a token may only be triggered once by one particular event occurrence.

7.4.3 Transaction Detection

The event concept of UAL defines transaction events which are to be emitted upon the start and the end of a transaction call. Since transactions are implemented as function calls, it is not possible to monitor transactions from outside without annotating the DUV. However, the UAL framework comes with helper classes which reduce the overall effort to be spent for these additional annotations. Furthermore, the implementation of a transaction detection is most likely to be reusable for standard communication interfaces and thus, rather represents a one-time effort.

The UAL base library provides an interface class which implements the callback interface to the event handler among other helper functions. Access to the event handler can be obtained, by inheriting from this base class. Generally, either proxy modules or in situ annotations can be used to implement transaction detection:

In the first approach, the transaction detection is encapsulated in so called proxy modules. The advantage of this approach is that the functional blocks of a DUV need not be changed. The implementation of proxy modules relies on the Proxy pattern described in [66]. A proxy module can be inserted in between two communicating modules. Thus, a proxy has to implement the same transaction level interface which is used for connecting the two modules. This means that the transaction calls from an initiator are routed through the proxy. Thus, within a proxy, it is possible to intercept a transaction call. However, care should be taken because the proxy module may not change the original behavior of a transaction. Hence, a proxy has to accept a transaction call and pass it on to the real target, unchanged. However, having the proxy in between allows adding the callbacks to the event handler before and after the call to the real target transaction. All transaction arguments can be copied to member variables of the proxy, in order to provide a stable access of a monitor to transaction arguments and return values.

The second approach mentioned above, is more flexible, because it also enables the insertion of callbacks anywhere within functional blocks. Hence, the callback events can be annotated in critical regions of code in order to provide a hook for assertion checking. In situ annotations can also be used to wrap function calls which are not visible from outside the block.

Since the event emission in both approaches is modeled through callbacks which are immediately executed by the event handler (i.e., without introducing delta delays), it is possible to use annotations in the context of both `SC_THREAD` and `SC_METHOD` processes.

7.4.4 Runtime API

The UAL base library also offers a runtime control API which can be instantiated for instance from within a SystemC testbench. The API offers the following self explaining control access functions:

Assertion Control	Coverage Access
`$UAL_reset(UAL_name)`	`$UAL_success(UAL_name)`
`$UAL_disable(UAL_name)`	`$UAL_real_success(UAL_name)`
`$UAL_enable(UAL_name)`	`$UAL_vacuous_success(UAL_name)`
`$UAL_disable_trace()`	`$UAL_failure(UAL_name)`
`$UAL_enable_trace()`	
`$UAL_set_trace_file_name()`	
`$UAL_ignore_severity(severity_level)`	

Table 7.1: Runtime API Functions

The access functions in the left column represent control functions. These functions have a *void* return type. The functions in the right column provide access to the assertion coverage data and thus, return integers.

A `UAL_name` is a hierarchical name which consists of up to three segments: `<bind_instance_name>,<UAL_monitor_name>,<UAL_directive_name>`

The first segment is the SystemC name of the instance of the generated bind module. The SystemC name represents the hierarchical path to this instance starting from the toplevel hierarchy. The second segment is the local name of a target declaration of a UAL monitor from the bind specification. The third segment is the name of the instance of a verification directive within the addressed monitor.

If the `UAL_name` is empty the access functions in the left column of Table 7.1 affect all instantiated monitors. If only the first segment is specified these functions apply to all monitor instances of the addressed bind instance. If also the second segment is specified the corresponding access functions apply to all assertions in the addressed monitor instance within the addressed bind instance. By specifying all three segments, one particular verification directive is addressed to which the function is applied. Note that the full name is obligatory for coverage access functions.

7.5 Binding

This section describes how the binding specification, introduced in Section 7.2, is implemented in the generated SystemC bind module. A bind module instantiates all monitors declared as targets in the bind specification. First, it is described to what interface the different kinds of UAL ports expand to before explaining how the mapping to actual design elements is accomplished.

State Mapping

A monitor port of kind *state* has the most simple representation in the generated monitor interface. The constructor of a monitor is added an argument which is a pointer of the port's type. Hence, when connecting this port, a pointer to the actual target object has to be passed to the monitor constructor. The access to design elements outside the bind module is accomplished through the use of the SystemC `simcontext` function. This function requires the hierarchical SystemC name of the module which has to be accessed. This hierarchical name is specified in the targets section of a bind file.

Event Mapping

Any event in the UAL implementation is resolved through a string. The compiler constructs a unique event identifier out of the instantiation path name of the target which owns the event and the event name itself. This name is passed to the constructor of a monitor which provides a string argument for a port of kind *event*. The monitor in turn passes the value to the underlying structures. This value is used in event observers for the registration with the event handler.

As mentioned earlier, the UAL base library offers an event detector for translating the occurrence of a particular SystemC event to a callback. Such an event detector is instantiated in the bind implementation for any SystemC event. The event detector is provided with the corresponding pointer to the desired event, and its name which is specified in the RHS operand of the mapping operator (`=>`).

Signal Mapping

A port of kind *signal* is implemented as one string argument as identifier for the signal value-change event, and one pointer of the same type as the corresponding port. The event detection of signals is accomplished again using an event detector.

Transaction Mapping

A port of kind *transaction* is implemented with two arguments of type *string* for the corresponding start and end events, as well as one pointer for each transaction argument and one for its return value. The transaction detection is obtained by proxies, as described in Section 7.4.3, which are not part of the bind module.

Private Data Access

In some cases it might be necessary to monitor values of data objects which are declared as private objects in the target. An external access to such objects is prohibited by C++ semantics. However, it is possible that a public access function is provided which returns the value of the object. The mapping operator of a bind specification hence, allows mappings to public access functions as well. However, mapping to a public access function does not yield a permanent connection to the target objects value. Therefore, an instance of a bind module registers with the event handler as well and provides access to a Bind Sampling function (see Figure 7.3). This function is called by the event handler on the occurrence of any event prior to propagating the event to the assertion evaluation engine. This way, the event handler ensures that the ports which are mapped to private objects are updated before starting the assertion evaluation. The access to private data objects hence, results in additional member variables in the bind implementation. Pointers to these variables are passed to a monitor's constructor. The sampling function iterates over all public access functions which are referenced in the mappings section of a bind specification and thus updates the values of the additional data members of the bind specification.

7.6 UAL Compiler

The second basis of the UAL application framework is the UAL compiler. The compiler's main task, is to parse a UAL description and a binding specification and generate the corresponding assertion implementation in SystemC. To accomplish this task the compiler performs syntax checks based on the formal grammar given in Appendix B and creates an internal data structure which is furthermore analyzed to check the semantics of a UAL description. For generating the implementation the compiler interprets the internal data structure and computes all necessary configuration parameters which are required by the elements provided in the UAL base library. Furthermore, the compiler generates SystemC code that represents the whole assertion structure including the mapping of UAL base library elements.

In addition to these tasks the compiler also supports the compilation of selftest specifications. Here, a selftest specification is parsed and checked for syntactical and semantical correctness. The compiler creates a whole automated regression environment for performing selftests.

The compiler is implemented in C++ in order to preserve the possibility to reuse the internal data structures for supporting direct interpretation of UAL assertions specified in SystemC models.

8 Application

This chapter provides an overview of the steps necessary to ramp up ABV based on UAL for a specific design. Furthermore, an example proxy specification is given, to illustrate the structure of proxies. Following that, a detailed application example is given to clarify the monitor writing and binding process. In connection with that, more applications are shown which focus on the special capabilities of UAL. This chapter closes with a general performance analysis, reflecting on simulation runtime impact and code efficiency.

8.1 Application Flow

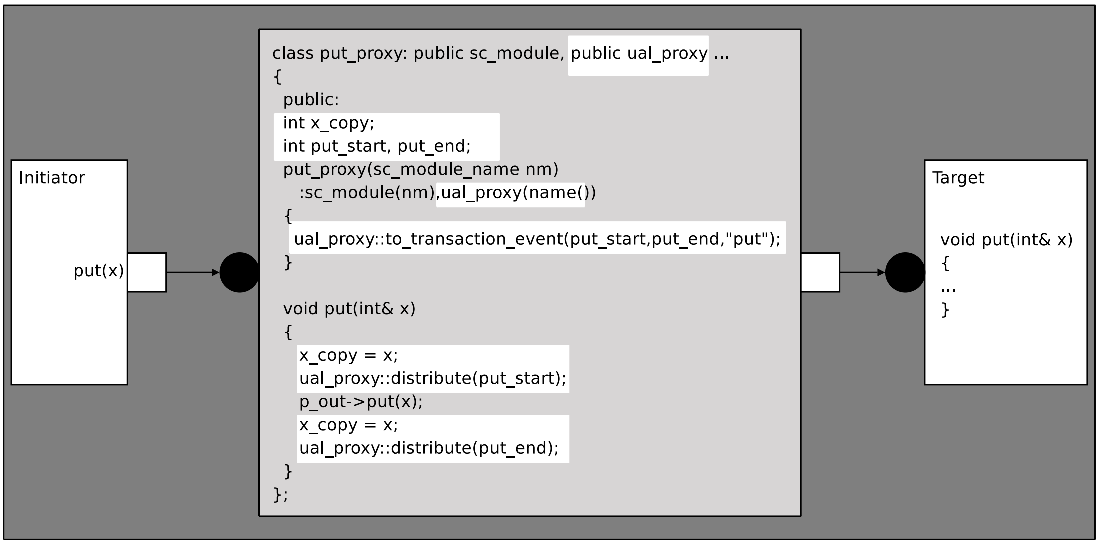
This section summarizes the steps a user has to perform in order to utilize the UAL application framework.

1. Proxy Development
    a) Writing proxy modules for all transaction interfaces of interest
    b) Annotation of the DUV with proxy instances
2. Description of basic UAL monitor
3. Specification of bindings
4. Compilation of UAL monitor and binding
5. Instantiation of generated bind file in the top level and in classes if needed
6. System compilation and simulation

As these steps show, the effort for integrating ABV using the UAL framework is feasible. Writing proxies can be considered a one-time effort because in an industrial setup there exists only a limited set of transaction interfaces. Proxies, that have been written for a transaction interface can easily be re-used.

8.2 Proxy Example

Figure 8.1 shows an example SystemC implementation of a proxy for detecting transactions and emitting the corresponding transaction events.



```
class put_proxy: public sc_module, public ual_proxy ...
{
  public:
  int x_copy;
  int put_start, put_end;
  put_proxy(sc_module_name nm)
     :sc_module(nm),ual_proxy(name())
  {
   ual_proxy::to_transaction_event(put_start,put_end,"put");
  }

  void put(int& x)
  {
     x_copy = x;
     ual_proxy::distribute(put_start);
     p_out->put(x);
     x_copy = x;
     ual_proxy::distribute(put_end);
  }
};
```

```
void put(int& x)
{
...
}
```

Figure 8.1: Transaction Detection Proxy

A proxy module has to be instantiated in between two communicating modules, providing the same functionality but acting as a verification hook as well. A proxy needs to offer the same interface as the target module and has to provide an implementation of the transactions which are defined for that interface. In the example in Figure 8.1, the proxy intercepts calls to the transaction called *put*. When developing a proxy, the following steps have to be done:

- A proxy module needs to inherit from an UAL helper class (*ual_proxy*) in order to obtain access to the UAL event handler.
- A member variable for each transaction argument and return value (not existent in the example) has to be declared (*x_copy*).
- Two member variables of type *int* are required for each transaction in order to represent transaction events. The content of these variables reflects the corresponding event identifiers which are set by the call to function *to_transaction_event()*.
- The SystemC hierarchical name of the proxy module has to be passed to the helper class in the initialization phase at construction (*ual_proxy(name)*).

