Complete Functional Verification

Translation of dissertation from April 2009

Jörg Bormann

Joerg.D.Bormann@web.de
Gistlstraße 55a
82049 Pullach
Germany
+49 1577 356 4108

Summary

The dissertation describes a practically proven, particularly efficient approach for the verification of digital circuit designs. The approach outperforms simulation based verification wrt. final circuit quality as well as wrt. required verification effort. In the dissertation, the paradigm of transaction based verification is ported from simulation to formal verification. One consequence is a particular format of formal properties, called operation properties. Circuit descriptions are verified by proof of operation properties with Interval Property Checking (IPC), a particularly strong SAT based formal verification algorithm. Furtheron, a completeness checker is presented that identifies all verification gaps in sets of operation properties. This completeness checker can handle the large operation properties that arise, if this approach is applied to realistic circuits. The methodology of operation properties, Interval Property Checking, and the completeness checker form a symbiosis that is of particular benefit to the verification of digital circuit designs. On top of this symbiosis an approach to completely verify the interaction of completely verified modules has been developed by adaptation of the modelling theories of digital systems.

The approach presented in the dissertation has proven in multiple commercial application projects that it indeed completely verifies modules. After reaching a termination criterion that is well defined by completeness checking, no further bugs were found in the verified modules. The approach is marketed by OneSpin Solutions GmbH, Munich, under the names "Operation Based Verification" and "Gap Free Verification".

Table of Contents

1 Preliminaries

1.1 Note about the translation

This document is a translation of my thesis “Vollständige funktionale Verifikation” to obtain the level of Dr.-Ing. at the University of Kaiserslautern from April 2009. The translation was prepared in May 2017. I took the freedom to correct errors and to update one or the other remark.

To save some effort, the translation started from the output of a public domain version of Google Translate. Since I could only spend a limited amount of effort on the translation, I skipped the appendix with the proofs. Please contact me in case of need.

The original German text can be obtained from *kluedo.ub.uni-kl.de/volltexte/2009/2356.*

1.2 Foreword

This work has been developed in an industrial environment, initially at the Central Research and Development department of Siemens AG, then at Infineon, and finally at the startup OneSpin Solutions GmbH, which further develops and markets these ideas under the names "Operation Based Verification" and "Gap Free Verification".

This has resulted in many ideas of this work being intensively examined. Without the convincing outcome from pilot projects for commercial deployment, I would never have had the chance to work out the complete verification. In these projects, our customers, both internal in our Siemens and Infineon times, as well as the real customers, had a large share. I am grateful to them for their challenges, for the insights they have given me into their processes, and sometimes also for the enthusiasm with which they have taken our results, although in most cases these results revealed errors in their circuit designs and led to additional work.

The work presented here has been created in a high-profile environment. So I could build on a high-class technology, which has many mothers and fathers. To name them individually would be beyond the scope of this preface.

Our front-end people developed and supervised the precise and powerful programs that read the circuits descriptions, which cope with all the finesse and all the sizes of the RTL descriptions presented to them. The tools developed in our prover group are among the most powerful in the world. They spoiled us in such a way that today we become impatient even with large circuits, if a proof takes longer than 5 minutes. A very important development task for a formal verification tool is the preparation of user input and the intuitive feedback about the results of the proof. The colleagues concerned have created a product that is pleasant and efficient to use and which is able to deal with highest complexity requirements. As a crown, a professional graphical user interface was created, which presents the functionality in an easily understandable way. All of this technology, combined in the OneSpin 360 MV product, was at my disposal to develop the ideas presented here, which in turn influenced the product. For this I am grateful to the developers.

In addition to the developers, I would like to thank Sven Beyer, Martin Freibothe, Steven Obua, Jens Schönherr and Sebastian Skalberg, who have been with me in the group for methodology, technology and advanced applications of OneSpin Solutions GmbH for their ideas, feedbacks, and further developments. I had further fruitful discussions with our academic cooperation partners. I would particularly like to thank Professor Wolfgang Kunz for the technical and scientific discussions, for the support of this thesis and for the feedback I received as a doctoral student. I also thank Dominik Stoffel for his many improving hints.

The work was supported by the German Federal Ministry for Research and Development within the framework of the research projects VALSE, Herkules and Verisoft, which also supported our contacts in the academic world.

This fertile high-caliber environment goes back to Professor Wolfram Büttner. His strategic skill and deep technical insight have attracted many of us to work in this field. He kept this working group together for over 10 years under very changing conditions. He frequently initiated the important considerations that led to technical progress. This culminated in the startup OneSpin Solutions GmbH, which he founded. However, his share in this work goes far beyond business. Rather, he has helped me in countless conversations and many common lunches to develop the ideas and to pinpoint them, as well as to evaluate the technology in its practical relevance. For this I thank him, as well as for his suggestion on this thesis, the many feedbacks, also on previous versions, as well as for the overall support.

I would like to thank today's management of OneSpin Solutions for their support and discussions during the preparation of this work, especially Michael Siegel for his helpful comments.

A consequence of the industrial environment of this work is the patent protection, which was obtained or applied for for some of the concepts and inventions related to this work, e.g. for the completeness checker [Busch / Bormann 2005] and the generation of complete property sets for the verification of processor architectures against processor pipelines [Bormann / Beyer / Skalberg 2006]. I must point out that the unauthorized use of the patented parts of this work for research purposes or in industrial applications constitutes a breach of patent rights which would have legal consequences.

A company like OneSpin Solutions GmbH is driven by explicitly stated or anticipated customer requirements. I would therefore like to emphasize that I wrote this thesis independently of the company's opinion and strategy. The opinions and conclusions published here are my own and have been published independently of the official strategy of OneSpin Solutions GmbH. Accordingly, this work does not make any statements about development plans or goals of OneSpin Solutions GmbH.

In the course of this work, my family had to do without husband and father even more than before. Now and then my wife and children hat to listen to reports of particularly good or particularly bad progress. I thank Beate, Ines and Joachim, for patience, forbearance and support.

2 Functional verification

2.1 Formal Verification and the Verification Gap

The complaint about the "verification gap" has accompanied the semiconductor industry for at least 15 years. Specifically, this complaint means that nowadays approximately 70% of the R & D cost for an ASIC are not spent on the actual circuit design, but on the verification, i.e. the quality assurance of the design. Despite the enormous cost, the success of the verification is rather modest: On average, an ASIC project needs 1.7 respins where the lithography masks for the production are replaced by improved ones. The cost of respins in the current technology with structure sizes of 45 to 65 nm are in the range of 1 million euros. This high rate of failures is no surprise if one considers that the main workhorse of the verification is still simulation, although the simulation models are about 1 million times slower than the finished chips. Even if many computers are involved in simulations for months, only a few minutes of the lifetime of a device are examined during the entire verification phase. This does not allow testing all critical situations such that functional flaws remain in the design and lead to respins.

The problems culminate particularly in functional verification which shall ensure that the function provided to an end user is correct. Functional verification does not consider any physical interference effects or errors from the introduction of production-specific functionality such as test logic, clock tree balancing, etc.

Almost as old as the complaint about the verification gap is the dream that formal verification could close the gap, or at least help to reduce it. In fact, formal verification is very successful in the form of combinatorial equivalence verification [Filkorn 1992, Brand 1993, Kunz 1993, Kuehlmann / Krohm 1997, Lohse / Warkentin 1998, Bormann / Warkentin 1999, Hoereth / Müller-Brahms / Rudlof 2002] and has almost pushed out the netlist simulation that was used before.

But with combinatorial equivalence verification, it is only demonstrated that the circuit function is retained. It is not shown whether this function is correct in the sense of a specification. Therefore, combinatorial equivalence verification can only be used for process steps that are executed after the definition of the circuit function. Thus, combinatorial equivalence verification is not a functional verification.

2.2 Assertion-Based Formal Verification (ABFV)

Formal verification has repeatedly demonstrated how useful it can be for functional verification by making significant contributions to the design or verification of particularly complex building blocks such as processors [Beyer 2005, Buckow / Bormann 1993] or filters [Busch 1991]. But then it was used by highly specialized experts, who applied theorem provers, i.e. highly complex proof tools. Approaches to make theorem proving available to a larger user community have so far failed commercially [Bombana et al. 1995].

After 20 years of intense research in this field [Bryant 1986, Clarke / Emerson / Sistla 1986, Mayger / Harris 1991, Filkorn 1992] only the assertion-based formal verification (ABFV) is regarded acceptable. There are now commercial ABFV tools [Solidify undated, IFV 2005, Jasper 2007, 0-in 2005, Magellan undated, Conquest undated], which are based on the research tools of the 1990s [McMillan 1992, Bormann et al. 1995].

The experience with the commercial ABFV tools is however sobering. Until recently, ambitious design projects have still largely been carried out without the use of formal functional verification [Shimizu et al. 2006]. Raj Mitra, who heads the formal methods at Texas Instruments [Mitra 2008], recognizes the benefits of formal verification, but criticizes among other issues the low circuit complexity that the tools can handle which requires the verification engineers to develop a deep understanding of the circuit. Mitra also mentions the unreliability with which formal results can be achieved in a given period of time. This uncertainty arises because user-defined abstractions [Kroening / Seshia 2007] have to be tried and tested in order to obtain verification results. Mitra considers it self-evident that formal verification can not fully investigate all functionality of a circuit.

Whether ABFV will ever meet the hopes set for formal verification seems questionable against the background of statements by the architects of ABFV tools that today, after many years of intensive research, the tools still have complexity limits of around 1000 state variables [Jain et al. 2007]. This is not a particularly high improvement compared to the complexities of the late 1990s, when 300 state variables were reached on the computers at that time. It is questionable how much of the improvement since then are related to the research efforts in the field of formal methods and how much comes from the pure improvement of the computer performance and the use of larger memories. With Moores Law about the growth of the complexities of integrated circuits [Moore 1965] the development of ABFV did not keep pace at all.

Other authors complain about the disconnect between ABFV and simulation-based verification [Bailey 2007]. One indication of this is the uncertainty how much simulation can be replaced by the formal verification of a given set of assertions. Another problem is that the transaction-based view of today's simulation test benches does not find any resonance in the ABFV: Assertions usually investigate local aspects of the behavior of individual signals and ignore the overall picture that this signal behavior should be part of. This may be owed to the complexity limitations of the ABFV tools, but the disconnect extends even to the fact that the standard languages PSL [PSL 2004] or SVA [System Verilog 2005] of ABFV are opposed to a transaction-based approach by formal tools (see section 4.3.12). Furthermore, the user is only given case-by-case ideas about how assertions are actually to be set up [Foster et al. 2003, Foster / Krolnik 2008].

In addition, ABFV requires the development of deep circuit knowledge, although many verification engineers can not or do not want to be able to look into the modules to be verified. This is the result of a long-practiced division of tasks between verification engineers and circuit engineers.

2.3 Today's Work Split Between Design and Verification

Today's verification practice follows the methods already used 20 years ago in the design of the Alpha chip: Use of simulation, while the modules of the circuit are treated as black boxes. In order to detect a fault, it must be observable on the output signals of the module during the simulation.

This observation based quality assurance is also common in other fields of technology. However, it is often supplemented by a quality assurance that is based on the understanding of the implementation of the product and reasoning about the sensibility of the implementation. In the circuit design, this second procedure of quality assurance would correspond to a code review or the complete verification to be presented in this thesis. Codereviews are known as methods for achieving high quality standards [EN61508 2001], but are used very rarely.

The main weakness of an observation based quality assurance is that errors are only identified with certain probabilities. However, the circuit design has so far arranged itself with this. Circuit verification has been improved by methods with which the occurrence probability of the errors has been gradually increased. Random pattern simulation, hardware acceleration, or simply faster simulation algorithms have made it possible to simulate more in the same time span. By coverage criteria, input stimuli can be selected more carefully and simulate a larger set of situations.

However, every single chip is orders of magnitude faster than the simulation during its design phase, and millions of such chips may be put into operation. Hence, even errors with the smallest observation probabilities are revealed. As long as one is satisfied with gradual improvements in the observation probability of errors, errors will continue to occur in the field, at least those with a too low observation probability to discover them during the verification phase.

The striking advantage of the simulation-based black box verification is, however, that it allows a task split between design and verification: Ideally, the verification code (i.e. the test bench) is developed independently of the implementation and is therefore usable even if the implementation is completely changed. The verification engineer is not compromised by the knowledge of the implementation, and thus runs less risk of taking over wrong views from the designer and thereby accepting errors. Since the verification code is created independently of the implementation it is therefore available early in the design process. Ideally, it is already used during the circuit design for fast quality assurance of newly developed circuit parts.

This task split between design and verification is now deeply rooted: The verification tools and methods are entirely focused on the phenomena at the module boundaries. They deal with interfaces and the transactions there. Accordingly, the tools and methods are concerned with transactions, their generation and their identification. There are, for example, transaction-based reference models and scoreboards to compare transaction sequences.

Verification engineers look at circuits as objects that are described by the sequences of the transactions on their interfaces. The actual circuit description plays a subordinate role. This goes to the point that verification engineers do not understand circuit descriptions because they neither know circuit description languages (such as VHDL or Verilog) nor do they have an overview of common circuit structures. An understanding based quality assurance is not possible for such verification engineers.

Designers, of course, know the internals of circuit designs. However, they are supposed to provide their circuit designs quickly, so that the verification teams can begin their work. Quality issues are less important than the design speed. The designers pass the code early to the verification engineers, and then continue with the next modules. Errors are discovered later than if the designers themselves had carried out some quality assurance. The late discovery requires the designers to re-familiarize themselves with half-forgotten code. The error corrections become more protracted and uncertain.

This strong task split between designers and verification engineers and the associated lack of information is regarded by opinion-makers in industry as a serious structural problem [Kranen 2008].

All in all, RTL code is often difficult to understand. If the code was software, it would often fail to meet the principles of sustainable, maintenance-oriented development. This becomes obvious at company-internal IP blocks, which are developed for use in various ASICs. Such modules are maintained, i.e. they undergo several revisions, usually also by various responsible persons. During this maintenance, badly understandable code is problematic, because the revisions are then carried out on the basis of an imperfect understanding. This reduces the quality of the module with every revision.

If errors in the IP blocks then accumulate, this loss of quality is often attributed to the age of the testbench and its outdated technology, although the testbench was originally sufficient to ensure the quality of the first RTL version and the reuse was originally advertised as a measure to increase the quality.

2.4 Today's verification methodology and ABFV

The work split between the circuit design and the verification and the deep internalized practice of looking at modules as a black box in the verification process, leads to acceptance problems of ABFV. It starts with the fact that it is not clear who is to write and prove the assertions.

One class of assertions is characterized as designer assertions. These are assertions which do not actually state anything about the specified function of the module, but about individual implementation details, often simply about the local interplay of some code lines. Such assertions are written by designers. They are easy to prove: It is clear to the designer which local aspect of the function is to be checked, the formalization by an assertion is simple, and the relevant circuit part is small so that there are no complexity problems when using the formal tools. But such assertions also do not verify much, even if they occur in hundreds or thousands.

A more interesting class of assertions are high level ssertions, which are used to prove specified aspects of the function of a module. In the specification, these aspects always look very succinct, but the devil is in the details. In the implementation, the developer may have inserted intended exceptions to the concise formulation of the specification. These exceptions must either be incorporated into the assertion, or they must be related to environmental constraints, which in turn must be formalized. The intended exceptions, however, have to be distinguished from the unintended exceptions, because the latter point to the errors that the verification should reveal. Both the identification of the intended exceptions as well as the differentiation of unintended exceptions requires knowledge of the circuit.

Where circuit knowledge is not to be acquired, the verification of high-level assertions is limited to those that can be purchased as a prefabricated verification IP. Such assertions are often related to protocol compliance. This verification task is also well addressed by simulation, because many approaches are based on the extraction of transactions.

But even if a verification engineer has set up a first trustworthy formalization of a high-level assertion, complexity problems can still occur, and they are the worse the more a high-level assertion indicates the correctness of the circuit. In order to master such complexity problems, the verification task is split or a more abstract version of the task is prove. Both can not be carried out successfully without the knowledge of the circuit and this probably leads to Raj Mitra's complaint [Mitra 2008] about the unpredictability of the effort for a formal verification.

All in all, it is not possible to formally investigate high-level assertions without knowing the circuit to be examined. This is why ABFV of high-level assertions is somehow sitting between the chairs: developers do not want to invest effort to formalize the high-level assertions properly, and the verification engineers lack the time, experience and knowledge to get into the circuit functionality. Moreover, as long as the simulation is the main source of insight during the verification, effort and benefits of high-level assertions are even in an unfavorable ratio. Accordingly, the concept of verification of high-level assertions is not widely used to date.

2.5 Complete Verification

The work to be presented in this dissertation takes a different approach than the ABFV. Firstly section 3.2 presents a generally applicable idea of what is to be verified: Operations of the circuit are introduced as the central means to structure the verification. Operations are parts of the execution of the circuit over a shorter period of time. They form useful building blocks of the functionality. The abstraction of circuit behavior by operations is so similar to the abstraction of interface behavior by transactions [Cai / Gajski 2003] that some authors write of transactions of a module instead of operations. For example, an operation of a bus interface describes the generation of a transaction on the bus, operations of processors describe the fetching and execution of an instruction in the processor pipeline, and the operations of an arbiter describe the arbitration cycles.

Many operations can be represented by generalized timing diagrams (see section 4.3.11) [Bormann / Spalinger 2001]. An example of generalized timing diagrams is shown in Figure 7. These timing diagrams are easily described in the language ITL [Siegel et al. 1999] (see section 4.3) or expressed on the base of the SVA library Tidal [Bormann 2007] as temporal properties. These temporal properties are called operation properties.

A property checker proves that the circuit performs the operation as described in the operation property. For this proof, powerful tools [OneSpin undated] are available, which prove properties over modules of medium size in the range of minutes. The largest module tested with this approach consisted of significantly more than 100,000 lines of RTL code.

The property verifier verifies operations by examining them from every state. However, some of its performance is bought by the fact that the property is also verified from states which can not occur during the operation of the circuit. If such unreachable states lead to counterexamples, they must be excluded by means of reachability conditions. Such reachability conditions can sometimes be determined automatically. Where automatic procedures fail, hints from the verification engineer are required. These hints can be developed from the circuit knowledge or they are based on indications from the designer of the circuit. Such hints will be justified later within the framework of the methodology, such that erroneous hints will not corrupt the verification. This approach makes it possible to transform the verification problem into a SAT problem and then to benefit from the high performance of SAT proofs. This proof approach is called Interval Property Checking (IPC).

The generalized timing diagrams suggest a kind of jig saw puzzle game [Bormann / Spalinger 2001]: If an input trace to the circuit is given, one can try to predict the circuit behavior by only using operation properties. If this allows unique determination of an output trace for each input trace, the set of operation properties is called complete. It then examines every aspect of the circuit function. The description of an automated completeness checker is part of this thesis. A complete set of proven operation properties provides an equivalent, more abstract circuit description and can be the basis for further abstractions up to a variant of a transaction model that is suitable for further formal verification. Thus, the abstraction level of the circuit description for formal verification comes close to that of the circuit description for simulation-based verification.The goal of the verification is a complete circuit description. It is therefore referred to as complete verification. Complete verification thus represents an understanding based quality assurance in accordance with the characterization in section 2.3. Occurrence probabilities for errors as they impede the simulation, are not an issue here. If an error is observable, it will be found.

To automatically prove the completeness of a set of operation properties a completeness checker was developed. At the submission date of the corresponding patent [Bormann / Busch 2005] it was the the first of its kind [Claessen 2006]. It solved a core problem of the formal functional verification ([Katz et al. 1999, Hojati 2003, Hoskote 1999]): When do you have a sufficient number of assertions formulated to fully cover the circuit functionality? Chapter 5 describes this completeness checker, proves the approach and illustrates it with an example.

The completeness checker examines operation properties over a circuit part in which the corresponding operations are carried out successively. Therefore, there are no concurrent operations. Such a circuit part is called cluster. Of course, larger circuits include concurrent operations. For complete verification, such circuits are broken down into multiple clusters. The composition of complete verification of the clusters to a complete verification of the overall circuit is only possible under certain requirements to modeling and so-called. integration assumptions, which contain conditions and assurances about behavior at the interface of a cluster. Chapter 6cdescribes and proves the conditions, and illustrates them by examples.

2.6 Application of Complete Verification

Since 1999, verification projects with complete verification and similar, less mature predecessors approaches have been in regular productive use at Siemens, Infineon and other semiconductor manufacturers and system houses. A selection of projects is listed in Tabelle 1. Almost all circuits that were verified in these projects functioned properly after the fabrication of the respective ASICs or the burning of the corresponding FPGAs.

In the few projects where circuit faults were overlooked, the escape was traced back to human failure in the application of the underlying methodology: For example, wrong constraints restricted the input traces too much and thus masked errors, or circuit errors were wrongly accepted by inaccurate descriptions of the expected output behavior.

One of the reasons for the high resultant quality of the approach is that – whereever possible – user input is treated as a hint to simplify proofs, but needs to be cross checked in turn. This minimizes the probability that human error impacts the final circuit quality.

The procedure described here is capable to take the full verification task for a module. Contrary to ABFV, complete verification is therefore no complement to simulation-based verification on module level, but an alternative. In extreme cases, complete verification reduces the application of simulation to the exploration of the hardware design to provide a designer with a feeling for the newly designed circuit. However, it is up to a verification responsible, how she or he defines the mix between simulation and complete verification. But now, similar to the quality assurance in other engineering disciplines, a proper mix of observation and understanding based quality assurance (in the sense of paragraph 2.3) can be configured.

This opens up considerably higher effort budgets for complete verification than for ABFV. As part of these effort budgets a deep familiarization with the circuit design is possible with a favorable ratio of costs and benefits. This familiarization with the circuit design is simplified by the structuring of the approach. Since the verification engineer is focused on one operation after another, he / she will not simultaneously be confronted with all the problems, as is the case with a verification by High Level Assertions.

2.7 Simulation-Based and Complete Verification

As stated above, complete verification is an independent approach for circuit verification. It is therefore in competition with simulation-based verification approaches for module verification, such as Coverage Driven Random Pattern Simulation [Bergeron et al. 2005]. In comparison, complete verification has a number of attractive characteristics, which should be be sketched briefly in the sequel. After a detailed presentation of complete verification these benefits will be described in section 3.6 in more detail.

Complete verification reaches the highest quality for the modules that were verified with this methodology, while other verification procedures leave errors in design, because the errors were either not stimulated, or because they were overlooked by the checking mechanisms.

The quality gain is not only realized in the implementation but also in the specification, because the completeness checker points to specification gaps. The competition processes have no way to structured identification of specification gaps.

To prepare and execute complete verification, a process has been defined, which starts with the planning of a verification project and provides recommendation about how to develop the operation properties. Good project planning leads to reliable execution times. The quality of project planning only affects the punctuality of the project, but not the quality of the circuit after complete verification. For simulation-based methods, there are also thoroughly defined processes. Here, verification requires the collection of all the verification objectives. The quality of the verification plan affects both the schedule and the final circuit quality.

The end of a complete verification project is reached when a logical, i.e. uncompromisable termination criterion is met, which is automatically checked by the completeness checker. Simulation-based verifications are terminated on the base of heuristic criteria such as coverage figures and fault finding curves.

Productivity and termination criteria are interrelated. Unskilled execution of complete verification or application on poor RTL code leads to extended verification times but does not impact the final circuit quality. Different to that, the termination of a verification by simulation is always a matter of interpretation. But experienced users verify well written RTL code with complete verification faster than through simulation, despite the additional quality gain. A rough estimate is a productivity of 2000 to 4000 lines of RTL code that can be examined with complete verification in a person month. In single cases, 8,000 lines per person month were reached.

Besides the unusual approach the biggest acceptance hurdle of complete verification is its close relationship with the concrete implementation, which makes verification experts suspect that errors might escape because misinterpretatios are consistently inserted into RTL and verification code. But experience shows that, despite this close relationship, such escapes are very rare. But implementation changes often require changes also in the verification code. This is not a problem with incremental changes as part of a design and verification project. But strong changes may result in substantial effort, whereas in simulation-based verification effort can be saved by reusing the testbenches. But such effort comparisons need to observe, that re-use of a testbench would need to be complemented by the adjustment of the coverage criteria and the stimuli, if the new circuit shall have the same quality as the old one. This adaptation effort is however often saved.

2.8 Complete Verification and ABFV

The fundamental difference between ABFV and complete verification arises from their different roles in a verification project: ABFV supports a verification which is carried out mainly by simulation while complete verification can handle the full verification task.

But both approaches have in common the central usage principle: Assertions and properties are formalized, they are verified with an automatic proof tool against the circuit, the proof tool constructs a counter-example, this is analyzed by the user and the analysis leads to the identification of a circuit failure or to adaptation of the assertion or property.

ABFV makes no requirements about the form of assertions, while complete verification requires a certain structure of the properties and how they should be formalized. The circuit quality that can be achieved with ABFV thus depends to a greater degree on skills and ingenuity of the formal verification engineer. But it requires little training to write and prove assertions. For each type of circuit and usually also for each implementation strategy an own set of assertions will be developed, without there being a guiding idea of the general structure of an assertion. Accordingly, there is a lot of literature about assigning useful assertions to circuit classes on a case by case base. For the complete verification the methodical base is wider, and then the verification tasks arise in a natural way.

Because of the diversity of assertions, universal verification tools are available for it. Assertions can be of widely varying complexity. The circuit itself plays a big role, so that even syntactically similar assertions can lead to very different resource requirements of the provers. Users either get accustomed to prover failures without apparent reason, or they develop a feeling for the resource requirements of a prover and start it only on appropriate assertions. Unsuitable assertions can be further investigated by simulation.

The operation properties of the complete verification can be proven with IPC which leads to moderate resource demands, at least in comparison to the circuit size. The prover rarely fails to produce proof or a counter example. If it does, the operation in question needs to be subdivided into smaller operations.

Assertions play an important role also in complete verification, for example as part of a specification. But they also come into play to summarize interim results. Some assertions are easily verified by means of ABFV. However, where this fails, assertions can be proven by showing that they are met by all operations of a complete set of operation properties.

Tabelle 1: Project with complete verification

Funktion	**Projekt**
Prozessoren	TriCore2 (super scalar, 32 Bit, automotive) [Bormann et al. 2007] Multithreaded network processor [PPv2 2008] IEEE floating point processor

	Weakly programmable IP [Loitz et al. 2008]
Peripherals	Infrared interface One-Wire Interface Touch Screen Measurement Interface USB Master Interface, Counter UART Interrupt Controller A/D Converter Controller Flash Card Data Port Camera Interface Multimedia Card Interface Configurable Arbiter DMA Controller
Bus Interfaces	Bus Arbiter AHB (Master-, slave interfaces, bridges, multilayer bus [Bormann/Blank/Winkelmann 2005]) Interfaces with proprietary protocols Protocol adaptation of a legacy processor [Bormann/Spalinger 2001] CAN, LIN, Flex Ray, AXI SRC Audiobus Interface Communication infrastrcuture of a massively parallel multi processor system HDLC Controller [Bormann/Blank/Winkelmann 2005]
Memory Control-lers	SDRAM Controller Advanced Memory Bus SATA Caches Flash Memory Interface
Error Correction	ECC Robustness against distortions of board to board communication
Telecom	AAL2 Termination Element Address management of an ATM Switch Sonet / SDH Frame Alignment [Thomas et al. 2004] Path Overhead processing of a Multi-Gigabit-Switch DSP Coprocessor-ASIC for correlation computation [Winkelmann et al. 2004]

3 Overview over Complete Verification

A special feature of the approach of complete verification outlined in section 2.5 is its similarity to transaction-based verification, i.e. the currently used simulation-based verification approach [Bergeron et al. 2005]. With respect to this similarity this approach is presented in the sequel from an application perspective.

A brief outline of transaction-based verification by simulation is given in section 3.1. The complete verification is a symbiosis of the operation based circuit description presented in section 3.2, the proof method “Interval Property Checking” (IPC) described in section 3.3 and the completeness checker of section 3.4, which is complemented by the compositional approach of section 3.5 to a process which has in principle no size restrictions. The pros and cons of the practical application of complete verification are discussed in section 3.6.

3.1 Transaction-Based Verification by Simulation

3.1.1 Transactions

RTL describes calculation steps and the interaction of these calculation steps. Each operator in an RTL description is a calculation step, and control statements enforce proper interaction of these calculation steps.

Functional verification checks whether the individual calculation steps in RTL interact as requested by a specification. In this respect functional verification corresponds to numerous other verification tasks: In the design of cells from transistors it is verified whether the interaction of the transistors implements the desired function of the cell. During verification of netlists against RTL it is examined whether many such cells interact properly to implement coarser calculation steps of the RTL (such as multiplication).

If simulation is used for functional verification of a module, main attention is put on the behavior of the interface signals of the module, that are those signals which connect the modules and transmit e.g. read or write requests or acknowledgments. This behavior appears at first glance confusing. But on closer inspection, it turns out that this behavior follows a few basic patterns. These basic patterns are obtained when the interface signals are grouped in busses, and when each bus is considered separately. The basic patterns are then called transactions of these buses. The transactions describe the request of module activities or services provided by other modules.

The available transactions, the allowed sequences, and their implementation by sequences of values of signals of a bus is given by a protocol specification. The concept of transactions is fundamental for today's simulation-based functional verification.

Transactions have parameters. Essential parameters are data, addresses or communication direction. Other parameters can give information about the simulation time at which the transaction begins and ends and the time between synchronization events defined by the protocol. Moreover, there may be protocol-specific parameters such as the type of a base transaction of the AHB protocol [AHB 1999].

The usage of the word “transaction” is not unique [Cai / Gajski 2003]. Instead there is a hierarchy of transactions. E.g., a basic transaction of the AHB protocol can be subdivided into address and data phase, preceded - depending on viewpoint – even by an arbitration phase.

These phases are transactions at a lower level of abstraction. On a higher level of abstraction, several basic AHB transactions connect to a burst transaction.