- Each transaction which shall emit events has to be registered with the UAL event handler using the *to_transaction_event()* function offered by the helper class. The variables which reflect the corresponding transaction events have to be passed to that function as well as the name of the transaction. This transaction name shall be referenced in any bind specification. Through calling this function a unique identification value is computed by the UAL framework for the transaction events. This value is stored in the according variables passed to the registration function.
- The implementation of the intercepted transaction has to contain a statement for copying transaction arguments to the local member variables. This statement has to be placed prior to emitting a transaction event (*x_copy = x;*).
- A transaction event is emitted using the function *distribute* offered by the helper class. The argument passed to it has to be the corresponding member variable which holds the identification value of that event.
- The call to the actual transaction has to be placed in between the emission of the start and the end event of a transaction.

8.3 CPU-Queue Example

Figure 8.2 depicts the application model including the proxy modules used for the transaction detection. The model consists of a queue of sixteen subsystems, each

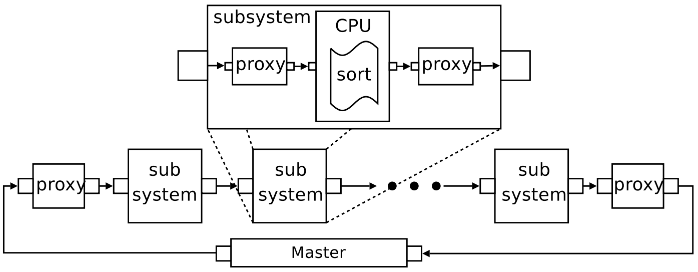


Figure 8.2: CPU Queue

including one CPU and I/O ports for data transfers. The communication between a CPU and its I/O devices is based on blocking transactions for reading and writing to the peripherals. A CPU blocks when the addressed device is not ready for that access. This means that an IN device can only be read by the CPU if its data register contains valid data and an OUT device can only be written if its data register is empty. The output port of a subsystem is connected to the input port of the next subsystem. The input port of the first subsystem is accessed from the outer driving module. The output port of the last subsystem is connected to the outer module's input port.

The software running on the CPUs implements a distributed algorithm for sorting non-zero values.

At first the number of data values to be sorted is read in and then passed on to the next subsystem. This value determines the number of iterations of the implemented loop. Following that, the first sort value is read in and stored within the *R0* register of the CPU. Then the second sort value is read in to register *R1.* After a comparison between *R0* and *R1* the greater of both values is sent to the next subsystem. Then the execution loops back to reading in the next sort value. Once all iterations are done, the subsystem sends out the sorted value.

When the first sorted value propagates to the output of the array, all remaining values are expected to arrive with exactly ten time steps distance at the output.

Further details of the queue system are not relevant for the remainder of this example.

8.3.1 Assertions for the CPU Queue

Many properties have been specified for this system. In the following two properties are highlighted to illustrate basic capabilities of UAL:

- Correct Node Sorting: Within the loop of the sort algorithm in one instance of a subsystem, a value that is read in is propagated to the output if it is greater than the value stored in the CPU's *R0* register and vice versa.

- Correct Transaction Stream: Pushing seventeen values in the array implies seventeen values at the output of the array where the first value pushed in equals the first value at the output. Additionally, the last sixteen values have to have a temporal distance of ten time steps to each other.

The first property is formulated regardless of time, whereas the second property requires further time information. A complete UAL description including a bind specification is described for the first property. For the second property the corresponding sequence and property description is illustrated.

8.3.2 Correct Node Sorting

The correctness of a sort step has to be checked for each instance of the subsystem. Hence, it is necessary to specify a monitor which incorporates the corresponding property. This monitor is bound to each subsystem by using the bind to class concept introduced in Section 7.5.

Monitor and Ports Section

For monitoring the behavior of the sort algorithm it is necessary to specify an interface which provides access to the following elements of a subsystem:

- CPU registers *R0* and *R1*
- CPU I/O transactions *READ* and *WRITE*

The UAL specification of the monitor *sort_mon* and its interface is given in Listing 8.1.

```
 1 monitor sort_mon
 2   ports
 3     state int R0;
 4     state int R1;
 5     transaction void READ(int data);
 6     transaction void WRITE(int data);
 7   endports
 8   sequences
 9   ...
10   endsequences
11   properties
12   ...
13   endproperties
14   verification
15   ...
16   endverification
17 endmonitor
```

Listing 8.1: Monitor for Checking Sort Algorithm: Interface

Lines 1 and 17 in Listing 8.1 show the declaration delimiters for a monitor. Lines 2 to 7 yield the *ports* section. The remaining sections are described later. The *ports* section includes two ports for accessing the state variables which represent the *R0* and *R1* registers of a CPU (lines 3, 4). The *ports* section also includes the declaration of two transaction ports. As can be seen, the notation following after the keyword *transaction* represents a function header notation in C++ style. The ports declared here, can be referenced from anywhere within the other sections of the monitor.

Sequences Section

Sequences are specified in the corresponding *sequences* section of a monitor (between lines 8 and 10 in Listing 8.1). Listing 8.2 contains two sequence declarations.

```
 1 sequence sort_data_in(
 2     ref int L1, ref int L2,
 3     state int CPU_R0, state int CPU_R1,
 4     transaction void CPU_READ(int data)
 5     )
 6   #1{CPU_READ'END@(CPU_R1!=0)}{true , L1=CPU_R1, L2=CPU_R0};
 7 endsequence
 8 sequence sort_data_out(
 9     ref int L1, ref int L2,
10     transaction void CPU_WRITE(int data)
11     )
12   #1{CPU_WRITE'END}{(L1>L2) ? (CPU_WRITE.data == L1)
13                             : (CPU_WRITE.data == L2)};
14 endsequence
```

Listing 8.2: Monitor for Checking Sort Algorithm: Sequences

Sequence *sort_data_in* is meant to be used as an antecedent and sequence *sort_data_out* as a consequent of an implication property which is described in the next section.

Sequence *sort_data_in* matches whenever the CPU fetches data to be sorted. All objects which need to be accessed by a sequence have to be declared in the corresponding argument list of a sequence. Two references to local variables *L1* and *L2* are declared on line 2. These variables are passed in by reference which means that any manipulation of these variables is visible from outside the sequence. On line 3 two arguments of kind *state* are declared. These arguments are used for connecting the sequence to the corresponding ports *R0* and *R1* through a property. The argument of kind *transaction* is meant to provide access to the transaction port *READ*. The declaration of these arguments is similar to the declaration of ports in the *ports* section.

The sequence specification on line 6 states that on every occurrence of an ending read transaction where the *R1* register of the corresponding CPU carries a non-zero value leads to the copying of the register values into the local variables which are passed in by reference. Since, the actual sorting state in the sort algorithm is recognizable by the value of register *R1*, constraining the according transaction event with this condition allows detecting exactly when a CPU fetches data to be sorted.

Sequence *sort_data_out* is only evaluated on a match of sequence *sort_data_in* due to the semantics of an implication (see List. 8.3). The local variables passed in by reference carry the values assigned in sequence *sort_data_in*. The actual mapping of the sequence arguments is done within a property declaration as shown in Listing 8.3. The sequence specification states that on the end of a write transaction issued by a CPU it is expected that the greater of the two values stored in the local variables is transported out via the write transaction. Any violation to this expectation yields an error. If the local sorting in a node is violated sequence *sort_data_out* produces a not-match.

Properties Section

Properties are declared in the *properties* section (between lines 11 and 13 in Listing 8.1). The property which specifies the expected behavior of the sort algorithm is declared according to Listing 8.3.

```
 1 property p_sort_val(
 2     state int CPU_R0, state int CPU_R1,
 3     transaction void CPU_READ(int data),
 4     transaction void CPU_WRITE(int data)
 5     )
 6   int L1, L2;
 7   sort_data_in(L1,L2,CPU_R0,CPU_R1,CPU_READ)
 8   |->
 9   sort_data_out(L1,L2,CPU_WRITE);
10 endproperty
```

Listing 8.3: Monitor for Checking Sort Algorithm: Property

As illustrated in Listing 8.3, the interface of property *p_sort_val* is the aggregation of the argument lists of the sequences described above, excluding the local variables. These are declared on line 6. Within the declaration sequence *sort_data_in* is instantiated as an antecedent of an implication operator on line 7 and 8. The sequence *sort_data_out* is instantiated as the corresponding consequent of the implication operator on line 9. The arguments from the property interface as well as the declared local variables are mapped to the sequence arguments. Through the implication it is ensured that sequence *sort_data_out* is only evaluated on a match of sequence *sort_data_in*.

Verification Section

In order to enable the evaluation of property *prop_sort_val*, it is necessary to specify an association with a verification directive within the *verification* section of a monitor

```
assert_cover a_sort(ERROR, ”Sort failed!”) =
      p_sort_val[AnyMatch,ReportOnRestart](R0,R1,READ,WRITE);
```

Listing 8.4: Monitor for Checking Sort Algorithm: Directive

(between lines 14 and 16 in Listing 8.1). Listing 8.4 shows the declaration of an *assert_cover* directive which is the most general directive provided in UAL.

With the *assert_cover* directive it is achieved that the first violation of property *p_sort_val* leads to a stop of the simulation due to the default setting for severity level *ERROR*. Furthermore, the violation is reported with the specified message ”Sort failed!”. Property *p_sort_val* is parameterized with the antecedent sequence mode *AnyMatch* and the property mode *ReportOnRestart*. The antecedent mode ensures that any match of sequence *sort_data_in* is considered for the evaluation of sequence *sort_data_out*. The property mode *ReportOnRestart* makes the property more strict by expressing that the antecedent sequence is not expected to match twice while the consequent sequence is being evaluated. This checks a constraint of the sort algorithm; a CPU may not fetch two values to be sorted from its *IN* device before sending the first comparison result via its *OUT* device.

Binding

After having specified the complete monitor in UAL, it is necessary to define how the monitor has to be connected to a subsystem. To accomplish this, the bind specification language is used. Listing 8.5 shows the complete binding specification for the described monitor, utilizing the bind to class concept.

The *targets* section from line 2 to 7 declares scopes of all objects which have to be linked together. The keyword *class* on line 2 indicates that a binding to class *subsystem* is specified. All paths specified within the *targets* section are suffixes to the hierarchical path of any instance of class *subsystem*. For example, the target declaration in line 3 indicates that the relative path to the CPU instance from the subsystem is *cpu_i*, which means that the CPU is directly instantiated within the subsystem.

Two proxy modules for detecting the corresponding transactions are part of the subsystem. In order to link to the transactions intercepted by the proxy modules, it is necessary to declare these proxies as targets as well.

Within the *mappings* section in line 8 to 16, the monitor is linked with the targets and their corresponding members. The CPU registers *R0* and *R1* are direct members of class *cpu_mod*. Hence, mapping the corresponding monitor ports to these

```
1  bind sort_mon_bind
2  targets(class subsystem)
3   module cpu = cpu_i(cpu_mod,"cpu_mod.h");
4   module IN_pxy = IN_pxy_i(IN_pxy,"IN_pxy.h");
5   module OUT_pxy = OUT_pxy_i(OUT_pxy,"OUT_pxy.h");
6   monitor s_mon = sort_mon(sort_mon,"sort_mon.ual");
7  endtargets
8  mappings
9  s_mon.R0     =>  cpu.R0;
10 s_mon.R1     =>  cpu.R1;
11 // target transaction signature
12 // void READ(int s_data);
13 // void WRITE(int s_data);
14 s_mon.READ  =>  IN_pxy.READ(data     => s_data);
15 s_mon.WRITE =>  OUT_pxy.WRITE(data   => s_data);
16 endmappings
17 endbind
```

Listing 8.5: Bind to Class Example

registers requires referencing the target *cpu* which represents the CPU and using the '.'-operator for hierarchically accessing the corresponding registers. This is specified in the RHS expressions of the mapping operator (`=>`) in lines 9 and 10. Note that on the LHS expressions of the mapping operator the monitor ports are also accessed using the '.'-operator on the target *s_mon* which represents the monitor.

Lines 14 and 15 show the mapping of the monitor transaction ports to the actual transactions. In the target, transactions are considered as member objects as well. In addition to the regular mapping, it is necessary for transactions to map also the corresponding arguments. Mapping the return value of a transaction is done the same way as an argument mapping. Only the keyword `RET` is used as the LHS expression of the mapping operator.

8.3.3 Correct Transaction Stream

In the example described in this section, it shall suffice to show the property and sequence descriptions only. The correctness of the transaction stream defines that pushing seventeen values in to the queue requires seventeen values to propagate out of the queue, while the first value pushed in has to be the first value to be pushed out and that the sorted data has to propagate out every ten time units. The transaction for pushing values into the system is called *SEND* and the transaction which is called by the queue for pushing data out is called *REC*.

The required behavior can be best formulated using an implication property as shown in Listing 8.6. The antecedent of such an implication has to detect the seventeen occurrences of the transaction *SEND* and the consequent has to detect the seventeen occurrences of the transaction *REC* while performing other checks.

```
1 property stream_prop[FirstMatchPipe,Overlap](
2     transaction void SEND(int data),
3     transaction void REC(int data))
4     int cnt;
5   stream_in(cnt,SEND) |-> stream_out(cnt,REC);
6 endproperty
```

Listing 8.6: Stream Property

The antecedent sequence is shown in Listing 8.7.

```
1 sequence stream_in(
2     ref int cnt,
3     transaction void SEND(int data))
4   #1{SEND'START} {$l_event(SEND'START), cnt=SEND.data}
5   #1{SEND'START} {$l_event(SEND'START)}                 //1
6   ...                                                    //...
7   #1{SEND'START} {$l_event(SEND'START)};                //16
8 endsequence
```

Listing 8.7: Antecedent of Stream Property

The sequence specification from line 4 to 7 shows an abbreviated[1] chain of delay operators which are all sensitive to the occurrence of the start event of transaction *SEND*. Since the first value to be pushed in is also required to flow out of the queue, it is necessary to store the value in a local variable for later comparison in the consequent. The first delay operator is followed by 16 further delay operators. Hence, if the transaction *SEND* starts seventeen times, the sequence matches. Since each occurrence of a starting transaction *SEND* leads to the creation of a new evaluation thread, the sequence would match again with the 18^{th} occurrence of a starting transaction *SEND*. This occurrence however, would represent the start of another sending procedure with new data. Hence, it is necessary that the sequence matches only with consecutive blocks of seventeen transaction occurrences. To achieve this, the mode for the sequence is set to *FirstMatchPipe* in the property in Listing 8.6 on line 1. This is also the reason for expressing the sequence with sixteen delay operators with a single-step configuration and equivalent Boolean propositions instead of using just one delay operator with a multi-step configuration of value sixteen. All threads created while detecting the first occurrence of seventeen starting transactions are forced to a not-match due to consumption attempt conflicts. The attempted

[1] Future versions of the UAL will provide syntax sugaring.

consumptions of these starting transactions are enforced by the use of the function *$l_event* in combination with mode *FirstMatchPipe*.

Listing 8.8 shows the sequence declaration of the consequent sequence.

```
1 sequence stream_out(
2     ref int cnt,
3     transaction void REC(int data))
4   #1{REC'END}{cnt==REC.data}
5   #1{REC'END}{true}
6   #15{REC'END@($delta_t == 10);
7        REC'END@($delta_t < 10),timer(11)
8       }{true};
9 endsequences
```

Listing 8.8: Consequent of Stream Property

This sequence requires only the transaction *REC* in the sensitivity of the delay operators in line 4 to 7. Since, this sequence is only evaluated on matches of the antecedent sequence, the first delay operator sensitive to the end of transaction *REC* is supposed to trigger with the first occurrence of transaction *REC* after the seventeen occurrences of transaction *SEND*. Hence, the Boolean layer expression describes that the data argument of transaction *REC* has to be equal to the first value pushed into the queue. This value is stored in the local variable *cnt* by the antecedent. Furthermore, it is necessary to check that the time in between two occurrences equals exactly ten time units. Through the delay operator in line 5 the sequence synchronizes to the second occurrence of an ending transaction *REC*. By using the time constraint in the positive sensitivity in line 6 it is specified that the delay operator is only triggered with the occurrence of an ending transaction *REC* exactly 10 time units later. In order to detect a violation of the required timing a negative sensitivity is used. Here, the use of the time constraint allows detecting that transaction *REC* occurs too early. The use of the timer expression allows detecting that a transaction *REC* occurs either too late or not at all.

8.4 Transactor

The example in this section applies to a simplified transactor which translates a RTL synchronous read protocol to a transaction call on a TLM. A read transaction on RTL is indicated with the signal *R_EN* being high at the positive edge of a clock signal *CLK*. The data to be read has to be stable at the next clock edge on signal *DATA*. Hence, the whole read transaction at the TL side has to happen in between two edges of the same clock signal. The two sequences shown in Listings 8.10 and

8.11 can be used for describing such a behavior within a UAL implication property. The corresponding property is shown in Listing 8.9.

```
property p_trans(
    signal sc_signal<bool> CLK,
    state sc_signal<bool> R_EN,
    state sc_signal<sc_uint<8> > DATA,
    transaction void READ(sc_uint<8> data))
  read_rtl(CLK,R_EN) |-> read_tl(CLK,DATA,READ);
endproperty
```

Listing 8.9: TL Read-Protocol

Listing 8.10 shows the antecedent sequence for detecting the initiation of a read transaction on the RTL protocol.

```
sequence read_rtl(
    signal sc_signal<bool> CLK,
    state sc_signal<bool> R_EN)
  #1{CLK'POS}{R_EN};
endsequence
```

Listing 8.10: RTL Read-Protocol

The example shows that if the events of a SystemC signal need to be used it is necessary to use the UAL kind specifier *signal* in addition to the type *sc_signal<bool>*.

Listing 8.11 shows the consequent sequence for detecting the transaction level implementation of the read protocol.

```
 1 sequence read_tl(
 2     signal sc_signal<bool> CLK,
 3     state sc_signal<sc_uint<8> > DATA,
 4     transaction void READ(sc_uint<8> data))
 5     sc_uint<8> l_dat;
 6   #1{READ'START;CLK'POS} true
 7   #1{READ'END;CLK'POS}
 8       {true, l_dat=READ.data}
 9   #1{CLK'POS}{DATA==l_dat}
10 endsequence
```

Listing 8.11: TL Read-Protocol

The specification in lines 6 to 9 shows that the sequence matches only if the transaction *READ* starts (line 6) and finishes prior to the occurrence of a positive edge of the signal *CLK* (lines 6, 7). Further on, the payload of transaction *READ* sampled into local variable *l_dat* (line 8) at the end of transaction *READ* has to be equal to the signal value *DATA* at the next occurrence of a positive edge (line 9).

The example shows, that triggering with clock edges is supported in UAL as well, and that this trigger can be combined with for instance triggers that represent transaction events. Hence, it is possible to specify sequences in UAL which can combine RTL and TL behavior. Since this is also the task of transactors, UAL is suitable for specifying assertions on the correctness of these transactors improving the quality of transactor IP and hence, verification.

8.5 IP Integration Verification

Another interesting application for UAL assertions for TL systems is the possibility to perform IP integration checks. This includes checking that the address decoding is implemented correctly or checking that a third party IP with different interfaces is wrapped correctly through adapters.

8.5.1 Address Decoding

For checking correct address decoding, an implication property, as shown in Listing 8.12, can be specified which checks that a transaction initiated by the system master leads to an invocation of the correct transaction at the target. The example system contains a master and two slave IP modules (*IO*, *Uart*).

```
property AddrDecProp(...)
    bool rw;
    unsigned long r_a;
    unsigned long w_a;
  MasterAccess(rw,r_a,w_a,...)
  |->
  MatchAddressing(rw,r_a,w_a,...);
endproperty
```

Listing 8.12: IP-Address Decoding: Property

Listing 8.13 shows the corresponding antecedent sequence which matches always when the master issues a write (*mWrite*) or read (*mRead*) transaction. Along with these calls, the sequence stores information in local variables for later use in the consequent. The example shows, that the combination of an event OR operator and the UAL Boolean layer function `$l_event` allows to store the information of which transaction has been initiated by the master in a local variable (*rw*).

```
sequence MasterAccess(
    ref bool rw, ref unsigned long r_a,
    ref unsigned long w_a, ...)
  #1{mWrite'START | mRead'START}
      {true, rw = $l_event(mRead'START)
           , r_a = mRead.addr, w_a = mWrite.addr};
endsequence
```

Listing 8.13: IP-Address Decoding: Antecedent

Listing 8.14 shows the corresponding consequent sequence. It matches only if the correct transaction on the slave side is called. This is achieved by checking the local variable and the address range of the corresponding slave module. The sequence consists of one delay operator which is triggered by any of the slave transactions. Furthermore, by using the events of the master transactions, the evaluation can be forced to a not-match because either of the slave transactions has to start prior to another invocation of a master transaction. In this sequence, the combination of the event OR operator with function `$l_event` allows the formulation of checks depending on events triggering the delay operator.

```
sequence MatchAddressing(
    ref bool rw, ref unsigned long r_a,
    ref unsigned long w_a, ...)
  #1{ IORead'START    | IOWrite'START |
      UartRead'START | UartWrite'START;
      mWrite'START | mRead'START}
  {
  ($l_event(IORead'START) && rw && (r_a>=0x24 && r_a<0x34))
  ||
  ($l_event(IOWrite'START) && !rw && (w_a>=0x24 && w_a<0x34))
  ||
  ($l_event(UartRead'START) &&  rw && (r_a>=0x0 && r_a<0x24))
  ||
  ($l_event(UartWrite'START) && !rw && (w_a>=0x0 && w_a<0x24))
  };
endsequence
```