3.1.2 Testbenches

The testbench of a simulation-based functional verification includes all code which has been developed to verify the given circuit. Figure 1 shows a typical transaction-based verification testbench, which is described below.

Directed or random tests generate sequences of signal values, called waveforms, on the buses of the circuit. The circuit code might contain assertions. These assertions can examine any signal in the code. However, the bulk of the verification task is done by examination of the interface signals of the module. To that end, the testbench converts the waveforms of the interface signals into series of transactions by so-called transaction extractors. Each transaction is described by its parameter set. During the extraction the extractors also check the compliance of the waveforms with the protocol specification, e.g., whether write data is indeed kept stable until the write was confirmed. The examination of the module functionality is then reduced to the investigation of the transactions. E.g. the test bench checks the parameters of the transactions, or whether the sequence of transactions is correct.

A distinction is made between incoming and outgoing transactions of the module. Incoming transactions are those in which a neighbor module has taken the initiative to a data transfer.

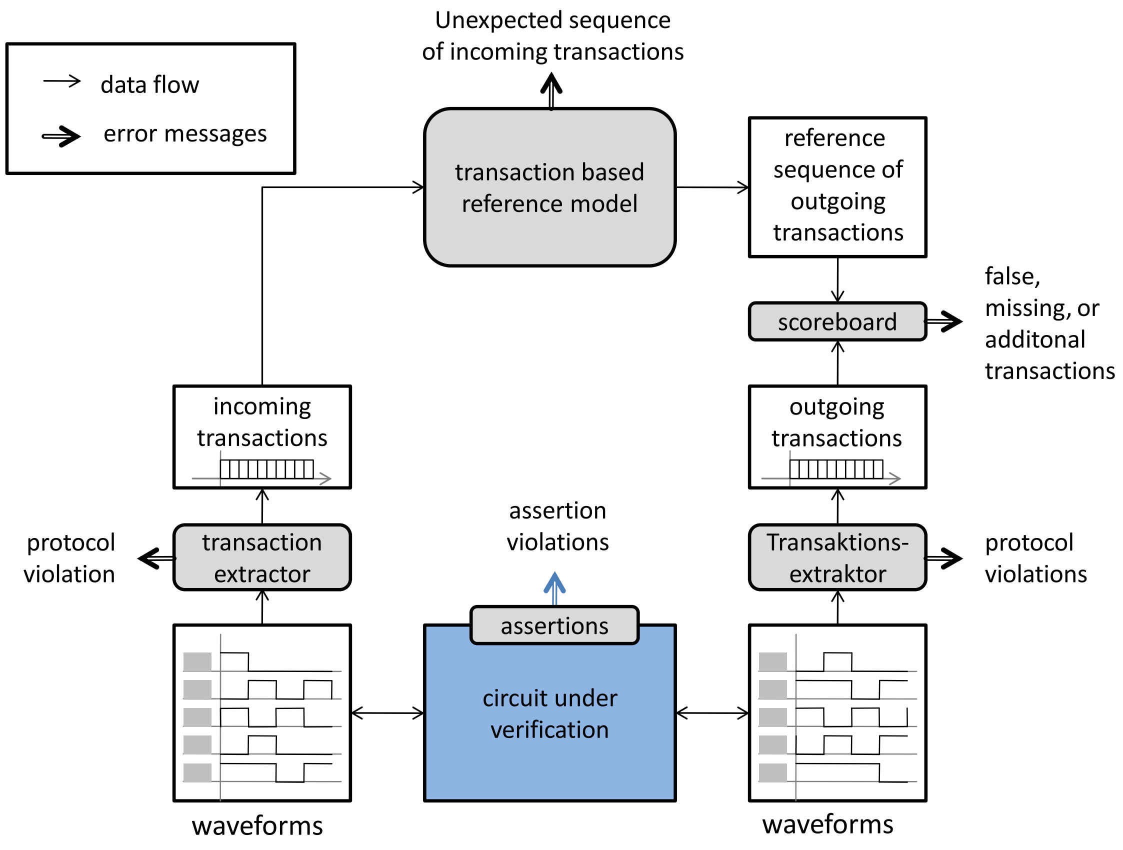


Figure 1: Transaction based testbench for simulation

Outgoing transactions are those in which the module under verification has taken the initiative to a data transfer. If a bus interface of a module receives incoming transactions, it is called slave or target interface, when it produces outgoing transactions, it is called master or initiator interface.

The terms "incoming" and "outgoing transaction" do not describe the direction of the data transfer. For write transactions, data flows from the master to slave interface, and for read transactions from the slave to the master interface. Some modules have only slave interfaces, such as memory modules, others may only have master interfaces, such as processors, or they have both types of interfaces, such as bus bridges.

In a typical transaction-based simulation verification, the incoming transactions of the circuit under verification are also processed by a transaction-based reference model that provides reference data to incoming read transactions and generates sequences of outgoing reference transactions. Such a model that operates on transactions is called circuit Transaction Level Model (TLM) of the circuit.

A sequence of outgoing transactions generated by the TLM need not exactly match the sequence of outgoing transactions that the transaction extractors generate from the waveforms at the interface of the examined module. Instead, the testbench will check for a suitably defined similarity. Testbench components that implement this similarity are called scoreboards. These compare the sequence of transactions that is generated by the implementation with the sequence of transactions from the TLM model and thus verify the module. A scoreboard can e.g. be tolerant wrt. local rearrangements of the series of transactions, but reports an error when the module fails to produce a transaction or produces incorrect or additional transactions.

The testbench sketched in Figure 1 is particularly suitable for the test of a module within a system simulation, in which the module is stimulated by neighboring modules. In this case, the transaction extractors actually extract the transactions from the signal behavior between modules.

Contrary to that, the verification of an isolated module requires that the testbench generates incoming transactions. To that end the generators on the master side of the circuit to be tested are described abstractly. They consist either of routines that e.g. read a text file with the parameters of transactions to be executed and pass them to the TLM and a convertor into the signal level description. The generators can also determine the parameters of the transactions at random.

Coverage measurements will determine how intensively the circuit is examined by the verification, i.e. how many operating situations were simulated. Coverage may be determined on the base of the RTL code of the circuit under test. On the other hand, straight functional coverage, i.e. the simulation of certain functional processes are often derived from the parameters of the transaction, so that the coverage measurement uses the one level of abstraction, which is particularly suitable for the given coverage condition.

3.2 Levels of Abstraction of Complete Verification

In this section the approach of complete verification is presented from an application perspective. Figure 2 shows the associated levels of abstraction and the refinement relations between them.

The role of the transaction-based reference model in the simulation is taken by a transaction automaton, which will be introduced in section 3.2.2. The transitions of the transaction automaton are triggered by incoming transactions and they describe which outgoing transactions are produced. The transaction automaton does not model how the incoming and outgoing transactions are implemented by signals and clocks.

A refinement of the transaction automaton represents the transactions through their signal behavior. The so refined automaton is called operation automaton and corresponds to the interaction of transaction extractors, a special scoreboard and the TLM reference model. The operation automaton is described in section 3.2.3.

A circuit is verified by demonstrating that its operation automaton meets the specification and has the same input / output behavior like the module under test. For this purpose a mapping of the states of the circuit to the states of the operation automaton is required, similar to the state mapping of combinatorial equivalence verification. This state mapping leads to operation properties, which are discussed in section 3.2.4. In section 3.2.6 the abstraction levels of complete verification will be compared with other abstraction techniques commonly used in circuit design and formal verification.

The proof of the operation properties with IPC is covered by section 3.3 together with the related reachability issues from a user perspective.

In carrying out complete verification in practice, the verification engineer always focuses on the verification of one operation at a time. Technology and theory to answer the question whether the entire circuit function has been verified, is presented in section 3.4. Initially this

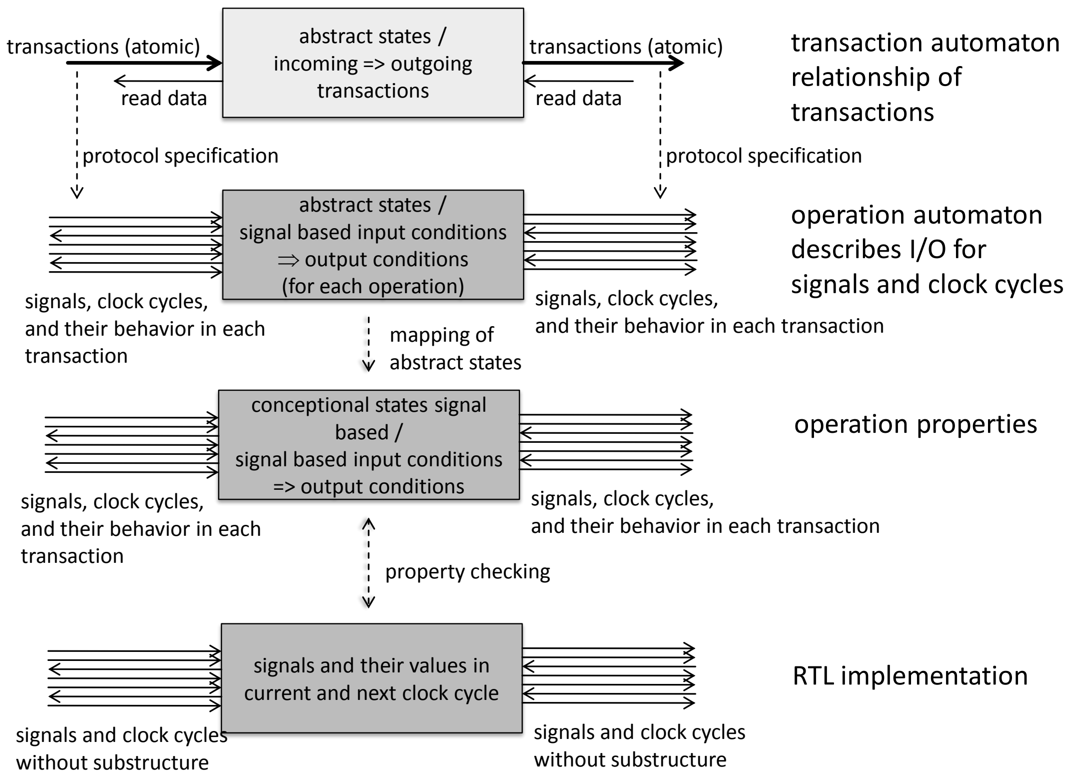

transactions (atomic)
abstract states / incoming => outgoing transactions
transactions (atomic)
read data
read data
transaction automaton relationship of transactions
protocol specification
protocol specification
abstract states / signal based input conditions ⇒ output conditions (for each operation)
operation automaton describes I/O for signals and clock cycles
signals, clock cycles, and their behavior in each transaction
signals, clock cycles, and their behavior in each transaction
mapping of abstract states
conceptional states signal based / signal based input conditions => output conditions
operation properties
signals, clock cycles, and their behavior in each transaction
signals, clock cycles, and their behavior in each transaction
property checking
signals and their values in current and next clock cycle
RTL implementation
signals and clock cycles without substructure
signals and clock cycles without substructure

Figure 2: Abstraction levels and their relationship

concerns only circuit components that perform operations in sequence, which are called clusters. The completeness checker in section 3.4.3 detects verification gaps in sets of operation properties about clusters, or it proves the absence of such gaps. Section 3.5 extends the notion of completeness of clusters to arbitrary concurrent functionality because it can be composed of multiple clusters.

3.2.1 Example

The concept introduced in the sequel will be illustrated by an example that relates to a memory interface. As Figure 3 shows, the interface provides the data transport between a processor and SDRAM. SDRAMs are memory blocks the memory locations of which are arranged in a matrix. Hence they have a row and a column address. The column address is usually represented by the upper half of the signal of the address bus, the row address by the lower half.

The control inputs of an SDRAM are usually called cs_n, we_n, ras_n and cas_n and remind of names such as chip select or write enable. But their mode of operation is better understood when the four signals are regarded as a command bus to the SDRAM. It conveys commands such as, e.g., "row activate", with which an address row is activated, as indicated by the condition

```
      cs_n = 0 and ras_n = 0 and cas_n = 1 and we_n = 1
```

Accordingly, there are read and write commands that cause the transmission of data bursts, a stop command that terminates the transmission of a burst, a precharge command to end the activation of a line, i.e. the line is closed and a NOP command, in which nothing happens.

Reading or storing a piece of data is requested by the processor through the activation of the request signal req and an appropriate value on the rw signal. The address is then found on the signal address and the write and read data on the signal rdata or wdata. The processor must keep its output signals stable until they are acknowledged by the memory interface by activating the ready signal. This activation also validates the read data.

Reading or storing usually requires multiple such commands to the SDRAM, which successively activate a row of the memory array, then access it with a burst read or write to the col-

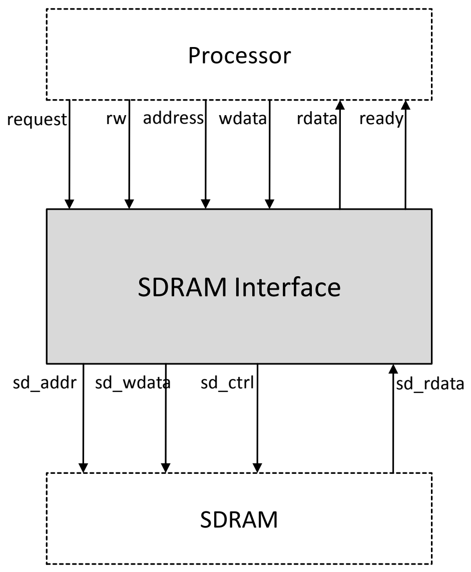


Figure 3: SDRAM interface and its system integration

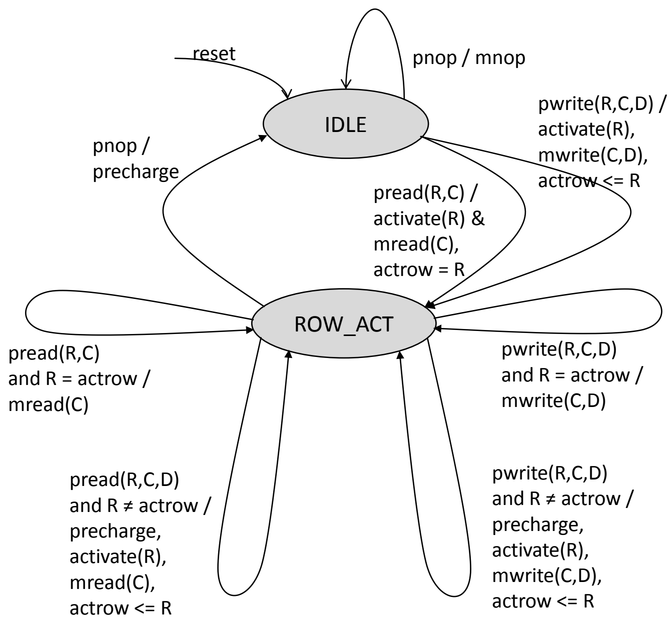
umn address of the memory location, and finally close the memory row. The corresponding commands of the command bus are called "row activate", "read" or "write", and "precharge". The commands are are accompanied by values on the sdram_addr address bus and the write data bus sdram_wdata, or they make the SDRAM provide valid values on the read data bus sdram_rdata after a number of clock cycles. The individual SDRAM commands must have certain, memory-type-dependent time distances from one another. After activation of a row several read and write operations can be performed on the elements of this row without closing and re-enabling this row.

3.2.2 Transaction Automaton

Starting point of complete verification is a transaction-based automaton representation of the specification, which is called transaction automaton. This is quite similar to the TLM reference model of the simulation-based verification. The automaton processes incoming transactions and generates related sequences of outgoing transactions. Differences in the levels of abstraction of TLM-reference models and transaction automata are discussed in this section.

The states of the transaction automaton are called "abstract conceptual states". Similar to the division of circuits into control and data paths the abstract conceptual states of transaction machines consist of a control portion and a data portion. The control part of the state is called important abstract state of the transaction automaton. The data portion is called visible state because this is how the circuit stores the data that is visible in the specification. The variables in which the visible states are stored, are called visible registers.

Figure 4 shows the transaction automaton of the SDRAM interface. There are two important states, namely IDLE and ROW_ACT. The latter reflects the fact that a memory row is activated, and subsequent access operations to the same memory row can therefore be carried out more quickly. In state ROW_ACT there is also a visible state actrow which gives information about the currently open memory row. In idle state this variable is meaningless.



reset
IDLE
pnop / mnop
pwrite(R,C,D) / activate(R), mwrite(C,D), actrow <= R
pnop / precharge
pread(R,C) / activate(R) & mread(C), actrow = R
ROW_ACT
pread(R,C) and R = actrow / mread(C)
pwrite(R,C,D) and R = actrow / mwrite(C,D)
pread(R,C,D) and R ≠ actrow / precharge, activate(R), mread(C), actrow <= R
pwrite(R,C,D) and R ≠ actrow / precharge, activate(R), mwrite(C,D), actrow <= R

Figure 4: Transaction automaton of the SDRAM interface

In general, a transaction automaton executes a state transition depending on the abstract conceptual state and - if the module has slave interfaces - from incoming transactions. During this state change the automaton determines the read data for incoming read transactions. Further, the automaton generates outgoing transactions if the module features a master interface. The concept of time of a transaction automatorn is therefore event driven.

When a transaction automaton is to be displayed by the transition graph of an automaton, the common representation of circuits with control and data content is used: Node of the transaction graphs are only the important states and the behavior of the data path is represented by appropriate expressions that are either part of the conditions under which a transition is carried out or determine the data or addresses that are issued by the circuit during the transition. The remainder of this presentation will work with this coarsened view in which the transaction machine often contains relatively few important states and few transitions.

The graph of the transaction automaton of the SDRAM interface is shown in Figure 4. In the illustration of the transaction automaton it has been assumed that the processor bus can execute the transactions pread (R, C), pwrite (R, C, D) and pnop which may occur in any order. R is the row, C the column address of the memory and D stands for the write data. For the sake of simplicity the read data remains unmentioned. The memory bus between the interface and the SDRAM can execute the transactions mnop, activate (R), mwrite (C, D), mread (C), precharge, which should contain the respective SDRAM commands while adhering to the associated timing conditions. The single transaction is characterized by an expression of the form <condition> "/" <action>. The <condition> must be met before the transition is executed, and the execution of the transaction leads to the activities <action>. The transactions specified are sent sequentially, and the visible registers get assigned the new visible state.

In Figure 4 there is e.g. a transition that is performed when the SDRAM interface is in the abstract conceptual state ROW_ACT when a pwrite (R, C, D) transaction is on the processor bus, and if the row address R is equal to the address actrow that is part of the visible state and determines the address of the activated memory row. In this case, the SDRAM interface produces a transaction mwrite (R, D) to the SDRAM and further remains in the abstract conceptual state ROW_ACT without changing the visible state actrow.

Each transition of the transaction automaton is related to a verification task about the module under test: It has to be shown that if the actual module is in a state that implements the important start state, and if it receives the incoming transaction of the transition, and if the condition between visible start state and incoming transition is satisfied, it generates the appropriate outgoing transaction with the appropriate data and finally enters the appropriate new important state and adapts the visible state appropriately.

3.2.3 Refinement to Operation Automaton

To investigate these verification tasks, the relationship between the transaction automaton and the input / output behavior of the circuit under test must be clarified. To that end the transactions that were previously considered as atomic must be expressed by the behavior of the input and output signals of the module to be tested. The essential information for this refinement comes from the specifications of the protocols of the slave and master interfaces. These specifications define what behavior of input signals the module may expect from its environment and how the output signals of the module should behave in releation therewith. This refinement step corresponds in simulation based verification to the extension of the the TLM reference model by the transaction extractors.

After the protocol specification has been introduced, the description is initially independent of a concrete implementation and does not specify when the module under verification produces the synchronization events that it should produce according to the protocol specification, and it does not specify the temporal relationship between incoming and outgoing transactions. Such degrees of freedom are expressed in the simulation-based verification by the scoreboard that accepts the temporal relationships allowed by the specification and rejects the others. For formal verification, these degrees of freedom are difficult to express. The approach presented is therefore slightly different: It requires the timing relationship implemented by the RTL.

By introduction of this information into the transaction automaton, a refined automaton is generated, which is called Operation Automaton. Its states are still the abstract conceptual states of the transaction automaton. But the operation automaton describes the input / output behavior of the circuit precisely for signals and clock cycles. A transition of the refined automaton is triggered when a condition is met about the input signals of the circuit, and they generate the appropriate behavior on the output signals, which is again described by a condition. In general, both conditions are sequential and overlap in time.

A transition of the operational automaton is thus characterized by an abstract conceptual start and end state and conditions about the concrete input and output signals. Such a transition will be referred to as an operation.

On top of all this, the refinement defines the temporal relationship between successive operations. To that end every operation is assigned a symbolic start time point t and a symbolic reference point T_{ref} at which all subsequent operations begin. The reference time point of an operation is determined by a time offset relative to the start time t. This time offset may depend on the visible state at the beginning of the operation, as well as on the behavior of the input signals.

In the context of a verification according to Figure 1 the concept of an operation automaton combines the roles of the Transaction Level Model, the scoreboard and the transaction extractors. The operation automaton could for example be executed by simulation in parallel with the module to be verified. Then, the verification would be to check whether the outputs of the module to be verified meet the conditions of the output signals generated by the operation automaton.

The refinement of the transaction automaton can be chosen to be so accurate that the operation automaton assigns each input trace exactly one output trace. In this case, the verification checks the equivalence between the operation automaton and the circuit to be verified.

Figure 5 presents the operation that results from the transition of the transaction automaton at the bottom left in Figure 4. The protocol of the processor bus is the simple request-ready protocol that was already introduced above. After the activation of the request signal all the other signals must be kept stable until the SDRAM acknowledges the access with a $ready$ pulse and simultaneously validates the values on the $rdata$ signal. On the SDRAM bus a precharge, activate, read and stop command are issued under the required time conditions. The stop command is required because the read command would otherwise create a burst of read data.

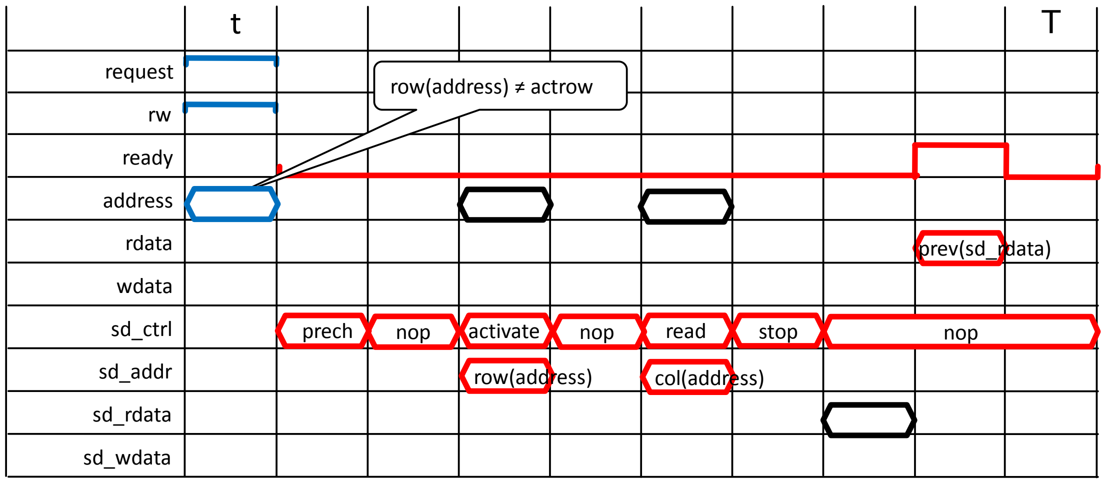


Figure 5: Operation "Read with change of memory row"

In the representation of the operation by Figure 5, the input condition of the operation is shown in blue. It contains the comparison between the row address of the transaction and the visible state. The description of the output signals is shown in red. The black lozenges mark signal values that are referenced elsewhere in expressions and describes itself neither an assumption nor a commitment.

This representation also describes the output of read data to the processor via the signal *rdata*. From the values of the signal that are marked black it is clear that the operation takes advantage of the stability of the processor bus signals until the *ready* pulse. Further, the description contains the determination of the reference time point T_{ref}, which is $t + 9$. In addition to the conditions on the input and output signals corresponding to Figure 5, the operation is characterized by its important start and end state row_act, and by the visible state actrow, which is set to the new row address in the course of the operation. The information about the conceptual state is not represented in Figure 5.

3.2.4 Refinement to Operation Properties

If the just described equivalence verification between operation automaton and actural RTL circuit is to be carried out with the quality of formal verification, additional measures are needed in order to examine practically relevant circuit sizes. For the approach described here, the abstract conceptual states are related explicitly to the states of the circuit and thereby the equivalence comparison is partitioned.

This corresponds to the procedure for combinatorial equivalence verification [Filkorn 1992, Kuehlmann / Krohm 1997, Lohse / Warkentin 1998, Bormann / Warkentin 1999, Hoereth / Müller-Brahms / Rudlof 2002]: There the original verification task is also to provide a proof that the circuits have equal input / output behavior. This is initially a sequential verification task, which needs to consider input and output traces of any length. This sequential verification task, however, can only be solved with algorithms that already fail on fairly small circuits. Therefore, a preprocessing step relates the two circuit descriptions by mapping the state signals of one circuit description to the state signals of the other. Through this mapping the original verification task is divided in time: The verification assumes same states at any initial time t and tries to prove the equality of the state signals at time t+1 (and of course also of the output signals). This decomposed verification task can be treated with SAT algorithms which are quite powerful and thus allow the examination of large circuits. The reduction to a combinatorial problem is admissible because design steps like synthesis, netlist optimizations,

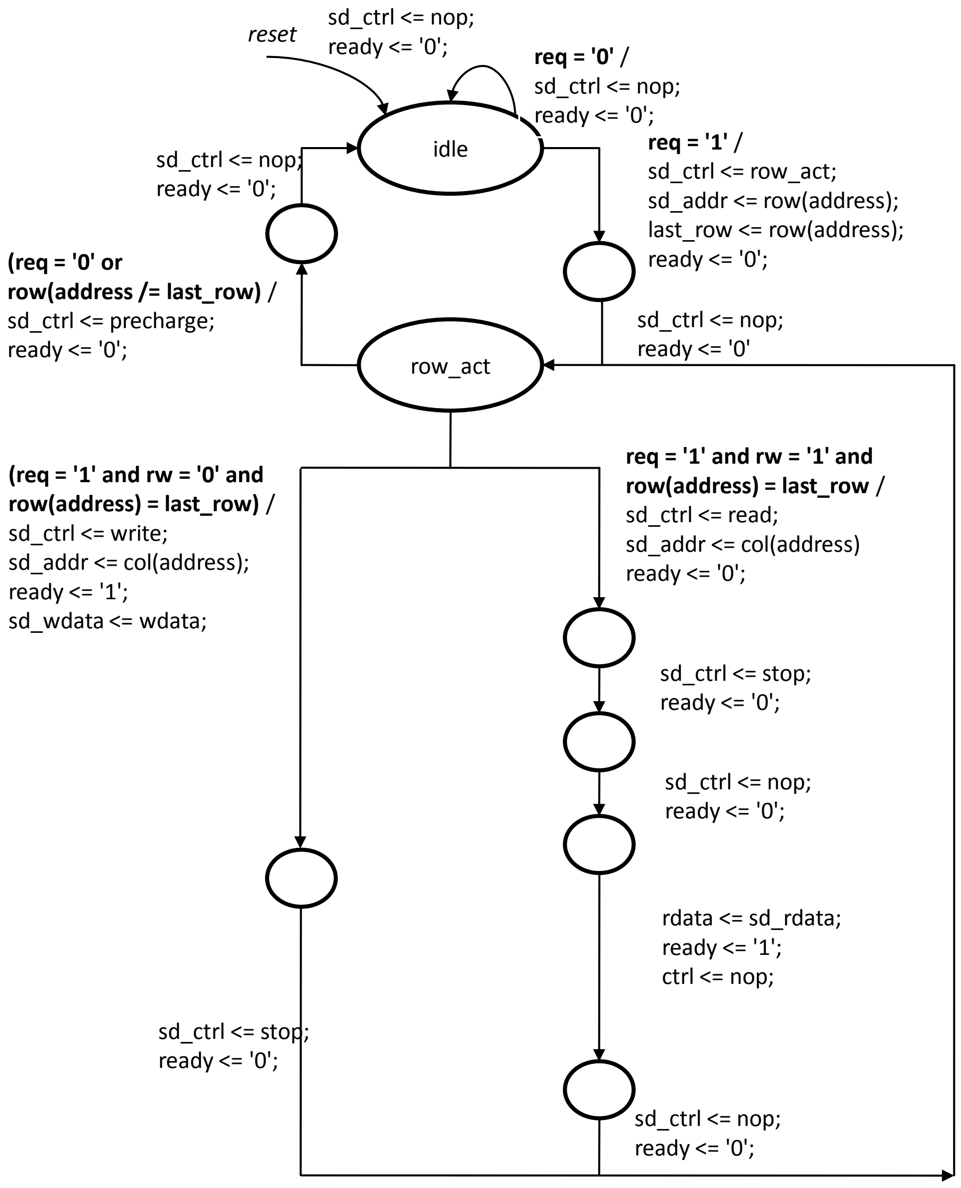


Figure 6: Implementation of SDRAM interface

insertion of test logic or last minute changes should not change the state encoding of the circuit.

The power of this approach is in its pessimism. Every now and then it returns negative comparison results, although the circuit descriptions are equivalent. But if the method proves two circuits equivalent, then they reliably have the same input and output behavior. Wrong mapping of state signals can not cause false positive results, which might mask an error. However, false negative results are possible by which circuits with actually the same input / output behavior are blamed to be different. Then counterexamples are generated that contain hints to the wrong state mapping.