Listing 8.14: IP-Address Decoding:Consequent

8.5.2 Correct Wrapping

The *UART* slave device is a third party IP which ships with a different TL interface then the rest of the system. Hence, for integrating this component, it is necessary to develop wrappers or adapters which bridge the different interfaces. This includes pure name mapping of transactions as well as mapping of payload types. UAL can be used for checking the correctness of a transaction call passed through the adapter. An implication property, as shown in Listing 8.15, is used for checking the correct wrapping of a write transaction.

```
property CorrectWrapping(...)
      MasterWriteUartAccess(...)
      |->
      UartWriteAccessViaAdpt(...);
endproperty
```

Listing 8.15: Correct Wrapping: Property

Listing 8.16 shows the specification of the antecedent sequence. This sequence matches if a master transaction is initiated which addresses the third party IP.

```
sequence MasterWriteUartAccess(...)
  #1{mWrite'START@((mWrite.addr>=0x0) && (mWrite.addr<0x24))}
     true;
endsequence
```

Listing 8.16: Correct Wrapping: Antecedent

This example shows, that the event CONSTRAINT operator allows considering only transactions which address the third party IP. The evaluation is only triggered by those start events of transaction *mWrite* where the address equals to the address of the IP.

```
sequence UartWriteAccessViaAdpt(...)
      #1{UartAdptWrite'START;mWrite'END} true
      #1{UartWrite'START;mWrite'END} true
      #1{mWrite'END} true;
endsequence
```

Listing 8.17: Correct Wrapping: Consequent

Listing 8.17 shows the consequent sequence which is triggered by the corresponding events. When a write transaction is initiated by the master module, it is supposed to start the write transaction in the adapter which in turn has to invoke the transaction in the IP. The transaction call chain has to reach the IP before the write transaction of the master completes. The sequence matches along with the corresponding transaction events. Hence, even without specifying any checks in the Boolean layer expressions, UAL sequences allow matching with sequences of event occurrences. In RTL assertion languages the trigger expressions are means to an end for checking Boolean propositions. In this example, the Boolean layer expressions are omitted. However, if also payload transformations are to be checked this can be done along with the matching of the call chain.

This example, though simple, shows also that considering both start and end of transactions allows checking more concise relations of transactions, such as inclusions. Properties, similar in structure to this example have been successfully applied within the European funded project SPRINT [20]. Here, a TL IP model from the company ST Microelectronics has been integrated via wrappers into an SoC platform which is modeled using the in-house TL interface standard from Infineon Technologies. UAL assertions were used for verifying the correct integration of the foreign IP.

8.6 Control and Data Flow Verification

In this section, both a UAL example for control flow and data flow checking is presented.

8.6.1 Control Flow Checking

In order to avoid polling devices by a master for managing the communication control, it is possible to defer the control to the receiving device by utilizing interrupt based synchronization. On TL, interrupts can be modeled either as transactions themselves, by events, or by usual signals. Here, the latter case is considered. Since UAL allows the triggering of assertion evaluations on the basis of regular signal events, it is possible to specify interrupt sequences and thus properties which describe the intent of an interrupt based communication protocol.

One common interrupt based protocol is that a device once, when it has been configured by a master to start an action, can signal how much data it is ready to receive from the master. In the example here, data burst requests are considered. A master writes a configuration value to one register of a device and indicates how many data packets in terms of bytes are ready to be sent. Since, the device can not process all packets at once, the data transfer is organized in data bursts with a burst size equal to 2 words and hence, 8 bytes. In order to allow that the master does not have to keep track of how much data has already been processed by the device, it relies on the interrupts issued by the device.

The device issues burst requests using the interrupt *BURSTReq*. However, the device has to notify the master when it is ready to receive the last burst of the whole transmission. This is signaled by the device through the interrupt *LBURSTReq*. The number of interrupt requests strongly depends on the packet size which always has to be a multiple of the burst size. Sequences are used for formulating an implication property which states that a transmission initiated by a master requires the correct generation of burst request interrupts. Listing 8.18 shows both sequences and the corresponding property.

```
 1 sequence burst_ante (...)
 2      #1{WRITE'START} true;
 3 endsequence
 4
 5 sequence burst_conseq (...)
 6      sc_uint<32> L1;
 7      #0{}{(WRITE.data % 8 == 0), L1 = WRITE.data}
 8      #1{BURSTReq'POS %(L1-1) ; LBURSTReq'POS} true
 9      #1{LBURSTReq'POS;BURSTReq'POS} true
10 endsequence
11
12 property burst_prop (...)
13      burst_ante(...)
14      |->
15      burst_conseq(...);
16 endproperty
```

Listing 8.18: Control-Flow

Within the consequent sequence (line 5 -10), it is checked that the master always indicates a correct packet size, which has to be a multiple of the burst size. By using the zero-delay operator in the consequent, it is accomplished that the property which is built on top of these sequences fails if the value does not meet the requirements. An event ACCUMULATOR operator (`..'POS %(L1-1)`) is used in the second delay operator in line 8. The accumulator expression results to the number of bursts derived by the packet size which have to be issued by the interrupt *BURSTReq*. Note that if the packet size yields only one burst no occurrence of interrupt *BURSTReq* is expected. The ACCUMULATOR operator hence, allows to consider all bursts which are not the last as one single abstract trigger. This shows that the data dependent temporal behavior allowed by the interrupt based protocol is not considered in the sequence layer and moved to the event layer. Thus, a static structure of the overall sequence can be preserved. Temporal delay values in common assertion languages are static values computable at compile time. Hence, the specification of data dependent behavior has to be tediously broken down to smaller specifications according to the divide and conquer principle. This sometimes involves having to formulate all possible scenarios. UAL though also requiring static values for delay operators, offers the specification of dynamic temporal behavior through the event layer, keeping an assertion description closer to the abstract functionality of a model.

8.6.2 Data Flow

FIFO components are very commonly used for decoupling sender from receiver. Not only FIFOs are used within TL modeling as communication channels to manage synchronization. FIFOs are also used in HW for modeling the communication of two blocks which operate at different speeds. Also computation is done in a pipelined way which is from the data flow point of view the same as a FIFO, except that values which are pushed in are also modified according to some algorithm and that input and output values may be of different types.

In this section, a property for a FIFO module is described which checks the correct data flow through the FIFO. The module does not provide access to its current fill stage. Furthermore, the FIFO allows a word aligned access for writing and a byte aligned access for reading data. The capacity of the FIFO in terms of bytes amounts to 128. In the following the antecedent and consequent sequences and the property on top are shown which together formulate the desired behavior that data written to the FIFO flows out of the FIFO in a guaranteed amount of ”time” measured in numbers of fetch accesses, and that the FIFO is order preserving in terms of bytes and thus, words as well.

Listing 8.19 shows the antecedent sequence which matches whenever a word is written to the FIFO. Also the word is stored in the local variable *l_dat*.

```
sequence write_access(...)
  #1{WRITE'END} {true, l_dat = WRITE.data};
endsequence
```

Listing 8.19: FIFO Data Flow: Antecedent

The consequent is shown in Listing 8.20. The maximum duration of the least significant byte of a written word in the FIFO amounts to 125 fetch accesses. This means, after writing a word into the FIFO, its least significant byte has to propagate with the 125th fetch access at latest. Since the duration can also be shorter depending on the filling stage of the FIFO at the write access, it is necessary to specify a delay range in terms of fetch accesses. Once, the least significant byte flows out of the FIFO, it is expected that the remaining bytes flow out with the next three fetch accesses.

```
sequence fetch_access(...)
  #{1:125}{READ'END}{READ.data==(l_dat & 0x000000FF)}
  #1{READ'END} {READ.data==((l_dat & 0x0000FF00) >> 8)}
  #1{READ'END} {READ.data==((l_dat & 0x00FF0000) >> 16)}
  #1{READ'END} {READ.data==((l_dat & 0xFF000000) >> 24)};
endsequence
```

Listing 8.20: FIFO Data Flow: Consequent

Listing 8.21 shows the corresponding property.

```
property p_FIFO_df[AnyMatch,PipeOrdered](...)
     sc_uint<32> l_dat;
     write_access(... ,l_dat)
     |->
     fetch_access(... ,l_dat);
endproperty
```

Listing 8.21: FIFO Data Flow: Property

As can be seen, the property mode is set to *PipeOrdered*. This mode ensures that it is checked that the FIFO preserves the order of written words and that the pipeline a FIFO represents is recognized in the property evaluation. Hence, even if two equal words are written to the FIFO consecutively the property mode ensures an unambiguous evaluation.

8.7 Performance Analysis

8.7.1 Runtime Performance

For obtaining an impression of the overall runtime performance impact of UAL assertions in comparison to a simulation without any assertions several design applications are analyzed. The following application designs are used:

- PV FFT Device: seven computation nodes perform a distributed FFT algorithm
- PVT AMBA-AHB Peripheral Synchronization Device: This device synchronizes peripherals with an AMBA-AHB bus and buffers the bidirectional data transfer between a master and a peripheral
- PVT AMBA-APB Timer Device: Configurable timer device with 32 programmable timer interrupts

- PV Switch Device: Programmable switch providing serial and parallel channel switching
- PV - RTL Mixed Switch Device: Same functionality as PV Switch device; some TL blocks are replaced with a corresponding RTL model.
- PVT CPU Queue running Sort Algorithm: Distributed sort algorithm (see Sec. 8.3)
- RTL Processor Model: RTL model of the CPU device from the CPU Queue example

Figure 8.3 shows a diagram which yields the factors of the runtime performance impact on each application. These results are obtained through an automated regression-suite, which repeats each simulation for several times and calculates the average values to reduce the influence of statistical variances caused by task switch jitters of the host system.

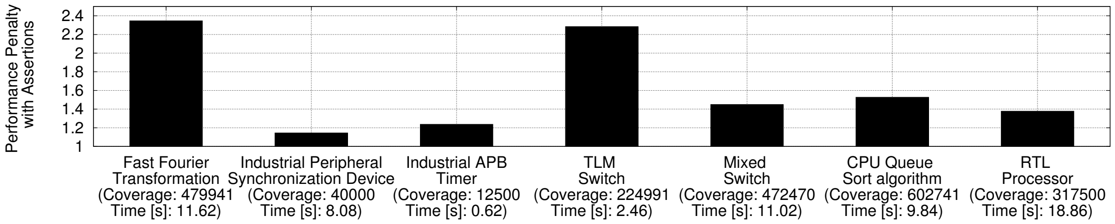


Figure 8.3: Performance Results

The coverage values shown below the performance penalty bars in Figure 8.3 are accumulated coverage results of every assertion in the system. Hence, the coverage can be considered as indicator for assertion activity. The time values listed in Figure 8.3 yield the mean values of simulation runtime of each application design running without assertions.