The approach of complete verification is similar: The conceptual states of the operation automata are mapped to states of the circuit to be tested. By this the operation automaton is refined to a machine whose states are given by conditions about the concrete signals of the implementation. These state conditions are called concrete conceptual states. The advantage of this refined automaton is that each transition and thus every operation can be compared with the RTL in a separate verification task. This verification task assumes that the circuit is in a concrete start state of an operation, and proves that the circuit behaves according to the operation and enters a concrete conceptual state at the reference time point. This verification task can be expressed by a property. This property is called operation property. The automaton formed by the set of all operation properties differs from the operation automaton only with respect to the states. The transitions and reference time points are the same. The operation automaton is therefore sometimes identified with the set of all operation properties.

The state mapping is a function that maps implementation states to the conceptual states and a special value $\perp$. This special value is for those implementation states that represent no conceptual state. In practice, there is for every important state a predicate to determine if the circuit entered the important state, and a separate function that maps the behavior of the circuit to the corresponding visible state.

The state mapping has to be defined. While this can be automated for combinatorial equivalence checking, it is usually provided manually by the user of complete verification. However, the user is working with a safety net, corresponding to the pessimistic attitude of equivalence verification: Incorrect user input can only lead to a negative result of the equivalence verification between operation properties and implementation, although a correct user input may actually show equivalence. It is not possible to falsely obtain a positive result by wrong user input.

To determine the state mapping for the SDRAM interface, the implementation must be examined. Figure 6 describes the state transition graph of the implementation. A transition of this graph corresponds to one clock cycle. The transition is characterized by assignments to registers that often drive the output signals, and where applicable by the conditions under which the transition is executed. The state mapping obviously maps $state = idle$ to the important state IDLE and $state = row\_act$ to the important state ROW_ACT, as well as the contents of the register $last\_row$ to the visible state actrow.

3.2.5 Representation of Operation Properties

Based on this state mapping the operation property belonging to Figure 5 can be formalized. It is expressed in the language ITL which is so closely related to timing diagrams, that properties described in this language can frequently be visualized very intuitively.

ITL distinguishes between an assume and a proof part. The assume part corresponds to the left side of an implication and the proof part to the right side. The implication will be checked for all time points $t \geq 0$. Assume and proof part consist of temporal conditions, which are formed from a state predicate and a temporal specification. The temporal specification references time explicitly, relative to an arbitrary but fixed point in time t.

This explicit specification of time points allows that assume and proof part overlap in time. This is important for operation properties, because it may happen that different operations are selected by values of input signals at relatively late times, such as e.g., read accesses of a processor under normal operation or once they are terminated by an error message.

In this regard, ITL has advantages over the conventional use of SVA or PSL, wherein the antecedent (that corresponds to the assume part of ITL) and the succedent (which corresponds to the proof part) describe successive time intervals which overlap by at most one clock cycle in the middle. Another advantage of ITL is the absence of operators that allow different matches for a given initial time point of the property. This is a source of errors even in user literature about SVA.

The graphical representation expresses the assume part by blue lines and the proof part by red lines. If the property is proven, its graphical representation can be printed on a slide and pulled over any simulation run of the SDRAM interface. Wherever the blue lines match, the red lines will match either.

The textual representation of the operation property of Figure 5 is

```
property read_new_row is

assume:
at t:       state = row_act;
at t:       request = '1';
at t:       rw = '1';
at t:       address /= last_row;

prove:
at t+9:             state = row_act;
at t+9:             last_row = prev(row(address));

during [t+1, t+7]:        ready = '0';
at t+8:             ready = '1';
at t+9:             ready = '0';
at t+8:             rdata = prev(sd_rdata);

at t+1:             sd_ctrl = precharge;
at t+2:             sd_ctrl = nop;
at t+3:             sd_ctrl = activate;
at t+3:             sd_addr = row(address);
at t+4:             sd_ctrl = nop;
at t+5:             sd_ctrl = read;
at t+5:             sd_addr = col(address);
at t+6:             sd_ctrl = stop;
during [t+7, t+9]:        sd_ctrl = nop;

end property;
```

The functions row and col extract the row and column address from the full address. Figure 7 presents the related graphical representation. The property can only be proven if it is assumed that the circuit environment behaves according to the protocol. This is formalized in the following two constraints:

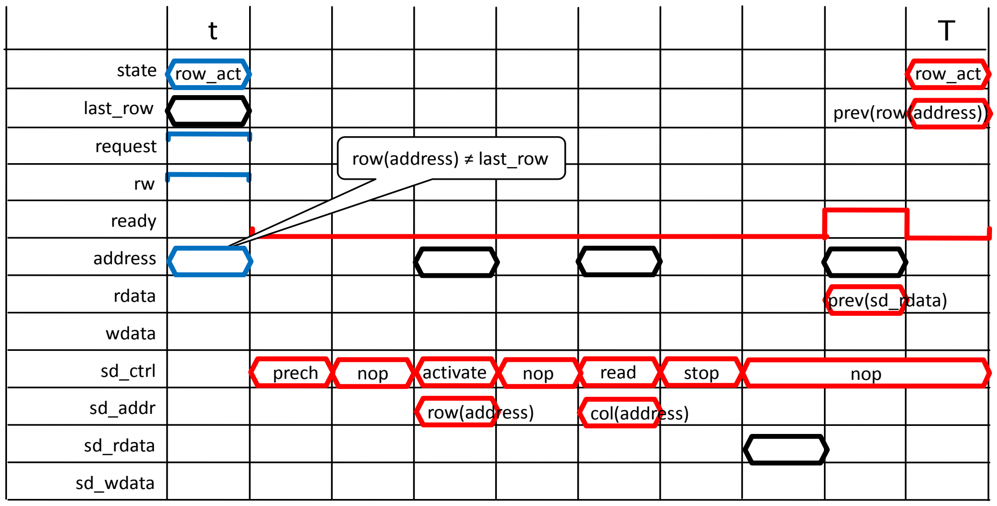


Figure 7: Graphical representation of the property

```
constraint no_reset :=
      reset = '0';
end constraint;

constraint processor_protocol :=
      if request = '1' and ready = '0' then
            next(request) = '1' and
            next(address) = address and
            next(rw) = rw and
            if rw = '0' then next(wdata) = wdata;
      end if;
end constraint;
```

3.2.6 Discussion

Operation properties, operation automata and transaction automata are circuit descriptions with increasing level of abstraction. The operation properties provide only reduced information about the behavior of a few internal signals of the circuit. Operation automata do not mention internal signals at all, but describe only the effect of internal signals to the sequence of operations and to the flow of information between successive operations. But operation properties and operation automata represent the input / output behavior precisely in terms signals and clock cycles. Only the transition to transaction automata also abstracts from input / output behavior of the circuit to atomic transactions.

Operation properties and operation automata therefore provide the input / output behavior of the circuit as precisely as the original RTL description. But in comparison with the RTL, the operations are more structured in the sense that they represent related activities on the output signals in their relationship with each other and with the input signals. This is less an abstraction than a structuring of the input / output behavior. This structuring turns the usually extremely complex variety of possible output actions of RTL's into a manageable number of operations. This allows understanding the circuit function more easily and it is easier to determine whether the operation of a circuit is correct.

The understandability gain by the transition from RTL to operations corresponds roughly to the improved understandability of RTL compared to a netlist description. Compared to netlists, RTL can be more easily understood because individual signals are grouped into bit vectors and complex logics with many gates and connecting internal signals are condensed to powerful or at least more intuitive operators. In the relationship between netlists and RTL, values of bit vectors and the powerful operators about the bit vectors play the role of transactions and operations in the relationship between RTL and an operation automaton.

But the improved structuring of the input / output behavior in terms of operations or operation properties and the associated abstraction of internal behavior does not only lead to a better understanding of the circuit. On top of this it will be shown in the following sections that operations also lead to proof tasks, which can be solved with a particularly suitable proof technique. As a consequence, given circuits can be verified with lesser technical effort, or larger circuits can be analyzed with the existing resources.

This effect shall also be achieved with other abstractions that are discussed in the literature, such as predicate abstraction and abstraction refinement [Kroening / Seshia 2007, Clarke et al. 2000, Wang et al. 2006, Chauhan et al. 2002, McMillan / Amla 2003, Gupta et al. 2003, Jain et al. 2008]. However, the goal of these methods is always to suitably reduce the model of a circuit. Complete verification avoids such reductions of the model. All properties are proven on an internal representation of the entire RTL. The only measure to reduce the size of a model is compositional complete verification, by which a circuit is broken into sub-circuits, which are then verified separately.

3.3 Proof Technology

In complete verification the proof algorithms and the methodology are particularly well matched. The proof technique of choice is Interval Property Checking (IPC), a proof algorithm that can often prove operation properties within minutes. IPC is applicable, because complete verification mitigates the disadvantage of IPC that reachability conditions must be given explicity, to characterize the states that can occur during normal operation of the circuit. Complete verification usually needs comparatively few and simple reachability conditions. In addition, user-specified reachability conditions are cross checked, so that the verification cannot be corrupted by accidentally incorrect user input.

Hence the good match between proof technique and the methodology of complete verification allows that even large circuits can be examined with operation properties, while the overhead of determining the reachability conditions remains acceptable. The effort is usually so low that the complete verification is more productive than the simulation-based verification, even if the quality improvements are not accounted for.

For general assertion based formal verification, IPC usually needs more elaborate reachability conditions. In ABFV tools the reachability analysis is therefore largely automated. Therefore, only smaller circuits can be examined with sometimes very high proof times and / or it is accepted that the examination is less rigid than a full proof.

IPC is treated in section 3.3.1. Reachability conditions are exemplified in section 3.3.2, and their determination and justification is discussed in sections 3.3.3 to 3.3.5.

3.3.1 Interval Property Checking (IPC) and Bounded Model Checking (BMC)

In order to verify a circuit, it must be ensured that each operation property derived from the operation graph is satisfied for every time point $t \geq 0$.

The corresponding proof can in principle be created by every formal proof method. Interval Property Checking (IPC) is however particularly useful because it can analyze large circuits quickly and quickly comes to a conclusion. Interval Property Checking is a SAT-based method [Ganai / Gupta 2007] to verify digital circuits [Lohse / Warkentin 2001]. The circuits are represented by Mealy automata with bit valued input, output and state variables. The mapping of RTL code to such automata is prior art [Bormann 1995].

Interval Property Checking is suitable for a property A, that relates input, output, and state variables of the circuit for time points within a finite time interval $[t, t+n]$. The properties are examined for all time points $t \geq 0$ and for all traces. For this, the state transition and output function of the Mealy automata are unrolled $n+1$ times (see Figure 8). This creates instances of the functions for the time points $t, t+1$, etc., up to $t+n$ which substitute the variables of the property. This results in a Boolean function A' whose arguments are the instances of the input variables of the Mealy automaton for each of the time points t to $t+n$ and the instance of the state variables at the time t. The circuit meets the original property A if the Boolean function A' is 1 for all arguments.

Zeros of A' point to traces that violate A. These so-called counter-examples are characterized by a state at the time t and by the sequence of input values at the time points t to $t + n$. No check is made if the state at time t is reachable. If it is not reachable, the counter-example is unrealistic, i.e., it cannot occur during operation of the circuit. Then the proof must be repeated under additional reachability conditions. This characteristic limits the application of IPC to processes in which reachability conditions are determined and validated separately.

The zeros of A' are determined with a special class of proof algorithms, called SAT solvers. These algorithms robustly handle problems with millions or more variables. For this dissertation, the exact function of SAT solvers is irrelevant, they are presented e.g., in [Ganai / Gupta 2007]. Known SAT solvers are GRASP [Marques-Silva / Sakallah 1996], Chaff [Malik et al. 2001], and MINISAT [Een / Sörensson 2003]. SAT solvers are continuously being improved [Brinkmann 2003 Wedler et al. 2007, Wedler et al. 2005, Novikov / Goldberg 2001, Novikov 2003, Novikov / Brinkmann 2005, Goldberg / Novikov 2002], and alternative proof algorithms are developed [Achterberg / Wedler / Brinkmann 2008].

IPC is related to Bounded Model Checking (BMC) as described by [Biere 1999]. Unlike IPC, BMC does not use an arbitrary state at the beginning of the instances of the state transition and output functions, but the reset state of the circuit. Therefore, the instances describe the clock cycle 0, 1, 2, etc. after reset of the circuit. BMC does not examine property A for every time point t. Instead, a BMC algorithm determines a limit N depending on the available resources and examines the property for all $t \leq N$. A counterexample of this examination is particularly useful, because it starts in the reset state and thus cannot be unrealistic, which was the problem with IPC due to an unreachable initial state. But if BMC does not find a counter example, the result is not a full proof because of the limitation to the time points $t \leq N$.

As long as ABFV is only used to support simulation-based verification, BMC is a good choice, because the users are mainly interested in counter examples. Since simulation cannot prove correctness of a circuit anyhow, a full proof for single assertions is not particularly helpful. Verification methods that use formal proof algorithms without aiming at full proof are frequently called semiformal.

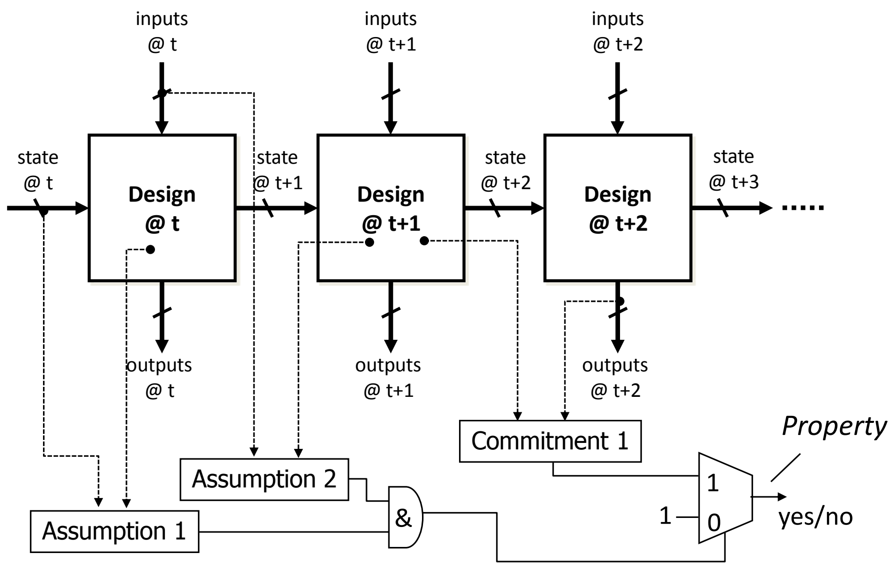


Figure 8: Generation of an IPC proof task

However, if more complex assertions must really be proven, e.g., to show compliance with a specification, the verification engineer is in a dilemma, because BMC is unsuitable, other verification approaches have lesser complexity boundaries, and the explicit determination of general reachability constraints is quite hard. A solution to this dilemma is described in section 3.4.4.

The proof of the property A with IPC requires only $n + 1$ instances of the state transition and output function and only one instance of the logic that represents A, while BMC requires $N + n + 1$ instances of the next state and output function, and N instances of the property logic. This shows that a BMC verification is more resource demanding than IPC, even though the proof is limited.

3.3.2 Examples of a Reachability Condition

The example of the SDRAM interface is actually too small to provide the user view onto IPC, unrealistic counterexamples and reachability conditions. Neither the run time benefits of an IPC proof can be demonstrated, nor the facettes of the reachability conditions required.

In order to present something it shall be assumed that the SDRAM interface of Figure 6 is modified according to Figure 9, i.e., by removing the assignments to the output register when the circuit in the state. This change is irrelevant since this register gets a ‘0’ assigned on all paths to the state *idle*. The assignments in state *idle* are therefore redundant.

Nevertheless, on the correspondingly modified circuit the property read_new_row of section cannot be proven directly by IPC. The counter-example examples shows that *ready* is '1' at $t + 1$. This counter example is irrelevant, because there are no input stimuli to the circuit such that $ready = 1$ in state *idle*. To acknowledge this, the start state condition of read_new_row must become $state\ =\ idle\ and\ ready\ =\ '0'$, and analogously all end states *idle* need to be adapted.

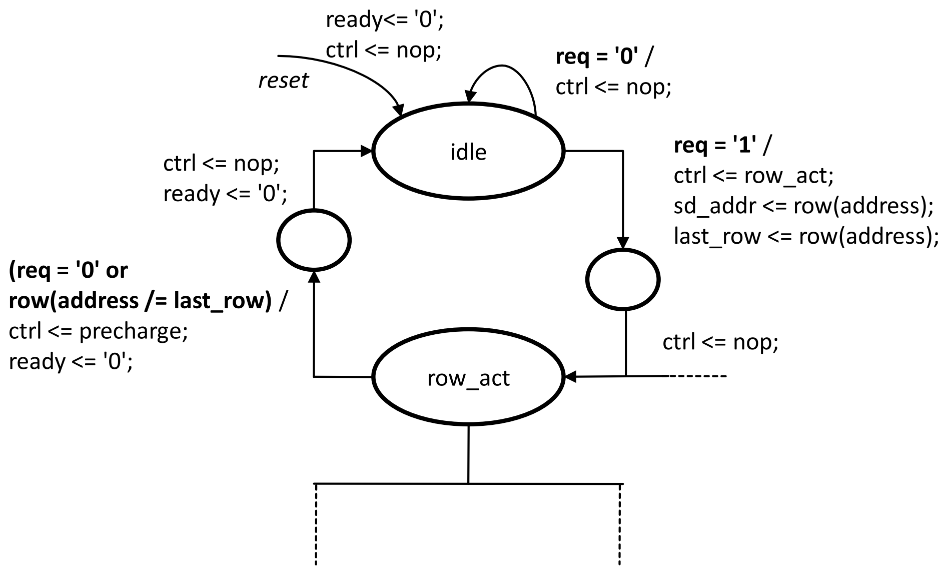


Figure 9: SDRAM interface with alternate implementation

3.3.3 Reachability Analysis

Reachability conditions can be determined automatically, or the user creates them manually.

The automatic determination of reachability conditions is called reachability analysis and is closely related to many formal proof algorithms. Reachability analysis is a very difficult research topic and a major reason for the slow development of formal verification over the past 15 years.

Some reachability analyzes strive to determine the set of reachable states exactly. These methods can only work on small circuits and have even then sometimes enormous run times.

Other methods determine the supersets and subsets of the set of reachable states. BMC's prologue of up to N instances of the state transition function of the Mealy automaton (see section 3.3.1) can be understood as a reachability analysis that determines a subset of the reachable states at the time t of the property A. Such underestimating methods are unsuitable for complete verification, because they mask counter examples that start in those states that were not discovered.

By contrast, methods for the determination of supersets of reachable states can be well integrated into the complete verification. Particularly well was this done with the method described in [Nguyen et al. 2008, Nguyen et al. 2005a, Nguyen et al 2005b]. Other options arise from the work of [Case / Mishchenko / Brayton 2006, Stoffel et al. 2004, Wedler et al. 2003, McMillan 2003].

3.3.4 Justification of User Defined Reachability Conditions

The automatic reachability analysis usually fails in such verification projects where the reachability conditions relate to creative design efforts on the basis of a global circuit understanding. Such reachability conditions are more easily determined by the human user, who reads them from suitable circuit documentation, obtains them from designer interviews, rebuilds the respective circuit understanding. For a complete verification, it is therefore essential that reachability conditions can be explicitly specified by the user.

Such a user input may contain errors. By the completeness checker that will be presented in section 3.4 will discover such errors. This prevents that the verification is corrupted by faulty user provided reachability conditions. Due to the low run times of the IPC proof algorithms, the user need not even be particularly careful when making guesses about the reachability conditions. If the complete verification can be carried out until the end, all user provided reachability conditions are also justified and otherwise, counter examples are created that help to debug the reachability conditions.

ABFV usually relies on automatically determined reachability conditions. If they are occasionally still entered by hand, it remains the responsibility of the user to justify the reachability conditions. Sometimes this justification is only provided informally. Other users check reachability conditions by simulation. In both cases, the reliability of verification is considerably impacted: The value of a formal verification of an assertion becomes questionable, if it is verified under the assumption of a reachability condition, that is only justified by a procedures that is considered so unreliable that formal verification was tried.

But reachability conditions may be justified separately. Typical approaches are induction proofs. These often require the identification of additional reachability conditions until an induction hypothesis is found that can be proven. In the ABFV this complex process leads to a dedicated reachability analysis for the corresponding assertion. It is likely that this process needs to be repeated if further similarly complex assertions are to be proven.

3.3.5 Reachability Conditions and Operation Properties

In connection with complete verification reachability conditions are only needed for the important states, because all operation properties start there. An operation automaton usually has only few important states, and therefore complete verification needs only few of such specific reachability conditions. This already simplifies the task of determining the reachability conditions.

A further simplification arises because usually a superset of the reachability conditions is sufficient. For example: If a register is initialized in all operations that start in some important state, then this register need not occur in the reachability condition. This is because even if the proof tool assumes an unreachable value at the beginning of the operation, this value will be overwritten by the initialization value, such that the unreachable value has no impact for the execution of the operation.

But in some implementation styles the registers are always reinitialized immediately after use, i.e. at the end of an operation. The example from section 3.3.2 reflects this situation. Such implementation styles lead to the reachability condition that the register has the initialization value in certain important states. In this situation, this reachability condition is really needed, but this is quite simple and easy to recognize.

If yet more complicated reachability conditions should occur in the important state, there is a good chance that they are known to the developer of the circuit, because usually the developers consider operations as self-contained functional units that start in the important states and implement them accordingly. This designer knowledge can be used in the development of the properties. For the reasons given in section 3.3.4 there is no risk of duplication of errors.

All in all, reachability conditions for the proof of operation properties can therefore usually be obtained quite easily. This statement should not rule out that inadequately implemented circuits include complicated reachability conditions that can be detected only with difficulty.

For reachability conditions for assertions, however, the situation is different: In the course of an operation intermediate results are calculated, stored and used. This can lead to complicated reachability conditions in the middle of an operation. Moreover, the designers are not aware of them because s/he has the flow of the operation in mind and does not think about the question, which signal values can occur simultaneously. Fortunately, such complicated reachability conditions from the middle of an operation are not necessary to prove the operation properties. But they may well be necessary for the proof of assertions that make e.g., statements about signal behavior in the middle of an operation. Such reachability statements are not even trivial in relation to individual operations. It becomes particularly difficult, however, if an assertion makes a statement over several operations. In this case the process of developing the reachability condition is no longer properly structured and unrealistic counterexamples during the development of the same reachability condition may show behavior from different operations. This can make the development of a general reachability condition a cumbersome task. In this case it may be advantageous to first carry out a complete verification before the the assertion is proven with the methods described in section 3.4.4.

3.4 Completeness Check

For complete verification a circuit is examined with the operation properties from section 3.2, fed into the proof algorithms of Interval Property Checking from section 3.3. The operation based examination of the circuit does not only help with the reachability problem and the complexity of the proof that was discussed in section 3.2. It also helps a user to structure her / his efforts for a formal verification and to focus on one operation and all its facettes at a time.

However, the concentration on individual operations gives rise to the risk that the interaction of the operations escapes from the view of the verification engineer. This might give rise to verification gaps. Examples are described in section 3.4.1. A measure to exclude verification gaps is to examine if the set of operation properties is complete. For this purpose it is investigated whether the verification code rigorously checks the values of the output signals at every time point. The rigor of the check is formalized in the concept of determinedness of section 3.4.2. The completeness checker from section 3.4.3 makes determination assumptions about the input signals and proves that the operation properties check the output signals with the rigor requested by the determination commitments.

The completeness checker allows complete verification of circuit parts, which process operations sequentially. For such circuit components section 3.5.6 introduces the the term cluster. Section 3.5.7 illustrates the conditions under which a complete verification of a set of clusters is also a complete verification of the circuit that consists of these clusters.

3.4.1 Verification Gaps

The development of operation properties follows the abstraction level from Figure 2 either top down or bottom up.

If a good specification is available a verification engineer will start with the highest level of abstraction, i.e. the transaction automaton. This will provide an overview of the transitions and conceptual states of this automaton. The transaction automaton will further be refined to an operation automaton and the operation properties. These steps are influenced by implementation specific information that captures design decisions as presented in sections 3.2.3 and 3.2.4.

The abstraction levels from Figure 2 are traversed bottom to top, when only inadequate specifications are available or if the implementation interpretes the specification in an idiosyncratic manner, so that the operations from the specification are not really found in the implementation. First, the operation properties are developed. During this development, the operation and transaction automaton are created piece by piece. The correspondence between the operation properties and the specification contents is continuously checked.

Regardless of the method, the verification engineer focuses on one operation at a time. This is not only beneficial for the proof technique of section 3.3. Rather, it helps the verification engineer to concentrate on a relatively small number of phenomena and to investigate them thoroughly before the next operation is processed. Thus, the single operation receives particularly high quality.

But during the verification it needs to be ensured that the operation properties together form an operation automaton. This can be impacted by a number of flaws during the development of the operation properties. All these flaws have the consequence that the operation properties accept unintended behavior. In this case, the verification has gaps. Some examples of such gaps will be presented in the sequel.

The most prominent verification gap is that some operations have not been verified at all. If a circuit has an error in such a forgotten operation, the error would not be detected.

Another type of verification gap is that the conditions about the input signals of the operations are formulated too restrictively. Then, the operation property checks less runs of the circuit than intended and lets other runs unexamined. The unexamined runs may hide implementation errors.

Further possible verification gaps are inadequate conditions about the output signals that allow multiple signal values, although only one is correct. For example, if the condition about sd_ctrl would be missing in the proof part of the property in section 3.2.5, the circuit would not be recognized as erroneous, if wrong commands would be passed to the SDRAM.

Furthermore, the predicates for important states can be formulated incorrectly, or there are different predicates being used for the same important state.

Finally, the calculation of the updated visible state may be described incompletely or it does not fit with the functions that extract the visible states from the traces of the circuit. For example, if the condition "at t + 9: last_row = prev (row (address))" were missing in the property in section 3.2.5, a subsequent read access could be verified with a property about a read to the same row address as before, or with a property about a read to a different row address. But the two properties provide different behavior e.g., of sd_ctrl and hence to not verify the read properly. An error in the logic that identifies row address changes might thus escape. When an error makes the circuit accidentally modify last_row during the first read operation, and if the 2nd read operation accidentally uses this modified value, the row address change would not happen and the SDRAM IF would write into the wrong memory location. Such errors can hardly be identified by simulation because in most cases the error only leads to some additional row address changes, but nevertheless the correct date is read. To read a wrong date during the simulation, the SDRAM must be accessed with a row address which coincidentally corresponds to the incorrect value in last_row, because only then the necessity of change in the memory line becomes apparent. Only then the wrong memory location will be read. This situation is unlikely in simulation, but the approach described here finds such errors and ensures on top of that, that the verification of all possible error scenarios is reliably taken care of.

3.4.2 Determinedness

To exclude the verification gaps above, it is sufficient to determine whether the conjunction of properties is satisfied by exactly one output trace for each input trace trace of the circuit. This completeness criterion is used e.g., in [Claessen 2006]. This test determines whether the verification engineer decided for a fixed output value at every point in time. If this is not the case, the verification must be improved.

This completeness criterion requests a bit much, because sometimes the designers do not want to uniquely specify some signals. E.g., many protocol specifications define when address and data signals shall be "valid", i.e. when the sending module must make the information available on the bus. Only then the receiving module should read them from the bus. Outside of these times, these signals may assume any value, and the concrete value is often a consequence of optimization criteria such as power or area consumption. The verification should better not check these values, because they might change during a design project, and this change should not modify the circuit function.

Conversely, it should also be ensured that the verification makes no assumptions about the input signals at times at which, e.g., a protocol specifies that the signals are invalid. The circuit must operate in the same way for all invalid signal values and therefore such signal values should again not be evaluated in the verification of the circuit.

In practice, it is therefore obviously necessary to allow a certain blur of the input and output traces of a circuit. Two input traces may be different, but should be treated equally by the circuit. For the same input trace the circuit may produce different output traces, but they may only differ in a way that is not recognized by the neighbouring circuits.

This blur is expressed by determination conditions. If the determination conditions specify which two input traces are to be regarded as equal, they are called determination assumptions. If the determination conditions define which two output traces are the same, they are called determination commitments. When the contents of visible registers are regarded as equal, is determined by local determination conditions.

In general, determination assumptions and commitments can be formed by conditional determination functions. The conditions mask situations where specific values of the traces are not relevant. The functions extract from the circuit traces all information relevant for the circuit operation, similar to the transaction extractors of section 3.1.2.

The basic syntactic building block for determination assumptions and requirements is the construct

```
if g then determined(e) end if;
```

with g as a condition and e as a function that extracts the information. The construct

```
determined(e);
```

is a short form of

```
if true then determined(e) end if;
```

The conjunction of determination assumptions and commitments is given by

```
determined(e);
determined(f);
```

Besides the determination assumptions and requirementscommitments there are also local determination conditions that specify intermediate proof goals of the completeness checker. They express that an operation property uniquely describes the value of a visible register at the end of the operation. Unlike determination assumptions and commitments these local determination conditions are therefore not examined at every time point, but only at selected, usually only one time point relative to the time point t of the operation property. The syntax is

```
at t: if g then determined(e) end if;
```

The following shall exemplify the terminology around determination at the SDRAM interface: The operation automaton must check the signal sd_ctrl against a unique value at every point in time. This is necessary to avoid that the SDRAM receives unintended commands. In addition, the signal $ready$ should always be uniquely determined, as is usual in many practically relevant protocol specifications, although this is not mandatory for the proper function of the protocol. This is achieved by determination commitments

```
determined(sd_ctrl);
determined(ready);
```

These commitments request that the operation properties always check these signals against a uniquely defined value.

The output signal sd_addr need only be checked against a unique value when the SDRAM needs the address, and that is the case when it receives the commands activate, read or write. Accordingly, the signal sd_wdata need only be valid if a write command is issued. The signal $rdata$ must only be determined when a read access is finished.