Generally, the measured activity of assertions is correlated with the overall impact. The performance impact increases as the number of exercised assertions increases. This is obvious, because the more assertions are exercised the more has to be computed by a simulator. All applications were also executed just with UAL callbacks and no assertions in order to measure the impact of callbacks alone. Here, the runtime increase is negligible for all examples. This means that the depicted impact in Figure 8.3 is caused by assertions reacting on events issued from a design. The performance of a model scales with abstraction, whereas the performance of the assertions is quite stable due to a similar structure of the implementation across different abstraction levels. Hence, for instance the impact of assertions with the RTL CPU application design is little despite the fact that the accumulated coverage results yield a high assertion activity.

Two peaks are visible in Figure 8.3 where the simulation with assertions is about twice the time compared to a simulation without assertions. The reason for these peaks is that the assertions used for these application designs are mostly run in pipelined mode, which involves more memory allocation / deallocation due to the increased amount of memory required for keeping track of the pipeline history.

Nevertheless, the slowdown in simulation caused by UAL assertions can be considered feasible when taking into account the methodological benefits of applying ABV as means for verification.

8.7.2 Lines of Code Analysis

As mentioned in Chapter 7, the UAL framework includes a compiler for generating the SystemC implementation of assertions specified in UAL and the required binding.

Table 8.1 shows a comparison of lines of code of UAL descriptions of assertions and binding compared to the lines of code which are generated after the compilation step for each application design. The lines of code of the UAL base library are not included.

Model / Lines of Code	**Monitor**	**Bind**	**Generated**	**Generation Factor**
	File	**File**	**Files**	(monitor+bind)/generated
Fast Fourier Transformation	198	34	4742	0.05
Periph. Synchronization Device	270	40	5102	0.06
Industrial APB Timer	96	12	1976	0.05
TLM Switch	190	32	3490	0.06
Mixed Level Switch	170	48	2303	0.09
CPU Queue Sort Algorithm	130	109	2005	0.12
RTL Processor	187	27	2344	0.09

Table 8.1: Lines of Code Comparison

The lines of generated code represent what would have to be created by a user manually if no language-compiler approach existed. Besides the tremendous effort this would pose for writing the code manually, it also has to be considered that the setup of all UAL base library components with correct parameter settings would have to be done by the user manually as well. Furthermore, an imperative assertion approach is not well suited for documenting a design with assertions, or to communicate design intent within teams. The ratio between UAL descriptions and the code generated out of it shows that the language-compiler based approach offers a high degree of abstraction for the assertion development.

8.7.3 Compiletime Performance

Table 8.2 shows the impact of the implementation of UAL assertions on the overall compile time of design in order to create an executable for simulation.

Model	**With assertions [s]**	**Without assertions [s]**	**Penalty Factor**
Fast Fourier Transformation	180.1	49.7	3.6
Industrial APB Timer	54.1	4.1	13.3
TLM Switch	102.9	8.2	12.6
Mixed Level Switch	102.5	46.1	2.2
CPU Queue Sort Algorithm	97.0	46.2	2.1
RTL Processor	148.8	87.1	1.7

Table 8.2: SystemC Compilation Time Comparison

This impact depends on the complexity of the design and the assertions generated by the UAL compiler. The two high penalty factors visible in Table 8.2 are caused by the fact that the complexity of the generated assertion implementation files is much higher than the complexity of the design model itself. The complexity of the assertion implementation also increases, since the modules offered by the UAL base library can not be pre-compiled due to a high degree of templating. Hence, most operators need to be compiled as well. However, the template approach for the UAL base library was chosen in order to keep the operators as generic as possible. In the development phase, this approach is more suitable, because changes can be applied quicker. For an industrial application of the UAL framework hence, it is possible to apply various optimizations in the assertion implementation by making the library less generic and offering more modules as statically or dynamically linkable libraries or making use of pre-compiled headers.

8.7.4 Experiences

Besides the more performance related aspects, the application of UAL assertions to the above mentioned designs also helped to gain more methodological experiences.

Through the application of temporal assertions for TL models and mixed abstraction models several bugs were detected in each design. Most bugs either originated through maliciously modeled interprocess synchronization, especially in PV models, and wrong SW interactions with the underlying HW. These sorts of bugs are hard to debug using a common C++ debugger, because it involves stepping through many components. This by itself is error prone and does not provide an abstract view on a system's functionality.

By analyzing message sequence diagrams given in the modeling specification of a design it was easy to formulate UAL assertions which check that these sequences are not violated. Due to the abstract transaction aware specification of properties with UAL, the discrepancies between an assertion description and the design specification are very little.

Furthermore, locating the origin of an already located bug requires less effort with UAL assertions. By incrementally formulating more and more assertions, it is possible to incrementally narrow down the location of a bug. This approach also has lead to the detection of more conceptual bugs by constantly reinterpreting the design specification with assertions. Due to the declarative description of behavior with UAL, a user is enforced to express the intended functionality with a different more declarative view than an imperative description used for modeling the design.

Within a system development project, the use of UAL assertions also helps to formulate contracts between HW and the SW running on it. For example, assertions which monitor initialization sequences of devices initiated by the SW yield errors of the SW itself due to wrong configurations. Hence, UAL assertions allow a SW developer to obtain more information of the HW when developing SW for a specific platform, making the overall SW development more easy.

The multi-abstraction capabilities of UAL also allow better verification of systems which are comprised of blocks modeled at different abstractions. When simulating both RTL and TL blocks which interact in the context of a system, UAL assertions, similar to the aforementioned transactor example, help checking the crossing of abstraction levels, especially with regard to timing and synchronization.

Hence, so far, the same advantages of ABV applied for RTL designs were also observed with the application of UAL to TLMs:

- Expressing design intent
- Formulation of design contracts
- Fast discovering of design bugs
- Shorter debug times

On top of these advantages, UAL assertions provide a better comprehension of more complex system activity including HW and SW interactions.

9 Summary and Outlook

Within this thesis, a novel approach for applying assertion based verification at modeling abstraction levels higher than the register transfer level was introduced. Requirements were identified which have to be addressed in order to accomplish this task. As a solution, the new universal assertion language (UAL) was introduced which supports the specification of temporal behavior of transaction level models including untimed, timed, and cycle approximate modeling paradigms. Furthermore, it was shown, that assertions formulated in UAL can be specified also across all the supported abstraction levels including the register transfer level. The semantics of the language were underlined by a formal high-level colored petri net model. Furthermore, a complete application framework was introduced which supports easy instrumentation of a system with assertions. The new concepts have been illustrated through various application examples.

The novel solutions which have been developed throughout this work along with the key findings have already been pre-published in several scientific conferences [37], [57], [67], [39], [68], [38], [69], [70], [71], [72].

Students who have significantly contributed to the implementation and its testing are mentioned as coauthors in the mentioned publications.

The scientific contribution of this work can be summarized as follows:

- A transaction aware declarative language for the specification of temporal assertions.
- Definition of an additional event layer for assertion descriptions which allows temporal sequence specifications across abstraction levels.
- Definition of a generalized concept of events, which goes beyond the notion of events in delta-cycle based concurrency paradigms.
- Interpretation of transactions in terms of event pairs which reflect the start and the termination of transaction calls.
- The description of temporal correlations of transactions regardless of the abstraction level of the underlying model.

- Active monitoring of model behavior based on self generated timing of assertions.
- Definition of verifiable temporal behavior at untimed abstraction levels.
- Specification of temporal sequences at a finer granularity than delta-cycles.
- Specification of dynamic and data-dependent temporal behavior
- Definition of flexible evaluation modes which can not be accomplished with common temporal logic semantics.
- A formal representation of temporal behaviors based on a high-level colored petri net model.

The introduced concepts have been tested in a real industrial environment with the courtesy of Infineon Technologies. The UAL application framework has been applied within the System Design Methodology group for improving the quality of transaction level models of productive designs.

The development of the whole UAL application framework is comprised of the following components:

- UAL Base Library Implementation: 7783 lines of SystemC code
- UAL Compiler: 14152 lines of C++ code

The successful application of UAL with a design still requires annotations to the model under scrutiny with callbacks for obtaining an event based representation of transactions and value changes of variables. Improving this, has to be the goal for further development. This work must include the development of automation techniques to reduce this effort and also to cope with legacy and proprietary code.

In addition to this, further work includes the following aspects:

1. The syntax of UAL at the moment is verbose and strict. Following versions could include a convenience layer on top of this grammar, in order to reduce the effort for specifying assertions even further. Such enhancements could be to allow the specification of sequence expressions inlined to a property declaration or to introduce a subroutine declaration section where more complex arithmetic and boolean functions can be declared and be used to formulate Boolean layer expressions, and further similar additions.

2. The key feature of UAL is the possibility to specify temporal behavior at different abstraction levels ranging from purely untimed TL modeling paradigms down to the RTL modeling paradigm. A promising next step is the development of a methodology to enable a reuse of assertions written for high-level models with lower level representations. This way it possible to enhance the verification of the equivalence of a TLM with respect to the corresponding RTL implementation. Such a methodology would require an at least semi-automated process for refining high-level assertion descriptions to RTL assertion descriptions while ensuring that the overall property to be observed does not change.

3. The introduced approach is a well representation of successful HW verification techniques at higher abstraction levels. While offering powerful capabilities to express abstract properties the approach presented here assumes a static structure of the underlying model. As more and more research is done for also migrating complete operating system development into a single abstract system-level representation of an embedded system it can be expected that also high-level SW modeling concepts will be utilized. Here, objects and processes are created dynamically at runtime after elaborating the system. Hence, for enabling the specification of assertions to keep track of temporal properties of such modeling styles it is necessary that the UAL framework is enhanced to also deal with dynamically created data structures and processes.

4. The evaluation semantics of UAL assertions are well suited for monitoring various sorts of communication patterns, including retransmissions and pipelined protocols. Within further work these semantics could be enhanced to also cope with more abstract scheduling algorithms applied in operating systems and bus arbitration handling where for instance, tasks or respectively requests are associated with priorities and can be preemptively suspended and resumed or respectively dropped.

Bibliography

[1] "ITRS Roadmap 2006 Update," p. 5, 2006. [Online]. Available: http://www.itrs.net/Links/2006Update/FinalToPost/01-SysDrivers-2006UPDATE.pdf

[2] R. Hodgson, "The X-Model: A Process Model for Object-Oriented Software Development," *Fourth International Conference "Software Engineering and its Application"*, 1991.

[3] "ITRS Roadmap 2006 Update," p. 16, 2006. [Online]. Available: http://www.itrs.net/Links/2006Update/FinalToPost/02-Design-2006Update.pdf

[4] Mentor Graphics, *Advanced Verification Methodology Cookbook.* Mentor Graphics, 2006. [Online]. Available: http://www.mentor.com/products/fv/_3b715c/

[5] *Wikipedia.* [Online]. Available: http://en.wikipedia.org/wiki/Moore%27s_law

[6] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, 19 April 1965.

[7] IEEE Computer Society, *SystemC LRM P1666.* [Online]. Available: http://www.ieee.org

[8] O. T. W. Group, *TLM 1.0.* [Online]. Available: http://www.systemc.org

[9] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification.* San Francisco, USA: Morgan Kaufmann Publishers, 2007.