The corresponding determination commitments are

```
if rw = 1 and ready = 1 then determined(rdata) end if;
if sd_ctrl = write then determined(sd_wdata) end if;
if sd_ctrl = read or sd_ctrl = write or sd_ctrl = activate
            then determined(sd_addr) end if;
```

The determination assumptions of the SDRAM interface require continuous determinedness of the signal $request$. The output signals rw and $address$ of the processor need only be determined when a request is present. Write data on the signal $wdata$ may only be assumed to be determined if it is a write request. The SDRAM provides its read data only if it has previously received a read command within a sufficient time interval. This provides the determination assumptions

```
determined (request);
if request = 1 then determined(rw) end if;
if request = 1 then determined(address) end if;
if request = 1 and rw = 0 then determined(wdata) end if;
if prev(sd_ctrl, 2) = read then determined (sd_rdata) end if;
```

The local determination condition for the property read_new_row from section 3.2.5

```
at t+9:     determined(last_row)
```

checks that this property uniquely determines the visible register $last\_row$.

3.4.3 Completeness Checker

A central component of this work is a completeness checker [Bormann / Busch 2005], for which a patent is pending. The completeness checker analyzes an operation automaton and uncovers verification gaps or confirms the absence of verification gaps. The use corresponds to a property checker. Like a Property Checker that receives assume and prove parts from the user and checks them against a circuit, the completeness checker receives determination assumptions and determination commitments, and checks them against an operation automaton. The completeness checker examines whether the operation properties only accept output traces that match the determination commitments if the input traces satisfy the determination assumptions.

The investigation is often limited to those input traces that can be generated by the environment of the circuit. The environment is described by constraints, or it is part of the verification. In the latter case, the environment assumptions are described by assertions, usually about neighouring parts of the circuit. In the sequel, the term local constraint shall be used as a collective term for the environment assumptions that are relevant for a completeness check of a set of operation properties.

Technically, the completeness checker firstly ensures that for each input trace a sequence of operations can be found, which covers the input trace. This sequence is called a chain. In further process steps, the completeness checker ensures that the output signals are uniquely determined by the properties modulo the determination commitments.

The completeness checker is especially implemented for the examination of an operation automaton or a set of operation properties. It is built on top of a property checker. This allows completeness checking on circuits of industrially relevant size. A detailed description of the completeness checker can be found in section 5 along with a detailed assessment of the prior art.

In practice, the completeness checker outputs counterexamples that point the user to verification gaps. In many cases, two sequences are shown that meet the determination assumptions but violate determination commitments. In other cases, the counter example consists of only one trace. Such a counter example shows that the operations or operation properties take the circuit into a conceptual state from where the chain of operations does not continue. This is demonstrated by specifying an input trace that satisfies all constraints of the verification, but violates the input conditions of all operations that start in this conceptual state. The completeness checker generates fairly detailed debug information, because it performs various specific tests on the operations or operation properties.

The counter examples point to verification gaps. This helps the user to complete the verification. In many cases, the verification gaps arise from specification gaps. Their closure improves the specification, such that even the quality of the specification benefits from the completeness check.

If the completeness checker can prove the determination commitments under the determination assumptions, the set of properties is called complete. This completeness criterion solves the problem to determine the quality of a set of properties that has repeatedly been mentioned by users and in the literature [Mitra 2008].

In the context of complete verification the completeness checker provides a logical criterion for the termination of a circuit verification. That criterion has never been applied previously in the industrial circuit verification. The satisfaction of this termination criterion can be documented in detail and is always traceable. This provides a much more founded quality proof for the circuit as currently possible with simulation-based methods.

3.4.4 Verification of Complex Assertions by the Completeness Checker

As already mentioned in section 2.8, assertions play an important role also in complete verification. They are used to capture dedicated verification goals as to justify local constraints or to summarize other important intermediate verification goals. As said earlier, the proof of assertions can become too complex for ABFV tools. In section 3.3.5 it was mentioned that the direct proof of assertions by IPC requires the provision of reachability conditions, and this can sometimes be prohibitively expensive.

One way out requires the presence of a complete set of proven operation properties. These relate to every input trace one or at most only few traces of internal and output signals. If the assertion is valid on all these traces, then the assertion is also true on the circuit.

This insight is the basis for an extension of the completeness checker. The basic idea is to treat the assertion as an additional output signal, and to prove about it that it always indicates that the assertion is true. Accordingly, the complete set of operation properties of a circuit provides an adapted induction scheme for verifying the assertion [Sheeran / Singh / Stalmarck 2000].

With such an analyzer for complex assertions, the complex reachability analysis involved in the proof of an assertion is replaced by the reachability analysis required for the proof of the operation properties and their completeness. In accordance with the considerations of section 3.3 this is much easier.

3.5 Compositional Complete Verification

The compositional complete verification [Beyer / Bormann 2008] is based on several sets of properties, that each completely verify a part of a larger circuit. This gives raise to the question whether these properties altogether completely verify the large circuit.
The answer seems trivial only at first glance. A second look reveals that the trivial answer usually involves circular reasoning. This can be seen as follows: When a verification shows that a module satisfies an assertion β under the constraint α and another satisfies assertion α under the constraint β it must not generally be concluded that the overall circuit satisfies the assertions α and β. Counter examples are presented in section 6.4.3.

To study related effects, integration conditions play an important role. They are descriptions of the interfaces of the property sets and will be introduced in section 3.5.1. Integration conditions exist both for those property sets that the completeness checker proved to be complete, as well as for sets of properties that have been formed according to the rules of compositional complete verification. Accordingly, this section develops a hierarchical process for the complete verification of circuits of any size.

To combine complete property sets of partial circuits to one complete set of the whole circuit, the properties need to be proven on suitable models, and the integration conditions of the partial circuits and the whole circuit need to satisfy a number of preconditions. Some of these preconditions are specific for the single integration condition, and some of them relate to the interaction of all integration conditions.

Since each circuit is usually part of a larger system, the preconditions of the individual integration conditions should be fulfilled even if no compositional verification is to be performed. Hence, corresponding tests offer themselves as plausibility tests for each complete verification and provide an attractive alternative to the examination of constraints wrt. contradicitions that is usual for ABFV. An overview of the plausibility tests is provided in section 3.5.3 and details are given in section 6.2.

The requirements about the interaction of the integration conditions of compositional complete verification have two important applications. In IP-based design of a system-on-chip (SoC) they lead to the verification of the communication between the IP blocks. This is described in section 3.5.5.

The complete verification of modules, such as the IP blocks themselves must often deal with multiple concurrent operation automata, which are implemented by various circuit components. These circuit parts are called clusters and are descrubed in section 3.5.6. The verification of a circuit with multiple clusters is presented in section 3.5.7.

Compositional complete verification is treated in detail in Chapter 6. The associated mathematical theory is set out in the Annex.

3.5.1 Integration Conditions

A complete verification is performed under constraints and determination assumptions. They include the conditions about the input behavior of a circuit under which the individual properties are proven and the conditions, which are used in the completeness check. Constraints and determination assumptions together provide information under which assumptions about the interface protocols the complete verification of the circuit was performed. Accordingly, this information is called integration assumption.

About the output signals of a circuit, assertions and determination commitments are proven. Therefore this information describes the protocol that behavior of the output signals of the circuit adheres to. Accordingly, this information is called integration commitment.

Integration assumptions and commitments are combined into integration conditions. They are formal descriptions of the protocol supported by the output signals of the circuit. A special feature of this approach is that integration conditions adequately express when a communication partner reads data and addresses from its input signals, or when it sends such information to the output signals. This sets the stage for testing that each communication partner in a SOC reads data and addresses only when they are sent by the other communication partner. This feature is fundamental for testing the communication between IP blocks and is not included in any other method for formal verification of communication between IP blocks.

3.5.2 Example

The integration conditions of the SDRAM interface from section 3.2.1 are the result if the constraints and the determination assumptions used during its complete verification and have the form

```
integration_condition of sdram_if is

assume:
      reset = '0';

      -- constraints of the processor_protocol
      if request = '1' and ready = '0' then
            next(request) = '1' and
            next(address) = address and
            next(rw) = rw and
            if rw = '0' then next(wdata) = wdata;
      end if;

      -- Determination assumptions of the SDRAM IF
```

```
        determined (request);
        if request = 1 then determined(rw) end if;
        if request = 1 then determined(address) end if;
        if request = 1 and rw = 0 then determined(wdata) end if;
        if prev(sd_ctrl, 2) = read then determined (sd_rdata) end if;

  guarantee:
        -- A potential assertion about pulses of the ready signal
        if prev(ready) = '1' then ready = '0'; end if;

        -- Determination commitments about the SDRAM IF
        if rw = 1 and ready = 1 then determined(rdata) end if;
        if sd_ctrl = write then determined(sd_wdata) end if;
        if sd_ctrl = read or sd_ctrl = write or sd_ctrl = activate
              then determined(sd_addr) end if;

  end integration_condition;
```

The integration assumption indicates the protocol that the SDRAM interface supports. This is expressed by the constraints of section 3.2.5 and the determination assumptions of section 3.4.2. In essence, the processor protocol requires that the processor request is maintained until the peripheral indicates that it served the request. The determination assumption states when the input signals are evaluated by the SDRAM interface.

As an example, the integration commitment is extended by an assertion that the SDRAM interface produces pulses on the ready signal, i.e. the ready signal is never active over two consecutive clock cycles. Such an assertion confirmes that a certain feature of the processor protocol is not used here, that allows immediate (combinatorial) responses to a request. Such an assertion could become important if, for example, a processor version is used, that can not deal with such immediate reactions.

Other assertions could be related to the protocol towards the SDRAM itself and e.g., assure the correct sequence of commands on sd_ctrl or compliance with time constraints between control commands.

3.5.3 Plausibility Tests

Integration assumptions should describe only those input traces that the circuit can actually receive, e.g., if it is part of a larger system. This is straightforward if the integration assumption contains no output and internal signals of the circuit under test.

However, in many cases the environment reacts on the behavior of the output signals of the circuit. That means, that the environment produces input signals to the circuit with a restricted behavior, and the restrictions are different depending on the circuit outputs. In this case, the integration conditions contain output and internal signals. Then the integration assumption is called reactive.

From the syntax, reactive integration assumptions may postulate behavior that can not be created by any surrounding system. For example, a reactive integration assumption may not only restrict the input signals of the circuit, but they can simultaneously ban certain values on the output signals, although there is no equivalent to do so in the physics of a digital circuit. In addition, the integration assumptions may postulate a behavior of the surrounding system that it could only show if it would not be causal, i.e. if it had clairvoyant abilities. Thirdly, the integration assumption could require a combinatorial path in the surrounding system that complements a combinatorial path in the circuit to form a combinatorial loop.

One of the conditions for compositional verification is that the integration assumption must not be pathological in any of the above scenarios. For a related criterion a formalization has been found that is intuitive also for users without formal education: There must be at least one equivalent circuit that implements the integration assumption and can be combined with the circuit under test, without producing combinatorial loops. Details are described in section 6.2.

The criterion therefore consists of two parts. On one hand, it requires that there is at least one such equivalent circuit. If it exists, the integration assumption is called implementable. The requirement of implementability of the integration assumption ensures firstly that it will accept all values of the output and internal signals of the tested circuit. Secondly it ensured that the integration assumption implies no clairvoyance, i.e. no dependency between current values of the input signals and values of output or internal signals at future clock cycles.

The second criterion requires that no combinatorial loops may be created if this equivalent circuit is connected to the to circuit under test. This criterion is called structural compatibility. Structural compatibility can be understood as a refinement of the requirement for causality. This refinement is about combinatorial dependencies between signals. As long as there are no combinatorial loops in a circuit, there is an order in which the operations of the circuit take place within a clock cycle. This provides a causality relation, and the requirement of structural compatibility ensures that the integration assumption fits to it.

In section 6.2 tests are presented which implement sufficient pre conditions to ensure that an integration assumption is implementable and structurally compatible with the circuit under test. These tests are regularly used as plausibility criteria that prevent a devaluation of a complete verification by inadequate integration assumptions.

Thus, the plausibility tests take a role that is played in ABFV by vacuity tests, that shall detect flaws in constraints, be it with or without the model of the circuit. In comparison, the vacuity tests of ABFV are fuzzier than the plausibility tests of complete verification because they only raise an alarm when a constraint prohibits all output values, and they are blind to non-causal dependencies between output and input signals of the tested circuit. In turn, the plausibility tests of complete verification already warn against a reactive constraint that accidentally restricts some output behavior.

3.5.4 Examples of Rejected Integration Assumptions

To demonstrate the value of the tests on implementability and structural compatibility, integration assumptions are presented below, which are rejected by them:

```
integration_condition of example1 is
assume:
      if next(outsig) then determined(insig) end if;
guarantee: …
end integration_condition;
```

would be rejected because the determination assumption is not causal.

The integration assumption

```
integration_condition of example2 is
assume:
      determined(insig);
      insig = outsig+1;
guarantee: …
end integratin_condition;
```

would be rejected if insig and outsig are bit vectors with equal width. Then the integration assumption prohibits namely that outsig ever becomes "1111 ... 1111". This is because ITL performs the addition in a numerically correct way which cannot be presented by insig due to the restricted bit width.

For the circuit of Figure 10, the integration condition

```
integration_condition of example3 is
assume:
      determined(in1);
      determined(in2);
      in2 = sum;
guarantee: …
end integration_condition;
```

will be rejected because the integration assumption is not structurally compatible with the model under examination. This is because any circuit that is equivalent with the integration assumption will introduce the combinatorial dependency that is indicated in Figure 10 by the dashed line, and this causes a combinatorial loop. It is useful to uncover this problem, because under the constraint in the integration assumption the unusual assertion

```
assertion unusual := out_sig = '0'; end assertion;
```

can be proven, which can be seen as follows: Since $sum = in1\ xor\ in2$, we get for $in1 = 1$ the relationship $sum = not(in2)$, which contradicts the constraint. Hence there must be $in1 = 0$, which proves the assertion. But the proof of this assertion appears absurd, because there is no circuit environment that is connected only with the signals sum and in2, and influences the signals in1 or out_sig.

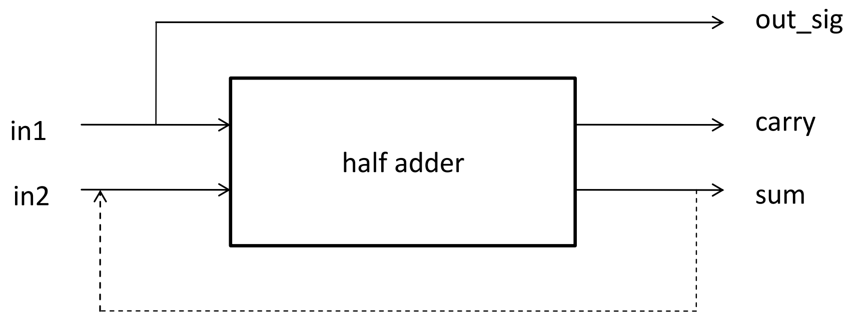


Figure 10: Structurally pathological case

3.5.5 IP Based SoC Design

A strategy for the rapid development of integrated circuits is the IP based design of a system-on-chip (SoC). Here circuit blocks are pre-designed, verified and made available in an IP library, potentially independently of final plans for the development of a specific ASIC. Such circuit blocks are, for example, processors, arbiters, memory controllers or bus systems. For such modules, the term IP block (IP = intellectual property) is used, which points out that the final product of their development is not a silicon circuit but only their production documents. IP blocks are realized on silicon as part of a SoC, which is created sometime later by another developer team from the production documents of multiple different IP blocks. This SoC development team is often referred to as the integrators.

As a basic set of production documents the IP developers will offer RTL, synthesis scripts and so on. They add value to their IP with further information, including integration information about the supported behavior of the input signals, or general descriptions about the behavior of the output signals. Frequently the integration information names the protocols that are supported by the module, and they may also indicate that the circuit does not support certain specialties of the protocol either on the input or the output side. The integration information of IP blocks thus contains information similar to the contents of the integration conditions of complete verification.

In IP-based SoC design, verification of the IP block itself is usually executed by the IP developer. Besides correct function this module verification must also show the correctness of the integration information. Today, this is done primarily by means of simulation and is therefore somewhat uncertain.

The integrators use the IP information to make a preselection of candidate IP blocks. After the interconnection of the IP blocks (by suitable RTL code) simulation is used to confirm the proper interaction with neighboring modules. Moreover, this simulation ensures that the interaction between the IP blocks actually has the function that is requested by the specification.

The integration conditions are results of complete verification as described above. They are precise and formally proven descriptions of the supported protocols that can be put automatically and formally into relation with each other before much effort is spent on the development of the SoC. This way, a formal verification of the communication in the SoC is possible, which is distributed like in simulation on the IP and SoC designers: The IP developers ensure on one side by execution of a complete verification that the integration conditions are correct. They also make sure that the integration assumptions are implementable and structurally compatible with their circuits.

The SoC designers on the other side ensure that the integration assumptions of the overall circuit are implementable and structurally compatible with the overall circuit. They also ensure the structural compatibility of integration assumptions of sub-blocks with the overall circuit. And of course they have to determine whether the integration conditions fit together. To that end firstly the integration assumptions of all blocks must be satisfied, taking into account the interconnection, the integration commitments of the neighboring blocks and the integration assumptions of the overall circuit. Secondly, the integration commitments of the overall circuit must by satisfied by the integration commitments of the sub-blocks and the integration assumption of the overall circuit. The mathematical formulation of these requirements is discussed in section 6.3.2.

If the SDRAM interface with its integration conditions from section 3.5.2 provides a processor with instructions (and not with data), the processor may have the following integration commitments:

```
integration_condition of processor is
assume:
      …
guarantee:
      p_rw = '1' and
      if p_request = '1' and p_ready = '0' then
            next(p_request) = '1' and
            next(p_address) = p_address
      end if;

      if p_request = '1' then determined(p_address) end if;
      …
end integration_condition;
```

The prefix p_ of the signal names should highlight that the outputs of the processor are initially not related to the input signals of the SDRAM interface. The processor signals must therefore be suitably substituted before an integration test can be performed. After this substitution the test can be executed. It seems an interesting facette that the determination assumption about the signal rw of the SDRAM interface is justified by the assertion about p_rw and not by a determination commitment.

3.5.6 Clusters

A complete verification of a larger circuit is usually split into multiple parts that are completely verified separately. This is always the case when the full circuit does not only execute one sequence of operations but there are multiple parts of the circuit that execute operations concurrently. In this case, the circuit must be verified by a number of operation automata.

The circuit parts that are verified by one operation automaton are called clusters. Two circuit elements usually belong to different clusters, if they process events from two independent sources.

Many circuits, e.g., the IP blocks from the previous section 3.5.5, consist of multiple clusters. Processor peripherals often consist of a configuration block which consists of the configuration registers and a slave interface to write or read these configuration registers. This block is often verified as a separate cluster, while the actual function of the peripheral is verified in one or more other clusters.

When complete verification is to be applied to processors, the entire pipeline is kept in one cluster. But if there is a more demanding prefetch unit, it usually does not work in the rhythm of the pipeline, but takes a life of its own. For example, the prefetch unit may be controlled by the allocation of the instruction bus. Accordingly, there are fifos between the prefetch unit and the decode stage of the processor pipeline, with which the different rhythms decoupled. In this case, the prefetch unit and the processor pipeline will be verified from the decode stage in different clusters.

Another reason to split a complete verification in different parts can be complexity limits of formal assertion checkers. But IPC can successfully treat relatively large clusters, such as a complete processor pipeline, so that the division of the circuit in clusters usually follows functional requirements rather than complexity problems.

3.5.7 Complete Verification of Circuits with Multiple Clusters

A circuit with multiple clusters is completely verified by executing complete verification of the clusters. It is obvious that this complete verification can be executed with the rules that were just introduced for the verification of a SoC that consists of multiple IP blocks. To this end, an own circuit model would be built for each cluster, and all relations between cluster inputs must be provided by explicit integration assumptions, which need to be justified by explicit integration commitments of the neighbour circuits.

But some effort related to explicit integration assumptions can be avoided, if the full circuit with its multiple clusters is known in advance. In this case neighbouring clusters can sometimes be used as implicit constraints. E.g., one cluster might have an output signal that is a delayed version of another output signal. If these two outputs are inputs to a second cluster, the relationship can be made explicit in an integration assumption of the second cluster. But this explicit integration assumption is not necessary if the model of the second cluster also contains the first cluster and how it derives one output from the other.

This approach is however limited. If the larger model contains dependencies from a cluster output to cluster inputs, i.e. through the environment of the cluster, then these dependencies may influence the formal verification. An example is given in section 6.4.3. It shows that dependencies from cluster outputs to cluster inputs must be removed during the model generation. This is done with the “signal cutting” operation that can optionally be executed during the model generation. With this signal cutting operation, an internal signal is turned into a primary output and a primary input. The primary output provides the value of the signal, the primary input is connected to all parts of the circuit that consume the signal.

The trick is to carefully select those cluster inputs that must be cut to become primary inputs of the verification. Two requirements determine the cluster inputs that must become primary inputs. One requirement is that cyclic dependencies between clusters are to be removed. The second requirement is that reactive constraints must not introduce such cyclic dependencies.

To determine the cluster inputs that must become primary inputs, a cluster graph is created. Nodes of this graph are the clusters, and if this cluster drives a signal that is used by another cluster, the cluster graph contains an edge from the cluster that produces the signal to the cluster that consumes it.

This cluster graph needs to be analyzed to determine a hopefully small set of edges, such that the cluster graph becomes cycle free, if the edges of the set are removed from the cluster graph. Every removed edge from one cluster to another corresponds to a number of signals from the first to the second cluster. All these signals need to be cut.

Complete verification is then executed separately for each cluster, but on the model with the cuts. Many cluster inputs will remain internal signals that are generated by other clusters. The complete verification of each cluster creates an integration assumption for that cluster. If an integration assumption contains a reactive constraint, all cluster inputs in this constraint need to become primary inputs by further signal cutting. This gives the model on which the compositional complete verification is executed with the approach described above.

The special feature of this approach is to use a model in which some signals being cut between clusters, but others may remain intact. The cuts are necessary to ensure that the complete verification of the entire circuit can actually be decomposed into a complete verification of the clusters. The advantage of this approach is that the verification can be performed on a model in which many input signals remain internal signals such that some integration constraints need not be made explicit. This saves effort.

This shall be exemplified by comparison with the example in section 3.5.2. It shall be assumed that the SDRAM interface is included in the processor and is developed and verified with it. Processor and SDRAM interface then form two clusters and their cluster graph is cyclic. It is advisable to eliminate the edge from the processor to the SDRAM interface to make the cluster graph cycle free. Then the verification of the processor and SDRAM interface is carried out on a common model in which the signals ready and rdata remain internal signals that connect the SDRAM interface to the processor. The verification is executed cluster by cluster, but it uses the same common model with a few cut signals. Thereby, the integration condition about the pulses of the ready signal is not required, which was necessary in section 3.5.2 was necessary to ensure proper processor function.

3.6 Complete Verification in Industrial Application

Operation automata and properties, IPC and the completeness checker form an advantageous symbiosis in this approach. The symbiosis has proven itself over nearly 10 years in industrial practice and has been made available in the product OneSpin 360 MV [OneSpin undated]. Since the completeness checker is used on operation properties, it produces resource friendly proof goals, such that it can handle properties about large circuits. IPC is particularly suitable for operation properties, in particular also because this requires only relatively simple reachability conditions. If such conditions are needed, the approach allows user input, by which a higher understanding of the circuit can be incorporated into the verification. If the user provides reachability conditions, the completeness checker ensures that they are not incorporated into the verification without proof. All in all, the symbiosis allows the independent formal investigation of modules, which forms an alternative to simulation-based module verification. The features of the formal approach are summarized in the sequel:

3.6.1 Highest Quality of Implementation

The approach described in this thesis has already been used in many verification projects. It achieved highest circuit quality with a significantly simpler procedure than with theorem provers and with a much higher productivity. In this regard, the complete verification is the first of its kind because in contrast to simulation, a properly configured complete verification does not allow that errors escape [Büttner 2007]: Simulation overlooks errors in unstimulated situations, but also stimulated errors are only discovered by the simulation when the simulation stimulus propagates the misbehavior to the interface of a checker, which moreover needs to be sensitive for the misbehavior. To mitigate the vulnerabilities of simulation-based verification some approaches are offered [Grosse / Hampton 2005, Certess undated] which are based on fault injection and mutation analysis [Offutt / Untch 2000].

Complete verification does not only mitigate such vulnerabilities, but it prevents them reliably and automatically. Errors escape complete verification only, if the specification was formalized in the same wrong way that was already used during the circuit development, or if the constraints on the primary inputs were formulated too restrictivly. This problem is shared by all verification procedures, as long as they start from informal specifications. For complete verification this case is quite unlikely because of the systematic alignment between informal specification, formalization and implementation.

Specifications that are formalized by properties are available for complete verification firstly in the form of reusable assertions about protocols, and secondly e.g., as formalized architecture descriptions for processors [Bormann / Beyer / Skalberg 2006], that were also the starting point for the development of fast simulators. If a verification starts out from such a formalized specification, the problem of falsely accepted errors is no longer relevant.

3.6.2 Quality of Specifications

If the specification is carefully translated into operation properties, the completeness checker also identifies specification gaps, and thus helps to complete the specification.

The approach is anyhow quite tolerant wrt. the quality of the specification, which is an advantage for industrial use. In various example applications little written information about the circuit was available apart from the module name. Even then it is still possible to work effectively provided that an authority like a system architect can make binding statements about the expected behavior. If there is no such authority, the verification reduces to plausibility tests e.g., about the synchronization of the circuit with its environment. Even these plausibility tests are still quite powerful verification tools because of the intuitive representation of operations of the circuit.

The final circuit description by operations states precisely and on a transaction level, how the specification was interpreted.

3.6.3 Constraining

Complete verification is designed so that each hint provided by the user must be proven. Constraints are therefore only required on the primary inputs of the full circuit that is to be completely verified. Here, they must also be known for simulation-based approaches. The application of complete verification allows that the designer focusses on one operation after another, and hereby always takes care of a few constraints and their exact formulation at a time. This already simplifies the correct constraining for complete verification compared to other formal procedures. On top of that, complete verification provides more critical plausibility checks about the constraints than other formal procedures. When integrating IP blocks to SoCs, compositional complete verification ensures that the constraints of the different IP blocks are indeed justified by the neighboring blocks.

3.6.4 Verification Process

Complete verification is an independent, highly automated verification process which is suited to provide far better quality than simulation-based methods. Complete verification is therefore an alternative to simulation, once the actual verification phase begins. In previous design steps, simulation appears indispensable to give designers a feeling for the circuit and its behavior.

Unlike ABFV the effort for complete verification therefore does not add to the effort for simulation-based verification, but complete verification can take over the full task and hence the full effort budget of the simulation based module verification.

Verification planning [Bormann et al. 2007] for complete verification is executed like project planning. The goal is to produce a net plan for the verification tasks that are related to the clusters. The completeness checker ensures full coverage of the specification during the execution of the verification. The net plan is the basis for progress reporting in the verification project. In contrast, verification planning for simulation requires much forethought [Bergeron et al 2005], many reviews and therefore more effort.

When carrying out complete verification, the verification engineer focuses on individual operations and their interaction, and thus on a level of abstraction similar to the transaction-based verification by simulation. This avoids the difference of abstraction levels which exists between ABFV and simulation based verification. The operations are taken from the specification, the degrees of freedom provided by specification are filled in according to the the implementation decisions. In addition it is taken from the implementation, how the operations interact with one another.

The examination of the circuit description is thus an integral part of complete verification. The process pursues the approach that an engineer applies anyhow, if he or she starts familiarization with a circuit and its description. The first part to be examined is the central control unit of the circuit and from there an understanding is built, how this control unit influences the data path. This understanding is reviewed, when the operation properties are developed step by step, when intermediate versions of the operation properties are verified, and when the completeness of the properties is checked.

Experience has shown that much less interaction is needed with the designers as in simulation-based verification projects or during application of ABFV [Mitra 2008]. Nevertheless errors localizations are usually very precise up to functionally tested proposals for corrections.

Applications of the approach over multiple years [Bormann / Spalinger 2001, Bormann 2003, Winkelmann et al. 2004, Thomas et al. 2004, Bormann / Blank / Winkelmann 2005, Bormann et al. 2007, Loitz et al. 2008] shows that the risk to accept errors is quite low, and that it is likely that all errors in the circuit description are found, which are course many more errors than by simulation-based verification.

3.6.5 Termination Criterion

One of the most striking features of complete verification is the existence of a termination criterion that is based on the proof that all functionality of the circuit is verified by at least one operation property. This is different from all other verification approaches which deploy more vague termination criteria which are satisfied before all functionality is verified. Hence all these approaches run the risk that errors escape because they were never stimulated or observed. Moreover, the operation based verification methodology provides a recipe how to achieve this termination criterion with a structured effort. The common termination criteria provide less insight about what to do to satisfy it.

The termination criterion of complete verification is reached, if the following requirements are fulfilled:

- All opertion properties are successfully checked against the RTL of the circuit.
- The completenss checker proves that the properties verify all circuit functionality.
- The complete verification of all clusters can be combined to a complete verification of the whole circuits.
- All predefined assertions are checked.

This is automatically checked in OneSpin 360 MV. Therefore it is the first verification tool with a non-heuristic termination criterion. If this termination criterion is satisfied, the properties and in particular the associated timing diagrams form a readable and proven design documentation.

Whether the verification really assures correctness, depends on whether the properties actually represent the specification, and whether the constraints actually model the environmental conditions under which the circuit should operate. The former can be ensured by reviews, the latter can be demonstrated by simulation or formal when the circuit is embedded in a larger system.

3.6.6 Productivity

Experienced verification engineers, who are willing and able to understand details of RTL descriptions, achieve this termination criterion and the related high circuit quality significantly faster than with simulation-based verification [Bormann 2003, Bormann / Blank / Winkelmann 2005], which even provides less quality. In many application examples experienced verification engineers completely verified on average 2,000 to 4,000 lines of RTL code per person month, the record is at 8000 lines per person month. Moreover, the computing power for the verification is much smaller than in simulation.

The approach abstains from complex custom abstractions, and thus avoids one of the reasons for upredictable efforts, that R. Mitra [Mitra 2008] complains about. The approach becomes slower when the RTL code is conceptually unclear, unstructured, or generally difficult to understand, i.e. badly described in the sense of Software Engineering. Such bad RTL code is sometimes created right from the first version, when the circuit concept was not well thought through and the circuit was subsequently improved incrementally due to feedback from e.g., simulation based verification. But more frequently, even worse RTL code arises in particular in the context of reuse, when if a circuit is repeatedly modified by several designers who did not fully familiarize with its concept before they started to modify the code. Complete verification can be particularly useful for such a circuit, because it allows rediscovering the concept, to detect unnecessarily complex or large code, and to ensure a verification of the corner cases that usually arise from such a development history.

3.6.7 Integration with Simulation

When completely verified modules are embedded in a larger system, that is to be verified by simulation, the verification plan need not state coverage goals for this module. It only needs to be checked sufficiently deeply that the system complies with the constraints of the complete verification. This provides a pragmatic approach for a cross-technology verification planning and coverage evaluation.

System-wide verification objectives usually require that the quality of simulation is monitored by coverage. The completely verified modules offer a simple way to define functional coverage on the basis of the operations of the modules: The coverage conditions are met whenever the corresponding operations are carried out. For this it is sufficient to turn the assumptions of the corresponding operation properties into functional coverage points. This avoids that the operations themselves are intensively investigated by the system simulation, which would be a redundant effort after a complete verification.

3.6.8 White-box Verification

Practitioners regard it a weak point of complete verification that it depends on the concrete implementation of the circuit. In fact, changes of the circuit RTL usually need to be accompanied by changes in the properties, because otherwise the test of some operation property fails.

This adjustment often requires further changes of the property set, to make the property set complete again. The change effort is usually low for successive versions during a design phase. It increases when circuit parts are discarded and reimplemented following an entirely different concept.

A different picture emerges in relation to circuit IP, i.e. reusable modules that are maintained over years and many revisions by various designers. In such a context there is often little time to propertly adapt simulation based verification. This leads to the phenomenon of ageing of the simulation-based verification, which has the consequence that the verification checks less and less critical situations of the modified functionality, and hence overlooks more and more errors. If this leads to escapes, i.e. error that are only detected during the product test, or even from in the field, the carelessness of the module verification is excused with the outdated verification technology, although this verification technology was well capable to secure a reasonable quality of the first version of the module.

Complete verification prevents the ageing of the verification. However, proper integration of complete verification should modify the development and maintenance processes of the circuit. Goal should be to closely interleave the RTL development and the related modification of the operation properties. Ideally, the RTL changes are planned on the higher abstraction level of the operation properties. This means that the operation properties are changed first, and then the RTL is suitably adapted. An advantage of such changed development and maintenance processes is the availability of a proven design documentation prior to every revision of the circuit RTL. This helps a (potentially new) developer to a quick familiarization with the concept behind the RTL. Experiments suggest that this does not only lead to highest quality of the IP after the changes, but it also reduces the amount of changes, and leads to conceptually clearer circuit description than what can be achieved with the state of the art approaches.

3.6.9 Does Complete Verification Help Against the Verification Gap?

Does complete verification meet the dream to reduce the verification gap with formal verification? The leaps in quality and verification productivity that were observed in many successful verification projects seem to suggest it. But it is a disruptive approach, which requires a completely different education of the verification engineers, which cannot use existing verification code, for which there is almost no verification IP, and which requires changes in the design, maintenance and verification processes.

4 Basics

4.1 Synchronous Circuits

All efforts on synthesis-based design are targeting digital circuits that are implemented by FPGAs or ASICs. Without loss of generality, this thesis focuses on synchronous circuits with one clock. Such circuits consist of combinatorial gates and flip-flops. The combinatorial gates shall be connected without forming combinatorial loops, such that they do not implement storing behavior. This work only considers circuits with flip-flops that are triggered by a rising clock edge.

Pecularities of the timing behavior of input signals shall be excluded. Therefore, it is always assumed that the circuit is (or can be) embedded into a larger circuit that satisfies these assumptions.

As usual in synchronous design, it is assumed that the latency of combinatorial paths from the output of a flip-flop to the input of another flip-flop is small enough, so that the input to the second flip-flop has a stable value, whenever a rising clock edge occurs. This assumption justifies both the zero-delay-descriptions that are usual for synthesis based design as well as its mapping to a Mealy automaton.

4.1.1 Automata

This Mealy machine represents the values of the signals of the circuit just before each rising clock edge [Bormann et al. 1995]. It is characterized by

- A set $\mathbb{i}$ of input variables and a set of values $\mathcal{W}(\mathbb{i})$ that these variables can assume. These variables should be the input signals of the circuit.
- A set $\mathbb{s}$ of state variables and a set of state values $\mathcal{W}(\mathbb{s})$ these variables can assume. These variables describe the values of the flip-flops in the circuit.
- A set $\mathbb{z}$ that contains the set of internal signals $\mathbb{u}$ and the set of output signals $\mathbb{o}$. All elements of $\mathbb{z}$ are output signals from an automata theoretic point of view. The sets of values are denoted $\mathcal{W}(\mathbb{z})$, $\mathcal{W}(\mathbb{u})$ and $\mathcal{W}(\mathbb{o})$. It is $\mathbb{z} = \mathbb{u} \cup \mathbb{o}$ and $\mathcal{W}(\mathbb{z}) = \mathcal{W}(\mathbb{u}) \times \mathcal{W}(\mathbb{o})$.[1]
- A state transition function $\Delta: \mathcal{W}(\mathbb{i}) \times \mathcal{W}(\mathbb{s}) \to \mathcal{W}(\mathbb{s})$,
- A function $\Lambda_{\text{out}}: \mathcal{W}(\mathbb{i}) \times \mathcal{W}(\mathbb{s}) \to \mathcal{W}(\mathbb{o})$ to compute the values of the outputs
- A function $\Lambda_{\text{int}}: \mathcal{W}(\mathbb{i}) \times \mathcal{W}(\mathbb{s}) \to \mathcal{W}(\mathbb{u})$ to calculate the values of the internal signals
- A condition $\varrho: \mathcal{W}^r(\mathbb{i}) \to \mathcal{B}$ about reset sequences[2]. These reset sequences are input traces of length r, that take the circuit into its reset state. It must be ensured that the circuit always executes one of the reset sequences before it processes normal inputs. This condition replaces the conditions about initial states that are usually introduced for automata.

The circuit firstly executes a reset sequence $(i_{-r}, i_{-r+1}, i_{-r+2}, \dots, i_{-1})$ which starts in an arbitrary state of the circuit, whereby the circuit produces a sequence $(s_{-r}, s_{-r+1}, s_{-r+2}, \dots, s_{-1})$ of states, which starts at an arbitrary state $s_{-r} \in \mathcal{W}(\mathbb{s})$ and ends in a reset state s_0. Starting from s_0 a sequence $(i_0, i_1, i_2, \dots)$ of input values is processed by the machine. This leads to

[1] Flip-flops are indeed represented twice, as part of the set of states $\mathbb{s}$ and additionally as a part of the set of internal signals $\mathbb{u}$.

[2] $\mathcal{B}$ represents the set $\{true, false\}$ of Boolean values.

values $(s_0, s_1, s_2, \ldots)$ of the flip-flop signals. The input stimulus $(i_{-r}, i_{-r+1}, i_{-r+2}, \ldots, i_{-1}, i_0, i_1, i_2, \ldots)$ therefore leads to sequences $(u_{-r}, u_{-r+1}, u_{-r+2}, \ldots, u_{-1}, u_0, u_1, u_2, \ldots)$ and $(o_{-r+1}, o_{-r+2}, \ldots, o_{-1}, o_0, o_1, o_2, \ldots)$ of values of internal and output signals, which are calculated according to

$$s_{j+1} = \Delta(\mathrm{s_j}, \mathrm{i_j})$$

and

$$o_j = \Lambda_{\mathrm{out}}(\mathrm{s_j}, \mathrm{i_j}) \text{ bzw. } u_j = \Lambda_{\mathrm{int}}(\mathrm{s_j}, \mathrm{i_j})$$

where $j \geq -r$. Δ is called the state transition function of the automaton and Λ_{out} the output function.

4.1.2 Traces

A sequence $(X^{(j)})$ of values of a set of signals is called a Trace X of these signals. Typically, the index j counts the cycles of a clock. The individual $X^{(j)}$ is called the j-th element of the trace. Traces can be finite or infinite. If a set $\mathbb{x}$ of signals is given, the set of all traces of $\mathbb{x}$ is denoted $\mathcal{W}^{\infty}(\mathbb{x})$. To form $\mathcal{W}^{\infty}(\mathbb{x})$, the signals shall be treated as free variables, i.e. $\mathcal{W}^{\infty}(\mathbb{x})$ should also contain those traces that would not be created by a circuit that implements the signals in $\mathbb{x}$.

A finite initial segment $(X^{(0)}, X^{(1)}, X^{(2)}, \ldots, X^{(n)})$ of a trace is called the trace prefix. n is called the largest index of the prefix. Two traces can have a common prefix. The separation index $SEP(X, Y)$ shall be the largest index of the largest common prefix of X and Y. If $X = Y$ then $SEP(X, Y) = \infty$.

Two traces X and Y can form a joined trace (X, Y) that consists of the pairs $\left((X^{(j)}, Y^{(j)})\right)$ of elements of the two traces. There shall be no distinction between a pair of traces and the trace of pairs of their elements.

According to section 4.1.1 an automaton M can be identified with a predicate M about traces. $M(I, S, U, O)$ is satisfied if and only if the automaton M can map an input trace I to the output trace O, the trace of the state variables S and the trace of internal signals U. This mapping is not unique, because firstly it reflects the reset behavior, and secondly is shall not be assumed that the reset state is unique. The trace of the state variables often plays no independent role, because this information is already contained in the trace of internal signals U. Therefore, the automaton is usually regarded as a predicate $M(I, U, O)$. A trace (I, U, O) that satisfies the predicate $M(I, U, O)$ is called a trace of the machine M. The Traces start with the index $(-r)$ to account for the reset sequence.

4.2 Assertions, Constraints and Properties

4.2.1 Roles of Assertions, Constraints and Properties

For the traces I, U and O of an automaton, conditions $\beta(I, U, O)$ can be defined that contain restrictions or expectations about the behavior of the automaton. These conditions are often built from formalizations $\hat{\beta}(t, I, U, O)$ that are stated relative to an arbitrary but fixed time point t. They should hold for all $t \geq 0$, i.e.

$$\beta(I, U, O) = \bigwedge_{t \geq 0} \hat{\beta}(t, I, U, O)$$

The conditions $\hat{\beta}$ may be assertions, constraints or operation properties. Constraints describe behavior that is not to be questioned, either because it shall be assumed that the environment of the circuit under test behaves accordingly, or because a verification should be limited to the situation described in the constraint. Constraints are used for the proof of assertions, of properties, or for the completeness check and they limit the proof accordingly. Constraints are global, i.e. they are valid for the entire verification.

Assertions contain either verification goals (see section 2.8), reachability conditions (section 3.3.3) or local constraints (section 3.4.3). Assertions and operation properties formalize proof goals as described in section 3.

4.2.2 Dependencies

In the sequel the term object shall be used to summarize assertions, constraints, operation properties, and completeness. A verification is executed by proving all objects except the constraints. Many of these proofs are executed under the assumption that other objects are valid. Practical experience (and common practice in maths) shows, that it is often helpful to postpone the actual proof of an object, e.g. because the concentration on one part of a circuit is helpful for the exact formalization and the proof of several objects.

The concept of dependencies helps to postpone the proof of certain objects in favour of a more suitable order in which they are worked at. To that end, the user annotates each object with the objects that are to be postulated during its proof. E.g., operation properties and many assertions are proven with IPC under the postulate of other assertions or constraints. Completeness and complex assertions are proven with the completeness checker, again postulating assertions, constraints and operation properties (see section 4.3.7). Except for potentially increased complexity of the algorithmic proof, the annotation of dependencies that are actually not needed for the proof does not harm.

To avoid circular reasoning, the graph of dependencies must be cycle free. If this is the case, the objects can be proven in any order that the user thinks best.

4.2.3 Reactive Constraints and Assertions

Constraints describe the behavior of the input signals of a circuit by providing a condition about the input trace that is satisfied if the input trace can occur during the operation of the circuit, and dissatisfied otherwise. Only simple constraints can be described by conditions, which depend only on the input signals of the circuit. In most cases, the circuit is part of a larger system with feedback from the output signals of the circuit to the input signals. These effects of the circuit environment are also formalized with constraints, but these constraints depend on output or internal signals. Such constraints are called reactive.

In compositional verification also reactive local constraints can occur, which are assertions from a global point of view. Analogously to the above, such assertions are called reactive assertions.

4.2.4 Safety and Liveness Conditions

A safety condition describes that an undesired situation never occurs. A liveness condition is met if a desired situation will eventually occur in the future.

A condition is a safety condition iff it can be refuted (i.e. proven to be wrong) on a finite prefix of a trace. Hence if a safety condition $B(L)$ is refuted on a trace L, then there is an index N such that $B(L')$ is violated on any trace L' that has a common prefix of length N with L. Typical safety conditions require that some Boolean condition about a choice of signals is always true, or that a second event always occurs after a first event occurs within a limited number of clock cycles.

Liveness conditions require the investigation of entire traces. A typical liveness condition requires that a start event is eventually followed by a reaction, without specification of a maximum waiting time. The algorithmic proof of liveness assertions is usually very expensive. Complete verification therefore avoids such assertions. This is possible because circuit verification uses liveness conditions in two situations for which there are alternative solutions:

The first situation is that the waiting time between the start event and the reaction is determined only by the circuit. In this case, the methodology of section 3.2.3 requires an exact determination of the waiting time which will be encoded in the corresponding operation property. By this, the operation property becomes a safety property and can therefore be verified with IPC.

In the second case, the circuit response occurs only after a second external event which must arrive after the start event. A counter example for this property consists of the start event and the second event, which arrives after a finite number of clock cycles. For the reasons described above, the circuit response is expected after a number of clock cycles that is again given by the circuit description. Hence, the counter example is again finite, which demonstrates that the property is a safety property. See section 4.2.5 for a discussion of how this behavior is captured in an operation property.

These considerations show that complete verification uses only safety properties and safety assertions. Liveness assertions may only occur as externally provided proof goals. They will be shown on the basis of operation properties and other safety assertions in a separate verification step.

4.2.5 Treatment of Endless Waiting Times

IPC, the proof technology that underlies complete verification (cf. section 3.3) cannot deal with endless waiting times between a start and an external event in the sense of the previous section 4.2.4. It is only possible to formalize finite waiting times, where the upper limit should be chosen small to keep complexity low. This limitation relates to the property check well as to the completeness check.

The user can however perform complete verification in full generality. To that end waiting periods must be divided into three phases which are described by separate properties: In the first phase the circuit enters the waiting period and reaches a waiting state. The second phase proves that the circuit remains in the waiting state if the external event is absent.The third phase assumes that the circuit is in the waiting state and the external event arrives, and proves that the circuit leaves the waiting state and continues to operate as expected. With this approach, all waiting states become conceptual states.

The application of this procedure is sometimes quite demanding, because an operation is split into three operations at a quite unnatural time point. For this time point the arguments from section 3.3.5 about the simplicity of reachability conditions between operations are invalid. Hence it can be quite expensive to identify the reachability conditions in the wait states. Further, the split makes it more difficult to understand an operation.

The alternative approach is to restrict the waiting time to a maximum that can be handled by the IPC-prover. This approach leaves a verification gap, because it does not reason about longer waiting periods. However, it is often obvious from the circuit behavior that this verification gap is irrelevant. The advantage of this alternative approach is the more comprehensive description of the operations and the related better understandability of the circuit behavior.

In the following description the theory is developed so that it is also applicable to operations with infinite delays.

4.3 Property Language ITL

A formal functional verification is the more accepted, the better the behavior to be proven can be expressed in the language that is accepted by the Property Checker. The goal in developing a property language should therefore be that a user can concentrate his/her energy on the identification of the behavior to be proven. As little effort as possible should be spent to actually formalize the proof goal. Deviations between intended and actually formalized behavior should be as rare as possible, and they should quickly become obvious during debugging. Furthermore, the property language must be able to represent the cause-effect relationships of the signals of the interfaces of a circuit that are involved in the processing of a transaction. To satisfy these demands, the property language ITL was developed [Siegel et al. 1999], based on experiences that have been made with a previous version [Bormann 1995].

ITL was backed up with the already introduced graphical representation (cf. Figure 7) [Bormann 2001, Paucke 2003], which closely follows timing diagrams and hence is much more intuitive than other graphical formalisms for describing circuit behavior [Schlör 2001, Peukert et al. 2001]. In addition, a method was developed to translate ITL properties into synthesizable simulation monitors [Bormann 2003, Beuer 2005].

4.3.1 Treatment of Time

A first example property has already been described in section 3.2.5. It shows that a main characteristic of the language is the explicit treatment of time, which is measured in clock cycles. All operation properties are formalized relative to an arbitrary but fixed point in time that is represented by the keyword "t". Finite offsets of t are expressed by $t + n$ or $t - n$ where n is an integer.

Variable offsets are e.g. required to describe how the circuit waits for a synchronization event from the communication partner during the execution of a transaction. They are represented by time variables that are defined relative to t. To describe what happens if a synchronization event does never arrive, the time variables can take the value infinity, which is represented by "$".

Time variables that can be infinite are called infinite time variables. Otherwise they are called finite time variables.

4.3.2 Timed Conditions

The properties are determined by conditions of the form

```
at <time point>: <Boolean condition>;
during [<time point1>, <time point2>]: <Boolean condition>;
within [<finite time pt1>, <finite time pt2>]: <Boolean condition>;
```

The first condition requires that the Boolean condition is true at the given time point, the second condition requires that it applies to all time points in the interval, and the third, that it becomes true at least once in the given interval. The intervals always contain their boundaries. If the left interval boundary is larger than the right, the interval is empty. Then the "during" condition is trivially satisfied and the "within" condition is trivially violated.

The time points have the form $tvar + n$ or $tvar - n,$ where $tvar$ can be t or a time variable. In within-conditions infinite time variables are not allowed. When the time specification after "at" contains a time variable and this time variable is set to infinity, the whole timed condition is trivially satisfied. If both limits of a during interval are infinite, the corresponding condition is trivially satisfied. If only the right boundary is infinite, the Boolean condition must be met for all time points starting from the left interval boundary.

The Boolean conditions have a syntax that is similar to either Verilog or VHDL. Depending on the underlying RTL language, ITL is also called VLI (VeriLog Interval Language) or VHI (VHDL Interval Language). The Boolean conditions are evaluated at the times indicated in each case. The return values of arithmetic bit vector operations are always wide enough to represent the exact result. Functions prev and next allow evaluation of expressions at other time points than the one indicated.

4.3.3 Basic implication

The basic structure of an ITL property is given by various text blocks. Two of them are particularly important: One block starts with the keyword "assume" and describes preconditions of the property. The other block with the keyword "prove" describes the proof goal of the property.

The assume part describes the situation that is to be checked with the property, and the proof part describes the expected behavior in this situation.

4.3.4 Assume and Proof Part of a Property

Assume and prove part of a property are formed by a number of timed conditions as in the preceding section 4.3.2. If these conditions are separated by a semicolon, they are combined by a conjunction.

Disjunction is expressed by the construct

```
either
      Condition
or
      Condition
or
      ...
end either;
```

Assume and proof part of a property are functions on the traces $L = (I, U, O)$ of the automaton that represents the circuit. The return values of the functions are 0 (false) or 1 (true). Besides the trace, t and all time variables $tvar_i$ are arguments of this function. The assume predicate $\hat{A}(t, L, T_1, T_2, \dots)$ indicates whether the assume part is satisfied on the Trace L for a given choice of time variables. Similarly, the proof goal predicate $\hat{C}(t, L, T_1, T_2, \dots)$ tells whether the proof part is satisfied.

4.3.5 Time Variables

Time variables are defined in a separate block that is introduced with the keywords "for timepoints". A time variable is introduced by the expression

```
        T j = B j + n j .. m j awaits w j;
```

B_j is either t or an already introduced time variable. n_j is an integer and m_j either an integer or \$, i.e., infinity. w_j is a waiting condition about input, output, or internal signals of the circuit. If B_j has a finite value, T_j is assigned the first value in the interval $[B_j + n_j, B_j + m_j]$ where w_j becomes true. If w_j is never satisfied in the interval, T_j becomes $B_j + m_j$ if m_j is finite, and $T_j = \$$, if $m_j = \$$. Also $T_j = \$$, if B_j is already infinite.

Because of their dependency on the the wait conditions, the T_j can therefore be considered functions $T_j(t, L)$ that return the value of the time variable for a given t and trace L. If these functions are substituted into the assume and proof predicate, $A(t, L) = \hat{A}(t, L, T_1(t, L), T_2(t, L), \dots)$ and $C(t, L) = \hat{C}(t, L, T_1(t, L), T_2(t, L), \dots)$ are created.

4.3.6 Freeze Variables

To describe data transport through the circuit, values can be frozen in a separate variable. They are then available throughout the property. Freeze variables are defined in a separate block that begins with the keyword "freeze". Syntax of the definitions is

```
        Fj = expressionj @ timepointj ;
```

Semantically, freeze variables behave as if they were substituted everywhere by the right side of the assignment.

If a freeze variable is defined relative to an infinite time variable T, it may be used only in expressions that become irrelevant if the time variable T assumes the value infinite. Such expressions are “at” constructs relative to T or relative to a time variables that depends on T, or during constructs, in which the left interval boundary has this shape.

4.3.7 Dependencies

Assertions and properties have an optional dependencies block, where dependencies according to section 4.2.2 are specified by listing the names of the dependencies.

A dependency is a predicate $d(t, L)$ with the definition of an assertion or a constraint. All dependencies must be satisfied on every trace of the underlying model at any time $\geq$. IPC assumes the dependencies on the examination window of the property or the assertion.

All dependencies are summarized to the condition about traces

$$D(L) = \bigwedge_{t\geq 0} d_1(t, L) \wedge d_2(t, L) \wedge d_3(t, L) \wedge \ldots$$

4.3.8 Assertions and Constraints

The syntax of assertions and constraints according to section 4.2 is

```
      constraint <name> :=
            <Boolean condition>;
      end constraint;

      assertion <name> :=
            <Boolean condition>;
            dependencies: <list of assertion- and constraint names>;
      end assertion;
```

with the Boolean conditions of section 4.3.2. These Boolean conditions may use $prev(\ldots)$ and $next(\ldots)$ to refer to signal values at the previous or next time point and they shall apply to all time poins after the reset sequence. Because of the reasons provided in section 4.2.4 ITL allows only the formalization of Safety assertions and Constraints.

4.3.9 Properties

The syntax of a property is:

```
      property <name>;
      dependencies: <list of assertions and constraints>;
      for timepoints: <time variable declaration>;
      freeze: <freeze variable declaration>;
      assume:
            <assume part>;
      prove:
            <prove part>;
      end property;
```

All blocks are optional except for the proof part. Their syntax and semantics were described in previous sections.

A property expresses the relationship

$$P(t, I, U, O) = (A(t, I, U, O) \Rightarrow C(t, I, U, O))$$

It holds on a trace (I, U, O) of an automaton, if it holds at any given time $t \geq 0$. A property holds, if it holds on all traces of the automaton that also satisfy the dependencies. This is summarized by the formula

$$\bigwedge_{I,U,O}\left((M(I,U,O) \wedge D(I,U,O)) \Rightarrow \bigwedge_{t\geq 0} P(t,I,U,O)\right)$$

where D summarizes all dependencies.

Since all dependencies are consequences of the constraints and the automaton, the proof of a property implies that all traces (I, U, O) of the automaton satisfy the property if they satisfy the constraints:

$$\bigwedge_{I,U,O}\left((M(I,U,O) \wedge Constr(I,U,O)) \Rightarrow \bigwedge_{t\geq 0} P(t,I,U,O)\right)$$

4.3.10 Example

As an example, a property about the execution of a read operation by a processor shall be presented. The processor is implemented by a pipeline with the stages dec (decode), ex1 (execute 1), ex2 and wb (write back). The respective stage can be stalled with the signals dec_stall, $ex1\_stall$, $ex2\_stall$ and wb_stall. If stalled, a stage does not pass any data to the next state. Accordingly the property contains time variables t_ex1, t_ex2 and t_wb which denote the time points when a stall is released and the instruction moves from the respective stage to the next.

The stall signals must satisfy some requirements to ensure that the pipeline works as expected. These requirements shall be summarized in the constraint stall_relation.

The ITL description of the property will be presented next and discussed in the following paragraph:

```
property read_no_violation is
dependencies: no_reset, stall_relation;

for timepoints: t_ex1 = t     + 1 .. $ awaits ex1_stall = '0',
                t_ex2 = t_ex1 + 1 .. $ awaits ex2_stall = '0',
                t_wb  = t_ex2 + 1 .. $ awaits wb_stall = '0';

freeze: instr_dec = instr @ t,
        read_val_ex2 = read_val @ t_ex2;

assume:
at    t:                opcode(ip_instr) == READ_INSTRUCTION;
at    t:                dec_valid = '1';
at    t:                dec_stall = '0';

at t_ex2:               violation = '0';

prove:
during [t+1, t_ex1]:    adr_out == address(instr_dec);
during [t_ex2+1, t_wb]: result_wb == read_val_ex2;
end property;
```

At the start of the property a read instruction shall wait in the dec stage which is not stalled. This gives the three conditions in the assume part for time t. Also at the time t the encoding of the instruction is expected on the signal instr and hence frozen in the freeze variable instr_dec, which allows access to the encoding whenever needed in the property. The read address is output in the ex1 stage. It is assumed that the data memory reacted when the EX2 stage can continue to operate. The reaction of the memory is either a message about violated access rights provided by the signal $violation$, or the read data itself. In the case of this property, the access rights are satisfied. Therefore, the data arrives at time t_ex2 and gets stored in the Freeze variable read_val_ex2. This freeze variable will be used to verify the data transport by comparison with the signal result_wb when the instruction is in the wb stage. It is shown that the output of the wb-stage provides the read data, for example, to the register file.

4.3.11 Graphical Representation

An important subset of ITL properties, namely properties without either- or within-statements can be represented graphically by color-coded timing diagrams [Bormann 2001, Paucke 2003]. This representation is very helpful especially for protocol-dominated circuits. An example of the graphical representation is given in Figure 11 which visualized the property of section 4.3.10. The intuitivity of this representation is a big advantage over other graphical formalisms [Schlör 2003, Peukert et al. 2001].

The color code in the generalized timing diagrams is

- blue for the assume part
- red for the proof part
- green for the waiting conditions of the time variables and
- black for the time points at which other conditions access the signal value. When accessed through freeze variables, their names are shown. If otherwise accessed, e.g., by $prev$ or $next$, no name is entered.

The time range in which a time variable can vary is indicated for each signal by a broader column and by an arrow in the first line of the timing diagram. The arrow is intended to indicate that the width of this cell is actually variable. In extreme cases, the column can get the

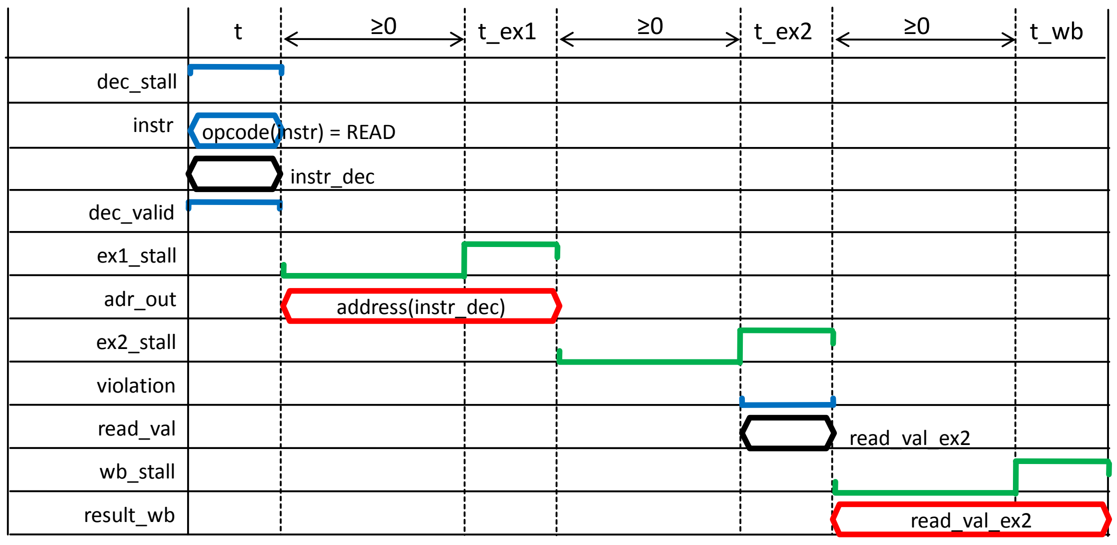


Figure 11: Graphical representation of the property read_no_violation

width 0 and thus vanish, when the wait condition is satisfied immediately. The maximum width of the cell is given by the upper bound for the wait time that is indicated above the arrow. When a signal reacts to a synchronization signal with some delay, the respective broader cell shifts corresponding to the delay. Figure 12 shows an example.