[10] B. Bailey, "Design complexity drives need for ESL," *EE Times*, 2004. [Online]. Available: http://www.eetimes.com/news/design/columns/eda/showArticle.jhtml?articleID=49900562

[11] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," in *Formal Methods in System Design (FMSD)*, vol. 19, no. 1. Kluwer, 2001.

[12] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," in *Advances in Computers*, vol. 58. Academic Press, 2003.

[13] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design.* Norwell: Kluwer Academic Publishers, 2003.

[14] L. Lamport, "Specifying concurrent systems with TLA+," in *Calculational System Design.* Amsterdam: IOS Press, 1999. [Online]. Available: citeseer.ist.psu.edu/article/lamport99specifying.html

[15] R. Hum, "How to boost verification productivity," *EE Times*, 2005. [Online]. Available: http://www.eetimes.com/news/design/columns/eda/showArticle.jhtml?articleID=57700323

[16] E. Gonen, "Psl in action - abv experience report," *Design Automation Conference*, 2004.

[17] H. Foster, "Unifying Traditional and Formal Verification Through Property Specification," 2002.

[18] J. A. Nacif, F. M. de Paula, H. Foster, C. N. C. Jr., F. C. Sica, A. O. Fernandes, and D. C. da Silva Jr, "An assertion library for on-chip white-box verification at run-time," in *4th IEEE Latin American Test Workshop (LATW)*, Brazil, 2003.

[19] K. Peterson and Y. Savaria, "Assertion-based on-line verification and debug environment for complex hardware systems," in *Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS)*, 2004.

[20] F. E. project under the Framework 6 (IST-027580 SPRINT-IP), "Open SoC Design Platform for Reuse and Integration of IPs (SPRINT)."

[21] IEEE Computer Society, *VHDL LRM Std. 1076-1987.* [Online]. Available: http://www.ieee.org

[22] Accellera, *Open Verification Library.* [Online]. Available: http://www.accellera.org/activities/ovl/

[23] Mentor Graphics, *0-in CheckerWare.* [Online]. Available: http://www.mentor.com

[24] Accellera, *Accellera PSL v1.1 LRM.* [Online]. Available: http://www.eda.org/vfv/docs/PSL-v1.1.pdf

[25] IEEE Computer Society, *SystemVerilog LRM P1800.* [Online]. Available: http://www.ieee.org

[26] ——, *IEEE Standard for the Functional Verification Language 'e'*, 2006. [Online]. Available: http://www.ieee.org

[27] J. Havlicek and Y. Wolfsthal, "PSL AND SVA: TWO STANDARD ASSERTION LANGUAGES ADDRESSING COMPLEMENTARY ENGINEERING NEEDS," *Design and Verification Conference (DVCON)*, 2005.

[28] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, *Verification Methodology Manual for SystemVerilog.* New York, USA: Springer Science+Business Media, 2006.

[29] Verisity, *Verification Reuse Methodology.* [Online]. Available: http://www.verisity.com/resources/whitepaper/erm.html

[30] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "FoCs: Automatic generation of simulation checkers from formal specifications," in *Computer Aided Verification*, 2000, pp. 538–542.

[31] A. Dahan, D. Geist, L. Gluhovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamdem, and Y. Lahbib, "Combining system level modeling with assertion based verification," *Sixth International Symposium on Quality of Electronic Design (ISQED'05)*, March 21 - 23 2005.

[32] T. Peng and B. Baruah, "Using Assertion-based Verification Classes with SystemC Verification Library," *Synopsys Users Group, Boston*, 2003.

[33] J. T. Inc., "Native SystemC Assertion (NSCa)," 2005. [Online]. Available: http://www.jedatechnologies.net

[34] A. Habibi and S. Tahar, "On the extension of SystemC by SystemVerilog Assertions," in *Canadian Conference on Electrical & Computer Engineering*, vol. 4, Niagara Falls, Ontario, Canada, May 2004, pp. 1869–1872.

[35] ——, "Towards an Efficient Assertion Based Verification of SystemC Designs," in *In Proc. of the High Level Design Validation and Test Workshop*, Sonoma Valley, California, USA, November 2004, pp. 19–22.

[36] A. Habibi, A. Gawanmeh, and S. Tahar, "Assertion based verification of PSL for SystemC designs," in *International Symposium on System-on-Chip*, Tampere, Finland, November 2004, pp. 177–180.

[37] 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.

[38] 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.

[39] W. Ecker, V. Esen, J. Smit, T. Steininger, and M. Velten, "IP Library For Temporal SystemC Assertions," in *Forum on Specification & Design Languages (FDL)*, Darmstadt, Germany, September 2006, pp. 301–308.

[40] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel, "Simulation-Guided Property Checking Based on Multi-Valued AR-Automata," 2001.

[41] R. J. Weiss, J. Ruf, T. Kropf, and W. Rosenstiel, "Efficient and Customizable Integration of Temporal Properties into SystemC," Lausanne, Switzerland, September 2005.

[42] D. Lettnin, R. J. Weiss, A. Braun, J. Ruf, and W. Rosenstiel, "Temporal Properties Verification of System Level Design," Erfurt, Germany, September 2005.

[43] P. M. Peranandam, R. J. Weiss, J. Ruf, and T. Kropf, "Transactional Level Verification and Coverage Metrics by Means of Symbolic Simulation," February 2004.

[44] A. Bauer, M. Leucker, and J. Streit, "SALT—Structured Assertion Language for Temporal logic," in *Proceedings of the Eighth International Conference on Formal Engineering Methods (ICFEM)*, ser. Lecture Notes in Computer Science, Z. Liu and J. He, Eds., vol. 4260. Berlin, Heidelberg: Springer-Verlag, Oct. 2006, pp. 757–776.

[45] ——, "SALT—Structured Assertion Language for Temporal logic," no. TUM-I0604, March 2006.

[46] 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.

[47] W. Thomas, "Automata on infinite objects," pp. 133–191, 1990.

[48] 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.

[49] ——, "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.

[50] 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.

[51] 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.

[52] A. Habibi and S. Tahar, "Design for Verification of SystemC Transaction Level Models," in *Design Automation and Test in Europe*, vol. 1, Munich, Germany, March 2005, pp. 560–565.

[53] 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.

[54] ——, "Assertion Based Verification of Transaction Level Models," in *MBMV*, 2006.

[55] A. Kasuya and T. Tesfaye, "Verification Methodologies in a TLM-to-RTL Design Flow," *DAC*, 2007.

[56] A. Kasuya, T. Tesfaye, and E. Zhang, "Native SystemC Assertion mechanism with transaction and temporal assertion support," *EDA Tech Forum*, 2006.

[57] 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.

[58] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, May 1994.

[59] 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

[60] 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.

[61] T. Steininger, *Automated Assertion Transformation Across Multiple Abstraction Levels*, to be published 2007.

[62] S. A. Kripke, “Semantical considerations on modal logic,” in *Proceedings of a Colloquium: Modal and Many Valued Logics, Acta Philosophica Fennica*, vol. 16, 1963, pp. 83–94.

[63] T. Kropf, *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[64] V. Stolz and F. Huch, “Runtime verification of concurrent haskell programms,” 2004. [Online]. Available: citeseer.ist.psu.edu/stolz05runtime.html

[65] C. Eisner, D. Fisman, J. Havlicek, A. McIsaac, and D. van Campenhout, “The Definition of a Temporal Clock Operator,” in *ICALP*, 2003, pp. 857–870.

[66] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* USA: Addison-Wesley Professional Computing Series, 2005.

[67] W. Ecker, V. Esen, M. Hull, T. Steininger, and M. Velten, “Requirements and Concepts for Transaction Level Assertions,” in *24th International Conference on Computer Design (ICCD)*, California, USA, October 2006.

[68] ——, “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.

[69] ——, “A Prototypic Language for Transaction Level Assertions,” in *Design & Verification Conference & Exhibition (DVCon)*, San Jose, California, February 2007.

[70] ——, “Implementation of a Transaction Level Assertion Framework in SystemC,” in *Design, Automation and Test in Europe (DATE)*, Nice, France, April 2007.

[71] W. Ecker, V. Esen, T. Steininger, and M. Velten, “On the Application of Transaction Level Assertions,” in *Ekompass-Workshop (edaWorkshop)*, Hannover, Germany, June 2007.

[72] W. Ecker, V. Esen, R. Schwencker, T. Steininger, and M. Velten, “Formal Techniques Speed Up Interconnect Verification of SystemC Virtual Platform Models,” in *Design & Verification Conference & Exhibition (DVCon)*, San Jose, California, February 2008.

Acronyms

ABV

Assertion **B**ased **V**erification

AVM

Advanced **V**erification **M**ethodology

BFM

Bus-**F**unctional **M**odel

BMC

Bounded **M**odel **C**hecking

CA

Cycle **A**pproximate

CC

Cycle **C**allable

CPU

Central **P**rocessing **U**nit

CTL

Computation **T**ree **L**ogic

DC

Duration **C**alculus

DPE

Data **P**rocessing **E**ngine

DUT

Design **U**nder **T**est

DUV

Design **U**nder **V**erification

EBNF

Extended **B**ackus-**N**aur **F**orm

EDA

Electronic **D**esign **A**utomation

ESL

Electronic **S**ystem-**L**evel

FL

Foundation **L**anguage

FLTL

Finite **L**inear **T**emporal **L**ogic

FPGA

Field **P**rogrammable **G**ate **A**rrays

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

HVL

Hardware **V**erification **L**anguage

HW

Hard**W**are

ITRS

International **T**echnology **R**oadmap for **S**emiconductors

LHS

Left **H**and **S**ide

LOC

Logic **O**f **C**onstraints

LRM

Language **R**eference **M**anual

LTL

Linear **T**emporal **L**ogic

OBE

Optional **B**ranching **E**xtensions

OSCI

Open **S**ystem**C** **I**nitiative

OVL

Open **V**erification **L**ibrary

PSL

Property **S**pecification **L**anguage

PV

Programmer's **V**iew

PVT

Programmer's **V**iew with **T**iming

RHS

Right **H**and **S**ide

RTL

Register **T**ransfer **L**evel

SoC

System **o**n **C**hip

SVA

System**V**erilog **A**ssertions

SW

Soft**W**are

TL

Transaction **L**evel

TLA

Temporal **L**ogic of **A**ctions

TLM

Transaction **L**evel **M**odel

TLTL

Timed **L**inear **T**emporal **L**ogic

UAL

Universal **A**ssertion **L**anguage

UML

Unified **M**odeling **L**anguage

URM

Universal **R**euse-**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

Glossary

Assertion

A verification component which observes the adherence of a design property

Assertion Based Verification

Dynamic or formal checking of design properties

Bounded Model Checking

A technique in formal verification which tries to prove an abstract formal behavior specification (property) against the state space of a design in terms of its state-transition paths of a certain length which is referred to as bound

Bus Functional Model

A component that translates between an abstract model of a communication protocol and its implementation model

Callback

Immediate notification without introducing delta-cycle delays

Clock tick

Either a positive or a negative edge of a clock signal

Coverage

A verification technique for measuring verification progress

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

Initiator

A module which invokes transactions

Linear Temporal Logic

A logic which enables the specification of temporal relations between Boolean propositions

Monitor

A verification component which incorporates one or more assertions or other checking components

Property

A formal specification of a desired behavior of a design

Register Transfer Level

Synthesizable, pin and cycle accurate hardware description

Scoreboard

A verification component which stores the expected responses of a DUV for comparison with the actual responses

Sequence

Regular expression which formulates a temporal pattern of Boolean propositions

Simulation kernel

Algorithm that defines the execution semantics of a modeling language

Software

A program which is executed on a HW platform

Sub-thread

A subordinate branch of one thread

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

Hardware Description and Verification Language (HDVL) based on and extending the Verilog HDL which includes design features, testbench features, and assertion features

Target

A module which implements transactions or a scope in a binding specification

Testbench

A verification component for simulation-based verification of a design, providing components for stimulus generation , design interconnect, and response checking

Thread

One evaluation of a sequence

Transaction

The encapsulation of a whole communication protocol into a function

Transactor

A component that translates between an abstract model of a communication protocol and its implementation model (see BFM)

Trigger

An event possibly derived from other events for triggering the delay mechanism of sequences

VHDL

The most common hardware description language in European semi-conductor industry

Virtual Prototype

A fully virtual executable model of a SoC

A Requirements Summary

This Appendix summarizes the requirements mentioned in Chapter 3 and categorizes the requirements according to whether it has not or not adequately been addressed yet.