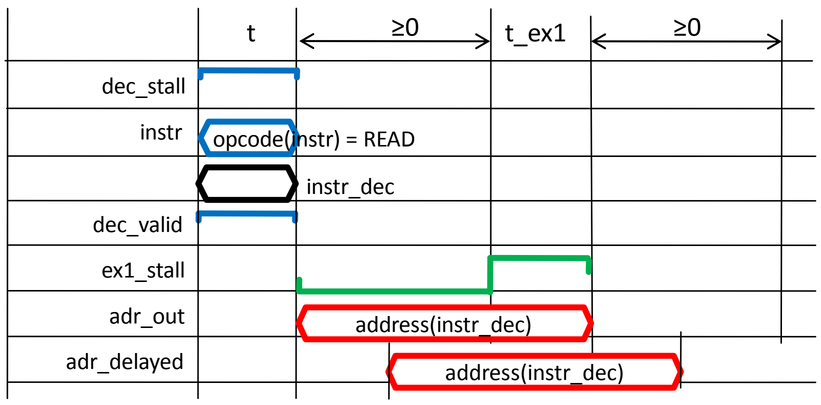


Figure 12: Delayed reaction on a synchronization event

4.3.12 Discussion

ITL is a proprietary language with features that make it particularly suitable for complete verification. Particularly important features are the Boolean implication and the explicit time. The specialty of Boolean implication is that assume part and proof part may overlap in time. This is important because it is required to reason about the protocol compliant execution of a full transaction, that usually involves several assumptions and requirements about the interface signals that interleave in time.

But Boolean implication requires a language feature that allows easily addressing the same point in time in the assume and the proof part. The explicit time expressions of ITL realize this, even when the common point is variable and determined by a time variable that is derived from a synchronization event.

A third important feature of ITL is its graphical representation. It eases formalization, understanding, and review of properties, in particular about protocol dominated circuits.

These three features are the reasons, why ITL still exists besides the two standardized property languages, PSL [PSL 2004], and the part of SystemVerilog [SystemVerilog 2005] to describe assertions, called SVA. Normal use of PSL and SVA abolishes above features: Sequence implication is used instead of Boolean Implication. In sequence implication the antecedent (comparable to the assume part of ITL) and the sukzedent (proof part) do not overlap in time so that the interplay of assumptions about input signals and requirements about output signals cannot be formalized in one property, although this interplay is typical for the execution of full protocol transactions.

Instead of explicit time expressions, SVA and PSL provide an opportunity to identify sequences of conditions that must hold on a trace successively. These sequences are formed by regular expressions. This allows expressing protocol related behavior only if the bus is internal, i.e. all modules that are connected to the bus are part of the circuit under formal verification. This is often the case in simulation, but not in formal verification, because such subsystems are usually too large for the formal tools. Formal verification examines the protocol compliance of a module at its primary interfaces. Then the examination of full transactions requires checking of more cause-effect-relations that are too complex for the sequence implication of SVA or PSL. In this situation SVA and PSL split the full transaction into smaller parts which can be expressed with sequence implication. The splitting impacts the conciseness of the properties, but accommodates complexity limitations of the formal tools. All in all, with SVA and PSL two property languages have been standardized, that are not well suited for a transaction based examination, although transaction based examination is key for simulation based verification.

When using less prominent parts of the language, some features described above can also be used, at least in SVA. This is made possible through the library Tidal [Bormann 2007], but the support by other formal tools is still to be explored.

In addition to the above strong points, ITL (and Tidal alike) have also some advantages in terms of usability. Most striking is probably the clear division of cause, effect and temporal structure in the assume-, prove-, and for-time point-blocks of the property code. In SVA and PSL the timing structure is mixed with the description of cause and effect. Since the timing structure is usually determined by external events, the succedents become difficult. Overall, SVA and PSL appear richer in snares, such that users have to invest more effort in clarifying the question of whether the intended property is actually correctly formalized in SVA or PSL.

5 Completeness and its Examination

The completeness checker is important for the automation of complete verification. It is presented below and compared to other approaches for determining the quality of a verification.

5.1 Methods for Determining the Quality of a Simulation-Based Verification

Integral part of verification projects are predictions about the already secured level of quality of the circuit under test. These predictions are already made long before the planned termination [Carter / Hemmady 2007] of the project. This quality prediction is based on statements as to how intensely the verification has examined the individual parts of the circuit, and how great the danger is that the circuit still contains an error.

5.1.1 Coverage

The quality of simulation based verification is mostly determined by the input coverage of simulations, which is often simply referred to as coverage. The Input Coverage makes a statement about the extent to which input traces of the simulation have examined the circuit in every possible situation. The result is condensed to a figure that is called the coverage measure.

Methods for measuring input coverage were first developed for software verification [Hirsch 1967, Belzer 1990], and then introduced in the nineties to hardware verification. Several of these methods now belong to the basic features of a verification environment. There are different approaches for determining the input coverage [Stuart / Dempster 2000]. Common to all is that they define a set of coverage conditions and a coverage target that indicates for each coverage condition, how often it should be met. The coverage measure is then a (possibly weighted) average ratio between the coverage targets and how often the related coverage conditions were actually met.

The various methods for determining input coverage differ with respect to the definition of the coverage conditions and coverage goals. A large group of methods is based on the RTL code. These methods are called code coverage or structural coverage. The simplest method, the Line or Statement Coverage, defines for each line of code the coverage condition that this line is executed and the coverage target that every line is executed at least once.

For branch coverage the conditions of the control statements (e.g., if and case) are examined. For each value that the control condition can assume, there is a coverage condition that is satisfied when the control condition has taken that value. The coverage target is in turn that each coverage condition is met once. Derivative methods (condition coverage, expression coverage focused expression coverage) do not examine the control condition as a whole, but their subexpressions. In path coverage, paths through the control graph of the RTL are determined in advance and the coverage conditions require that such a path is executed.

Other coverage metrics depend on the signals of the circuit. For toggle coverage the coverage conditions describe changes in signals and coverage goal is that for each signal a change in its value is observed. Triggering coverage analyzes which signal changes trigger the execution of a process.

Structural coverage provides effective guidance to best allocate the limited capabilities of simulation. But all coverage metrics described above are satisfied long before the circuit was really examined under all situations. Coverage is for example weak, when it comes to a function that is implemented by parallel processes that need to be examined with every possible time offset.

Code coverage is therefore complemented by functional coverage. The individual coverage conditions and coverage targets are explicitly specified by a verification engineer. This allows putting some focus on situations that the verification engineer regards critical and which are not addressed by the structural coverage. Obviously, the strength of the functional coverage depends on the definition of the coverage conditions. If a coverage condition has been left out, verification will not make an effort to trigger the respective situation.

5.1.2 Coverage Distributes the Attention of Simulation

A verification aims at 100% input coverage. The verification project leaders are used to the fact that over the time spent on the verification project, the coverage measure changes according to a saturation curve. Towards the end of a verification project, a coverage measure of 98% therefore might be dramatically low. Target are often values in the range of 99.9%. At 100% input coverage verification teams are usually happy, despite the shortcomings of the approach. For example, the search for errors due to parallel execution of processes is not structured by the above-mentioned coverage metrics.

Even from relatively small circuits onwards, simulation is in principle not able to consider each stimulus. Therefore, the coverage conditions partition the entire verification task into sort of a grid. Each element of this grid comprises a plurality of circuit executions that all satisfy the same coverage condition. To meet the coverage condition, it does not matter which of the circuit executions actually occurred.

For the circuit executions within one grid element it is implicitly assumed that the circuit would work similar and that therefore an error of the circuit could be detected in any of these executions. Which differences between the executions are considered irrelevant, is determined by the selection of the coverage conditions. A first division is made by the choice of method for code coverage, as adjusted by the individual production conditions of functional coverage.

The verification quality is higher the finer the grid is, i.e. the more detailed the coverage conditions are. Nevertheless, it is not useful to choose a particularly fine grid, because this would require satisfying a large number of coverage conditions, which in turn requires a large number of simulations, and thus a very high run time. Therefore the choice of coverage conditions and coverage goals must find a good balance between verification quality and the simulation effort.

When users of formal verification want to emphasize the advantages of their method against coverage driven simulations, they build a mental model. They postulate a coverage measure in which every input trace of a sufficient length[3] is mapped to a coverage condition that is satisfied if the circuit receives this input trace. When this coverage measure indicates 100% the verification examined all input traces, as is typical for formal verification. But for realistic circuits this coverage measure would never even reach even small values because there are simply so many coverage conditions that it is impossible to meet them all during a simulation.

[3] The length can for example be the diameter of the circuit according to [Biere et al. 1999].

But of course no one would seriously consider this mental model as an input coverage method for simulation, but just choose a coarser grid. This shows that coverage has also the purpose to appropriately distribute the limited capabilities of simulation over the whole circuit, and to concentrate it on particularly interesting spots. Coverage is not able to increase the capabilities of simulation, but it only allows a better use of its limited power.

5.1.3 Output Coverage

But the quality of verification is determined not only by input coverage, but also by the question of whether enough verdict generators have been incorporated into the simulation, be they assertions, monitors, checkers, or other mechanisms for detecting errors. Input Coverage influences only the probability that an error in the code is actually triggered. Whether the resultant misbehavior is really recognized, depends on the verdict generators. How accurately they examine the circuit behavior, is not captured by the Input Coverage. In extreme cases, a simulation could provide best values for the Input Coverage, although it does not contain a single verdict generator, such that the verification reports no error at all. Obviously, the verdict generators have significant contribution to the quality of verification.

In this thesis the term Output Coverage is used to describe the quality of all verdict generators. Input and Output Coverage provide orthogonal contributions to the quality of the verification. The combination of these actually independent values to a unified coverage measure statement is work in progress [Bailey 2007].

The output coverage is usually determined at the beginning of a verification project, i.e. during the setup of the verification plan, and specifically during the determination of the verification goals. During this phase, the verification engineers work informally: The specification is analyzed and divided into individual verification tasks. This step is critical, because if some verification goals are not identified, they will not be checked lateron, and related misbehavior is not recognized.

Because this process is informal, there is no fundamental tool support. Some tools help to document the relationship between specification and verification plan by cross-referencing. These tools identify parts of the specification which are not yet covered by an entry in the verification plan. But such tools cannot tell whether the ignored paragraphs really requested an entry in the verification plan, because they do not comprehend the text of the specification. Important quality assurance measures during the creation of the verification plan are therefore reviews by the engineers who are responsible for specification and circuit design.

In many design teams these reviews are the main measures to ensure decent output coverage. Automated tools for testing the output coverage of simulation-based verification [Certess undated, Grosse / Hampton 2005] are deployed by only few design teams. These tools can only be used when the testbench is largely completed including all their tests and checkers. The investigations are based on Mutation Analysis [Offutt / Untch 2000], that is, they inject errors into the circuit and investigate whether these errors are found by the simulation. If the injected errors are not found, deficits are often related to insufficient output coverage. If the verification project then still has time to improve the simulation, verification quality can be increased due to this feedback.

5.2 Output Coverage in Formal Verification

Formal verification does not have the problem of inadequate input coverage, except maybe for some semi-formal verification approaches, which will not be discussed further. A true proof acts as if the property or assertion in question is examined with all possible input traces. But this powerful examination method verifies often only quite simple statements. It seems that the assertions of ABFV fail to identify a lot of misbehavior because they only examine a very narrow range of functionality. This might lead to the impression that the advantage of optimal input coverage of formal verification is consumed by the disadvantage of inadequate output coverage. To oppose this, several approaches deal with the determination of the output coverage of formal verification. Some of these approaches are discussed below.

5.2.1 Fault Injection Method

The approach in [Hoskote et al. 1999, Hojati 2003] uses Mutation Analysis similar to the respective simulation based approach described above. Errors are injected into the circuit and it is checked whether then at least one of the properties is refuted by a property checker. If each injected error leads to a refutation, the quality of the property set is regarded sufficiently high. Otherwise, the ratio of injected errors is determined, that are detected by formal proof of the property set.

This method is heuristic. There is no guarantee that other errors than the injected ones are identified. The method requires both the properties and the circuit description.

5.2.2 Completeness of a Specification in Relation to a Given Circuit

The approach in [Katz et al. 1999] is based on a relation $\leq$ between so-called Kripke structures which is called simulation preorder. In the context of this thesis the Kripke structures are the same as automata. For two circuits and their automata M and $\bar{M}$ we have $M \leq \bar{M}$ iff $\bar{M}$ has all input signals of M, and if $\bar{M}$ has any additional inputs, then their traces can always be chosen such that $\bar{M}$ is sequentially equivalent to M. If $M \leq \bar{M}$ and $\bar{M} \leq M$ holds, M and $\bar{M}$ are sequentially equivalent, i.e. they have the same input and output behavior .

From a temporal logic formula φ considered in [Katz et al. 1999], an automaton $T(\varphi)$ can be generated which is called Tableau of φ, and which may be indeterministic. The Tableau $T(\varphi)$ satisfies the property φ, and has such a rich behavior that each circuit M that satisfies φ can be embedded in the Tableau in the sense of $M \leq T(\varphi)$.

If ψ is the specification of a circuit M then ψ is usually the conjunction of partial specifications. In the context of this thesis the partial specifications are be operation properties or assertions. Usual property checking examines if M satisfies the specification ψ. If this is the case, there is $M \leq T(\psi)$ by definition. To determine the output coverage of ψ, it is attempted to show

$$T(\psi) \leq M$$

For this attempt the states and transitions of $T(\psi)$ and M are matched one with the other. If M contains states or transitions without a mapping partner in $T(\psi)$, then ψ does not describe some part of the behavior of M, and hence ψ is not a complete specification.

ψ is called a complete specification of M in the sense of [Katz et al. 1999], iff $T(\psi) \leq M$ can indeed be proven. In this case, $T(\psi)$ and M are sequentiall equivalent. Obviously a second circuit $\bar{M}$ is sequentially equivalent to M if ψ is also a complete specification of $\bar{M}$. [Katz et al. 1999] does not discuss, if the proof of $T(\psi) \leq \bar{M}$ is still needed. But without related considerations it can only be stated that ψ is a complete specification for a given circuit

M. [Katz et al. 1999] makes no considerations whether ψ is complete independently of an implementation.

In comparison, the completeness concept of this thesis is independent of a concrete circuit, and only a characteristic of a set of properties. Besides that there are also application-oriented objections against [Katz et al. 1999]: To diagnose specification gaps, the tableau machines are to be compared sequentially with $T(\psi)$, hence counter examples can only be understood if the user understands Tableaus and knows how they are related to the original property ψ. Even if the BDD-based equivalence comparison algorithm of [Katz et al. 1999] would be replaced by more powerful algorithms of today, the tableau generation still appears to be an expensive step that limits the complexity of the circuits that can be examined with this approach.

5.2.3 Completeness Criterion

In this thesis output coverage is determined by checking completeness of the set of properties. The completeness criterion was introduced informally in section 3.4.2 and 3.4.3: A set of operation properties is called complete iff for every input trace that satisfies its determination assumptions, an output trace can be found that satisfies all properties and the related determination commitments. This completeness check must restrict itself to the examination of the properties with the signal names becoming free variables, and it must not use the concrete circuit.

As discussed in section 3.4.2 there is no loss of generality if the determination assumption

```
    if g then determined(e) end if;
```

is considered, where g is a condition and e an ITL expression that extracts at every time point τ the information contained in the input traces. Condition g and extraction function e will depend on input signals, but may also depend on output and internal signals. If two triples (I,U,O) und $(\bar{I},\bar{U},\bar{O})$ of traces are given, the determination assumption is calculated by

$$DA(I,U,O,\bar{I},\bar{U},\bar{O}) = \left(\bigwedge_{t\geq 0}\left(\sim g(t,I,U,O) \wedge \sim g(t,\bar{I},\bar{U},\bar{O})\right) \vee e(t,I,U,O) = e(t,\bar{I},\bar{U},\bar{O})\right)$$

where ~ denotes negation. For brevity

$$da(t,I,U,O,\bar{I},\bar{U},\bar{O}) = \left(\left(\sim g(t,I,U,O) \wedge \sim g(t,\bar{I},\bar{U},\bar{O})\right) \vee e(t,I,U,O) = e(t,\bar{I},\bar{U},\bar{O})\right)$$

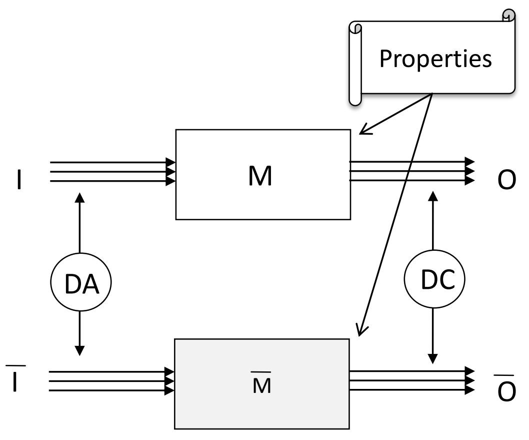


Figure 13: Completeness checking

shall be introduced as local determination assumption.

The determination commitment $DC(I,U,O,\bar{I},\bar{U},\bar{O})$ and its local counterpart $dc(t,I,U,O,\bar{I},\bar{U},\bar{O})$ shall be defined analogously.

The completeness checker examines the situation that any two circuits M and $\bar{M}$ are given, which both meet all operation properties. These two circuits shall receive input traces I und $\bar{I}$ and produce output traces O and $\bar{O}$ as well as traces U und $\bar{U}$ of internal signals. The traces shall satisfy the dependencies $D(I,U,O)$, $D(\bar{I},\bar{U},\bar{O})$ and the determination assumption $DA(I,U,O,\bar{I},\bar{U},\bar{O})$. The completeness checker examines whether the output traces meet the the determination commitments $DC(I,U,O,\bar{I},\bar{U},\bar{O})$ in this situation. If this holds for all possible input stimuli I und $\bar{I}$ the property set is complete. M und $\bar{M}$ then have the same input / output behavior at least modulo the determination assumptions and commitments. The situation is shown in Figure 13.

Thus, ultimately the following condition is derived for the completenss of a set the $\mathcal{P}$ of properties:

5-1

$$\bigwedge_{\substack{I,\bar{I}\in\mathcal{W}^{\infty}(\mathbb{i}),\\ U,\bar{U}\in\mathcal{W}^{\infty}(\mathbb{u}),\\ O,\bar{O}\in\mathcal{W}^{\infty}(\mathbb{o})}} DA \wedge \left(D \wedge \bigwedge_{P\in\mathcal{P}} \bigwedge_{t\geq 0} P\right) \wedge \left(\bar{D} \wedge \bigwedge_{P\in\mathcal{P}} \bigwedge_{t\geq 0} \bar{P}\right) \Rightarrow DC$$

Here the following denotational conventions are used: $DA = DA(I,U,O,\bar{I},\bar{U},\bar{O})$, $DC = DC(I,U,O,\bar{I},\bar{U},\bar{O})$, $D = D(I,U,O)$, $\bar{D} = D(\bar{I},\bar{U},\bar{O})$, $P = P(t,I,U,O)$ and $\bar{P} = P(t,\bar{I},\bar{U},\bar{O})$.

5.2.4 Alternative Formalizations of the Completeness Criterion

If determination assumptions simply require the equality of I and $\bar{I}$, and the determination commitmentds demand $O = \bar{O}$ the completeness becomes

5-2

$$\bigwedge_{\substack{I,\in\mathcal{W}^{\infty}(\mathbb{i}),\\ U,\bar{U}\in\mathcal{W}^{\infty}(\mathbb{u}),\\ O,\bar{O}\in\mathcal{W}^{\infty}(\mathbb{o})}} \left(D(I,U,O) \wedge \bigwedge_{\substack{P\in\mathcal{P},\\ t\geq 0}} P(t,I,U,O)\right) \wedge \left(D(I,\bar{U},\bar{O}) \wedge \bigwedge_{\substack{P\in\mathcal{P},\\ t\geq 0}} P(t,I,\bar{U},\bar{O})\right) \Rightarrow O = \bar{O}$$

This corresponds to the formalization in [Claessen 2006], which has been developed independently. The completeness criterion from section 5.2.3 is somewhat more general, because of its determination assumptions and determination commitments. This work goes beyond [Claessen 2006] because it provides a more efficient divide-and- conquer implementation of the completeness checker that is specialized for operation properties and produces counterexample that enable a more detailed analysis.

A further modification of the criterion 5-2 is to replace one of the conjuncts in the LHS by the model itself, which is anyway to be checked against the properties. Thus, it is easy to transform 5-2 into the equivalent formalization

5-3

$$\bigwedge_{\substack{I,\in\mathcal{W}^{\infty}(\mathbb{i}),\\ U,\bar{U}\in\mathcal{W}^{\infty}(\mathbb{u}),\\ O,\bar{O}\in\mathcal{W}^{\infty}(\mathbb{o})}} (D(I,U,O)\wedge M(I,U,O))\wedge\left(D(I,\bar{U},\bar{O})\wedge\bigwedge_{\substack{P\in\mathcal{P},\\ t\geq 0}} P(t,I,\bar{U},\bar{O})\right)\Rightarrow O=\bar{O}$$

An implementation of a completeness checker in accordance with this reformalization is attempted in [Große et al. 2008].

5.3 The Algorithm

The completeness checker receives a number of inputs which are described in section 5.3.1. Main entry is the set of operation properties, based on which the completeness checker performs a series of tests. There are several classes of tests that are characterized by a formula each. The formulas are instantiated according to the user inputs and then proved with a property checker. They are not be verified on the actual circuit, but on a model that contains two instances of type correct free variables for each signal of the circuit, according to the circuits representations M and $\bar{M}$ of the completeness criterion 5-1 in section 5.2.3. Section 5.3.8 provides the proof that the interaction of the tests actually implements the completeness criterion.

How the formulas are to be instantiated, will will be shown at the example that is known from section 3.2.1. The completeness checker input will be derived from section 3.4.2.

5.3.1 Inputs

The completeness checker needs the operation automaton with the operation properties. Therefore, an important input to the completeness checker is the set of properties $\mathcal{P}$ together with a successor relation $\gg$. The successor relation determines which properties may follow each other in the operation automaton. According to section 4.3 every property provides the time variables $T_i^P(t,I,U,O)$, the assume part $A^P(t,I,U,O)$ and the proof part $C^P(t,I,U,O)$. On top of that, according to section 3.2.3 every property P provides the reference time point $T_{ref}^P(t,I,U,O)$, which is assumed to be of the form $T+n$, where T is either t or one of the time variables $T_i^P(t,I,U,O)$.

The reset property R is a special element of $\mathcal{P}$ about the reset behavior according to section 4.1.1. The assume part of R is the condition $\varrho(I_{t-r}, I_{t-r+1}, I_{t-r+2}, \dots I_{t-1})$ that describes the reset sequence, such that $R(0,I,U,O)$ describes the reset behavior of the circuit.

The determination commitments are formalized in accordance with the $determined$ statements by local determination commitments $dc_k(\tau,I,U,O,\bar{I},\bar{U},\bar{O})$.

For each property and for each determination commitment the user has to provide the determination time interval in which the determination commitment should be satisfied. The determination time interval is specified by its first time point dl_k^P and its last time point dh_k^P. dl_k^P is of the form $t+n$ and dh_k^P is of the form $T+n$ where t is the start time point of the property and T is either t or a time variable $T_i^P(t,I,U,O)$. Correspondingly the determination time interval is a function $[dl_k^P(t,I,U,O), dh_k^P(t,I,U,O)]$ of t and the traces of the input, output and internal signals. The determination commitments of the reset property are requested for all time points ≥ 0, i.e. after the reset sequence.

To formalize intermediate results about the determination of the visible registers from section 3.2.2, the user can annotate local determination commitments $Dloc^P(t, I, U, O, \bar{I}, \bar{U}, \bar{O})$ to every property. They consist of one or multiple conditions in the form provided by section 3.4.2, which are required at single points in time relative to t or relative to a time variable T_i^P. The following definition shall simplify the notation:

5-4

$$Det^P(t) = Dloc^P(t, I, U, O, \bar{I}, \bar{U}, \bar{O}) \wedge \bigwedge_i \bigwedge_{\tau \in [dl_i^P(t,I,U,O), dh_i^P(t,I,U,O)]} dc_i(\tau, I, U, O, \bar{I}, \bar{U}, \bar{O})$$

Assertions and constraints among the Dependencies of the completeness proof form the condition $D(I, U, O)$.

Some of the user inputs are redundant. The successor relation $\gg$, the reference time points T_{ref}^P, the determination time intervals $[dl_i^P(t, I, U, O), dh_i^P(t, I, U, O)]$ and actually even the local determination commitments $Dloc^P(t, I, U, O, \bar{I}, \bar{U}, \bar{O})$ can in principle be automatically extracted from the properties. In fact, first research on completeness checkers [Busch 2005] dealt with such automatic extraction and developed related determination tests and case split tests. But these activities did not deal with visible registers and did not recognize successor tests. Thus they failed to prove completeness in the intuitive definition of section 3.4.3. Furthermore, the algorithms were so complex that only fairly small circuits could be examined. From today's perspective, the provision of the redundant user information is not a big effort, since the information captures the intentions of the verification engineer. In turn, the user input has the advantage that it allows reduction of the proof complexity, and that it allows more telling diagnoses if the completeness checker detects a verification gap.

5.3.2 Example

The algorithm shall be illustrated at the example from section 3.2.1 with the transaction automaton according to Figure 6. The property read_new_row of section 3.2.5 is written as

```
property read_new_row is
dependencies: no_reset, processor_protocol;

assume:
at    t:                  state = row_act;
at    t:                  request = '1';
at    t:                  rw = '1';
at    t:                  address /= last_row;

prove:
at    t+9:                state = row_act;
at    t+9:                last_row = prev(row(address));
during [t+1, t+7]:        ready = '0';
at    t+8:                ready = '1';
at    t+9:                ready = '0';
at    t+8:                rdata = prev(sd_rdata);
do_read(t, sd_ctrl, sd_addr, address);

end property;
```

after the introduction of the macro do_read with the definition

```
macro do_read(tt: timepoint; sd_ctrl: bit;
      sd_addr, address: bit_vector): temporal :=
at    t+1:               sd_ctrl = prech;
at    t+2:               sd_ctrl = nop;
at    t+3:               sd_ctrl = activate;
at    t+3:               sd_addr = row(address);
at    t+4:               sd_ctrl = nop;
at    t+5:               sd_ctrl = read;
at    t+5:               sd_addr = col(address);
at    t+6:               sd_ctrl = stop;
during [t+7, t+9]:       sd_ctrl = nop;
end macro;
```

The reference time point $TRef\_R$ of this property is $t + 9$. A local determination commitment

```
at t+9: determined(last_row);
```

comes with the property.

A further property shall be about write requests:

```
property write_old_row is
dependencies: no_reset, processor_protocol;

assume:
at t:       state = row_act;
at t:       request = '1';
at t:       rw = '0';
at t:       address = last_row;

prove:
at t+2:     state = row_act;
at t+2:     last_row = prev(last_row, 2);
at t+1:     ready = '1';
at t+2:     ready = '0';
at t+1:     sd_ctrl = write;
at t+1:     sd_addr = col(address);
at t+1:     sd_wdata = wdata;
at t+2:     sd_ctrl = stop;

end property;
```

The register last_row shall not be modified during this operation. Therefore it needs to be proven that its value at the end of the operation is the same as at the start. This is done with the second line of the prove part.

The property has the reference time point $TRef\_W = t + 2$ and the local determination commitment

```
at t+2: determined(last_row);
```

The properties can be proven under the dependencies

```
constraint no_reset :=
      reset = '0';
```

```
end constraint;

constraint processor_protocol :=
      if request = '1' and ready = '0' then
            next(request) = '1' and
            next(address) = address and
            next(rw) = rw and
            if rw = '0' then next(wdata) = wdata;
      end if;
end constraint;
```

The two operations may follow each other in arbitrary order. The successor relation is therefore $read\_new\_row \gg write\_old\_row$, $write\_old\_row \gg read\_new\_row$, $read\_new\_row \gg read\_new\_row$ and $write\_old\_row \gg write\_old\_row$.

Section 3.4.2 already stated the determination commitments

```
determined(sd_ctrl);
determined(ready);
if rw = '1' and ready = '1' then determined (rdata) end if;
if sd_ctrl = write then determined (sd_wdata) end if;
if sd_ctrl = read or sd_ctrl = write or sd_ctrl = activate
            then determined(sd_addr) end if;
```

and the determination assumptions

```
determined (request);
if request = '1' then determined(rw) end if;
if request = '1' then determined(address) end if;
if request = '1' and rw = '0' then determined(wdata) end if;
if prev(sd_ctrl, 2) = read then determined (sd_rdata) end if;
```

The determination time intervals for the read and write property are $[t+1, TRef\_R] = [t+1, t+9]$ and $[t+1, TRef\_W] = [t+1, t+2]$, respectively.

The completeness check needs only the dependency $processor\_protocol$.

5.3.3 Chains of Operation Properties

The basic idea of the completeness checker is to build chains of operation properties

5-5

$$\prod_{j\geq 0} P_j(t_j, I, U, O)$$

where $t_0 = 0$, $t_{j+1} = T_{ref}^{P_j}$, and then to prove that all behavior of the circuit is uniquely described by at least one chain, up to the determination commitments. P_0 is the reset property R and describes the circuit behavior after reset. The chains terminate as soon as a t_{j+1} becomes infinite, or the chains themselves are infinite.

A first sub-goal of the completeness checker is to prove that for each input stimulus that is permitted by the assertions and constraints among the dependencies of the completeness check, a property chain can be found such that the input trace satisfies all the assume parts. To this end the completeness checker generates and executes the case split tests that will be described more closely in section 5.3.4.

It shall then be verified that this chain determines the output trace O. To that end each property P_j is required to fulfill each determination commitment in the respective determination time interval. This is checked by the determination test, which also ensures that the determination time intervals of successive properties either seamlessly adjoin one another, or that the determination commitments even hold in the gaps between them. The determination test is presented in section 5.3.6.