A.1 List of Requirements

R 1 : ***Specification of transaction sequences***

R 2 : ***Runtime evaluation of assertions***

R 3 : ***Support of all SystemC and C++ base data types***

R 4 : ***Seamless access of assertions to modules and their internals***

R 5 : ***Compliance to SystemC***
OSCI SystemC compliant evaluation of assertions. No changes to the simulation semantics are allowed.

R 6 : ***Implementation on top of SystemC***

R 7 : ***Support of any event offered within SystemC***

R 8 : ***Linking assertions to any SystemC event***

R 9 : ***Tracking of implicit SystemC events***

R 10 : ***More granular time resolution than delta-time***

R 11 : ***Support of assertions in context of blocking and non-blocking transactions***

R 12 : ***Mechanism to link to transactions including return values and arguments***

R 13 : ***Support of OSCI TLM standard***

R 14 : ***Support of most popular abstraction levels***
Support of abstraction levels: PV, PVT, Cycle Approximate (CA), CC/ RTL

R 15 : ***Support of mix of abstraction levels***
Support of mix of abstraction levels from R 14

R 16 : ***Mechanism to link assertions to model state variables***

R 17 : ***Mechanism to capture assignment on model state variables***

R 18 : ***Mechanism to link to existing public state access functions***
Mechanism to link to existing public state access functions also in case of model state variables which are declared in a private context.

R 19 : ***Specification of partial orders on events***

R 20 : ***Specification of strict partial orders on events***
Strict partial orders for detecting absence of event occurrences.

R 21 : ***Time identity of events***
Time identity of events happening at the same simulation time

R 22 : ***Specification of temporal relations based on simulation time***

R 23 : ***Dynamic temporal behavior***
Capture dynamic temporal behavior, including dynamic time delays, dynamic amount of transaction calls, dynamic amount of event occurrences.

R 24 : ***Mechanism to link assertions to signals***

R 25 : ***Mechanism to reset assertion evaluations***

R 26 : ***Defined sampling of design states***
Sampling of design states with the occurrence of any transaction, any event, or at any simulation time.

R 27 : ***Read-Only access to design states***
Read access to design internals, write access must not be supported.

R 28 : ***No side-effects in DUV***
No side-effects in DUVcaused by assertions.

R 29 : ***Transaction detection mechanism***
Enabling the tracking of transaction occurrences.

R 30 : ***Selective transaction detection mechanism***
Detection of consecutive, partially overlapped, and fully overlapped transactions.

R 31 : ***Immediate notification of occurring transactions***
The state of a model may not change until the detection has been processed.

R 32 : ***Support of different request / response behaviors***

R 33 : ***Support of different pipelining behaviors***

1. *Detection of in-order pipelining*
2. *Detection of arbitrary-order pipelining*

R 34 : ***Declarative assertion notation***
Implementation as a declarative language that supports transaction aware abstract descriptions of design properties and assertions.

R 35 : ***Support of all RTL ABV paradigms***
Assertion evaluation needs to support the following RTL ABV paradigms:

1. *Match all possible alternatives*
2. *Match as early as possible*
3. *Overlapped evaluation*

R 36 : ***Accumulation of assertion results***
Coverage has to be provided to following assertion results:

- *Success: Increment coverage item on each success*
- *Vacuous Success: Increment coverage item on each vacuous success (failing antecedent expressions)*
- *Real Success: Increment coverage item only on real successes (succeeding consequent evaluations)*
- *Failure: Increment coverage item only when property fails (failing consequent evaluations)*

Explicit enabling of coverage needs to be specified in the context of an assertion.
Coverage results need to be accessible to other objects (testbenches).

R 37 : ***Support of Local Variables***
Support of local variables to transport data within one evaluation thread.

R 38 : ***Assertion runtime control***

R 39 : ***Support of severity levels***
The severity level of an assertion failure needs to be specified in the context of an assertion. The following severity levels need to be supported:

- *INFO*
- *WARNING*
- *ERROR*
- *FAILURE*

The meaning of a severity needs to be relayed to a simulation tool. Default simulator interactions on failing assertions are:

- *INFO — WARNING: Display notification to error output; continue simulation*
- *ERROR — FAILURE: Display notification to error output; halt simulation*

The severity levels can be used e.g. to filter outputs, i.e. ”show severity levels higher than WARNING”

R 40 : ***Support of user specifiable report messages***

R 41 : ***Packaging of assertions***
Assertions need to be encapsulated to allow for packaging to assertion libraries

A.2 Categorization

This section provides a categorization of the requirements listed in the previous section. Each requirement is categorized according to its status at the beginning of this work and its importance for accomplishing ABV at TL.

The status is marked with three different values:

addressed : the requirement has already been addressed in other ABV approaches

non-adequate : the requirement has already been addressed in other ABV approaches however, not to the extent required here

missing : the requirement has not been addressed at all

The importance of a requirement is marked by two values:

blocking : the fulfillment of a requirement marked as blocking is vital for accomplishing ABV at TL

useful : the fulfillment of a requirement marked as useful does not enable ABV at TL however, it does enhance the applicability

Table A.1 shows the categorization of the previously listed requirements.

Requirement	Status	Importance
R 1	non-adequate	blocking
R 2	addressed	blocking
R 3	addressed	blocking
R 4	addressed	useful
R 5	addressed	blocking
R 6	non-adequate	blocking
R 7	non-adequate	blocking
R 8	non-adequate	blocking
R 9	missing	blocking
R 10	missing	blocking
R 11	non-adequate	blocking
R 12	missing	blocking
R 13	missing	useful
R 14	missing	blocking
R 15	missing	blocking
R 16	non-adequate	blocking
R 17	missing	useful
R 18	addressed	useful
R 19	non-adequate	blocking
R 20	missing	blocking
R 21	missing	blocking
R 22	missing	blocking
R 23	missing	blocking
R 24	addressed	blocking
R 25	addressed	blocking
R 26	non-adequate	blocking
R 27	addressed	useful
R 28	addressed	useful
R 29	non-adequate	blocking
R 30	missing	blocking
R 31	missing	blocking
R 32	non-adequate	useful
R 33	non-adequate	useful
R 34	non-adequate	blocking
R 35	addressed	blocking
R 36	addressed	useful
R 37	addressed	useful
R 38	addressed	useful
R 39	addressed	useful
R 40	addressed	useful
R 41	addressed	useful