In addition, it must be ensured that any predecessor property P_{j-1} proves enough about the internal signals to ensure that the applicability of the successor property is determined, i.e. uniquely ensured. This is done by the successor test in section 5.3.5.

5.3.4 Case Split test

The case split ensures the existence of property chains for every input trace. Each single tests examines the situation that the $P(t, I, U, O)$ was satisfied on a trace I for an arbitrarily chosen point in time $t.$ Then the case split test examines if the assumption part of at least one of the possible successors properties of P will be met by I. To simplify the notation of the general case split test, D will be used instead of $D(I, U, O)$ and $|P| = A^P \wedge C^P$:

5-6

$$\bigwedge_{I,U,O} \bigwedge_{t \geq 0} D \wedge |P| \Rightarrow \bigvee_{Q \in \mathcal{P}: P \gg Q} A^Q\left(T_{ref}^P(t, I, U, O), I, U, O\right)$$

The test is performed on a trivial model in which free variables are used for the signals of the sets $\mathbb{i}, \mathbb{u}, \mathbb{o}$.

The current implementation of the test is limited to properties with finite time variables as described in section 4.3.1.

If the case split test applies to all properties, including the reset property, induction shows, that there is a chain (P_i) of operation properties for each input trace I such that each assume part in Formula 5-5 is satisfied by the input trace.

The typical verification gap that is revealed by the case split test is a behavior of the inputs that is not covered by any successor property of P. The diagnosis of such an error will provide a snippet of the input trace that satisfies assume and prove part of P as well as all dependencies. It will list the appropriately time shifted assume parts of all successor properties and demonstrate that none of them is satisfied by the snippet of the input trace. Usually the diagnosis either points to a missing property that describes the expected circuit behavior or to a missing dependency that rules out the input trace.

The completeness checker generates the corresponding proof tasks from the text of the original properties. For the example of section 5.3.2 this gives

```
property case_split;
dependencies: processor_protocol;
for timepoints: TRef_W = t + 2;
```

```
assume:
-- assume part of write_old_row
at t:       state = row_act;
at t:       request = '1';
at t:       rw = '0';
at t:       address = last_row;
-- prove part of write_old_row
at t+2:     state = row_act;
at t+2:     last_row = prev(last_row, 2);
at t+1:     ready = '1';
at t+2:     ready = '0';
at t+1:     sd_ctrl = write;
at t+1:     sd_addr = col(address);
at t+1:     sd_wdata = wdata;
at t+2:     sd_ctrl = stop;

prove:
either
      -- assume part of write_old_row, time shifted by TRef_W
      at TRef_W:        state = row_act;
      at TRef_W:        request = '1';
      at TRef_W:        rw = '0';
      at TRef_W:        address = last_row;
or
      -- assume part read_new_row, time shifted by TRef_W
      at TRef_W:        state = row_act;
      at TRef_W:        request = '1';
      at TRef_W:        rw = '1';
      at TRef_W:        address /= last_row;
or
      … (time shifted assume parts of other successor properties)
end either;
end property;
```

5.3.5 Successor Test

The successor test determines whether the predecessor property P determines the applicability of the successor property Q. This means that the input trace and the visible registers of P uniquely decide about the applicability of Q. The extent to which P determines its visible registers is given by the local determination commitment $Dloc^P$. The applicability of Q depends on the question whether its suitably time shifted assume part $A^Q(T^P_{ref}(t,I,U,O),I,U,O)$ holds. Since this test is about determination, it is executed on two sets of traces (I,U,O) and $(\bar{I},\bar{U},\bar{O})$ which are assumed to be similar modulo appropriate determination conditions. It is assumed that assume and proof part of P and the appropriately shifted assume part of Q hold on (I,U,O), and that assume and proof part of P hold on $(\bar{I},\bar{U},\bar{O})$. The successor test then aims at a proof that the assume part of Q also holds on $(\bar{I},\bar{U},\bar{O})$.

To simplify notation, the following abbreviations will be used: Determination assumption $DA = DA(I,U,O,\bar{I},\bar{U},\bar{O})$, dependency $D = D(I,U,O)$ and $\bar{D} = D(\bar{I},\bar{U},\bar{O})$, assume part $A^Q = A^Q\big(T^P_{ref}(t,I,U,O),I,U,O\big)$ and $\bar{A}^Q = A^Q\big(T^P_{ref}(t,I,U,O),\bar{I},\bar{U},\bar{O}\big)$, time variables $T^Q_i = T^Q_i\big(T^P_{ref}(t,I,U,O),I,U,O\big)$ and $\bar{T}^Q_i = T^Q_i\big(T^P_{ref}(t,\bar{I},\bar{U},\bar{O}),\bar{I},\bar{U},\bar{O}\big)$, further $|P| = A^P(t,I,U,O) \wedge C^P(t,I,U,O)$ and $|\bar{P}| = A^P(t,\bar{I},\bar{U},\bar{O}) \wedge C^P(t,\bar{I},\bar{U},\bar{O})$. The successor test then takes the form

5-7

$$\bigwedge_{I,\bar{I},U,\bar{U},O,\bar{O}} \bigwedge_{t\geq 0} (DA \wedge D \wedge \bar{D} \wedge |P| \wedge |\bar{P}| \wedge Det^P(t) \wedge A^Q) \Rightarrow \left(\bar{A}^Q \wedge \bigwedge_i T_i^Q = \bar{T}_i^Q \right)$$

The test is again performed on a trivial model with free variables for the signals of the sets $\mathbb{i}, \mathbb{u}, \mathbb{o}, \bar{\mathbb{i}}, \bar{\mathbb{u}}$ and $\bar{\mathbb{o}}$.

A situation that violates the successor test is described in Figure 14: The predecessor operation P has a conceptual end state $state = idle$, which is both partly covered by the start states of the successor properties Q_1 and Q_2. The end state of P is only partly covered, because the assume parts of Q_1 and Q_2 introduce a counter cnt to determine whose turn it is. This value should be determined by P. If it is not, two different circuits could be implemented that differently implement cnt during the operation of P, such that the two circuits produce fundamentally different output traces for the same input trace, which contradicts completeness. Only one implementation of cnt can be the desired one. It is a verification gap not to check this

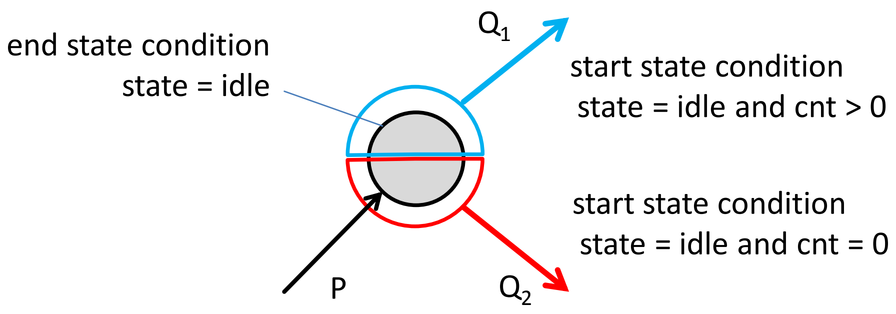


Figure 14: Counter example to the successor test

implementation. The successor test ensures that the verification engineer decides for one implementation and verifies it by the operation properties.

The counter example is related to one pair of predecessor and successor property P and Q. The counter example shows two sets of traces (I, U, O) and $(\bar{I}, \bar{U}, \bar{O})$ such that the traces are equal modulo determination assumptions for all times, modulo the determination commitments during the determination time intervals of P and modulo the local determination commitments of P. The counter example demonstrates that both sets of traces satisfy the assume and prove part of P, the dependencies, and the local determination condition $Dloc^P$. Furtheron, the counter example demonstrates that (I, U, O) satisfies the assume part of Q, but $(\bar{I}, \bar{U}, \bar{O})$ does not.

The analysis of the counter example may uncover a missing local determination condition, e.g. about cnt in the example above. Once the determination condition is added, the determination test of section 5.3.6 will insist on a proper proof about the value of the visible register. The counter example may also highlight a mismatch between the end state condition of P and the start state condition of Q.

The successor test for the predecessor property $write\_old\_row$ and the successor property $read\_new\_row$ from the example in section 5.3.2 is generated as follows, where the objects of $\bar{M}$ and indicated by the _c ssuffix.

```
property successor_test;
dependencies: processor_protocol, processor_protocol_c;
for timetpoints: TRef_W = t+2;

assume:
-- determination assumptions
during [t, TRef_W+9]:   request = request_c;
during [t, TRef_W+9]:   (request = 0 and request_c = 0) or rw = rw_c;
during [t, TRef_W+9]:   (request = 0 and request_c = 0)
                        or address = address_c;
during [t, TRef_W+9]:   (
                              (request = 0 or rw = 1) and
                              (request_c = 0 and rw_c = 1)
                        ) or wdata = wdata_c;
during [t, TRef_W+9]:   (
                              prev(sd_ctrl, 2) /= read and
                              prev(sd_ctrl_c, 2) /= read
                        ) or sd_rdata = sd_rdata_c;

-- determination commitments of write_old_row
during [t+1, t+2]:      sd_ctrl = sd_ctrl_c;
during [t+1, t+2]:      ready = ready_c;
during [t+1, t+2]:      (
                              (rw = 0 or ready = 0) and
                              (rw_c = 0 and ready_c = 0)
                        ) or rdata = rdata_c;
during [t+1, t+2]:      (sd_ctrl /= write and sd_ctrl_c /= write) or
                        sd_wdata = sd_wdata_c;
during [t+1, t+2]:      (
                              sd_ctrl /= read and
                              sd_ctrl /= write and
                              sd_ctrl /= activate and
                              sd_ctrl_c /= read and
                              sd_ctrl_c /= write and
                              sd_ctrl_c /= activate
                        ) or sd_addr = sd_addr_c;

-- lokale determination commitments of write_old_row
at t+2:     last_row = last_row_c;

-- assume part of write_old_row for first set of traces
at t:       state = row_act;
at t:       request = '1';
at t:       rw = '0';
at t:       address = last_row;
-- prove part of write_old_row for first set of traces
at t+2:     state = row_act;
at t+2:     last_row = prev(last_row, 2);
at t+1:     ready = '1';
at t+2:     ready = '0';
at t+1:     sd_ctrl = write;
at t+1:     sd_addr = col(address);
at t+1:     sd_wdata = wdata;
at t+2:     sd_ctrl = stop;

-- assume part of write_old_row for second set of traces
at t:       state_c = row_act;
at t:       request_c = '1';
at t:       rw_c = '0';
at t:       address_c = last_row_c;
-- prove part of write_old_row for second set of traces
at t+2:     state_c = row_act;
at t+2:     last_row_c = prev(last_row_c, 2);
at t+1:     ready_c = '1';
at t+2:     ready_c = '0';
at t+1:     sd_ctrl_c = write;
at t+1:     sd_addr_c = col(address);
at t+1:     sd_wdata_c = wdata;
at t+2:     sd_ctrl_c = stop;

-- time shifted assume part of read_new_row for 1st set of traces
at TRef_W:        state = row_act;
at TRef_W:        request = '1';
at TRef_W:        rw = '1';
at TRef_W:        address /= last_row;

prove:
-- time shifted assume part of read_new_row for 2nd set of traces
at TRef_W:        state_c = row_act;
at TRef_W:        request_c = '1';
at TRef_W:        rw_c = '1';
at TRef_W:        address_c /= last_row_c;
end property;
```

5.3.6 Determination Test

The determination test considers a situation similar to that of the successor test from section 5.3.5, but with addition that it successfully passed the successor test. The determination test proves that Q determines all output and visible registers at all required points in time. With the denotational simplifications from the previous sections and the abbreviations $T_{ref}^{P} = T_{ref}^{P}(t,I,U,O)$, $|Q| = A^{Q}\left(T_{ref}^{P},I,U,O\right) \wedge C^{Q}\left(T_{ref}^{P},I,U,O\right)$, $|\bar{Q}| = A^{Q}\left(T_{ref}^{P},\bar{I},\bar{U},\bar{O}\right) \wedge C^{Q}\left(T_{ref}^{P},\bar{I},\bar{U},\bar{O}\right)$ $dh_{i}^{P} = dh_{i}^{P}(t,I,U,O)$, $dl_{i}^{Q} = dl_{i}^{Q}\left(T_{ref}^{P},I,U,O\right)$, $T_{i}^{Q} = T_{i}^{Q}\left(T_{ref}^{P},I,U,O\right)$ und $\bar{T}_{i}^{Q} = T_{i}^{Q}\left(T_{ref}^{P},\bar{I},\bar{U},\bar{O}\right)$ the determination test has the following shape:

5-8

$$\bigwedge_{I,U,O,\bar{I},\bar{U},\bar{O}} \bigwedge_{t\geq 0} \left(DA \wedge D \wedge \bar{D} \wedge |P| \wedge |\bar{P}| \wedge |Q| \wedge |\bar{Q}| \wedge Det^{P}(t)\right)$$
$$\Rightarrow \left(Det^{Q}\left(T_{ref}^{P}\right) \wedge \left(\bigwedge_{i} \bigwedge_{\tau\in\left[dh_{i}^{P}+1,dl_{i}^{Q}-1\right]} dc_{i}\left(\tau,I,U,O,\bar{I},\bar{U},\bar{O}\right) \right) \right)$$

The first part of the RHS of the implication ensures that the property Q meets its determination commitments. The second examines gaps between the determination time intervals of P and Q and requires the the determination commitments hold there as well.

The test is performed on the trivial model that was already used for the successor test of section 5.3.5. The dependencies D and $\bar{D}$ and can provide additional information about the behavior of signals which may help to demonstrate the determination. For example, this might fill in gaps in the description of an output signal. Such gaps may e.g. appear between operation properties about processor pipelines whose individual stages can be stalled (see e.g. [Bormann / Beyer / Skalberg 2006]).

The counter example for a determination test presents two set of traces (I, U, O) and $(\bar{I}, \bar{U}, \bar{O})$ that satisfy the dependencies, the assume and prove parts of predecessor property P and successor property Q, as well as the determination assumptions about the inputs, the determination commitments about the outputs during the determination time intervals associated with P and the local determination commitments of P. The counter example will then show how Q allows the traces to be so different that they violate the determination commitments.

The usual verification gap that is uncovered by a determination test are insufficient descriptions of output behavior or visible states, mismatches between the time points where P determines a visible state and Q uses it, or determination time intervals that are not seamlessly adjacent.

A determination test for the properties write_old_row and read_new_row from the example of section 5.3.2 is relatively long. Therefore some parts are removed, that are identical to the successor test of section 5.3.5.

```
property determination_test;
dependencies: processor_protocol, processor_protocol_c;
for timetpoints: TRef_W = t+2;

assume:
-- determination conditions
... (see section 5.3.5)
-- determination commitments of write_old_row
... (see section 5.3.5)
-- local determination commitments of write_old_row
at t+2:           last_row = last_row_c;

-- assume part of write_old_row for the 1st set of traces
... (see section 5.3.5)
-- prove part of write_old_row the 1st set of traces
... (see section 5.3.5)

-- assume part of write_old_row for the 2nd set of traces
... (see section 5.3.5)
-- prove part of write_old_row the 2nd set of traces
... (see section 5.3.5)

-- time shifted assume part of read_new_row for 1st set of traces
at TRef_W:        state = row_act;
at TRef_W:        request = '1';
at TRef_W:        rw = '1';
at TRef_W:        address /= last_row;
-- time shifted prove part of read_new_row for 1st set of traces
at TRef_W+9:      state = row_act;
at TRef_W +9:     last_row = prev(row(address));
during [TRef_W +1, TRef_W +7]:      ready = '0';
at TRef_W +8:     ready = '1';
at TRef_W +9:     ready = '0';
at TRef_W +8:     rdata = prev(sd_rdata);
do_read(TRef_W, sd_ctrl, sd_addr, address);

-- time shifted assume part of read_new_row for 2nd set of traces
```

```
at TRef_W:        state_c = row_act;
at TRef_W:        request_c = '1';
at TRef_W:        rw_c = '1';
at TRef_W:        address_c /= last_row_c;
-- time shifted prove part of read_new_row for 2nd set of traces
at TRef_W+9:      state_c = row_act;
at TRef_W +9:     last_row_c = prev(row(address_c));
during [TRef_W +1, TRef_W +7]:      ready_c = '0';
at TRef_W +8:     ready_c = '1';
at TRef_W +9:     ready_c = '0';
at TRef_W +8:     rdata_c = prev(sd_rdata_c);
do_read(TRef_W, sd_ctrl_c, sd_addr_c, address_c);

prove:
-- determination commitments
during [t+3, tRef_W+9]: sd_ctrl = sd_ctrl_c;
during [t+3, tRef_W+9]: ready = ready_c;
during [t+3, tRef_W+9]: (
                             (rw = 0 or ready = 0) and
                             (rw_c = 0 and ready_c = 0)
                        ) or rdata = rdata_c;
during [t+3, tRef_W+9]: (sd_ctrl /= write and sd_ctrl_c /= write) or
                        sd_wdata = sd_wdata_c;
during [t+3, tRef_W+9]: (
                             sd_ctrl /= read and
                             sd_ctrl /= write and
                             sd_ctrl /= activate and
                             sd_ctrl_c /= read and
                             sd_ctrl_c /= write and
                             sd_ctrl_c /= activate
                        ) or sd_addr = sd_addr_c;

-- local determination commitments
at TRef_W+9:            last_row = last_row_c;
end property;
```

5.3.7 Tests about the Reset Property

The property chains (cf. section 5.3.3) need to start with the reset property R. This property has no predecessor property. Therefore, the tests need to be slightly modified.

To demonstrate that the application of the reset property is determined, a variation of the successor test is used.

5-9

$$\bigwedge_{I,\bar{I},U,\bar{U},O,\bar{O}} \bigwedge_{t\geq 0} (DA \wedge D \wedge \bar{D} \wedge A^R) \Rightarrow \left(\bar{A}^R \wedge \bigwedge_i T_i^R = \bar{T}_i^R\right)$$

Besides the already introduced denotational simplifications, the formula uses the abbreviations $A^R = A^R(0, I, U, O)$, $\bar{A}^R = A^R(0, \bar{I}, \bar{U}, \bar{O})$, $T_i^R = T_i^R(0, I, U, O)$ and $\bar{T}_i^R = T_i^R(0, \bar{I}, \bar{U}, \bar{O})$.

Finally, it must be shown that the reset property determines its output. This is done with

5-10

$$\bigwedge_{I,U,O,\bar{I},\bar{U},\bar{O}} \bigwedge_{t\geq 0} (DA \wedge D \wedge \bar{D} \wedge |R| \wedge |\bar{R}|) \Rightarrow Det^R(t)$$

using the abbreviations $|R| = A^R(0,I,U,O) \wedge C^R(0,I,U,O)$ und $|\bar{R}| = A^R(0,\bar{I},\bar{U},\bar{O}) \wedge C^R(0,\bar{I},\bar{U},\bar{O})$.

Both tests are again performed on free variables for the signals of $\mathbb{i}, \mathbb{u}, \mathbb{o}, \bar{\mathbb{i}}, \bar{\mathbb{u}}$ and $\bar{\mathbb{o}}$.

5.3.8 Proof

It shall be demonstrated that the completeness condition 5-1 is met, if the case split, successor, determination, and reset tests hold on a set of operation properties $\mathcal{P}$ which are assumed to hold on two different automata M and $\bar{M}$.

Two input traces I und $\bar{I}$ shall be given. They both contain reset sequences $(I^{(-r)}, I^{(-r+1)}, I^{(-r+2)}, \dots, I^{(-1)})$ and $(\bar{I}^{(-r)}, \bar{I}^{(-r+1)}, \bar{I}^{(-r+2)}, \dots, \bar{I}^{(-1)})$ and extend afterwards such that the synchronous circuits M and $\bar{M}$ generate traces $U, O, \bar{U}$ und $\bar{O}$ of internal and output signals that satisfy $M(I,U,O)$ and $\bar{M}(\bar{I},\bar{U},\bar{O})$. For these traces the determination assumption $DA(I,O,U,\bar{I},\bar{O},\bar{U})$ shall be met. It is to be proven that the determination commitment $DC(I,O,U,\bar{I},\bar{O},\bar{U})$ is also met.

The proof uses induction. The induction hypothesis is that there is a finite chain $(P_0, P_1, P_2, \dots, P_n)$ of properties and a finite sequence of time points $(t_0, t_1, t_2, \dots, t_n)$ with $t_0 = 0$, $t_{j+1} = T_{ref}^{P_j}$ such that internal and output traces are found that satisfy $\prod_{j=0}^{n} P_j(t_j, I, U, O)$ and $\prod_{j=0}^{n} P_j(t_j, \bar{I}, \bar{U}, \bar{O})$, the determination condition

$$\bigvee_{j} \bigvee_{\tau \in \left[0, dh_j^{P_n}\right]} DC_j(\tau, I, U, O, \bar{I}, \bar{U}, \bar{O})$$

the local determination commitment $Dloc^{P_n}(t_n)$ and $T_{ref}^{P_n} = \bar{T}_{ref}^{P_n}$.

Induction base: For the $n = 0$ the reset property is to be considered applied for $t = 0$. Its assume part is satisfied for I and $\bar{I}$. Because of the first test 5-9 about the reset property, potential time variables of the reset properties are equal for both copies of the traces. Hence the reference time points T_{ref}^{R} und $\bar{T}_{ref}^{R}$ are equal which proves the respective part of the induction hypothesis. Since the assume part of the reset property is satisfied for both copies of the traces, M and $\bar{M}$ behave for $t = 0$ according to the reset property. Hence formula 5-10 is applicable and proves $Det^R(0)$. The rest of the induction hypothesis follows from formula 5-4 and the conditions about the right boundary of the determination time intervals of the reset property.

Step from n to $n+1$: Let the induction hypothesis be valid for n. This means in particular that $A^{P_n}(t_n, I, U, O)$ und $C^{P_n}(t_n, I, U, O)$ hold. The case split test 5-6 therefore ensures for $t = t_n$ the existence of a property P_{n+1} the acceptance part of wich is satisfied for $t = T_{ref}^{P_n}(t_n, I, U, O)$. This defines t_{n+1} according to the induction hypothesis for $n+1$. From the successor test 5-7 follows that $A^{P_{n+1}}(t_{n+1}, \bar{I}, \bar{U}, \bar{O})$ is also satisfied. Hence M und $\bar{M}$ behave for $t = t_{n+1}$ according to the property P_{n+1}. The successor test 5-7 additionally shows that potential time variables of P_{n+1} assume the same values on both copies of the traces, i.e., $T_{ref}^{P_{n+1}} = \bar{T}_{ref}^{P_{n+1}}$. Moreover, the determination test can be applied to P_n and P_{n+1} for $t = t_{n+1}$ and provides because of formula 5-4

$$Dloc^{P_{n+1}}(T_{ref}^{P_n}) \wedge \left(\bigwedge_j \bigwedge_{\tau \in \left[dh_j^{P_n}+1, dh_j^{P_{n+1}}\right]} dc_j(\tau, I, U, O, \bar{I}, \bar{U}, \bar{O}) \right)$$

Together with the induction hypothesis for n this proves the induction hypothesis for $n + 1$.

Thus the validity of the induction hypothesis is shown for all n, for which t_n remains finite. Either the t_n remain finite for all natural numbers n, or there is a number N, for which $T_{ref}^{P_N}$ becomes infinite. In the first case $dh_i^{P_n}$ increases beyond all limits, because it is calculated relative to t_n. And t_n grows beyond all limits because $t < T_{ref}^P$ is required for all properties $P \in \mathcal{P}$. In the second case is $dh_j^{P_N} = \infty$ for all j for which there are determination commitments DC_j. This finally proves

$$\bigvee_j \bigvee_{\tau \geq 0} dc_j(\tau, I, U, O, \bar{I}, \bar{U}, \bar{O})$$

which is the definition of $DC(I, U, O, \bar{I}, \bar{U}, \bar{O})$ according to section 5.3.1. This completes the proof.

5.4 Discussion

The prominent role of the completeness checker to prove the completeness and hence the termination of a complete verification was presented in section 3.4.3 in detail and will not be repeated here. Instead, some peculiarities of the implementation of the completeness checker shall be pointed out.

5.4.1 Case Split Tests and Reactive Constraints

Figure 15 shows a slightly modified version of the property from Figure 7, that contains additional assumptions about $request$ and rw for time $t + 1$ to $T - 1$, that an engineer could easily include in the property. These additional assumptions are actually redundant, since they are consequence of other parts of the property and of two protocol constraints

```
if ready = '0' and request = '1' then next(request = '1') end if;
if ready = '0' and request = '1' then rw = next(rw) end if;
```

and of the fact that all operations deactivate the ready signal in $state = row\_act$.

If all other properties were changed correspondingly, the case split test would provide a counterexample that asks in principle for properties about the case that the signals $request$ und rw from the processor do not behave according to the protocol. This question is at first astonishing, because the respective protocol constraints are among the dependencies of the completeness check. But the astonishing behavior of the completeness checker is a result of formula 5-6 of the case split tests, because there the proof parts C^Q of the successor properties are not taken into account. Hence the completeness checker does not know about $ready = 0$ and hence it cannot use the reactive constraints.

In the verification practice the verification engineer should use reactive protocol constraints during the proof of the properties and remove those parts of the assumption from the property that are redundant to the protocol constraints. Therefore, the property of Figure 7 is better suited for the case split test.

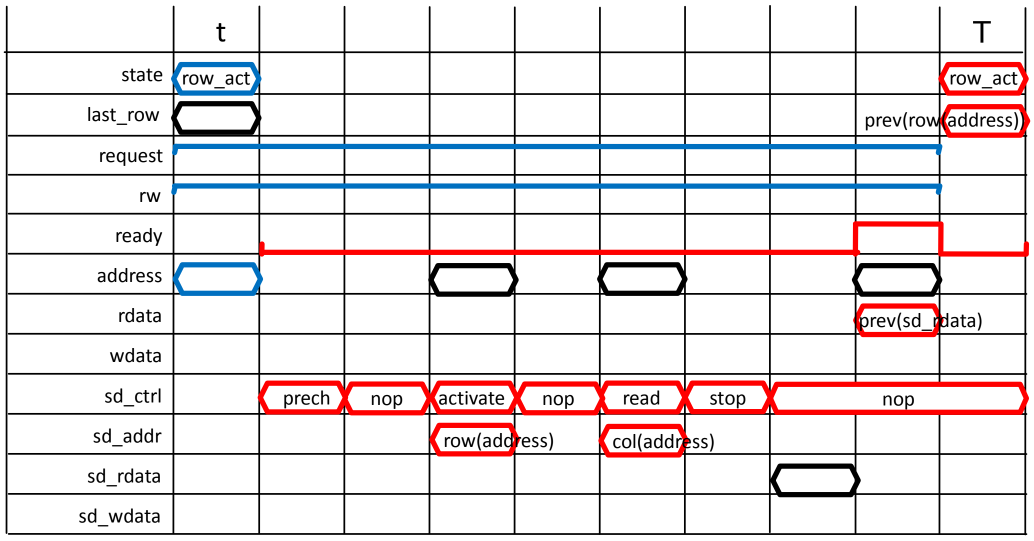


Figure 15: Modified SDRAM IF property (additional assumptions about request and rw)

5.4.2 Completeness Proof on Operation Automata

The completeness proof for operation automata is almost the same as for operation properties, although the operations of the operation automaton do not contain the important states. But their role is taken up by the successor relation $\gg$ and the reference time point T^{P}_{ref}. By this, the completeness checker can be applied to check the plausibility of an operation automaton, for which there is no implementation of the conceptual states yet. This plausibility check does not elaborate on the question of implementability of the operation automaton, but it is useful as an initial step of a development methodology where the properties are developed first and then the RTL.

5.4.3 Important states

The completeness checker identifies the conceptual states independently and automatically. The tests are defined such that the user need not highlight has with a specific syntax. In fact, in practice, sometimes quite unusual conceptual conditions have occurred. The conceptual states can include input or output signals, they can depend on output behavior even over a longer period, and they can even contain time variables. All this is not immediately compatible with the concept of a state and would constitute an additional difficulty in using complete verification. With the chosen approach the user need not care which part of the property is required to connect with the successor property, which part is responsible to check the output behavior, and if some parts of the property serve both purposes.

In relationship to important states it was already mentioned that the completeness checker does not really need them. It only checks in the successor test whether the conceptual states of successive properties fit together among themselves and in relation to the reference time points and the successor relation. But this compatibility is needed to check user provided reachability information (cf. section 3.3.4).

6 Compositional Complete Verification

Compositional complete verification [Beyer / Bormann 2008] is described in this chapter with two facets. The assembly of complete verifications will be presented at the example of the development of a system on chip (SoC) from IP blocks in section 6.1, and results in an efficient method to verify the communication of IP blocks of a SoC in section 6.3.

Section 6.4 is about the decomposition of complete verification into suitable clusters that is usually necessary for the verification of larger modules, such as many IP blocks. Different from the assembly, the entire RTL code is here available at least in a first preliminary version and it is to be completely verified. The approach is to decompose the circuit into parts which are called clusters (cf. section 3.5.6), and then to completely verify the clusters under observation of some restrictions that ensure the composition of the complete verification of the clusters to a complete verification of the entire block.

Of course a development and verification flow starts from IP blocks that are completely verified with the methods from section 6.4. Once the IP blocks are assembled in SoCs the complete verification follows section 6.3. Thus, the discussion in this thesis reverses the natural order of the design and verification process. Anyhow, the theory of the decomposition of a complete verification task is derived from the theory of the assembly of complete verifications, and this justifies reversed presentation in this thesis. At first reading of section 6.3 alls IP blocks can be viewed as consisting of only one cluster that is completely verified with the methods from the previous chapters.

Compositional complete verification is based on so-called integration conditions. The question, if subcircuits of a SoC communicate properly with one another, can be examined exhaustively by checking the integration conditions with an IPC-prover. Integration conditions exist for each complete verification, regardless of whether it was carried out on a single cluster, or whether it already comprises multiple clusters. Therefore, this chapter also paves the way for a hierarchical application of complete verification of SoC of arbitrary size.