Table A.1: Categorization of Requirements

B Language Grammar

B.1 Monitor Grammar

```
monitor               =  "monitor" identifier
                         ports_section
                         [ constants_section ]
                         sequences_section
                         properties_section
                         verification_section
                         "endmonitor" ;
                                                                        (B.1)
ports_section         =  "ports"
                         port_declaration { port_declaration }
                         "endports" ;
                                                                        (B.2)
port_declaration      =  kind type identifier [ "[" number "]" ]
                         [ transaction_parameters ] ";" ;
                                                                        (B.3)
constants_section     =  "constants"
                         constant_declaration { constant_declaration }
                         "endconstants" ;
                                                                        (B.4)
constant_declaration  =  type identifier "=" value ";" ;
                                                                        (B.5)
sequences_section     =  "sequences"
                         sequence_section { sequence_section }
                         "endsequences" ;
                                                                        (B.6)
properties_section    =  "properties"
                         property_section { property_section }
                         "endproperties" ;
                                                                        (B.7)
verification_section  =  "verification"
                         directive { directive }
                         "endverification" ;
                                                                        (B.8)
directive             =  directive_kind identifier "(" [ directive_parameter ] ")"
                         "=" property_instance ";" ;
                                                                        (B.9)
```

```
directive_kind            = "assert"
                             | "cover"
                             | "assert_cover"
                             | "assume" ;                                          (B.10)
directive_parameter       = severity_level
                            "," string
                            [ "," reset_event_expr ] ;                             (B.11)
severity_level            = "INFO"
                             | "WARNING"
                             | "ERROR"
                             | "FAILURE" ;                                         (B.12)
reset_event_expr          = event_expression ;                                     (B.13)
property_section          = "property" identifier property_interface
                            property_declarations
                            property_specification
                            "endproperty" ;                                        (B.14)
property_interface        = [ property_mode_list ] formal_argument_list ;          (B.15)
property_declarations     = { localvar_declaration } ;                             (B.16)
property_specification    = implication_property
                             | single_sequence_property                            (B.17)
implication_property      = sequence_instance "|->" sequence_instance ";" ;        (B.18)
single_sequence_property  = sequence_instance ";" ;                                (B.19)
property_instance         = identifier [ property_mode_list ]
                            param_argument_list ;                                  (B.20)
property_mode_list        = "[" [ sequence_mode "," ] property_mode "]" ;          (B.21)
property_mode             = "Restart"
                             | "NoRestart"
                             | "ReportOnRestart"
                             | "Overlap"
                             | "Pipe"
                             | "PipeOrdered"
                             | "Cover" ;                                           (B.22)
sequence_section          = "sequence" identifier sequence_interface
                            sequence_declarations
                            sequence_specification
                            "endsequence" ;                                        (B.23)
```

```
sequence_interface      =  [ "[" sequence_mode "]" ] formal_argument_list  ;
                                                                              (B.24)
sequence_declarations   =  { localvar_declaration }  ;
                                                                              (B.25)
sequence_specification  =  delay_operator { delay_operator } ";"  ;
                                                                              (B.26)
delay_operator          =  "#" steps sensitivity "{" condition { action } "}"  ;
                                                                              (B.27)
steps                   =  zero_step
                            | multi_step
                            | range_step  ;
                                                                              (B.28)
zero_step               =  "0"
                            | ( "{" "0" "}" )  ;
                                                                              (B.29)
multi_step              =  non_zero_number
                            | ( "{" non_zero_number "}" )  ;
                                                                              (B.30)
range_step              =  "{" number ":" number "}"  ;
                                                                              (B.31)
sensitivity             =  "{" [ pos_sensitivity ]
                           [ ";" neg_sensitivity ] "}"  ;
                                                                              (B.32)
condition               =  boolean_expression
                           [ "?" boolean_expression ":" boolean_expression ]  ;
                                                                              (B.33)
action                  =  "," identifier "=" localvar_expression  ;
                                                                              (B.34)
boolean_expression      =  expression  ;
                                                                              (B.35)
accumulator_expression  =  expression  ;
                                                                              (B.36)
timer_expression        =  expression  ;
                                                                              (B.37)
localvar_expression     =  expression  ;
                                                                              (B.38)
sequence_instance       =  identifier [ "[" sequence_mode "]" ] param_argument_list  ;
                                                                              (B.39)
sequence_mode           =  "AnyMatch"
                            | "FirstMatch"
                            | "FirstMatchPipe"
                            | "FirstMatchPipeOrdered"  ;
                                                                              (B.40)
operand                 =  lastevent
                            | value
                            | ( identifier [ "." identifier ] [ "[" array_index "]" ] )
                            | ( identifier [ "." "RET" ] )  ;
                                                                              (B.41)
```

```
factor               =  operand
                         | ( "(" expression ")" )
                         | ( unary_operator factor ) ;
                                                                    (B.42)
term                 =  factor { arith_operator factor } ;
                                                                    (B.43)
expression           =  term { boolean_operator term } ;
                                                                    (B.44)
pos_sensitivity      =  trigger_expression ;
                                                                    (B.45)
neg_sensitivity      =  trigger_expression ;
                                                                    (B.46)
trigger_expression   =  ( event_expression [ "," trigger_timer ] )
                         | trigger_timer ;
                                                                    (B.47)
trigger_timer        =  "timer(" timer_expression ")" ;
                                                                    (B.48)
event_operand        =  identifier [ "[" array_index "]" ] [ "'" event_kind ] ;
                                                                    (B.49)
event_constraint     =  "@(" boolean_expression ")" ;
                                                                    (B.50)
event_accumulator    =  "%(" accumulator_expression ")" ;
                                                                    (B.51)
unary_event_expr     =  event_accumulator
                         | event_constraint ;
                                                                    (B.52)
event_factor         =  ( event_operand [ unary_event_expr ] )
                         | ( "(" event_expression ")" [ unary_event_expr ] ) ;
                                                                    (B.53)
event_term           =  event_factor { "&" event_factor } ;
                                                                    (B.54)
event_expression     =  event_term { "|" event_term } ;
                                                                    (B.55)
lastevent            =  "$l_event(" event_operand ")" ;
                                                                    (B.56)
arith_operator       =  "&"
                         | "|"
                         | "+"
                         | "-"
                         | "/"
                         | "%"
                         | "*" ;
                                                                    (B.57)
```

```
boolean_operator          =  "&&"
                              | "||"
                              | "<="
                              | ">="
                              | "!="
                              | "=="
                              | "<"
                              | ">"  ;
                                                                        (B.58)
unary_operator            =  "!"
                              | "-"
                              | "~"  ;
                                                                        (B.59)
formal_argument_list      =  "(" [ formal_argument_decl ]
                             { "," formal_argument_decl } ")"  ;
                                                                        (B.60)
formal_argument_decl      =  [ "ref" ] [ kind ] [ type ] identifier
                             [ transaction_parameters ]  ;
                                                                        (B.61)
localvar_declaration      =  type identifier ";"  ;
                                                                        (B.62)
param_argument_list       =  "(" [ parameter_argument ]
                             { "," parameter_argument } ")"  ;
                                                                        (B.63)
parameter_argument        =  identifier [ ( "'" event_kind ) | ( "." identifier ) ]  ;
                                                                        (B.64)
transaction_parameters    =  "(" type identifier
                             { "," type identifier } ")"  ;
                                                                        (B.65)
kind                      =  "state"
                              | "event"
                              | "signal"
                              | "transaction"  ;
                                                                        (B.66)
```

B.2 Bind Grammar

```
bind_file                 =  bind_definition { bind_definition }  ;
                                                                        (B.67)
bind_definition           =  "bind" identifier
                             targets_section
                             mappings_section
                             "endbind"  ;
                                                                        (B.68)
targets_section           =  "targets" [ "(" "class" identifier ")" ]
                             target_declaration { target_declaration }
                             "endtargets"  ;
                                                                        (B.69)
```

target_declaration = *monitor_target*
| *design_target* ; (B.70)

monitor_target = "monitor" *identifier* "=" *identifier* ";" ; (B.71)

design_target = "module" *identifier*
"=" *identifier* { "." *identifier* }
"(" *identifier* [template] "," """ *filename* """ ")" ";" ; (B.72)

design_target_type = "module"
| "proxy" ; (B.73)

mappings_section = "mappings"
mapping_declaration { *mapping_declaration* }
"endmappings" ; (B.74)

mapping_declaration = *identifier* "." *identifier*
"=>" *identifier* "." *design_object* ";" ; (B.75)

design_object = *transaction_object*
| *array_object*
| *function_object*
| *variable_object* ; (B.76)

transaction_object = *variable_object* *parameter_mapping* ; (B.77)

array_object = *variable_object* "[" *number* "]" ; (B.78)

function_object = *variable_object* "(" ")" ; (B.79)

variable_object = *identifier* { "." *identifier* } ; (B.80)

parameter_mapping = "(" (*identifier* | "RET") "=>" *identifier*
{ "," *identifier* "=>" *identifier* } ")" ; (B.81)

B.3 Testbench Grammar

test_file = *test_definition* { *test_definition* } ; (B.82)

test_definition = "testbenches" *identifier*
testbench_section { *testbench_section* }
"endtestbenches" ; (B.83)

testbench_section = "testbench" *identifier*
testcase_section { *testcase_section* }
"endtestbench" ; (B.84)

```
testcase_section      =  "testcase" identifier testcase_parameters
                         test_stimulus { test_stimulus }
                         "endtestcase" ;
                                                                        (B.85)
test_stimulus         =  ( assign_stimulus
                         | event_stimulus
                         | wait_statement  ) ";" ;
                                                                        (B.86)
assign_stimulus       =  identifier [ "." identifier ] [ "[" number "]" ] "=" value ;
                                                                        (B.87)
event_stimulus        =  identifier [ "[" number "]" ] [ "'" event_kind ] ;
                                                                        (B.88)
wait_statement        =  "wait" "(" number ")" ;
                                                                        (B.89)
testcase_parameters   =  "(" "loop" "=" number ","
                         "reset" "=" reset_type
                         [ "," "trace" "=" ( "ENABLE" | "DISABLE" ) ] ","
                         expect_statement ")" ;
                                                                        (B.90)
expect_statement      =  "expect" "=" "[" identifier cover_assignment
                         { "," identifier cover_assignment } "]" ;
                                                                        (B.91)
cover_assignment      =  "(" cover_type "=" number
                         { "," cover_type "=" number } ")" ;
                                                                        (B.92)
cover_type            =  "REAL"
                         | "SUCCESS"
                         | "VACUOUS"
                         | "FAILURE" ;
                                                                        (B.93)
reset_type            =  "MONITOR"
                         | "COVERAGE"
                         | "NONE"
                         | "ALL" ;
                                                                        (B.94)
```

B.4 Common Grammar

```
string                =  """ string_characters """ ;
                                                                        (B.95)
filename              =  ( non_digit | digit | "-" | "." )
                         { non_digit | digit | "-" | "." } ;
                                                                        (B.96)
```

event_kind = "START"
| "END"
| "POS"
| "NEG"
| "CH" ;
(B.97)

type = *object_type* [*template*] ;
(B.98)

template = "<" ((*object_type* [*template*]) | *value*)
{ "," ((*object_type* [*template*]) | *value*) } ">" ;
(B.99)

object_type = *cpp_type*
| *sc_type*
| *ual_type* ;
(B.100)

ual_type = "callback" ;
(B.101)

cpp_type = *int_type*
| "double"
| ("long" "double")
| "float"
| "bool"
| "void" ;
(B.102)

timeunit_definition = "timeunit" "=" *time_number* *time_unit* ;
(B.103)

int_type = [*sign_type*] *cpp_integer* ;
(B.104)

cpp_integer = ("long" "int")
| ("long" "long" "int")
| ("long" "long")
| "long"
| ("short" "int")
| "short"
| "int" ;
(B.105)

sign_type = "signed"
| "unsigned" ;
(B.106)

sc_type = "sc_signal"
| "sc_int"
| "sc_uint"
| "sc_event"
| "sc_event_queue"
| "sc_time"
| "sc_bit"
| "sc_bv" ;
(B.107)

```
value           =  integer_value
                   | real_value
                   | ( "'" bit_value "'" )
                   | ( """ bv_value """ )
                   | boolean_value
                   | ual_value
                   | ( "(" array_value ")" )  ;
                                                            (B.108)
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.109)
ual_value       =  "$time"
                   | "$delta_t"  ;
                                                            (B.110)
integer_value   =  [ "-" ] number  ;
                                                            (B.111)
real_value      =  [ "-" ] number "." number  ;
                                                            (B.112)
bv_value        =  bit_value { bit_value }  ;
                                                            (B.113)
bit_value       =  "1"
                   | "0"  ;
                                                            (B.114)
boolean_value   =  "true"
                   | "false"  ;
                                                            (B.115)
time_number     =  "100"
                   | "10"
                   | "1"  ;
                                                            (B.116)
time_unit       =  "s"
                   | "ms"
                   | "us"
                   | "ns"
                   | "ps"
                   | "fs"  ;
                                                            (B.117)
array_index     =  ( identifier [ "." identifier ] )
                   | number  ;
                                                            (B.118)
```

```
number              =  "0"
                        | non_zero_number  ;
                                                              (B.119)
non_zero_number     =  non_zero_digit { digit }  ;
                                                              (B.120)
identifier          =  non_digit { digit  | non_digit }  ;
                                                              (B.121)
string_characters   =   { digit
                        | non_digit
                        | special_character  }  ;
                                                              (B.122)
digit               =  "0"
                        | non_zero_digit  ;
                                                              (B.123)
non_zero_digit      =  "1"
                        | "2"
                        | "3"
                        | "4"
                        | "5"
                        | "6"
                        | "7"
                        | "8"
                        | "9"  ;
                                                              (B.124)
non_digit           =  "_"
                        | letter  ;
                                                              (B.125)
letter              =  uppercase_letter
                        | lowercase_letter  ;
                                                              (B.126)
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.127)
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.128)
special_character   =  "-" | "/"
                        | "+" | "*" | "<"
                        | ">" | "=" | "$"
                        | "!" | "^" | "~"
                        | "{" | "}" | "["
                        | "]" | "(" | ")"
                        | "?" | "." | "’"
                        | "|" | "&" | ","
                        | ";" | ":" | "%"  ;
                                                              (B.129)
```