The theory of compositional complete verification is a suitable adaptation of the theories about models of digital systems and the associated Assume-Guarantee reasoning, which clarifies questions about the integration of components that were modeled according to these theories. The relevant mathematics is presented in the Appendix (section 8).

6.1 Compositional Complete Verification for IP-based SoC Design

6.1.1 Verification Tasks of IP-based SoC Design

The first step of a SoC design is the partitioning of the overall function into subtasks. These tasks are often taken over by IP blocks, i.e. pre-developed and pre-verified circuits. The SoC designer needs to select them, usually from a larger number of different IP blocks with similar functionality. Criteria for this selection are of course functional aspects, but also an assessment of the RTL quality, the usefulness of the documentation, royalties, chip area, power consumption, etc.

The functional aspects include the actual function, processing latencies, throughput and about the protocols at the interfaces. The question of the protocols is crucial to ensure the communication in the SoC design – interface partners should use the same protocol to avoid the cost and performance impact of bus bridges. Protocols have been standardized to reduce variability and potential mismatches between communication partners.

But standardization is inflexible. With standardized protocols it is much harder for the SoC designers to find a good trade-off between performance requirements of the communication and cost in terms of silicon area and power consumption. Today's SoC design practice therefore uses a compromise: Some features of the protocols are marked optional, sometimes explicitly, sometimes only as the consequence of common design practice.

These options must be taken into account during the selection of IP blocks. Otherwise, if one partner implements a feature that is never requested by the other partner, unnecessary power or silicon area is consumed. Alternatively, if one communication partner requests a feature that is not supported by the other partner, the communication might break. This error may only affect the two communication partners, which is bad enough. But it may also impact the whole bus system. Communication errors are often major functional failures of SoCs that can not easily be circumvented.

Such incompatibilities of IP blocks that pretend to support the same standard protocol must be ruled out during the selection. At the time of the selection the RTL code of the actual IP block is usually not available to the SoC designer, either because the SoC developer did not have time to deal with the code, or because business negotiations are not in a state are that allow the transfer of the highly sensitive RTL code. The SoC designers then rely on the documentation of the IP blocks and and compare it with respect to the interface behavior of those blocks that should communicate with one another. This alignment is prone to errors. The documentation may be incomplete, it does not necessarily represent the IP block correctly and it might be misunderstood by SoC designers.

Today, if such incompatibilities escape the attention of the SoC developer, the project is already lucky if they are discovered in the system simulation of the largely developed SoC. But even this is not certain, because an RTL based system simulation of the whole SoC runs so slowly that only few scenarios can be simulated. Thus, such communication errors are sometimes noticed only in the product test after the production of the first silicon sample, when the fix requires a respin.

But up to the point in time where the system simulation can be performed, much effort has been invested into the new SoC, and all of this effort has been carried out on the basis of the initial selection of the IP blocks. It is most likely that a sizeable amount of this effort went into the development of the communication infrastructure of the SoC. Frequently, the software development begins in parallel with the hardware development. Changes to the original selection of the IP blocks lead to additional effort to adapt the communication infrastructure and to adapt the already developed software. They should therefore be kept rare and small.

So it is very helpful for the SoC design to get reliable information about the compatibility of potential communication partners already during the selection of the IP blocks.

6.1.2 Formal Verification of the Communication of the IP Blocks of a SoC

The technique presented here allows using at an early stage reliable information about the compatibility of the interface behavior of IP blocks. Results about the compatibility of these blocks can be obtained in minutes. The related approach is a natural consequence of the question when the complete verification of separate subcircuits is a complete verification of the overall circuit.

Prerequisite is a kind of interface description of the complete verification of the IP block - similar to [Broy / Rumpe 2007]. This interface description consists of an integration assumption with constraints about behavior and determinedness of input signals, and from an integration commitment with assertions about the behavior and the determinedness of the output signals. Integration assumptions and integration commitments are collectively referred to as integration conditions. If the IP block is completely verified, the integration conditions are available. However, it is also worthwhile for an IP provider to set up the integration conditions even without a formal verification. This does not rule out discrepancies between the RTL code and the integration conditions, but allows for the compatibility test described in this chapter.

Based on these integration conditions some formal tests can efficiently detect if IP blocks are compatible with respect to their communication behavior. It could be imagined that these integration conditions are made freely available for each IP block. By this the proposed test can be performed without making RTL code to a yet undecided buyer of IP.

6.1.3 Protocol Descriptions by Integration Conditions

An example of an integration condition has been described in section 3.5.2. Basic syntax of an integration condition is

```
integration_condition of <module_name> is
assume:
      <determination assumptions and constraints>;
guarantee:
      <determination commitments and assertions>;
end integration_condition;
```

At least in the context of formal verification, integration conditions allow for the first time a description of the main raison d'être of protocols at all, namely to ensure that one module is ready to receive some data, when another module really sends it.

ABFV has no way to express this, and is therefore limited to behavioral aspects of the control and data signals. ABFV can express and check comfortably that a data signal is to remain unchanged in a certain situation. But whether this data signal then transmits a valid value, i.e. a value that should be evaluated by the receiver, can not be expressed by means of ABFV. Complete verification can reason about valid data values with the keyword "determined" of the integration conditions.

The limitations of ABFV become particularly striking at the rdata signal of the SDRAM interface. This signal is not subject to any behavior restriction that could be expressed in an assertion. The signal itself may behave arbitrarily, only that it sometimes transmits read data. This is is indicated by $ready\_o = '1'$. It But since there is no restriction in the behavior of rdata, ABFV descriptions of the processor interface will not mention rdata at all despite its prominent role in the protocol. In contrast, Integration conditions will contain the determination commitment

```
if ready_o = '1' then determined(rdata) end if;
```

which makes the completeness check require a proof goal about rdata whenever $ready\_o =' 1'$.

In simulation-based verification the transaction extractors from section 3.1.2 identify when a signal carries a valid value, because this becomes a parameter of the respective transaction. These extracted parameters are compared with the values calculated by a reference model. This is how simulation verifies that the modules indeed consistently forward data and addresses.

6.1.4 A Simple Compositional Completeness Proof

To provide a feeling for the intended reasoning of compositionsl complete verification, it is assumed that two blocks M_1 and M_2 are connected in series. M_1 shall be completely verified by the property set $\mathcal{P}_1$. The complete verification was executed under the integration assumption IA_1 and it proved the integration commitment IC_1. The same applies for M_2.

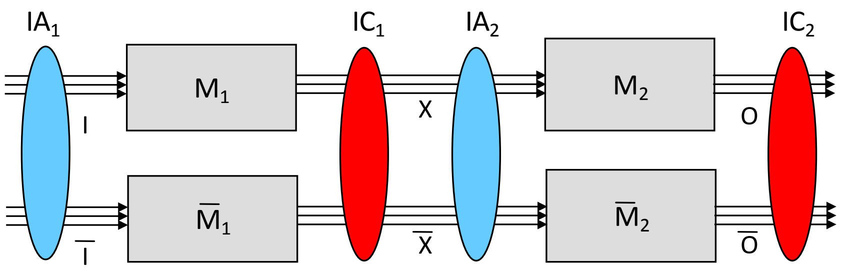


Figure 16: Cluster graph without cycles

To completely verify the overall circuit, it obviously must be ensured that IA_2 is satisfied whenever IC_1 holds. If this is the case, the circuit is completely verified by the property set $\mathcal{P}_1 \cup \mathcal{P}_2$. The Integration assumption of the full circuit is IA_1 and the integration commitment is IC_2.

To prove that the whole circuit is proven by $\mathcal{P}_1 \cup \mathcal{P}_2$ the criterion from section 5.2.3 must be shown. Hence, any two circuits M and $\overline{M}$ shall be compared, that both satisfy $\mathcal{P}_1 \cup \mathcal{P}_2$. Under the integration assumption IA_1 the property set $\mathcal{P}_1$ proves the integration commitment IC_1 and thus identifies the circuit parts M_1 and $\overline{M}_1$ as being completely verified by $\mathcal{P}_1$. M_1 and $\overline{M}_1$ show equal input/output behavior to the extent described by IA_1 and IC_1. Since IC_1 is satisfied, IA_2 holds also. $\mathcal{P}_2$ identifies circuit parts M_2 and $\overline{M}_2$ which satisfy under the integration assumption IA_2 the integration commitment IC_2. Hence IC_2 is satisfied. This shows that M and $\overline{M}$ from Figure 16 are completely described by $\mathcal{P}_1 \cup \mathcal{P}_2$ for the integration assumption IA_1 and the integration commitment IC_2.

6.1.5 Problem with cyclic dependency of IP blocks

However, the above-mentioned arguments can only be applied as long as there are no circular dependencies between the IP blocks. The scenario from Figure 17 shall demonstrate, that above reasoning leads to absurd results, if two circuits depend cyclically one on the other. The scenario is an anonymized, simplified and focused version of a customer problem [Beyer 2008].

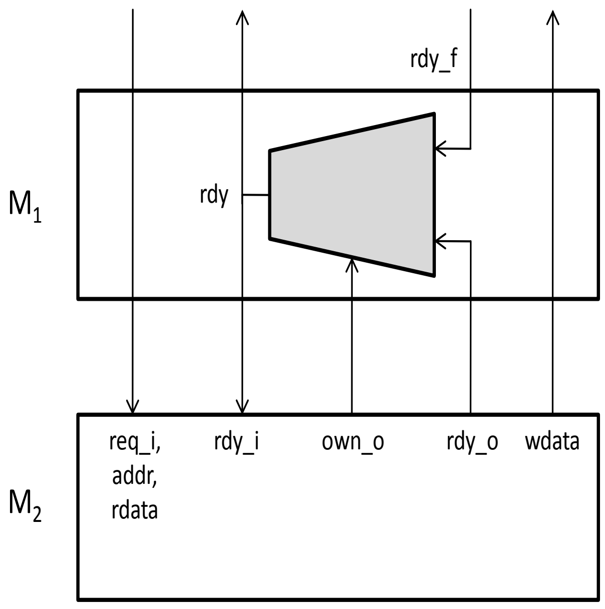


Figure 17: Problematic Completeness Check

The circuit under verification consisted of a bus system M_1 and a peripheral M_2. The only interesting part in M_1 is a multiplexor that is connected with M_2 as shown in Figure 17. The protocol requires a synchronization signal rdy that is distributed to all peripherals. The peripheral that is currently requested from the bus must control the synchronization signal by the multiplexor select own_o and the local synchronization output rdy_o. If $own\_o = 0$, the peripheral M_2 does not control the synchronization signal. This role is then taken up by another peripheral which is not part of this example. Its role is modeled here with a free primary input rdy_f of M_1.

For M_1 a complete verification was carried out with the following integration conditions (irrelevant parts were left out)

```
integration_condition of bus_system is

assume:
      determined(own_o);
      if own_o = '1' then determined(rdy_o) end if;
      if own_o = '0' then determined(rdy_f) end if;

guarantee:
      if own_o = '1' then rdy_i = rdy_o end if;
      determined(rdy_i);

end integration_condition;
```

The peripheral has been completely verified under the integration conditions

```
integration_condition of peripheral is

assume:
      if own_o = '1' then rdy_i = rdy_o end if;
      determined(rdy_i);
```

```
guarantee:
      determined(own_o);
      if own_o = '1' then determined(rdy_o) end if;

end integration_condition;
```

At first glimpse this looked as if the full system would be completely verified under the integration condition

```
integration_condition of system is

assume:
      if own_o = '0' then determined(rdy_f) end if;

guarantee:
      determined(own_o);
      if own_o = '1' then determined(rdy_o) end if;

end integration_condition;
```

But the completeness of the property set about the peripheral could be proven, although any proof goal about rdy_o was forgotten. Hence it was obvious that the properties did not verify rdy_o and misbehavior of rdy_o would not have been detected. This surprising behavior of the completeness checker is a result of the determination assumption about the signal rdy_i and the constraint

```
if own_o = '1' then rdy_i = rdy_o end if;
```

These appear in the determination test 5-8 where the constraint is part of the dependencies D and $\overline{D}$. Whenever a property expects $own\_o\ =\ '0'$ the determination commitment does not have any requests for the signal rdy_o. If a property expects $own\_o\ =\ '1'$ the constraint becomes applicable and ensures $rdy\_o\ =\ rdy\_i$ and $\overline{rdy\_o}\ =\ \overline{rdy\_\imath}$. The unconditional determination assumption about rdy_i means $rdy\_i = \overline{rdy\_\imath}$ and all this gives $rdy\_o = \overline{rdy\_o}$ which proves the determination commitment.

The error arose because the integration conditions of the overall circuit could only be derived using a circular argument about the integration conditions of the sub-modules. The circular reasoning was as follows: The completeness check about M_2 proved the determination commitment about rdy_o under the assumption that rdy_i is determined. The completeness check about M_1 proved the determination of rdy_i under the assumption of a conditional determination of rdy_o.

This chapter will describe the conditions under which complete property sets of cyclically interconnected IP blocks may be combined into one complete property set about the full circuit.

6.2 Plausibility Tests About Integration Assumptions

The absurd results from the previous section require measures to rule out cyclic reasoning. To this end, conditions were developed that include requirements about the structure of the integration assumptions. These requirements are a natural consequence of the wish that circuits should exist that behave as given by the integration assumption. Such circuits will be referred to as implementations of the integration assumption. Moreover, at least one implementation of the integration assumption must not even produce combinatorial loops when combined with the circuit under verification. The examples from section 3.5.3 show that the existence of such circuits is not trivial.

An integration assumption will be called implementable, iff there is at least one implementation of it. It will be called structurally compatible with the circuit, if at least one implementation of it avoids a combinatorial loop through the circuit and that implementation. Implementability and structural compatibility are fundamental prerequisites for integration assumptions to be realistic. The criteria are detailled out in sections 6.2.2 and 6.2.4 on the based of a formalization of integration conditions in section 6.2.1.

Section 6.1.5 demonstrates that it is not always easy to tell whether integration assumptions are indeed realistic images of possible circuit environments. It will be shown in the course of the following discussion that the integration assumption of the peripheral there is not implementable and hence violates one of the fundamental prerequisite of an integration assumption.

The prerequisites are not only important when complete verifications are to be merged. They are already very helpful for the verification of a single cluster. Here they may serve as a plausibility criterion that saves users from wasting effort on complete verifications with unsuitable constraints or determination assumptions. These plausibility criteria are much sharper than the vacuity tests that are often used in ABFV to warn users about inappropriate constraints or antecedents. A comparison between the two approaches is provided in section 6.2.5.

6.2.1 Formalization of Integration Conditions

Although the syntax of the integration conditions seems to refer only to one set of variables of the model, the formalization of the integration conditions must bear in mind that these conditions characterize complete verification. The completeness checking involved relates to the comparison of two arbitrary circuits that meet the relevant properties. Therefore, the integration assumptions need to restrict two traces I and $\bar{I}$ of input signals and the integration commitments should provide guarantees about two traces O und $\bar{O}$.

Having said this, the integration assumptions are defined in the notation of chapter 5 by

$$IA(I,U,O,\bar{I},\bar{U},\bar{O}) = \alpha(I,U,O) \wedge \alpha(\bar{I},\bar{U},\bar{O}) \wedge\ DA(I,U,O,\bar{I},\bar{U},\bar{O})$$

Here α should include all constraints that have been used in the complete verification of a module, i.e. the constraints of the dependencies D of the completeness check and the constraints under which the individual properties were verified against the RTL code.

Correspondingly, integration commitments are formalized by

$$IC(I,U,O,\bar{I},\bar{U},\bar{O}) = \zeta(I,U,O) \wedge \zeta(\bar{I},\bar{U},\bar{O}) \wedge\ DC(I,U,O,\bar{I},\bar{U},\bar{O})$$

formalized, with ζ being the conjunction of assertions about the output signals.

Integration assumptions may be reactive, but reactive formulae are particularly error prone and therefore benefit from the plausibility tests.

With the exception of the integration conditions all objects used in a completeness check occur in pairs. A further denotational simplification shall help here, where double letters refer to the pair. Hence, $II = (I,\bar{I})$, $OO = (O,\bar{O})$, $UU = (U,\bar{U})$, $MM(II,UU,OO) = M(I,U,O) \wedge \bar{M}(\bar{I},\bar{U},\bar{O})$, With a slight generalization, $\mathcal{P}$ will be replaced by

$$PP(II,UU,OO) = \bigwedge_{P\in\mathcal{P}} \bigwedge_{t\geq 0} P(t,I,U,O) \wedge \bigwedge_{P\in\mathcal{P}} \bigwedge_{t\geq 0} P(t,\bar{I},\bar{U},\bar{O})$$

Keeping in mind the proof of the assertion ζ and the formula 5-1, the completenss check thus be rewritten to

6-1

$$IA(II,UU,OO) \wedge PP(II,UU,OO) \Rightarrow IC(II,UU,OO)$$

6.2.2 Implementable Integration Assumptions

Given traces UU and OO of internal and output signals, an implementable integration assumption IA should only allow input traces II that could be produced by a circuit in the environment of the circuit under verification. To demonstrate the implementabiliy of an integration assumption an equivalent circuit N^{IA} needs to be found. N^{IA} receives as inputs the traces U, $\bar{U}$, O und $\bar{O}$ of the internal and output signals of the arbitrary circuits M and $\bar{M}$ of the completeness check. On top of this, N^{IA} may have any number of additional input signals, which are collected in the input trace F. The outputs of N^{IA} provide the input traces I and $\bar{I}$ to M and $\bar{M}$. N^{IA}, its interface and its connection to M and $\bar{M}$ as well as its relation to IA is shown in Figure 18.

If there are flip flops in N^{IA} is is assumed that they are reset to an appropriate value at time 0. If N^{IA} is considered as a separate circuit, the traces UU and OO become free variables. To be an implementation of IA, N^{IA} must satisfy two requirements on these free variables: Firstly, every trace trace II that N^{IA} produces a for a given F, UU, OO, must satisfy $IA(II,UU,OO)$. Secondly, for every trace II that satisfies $IA(II,UU,OO)$ there must be a trace F that makes N^{IA} produce II when F, UU, OO are fed into it. Potential candidates for equivalent circuits N^{IA}can be obtained from [Schickel et al. 2006, Schickel et al. 2007].

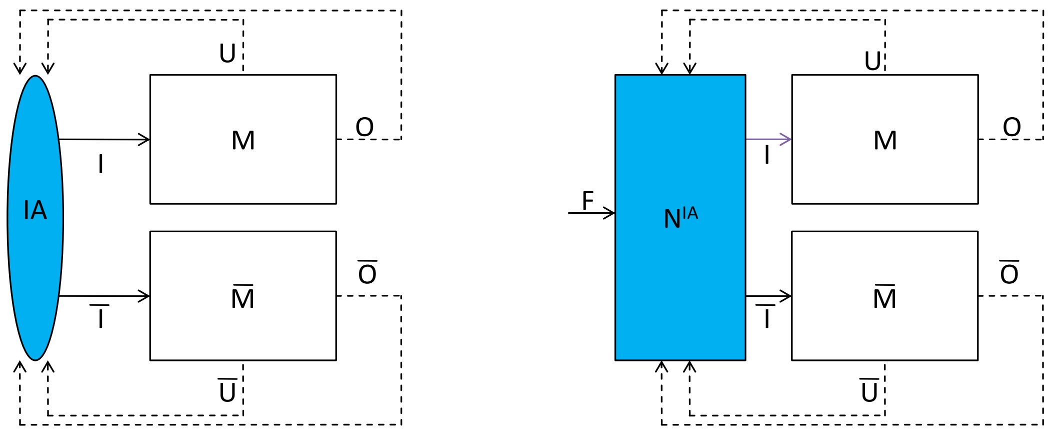


Figure 18: Implementable Integration Assumptions

One of the consequences of the definition is that non-reactive integration assumptions are always implementable.

Furthermore, an implementable integration assumption must not restrict the traces UU and OO. In other words, for every trace of internal and output signals there is always at least one input trace. This justifies the view that the integration assumption is in fact only a condition about the input signals, but potentially parameterized by the output and internal signals. Every element of this input trace may only depend on past and current values of UU, OO and F and must not depend on future values

A whole integration assumption need not be implementable, even if it is a conjunction of implementable parts. For example, it is easy to find a circuit that implements

```
if own_o = '1' then rdy_i = rdy_o end if;
```

It is equally simple to find a circuit with one input that implements

```
determined(rdy_i);
```

But the implementation assumption from section 6.1.5

```
integration_condition of peripheral is

assume:
      if own_o = '1' then rdy_i = rdy_o end if;
      determined(rdy_i);

guarantee:
      …

end integration_condition;
```

is not implementable, although it contains only of the two lines above. This can be seen as follows: The implementation assumption stands for the condition

$$(own\_o = '1' \Rightarrow rdy\_i = rdy\_o) \wedge \left(\overline{own\_o} = '1' \Rightarrow \overline{rdy\_\imath} = \overline{rdy\_o}\right) \wedge \left(rdy\_i = \overline{rdy\_\imath}\right)$$

In case of $own\_o = \overline{own\_o} = '1'$ it restricts the output trace to $rdy\_o = \overline{rdy\_o}$ in contradiction to the fact that implementable integration assumptions must not restrict the output signals. Consequently, it would already be the requirement of implementability of the integration assumption that would have uncovered the problem of section 6.1.5.

6.2.3 Criteria for the Implementability of Integration Assumptions

This section introduces criteria for the implementability of integration assumptions. For clarity, the trace UU of internal signals will no longer be written in the sequel. It can be considered as part of the trace OO.

To test the implementability of an integration assumption it must be proven that for every input trace II there is a trace OO of output and internal signals that satisfies $IA(II, OO)$. Such an existence proof can only be executed by SAT-provers under additional requirements about the structure of IA. The formalization of integration assumptions in ITL has been chosen to adhere to these requirements. The trick is that all integration assumptions are of a form that allows their transformation into a condition $\widetilde{IA}\left(II^{(t)}, II^{[0,t-1]}, OO^{[0,t]}\right)$ that maps past values of the input, output and internal traces and the current values of internal and output signals to the current value of the input trace. $\widetilde{IA}$ is related to the original integration assumption via

6-2

$$IA(II, OO) = \bigwedge_{t \geq 0} \widetilde{IA}\left(II^{(t)}, II^{[0,t-1]}, OO^{[0,t]}\right)$$

$II^{[n,m]}$ shall denote the list of values $II^{(n)}$, $II^{(n+1)}$, $II^{(n+2)}$, ..., $II^{(m)}$.

$\widetilde{IA}$ belongs to an implementable integration condition, if the elements of II can be determined one after the other, i.e. without a dead end situation where an unsuitable choice of some $II^{(t)}$ makes it impossible to find an $II^{(T)}$ for some $T > t$. Therefore it needs to be tested whether for every point in time t there is a value ii that satisfies $\widetilde{IA}\left(ii, II^{[0,t-1]}, OO^{[0,t]}\right)$. But even this test can become quite complex. Further optimization is therefore to split $\widetilde{IA}$ into subconditions $\widetilde{IA}_l$ that are responsible for the values of only few input variables, the trace of which will be denoted with II_l. For every l the subcondition $\widetilde{IA}_l$ may only depend on input variables that were determined by a $\widetilde{IA}_\nu$ with a smaller index. Hence, with $II = (II_0, II_1, II_2, \ldots II_n)$ and $II^{(t)}_{[0,l-1]} = (II^{(t)}_0, II^{(t)}_1, II^{(t)}_2, \ldots II^{(t)}_{l-1})$ it can be written

6-3

$$\widetilde{IA}\left(II^{(t)}, II^{[0,t-1]}, OO^{[0,t]}\right) = \bigwedge_{l=0}^{n} \widetilde{IA}_l\left(II^{(t)}_l, II^{(t)}_{[0,l-1]}, II^{[0,t-1]}, OO^{[0,t]}\right)$$

Based on this decomposition, a first implementability criterion can be provided: To that end a constant m shall be defined that provides the highest delay at which signals are evaluated in $\widetilde{IA}$. With the syntax from section 6.1.3, m provides the maximum depth with which prev operators are interleaved. The criterion shall be examined for all $l \in [0, n]$. It consists of a base case for $t \in [0, m-1]$:

6-4

$$\left(\varrho\varrho(II) \wedge \bigwedge_{k=0}^{t-1} \widetilde{IA}\left(II^{(k)}, II^{[0,k-1]}, OO^{[0,k]}\right) \wedge \bigwedge_{k=0}^{l-1} \widetilde{IA}_k\left(II^{(t)}_k, II^{(t)}_{[0,k-1]}, II^{[0,t-1]}, OO^{[0,t]}\right)\right) \Rightarrow \bigvee_{ii_l} \widetilde{IA}_l\left(ii_l, II^{(t)}_{[0,l-1]}, II^{[0,t-1]}, OO^{[0,t]}\right)$$

and a case for $t \geq m$:

6-5

$$\left(\bigwedge_{k=t-m}^{t-1} \widetilde{IA}\left(II^{(k)}, II^{[0,k-1]}, OO^{[0,k]}\right) \wedge \bigwedge_{k=0}^{l-1} \widetilde{IA}_k\left(II^{(t)}_k, II^{(t)}_{[0,k-1]}, II^{[0,t-1]}, OO^{[0,t]}\right)\right) \Rightarrow \bigvee_{ii_l} \widetilde{IA}_l\left(ii_l, II^{(t)}_{[0,l-1]}, II^{[0,t-1]}, OO^{[0,t]}\right)$$

To prove these formulae with SAT, the existential operator needs to be dissolved into a disjunction over all possible values of ii_l. This is possible if the signals of II_l consist only of a few bits. For wider variables an alternative implementability criterion is given below.

To examine condition 6-5 with a formal tool, the variables shall be created relative to t, i.e., there should be sets of variables for $OO^{(t)}$, $II^{(t)}$, $OO^{(t-1)}$, $II^{(t-1)}$, and so forth until $OO^{(t-2m)}$ and $II^{(t-2m)}$. Earlier elements of II and OO will not be evaluated in formula 6-5. The appendix (i.e. section 8) contains a proof that this criterion about integration assumptions guarantees the existence of an equivalent circuit N^{IA}.

In practice, many integration assumptions have a structure that allows an even simpler test about the implementability of an integration assumption, which is still applicable if the dissolution of $\bigvee_{ii_l} \widetilde{IA}_l\left(ii_l, II_{[0,l-1]}^{(t)}, II^{[0,t-1]}, OO^{[0,t]}\right)$ from the formulae 6-4 or 6-5 into a disjunction becomes too large.

This simplified examination is appicable to those $\widetilde{IA}_l$ that consist of a conjunction of constraints of the form

```
if $c_l\left(I_{[0,l-1]}^{(t)}, I^{[0,t-1]}, O^{[0,t]}\right)$ then $I_l^{(t)} = F_l\left(I_{[0,l-1]}^{(t)}, I^{[0,t-1]}, O^{[0,t]}\right)$ else $R_l\left(I_l^{(t)}\right)$ end if;
```

and determination assumptions of the form

```
if $g_l\left(I_{[0,l-1]}^{(t)}, I^{[0,t-1]}, O^{[0,t]}\right)$ then determined$\left(I_l^{(t)}\right)$ end if;
```

In this formula I stands for the trace of the input signals as before, O denotes the trace of output and internal signals. R_l, c_l and g_l are conditions, F_l an expression with the result type of I_l, such that the condition of the then-branch of the constraint can always be satisfied. The parameter lists indicate the variables that may occur in the respective expressions and conditions.

For the following considerations the indices and parameter lists of g, c and F will be omitted. $\bar{g}$, $\bar{c}$ and $\bar{F}$ denote application to $\bar{I}$ und $\bar{O}$. Then, the implementability test consists of a base case for $t \in [0, m-1]$:

6-6

$$\left(\varrho\varrho(II) \wedge \bigwedge_{k=0}^{t-1} \widetilde{IA}\left(II^{(k)}, II^{[0,k-1]}, OO^{[0,k]}\right) \wedge \bigwedge_{k=0}^{l-1} \widetilde{IA}_k\left(II_k^{(t)}, II_{[0,k-1]}^{(t)}, II^{[0,t-1]}, OO^{[0,t]}\right)\right)$$
$$\Rightarrow \left(R(F) \wedge ((g \vee \bar{g}) \wedge c \wedge \bar{c} \Rightarrow F = \bar{F})\right)$$

and a case for $t \geq m$:

6-7

$$\left(\bigwedge_{k=t-m}^{t-1} \widetilde{IA}\left(II^{(k)}, II^{[0,k-1]}, OO^{[0,k]}\right) \wedge \bigwedge_{k=0}^{l-1} \widetilde{IA}_k\left(II_k^{(t)}, II_{[0,k-1]}^{(t)}, II^{[0,t-1]}, OO^{[0,t]}\right)\right)$$
$$\Rightarrow \left(R(F) \wedge \left((g \vee \bar{g}) \wedge c \wedge \bar{c} \Rightarrow F = \bar{F}\right)\right)$$

These tests can be executed with SAT algorithms, without the growth of the formulae by the unfolding of the existential operators in 6-5.

These tests can be checked with SAT algorithms and the formulae do not swell as before through the unfolding of the existential quantifier. The remarks made above about the selection of the variables also apply to formula 6-7. Again, a theorem about this implementability condition is proven in the Appendix.

As an example, the second criterion shall be applied to the original integration assumption of the peripheral of the example in section 6.1.5. This was

```
integration_condition of peripheral is

assume:
      if own_o = '1' then rdy_i = rdy_o end if;
      determined(rdy_i);

guarantee:
      …

end integration_condition;
```

It is rdy_i which is determined by the integration assumption, furtheron $R = 1$, $g = 1$, $c = (own\_o = 1)$ and $f = rdy\_o$. The examination of $R(F)$ is trivial, but $(g \vee \bar{g}) \wedge c \wedge \bar{c} \Rightarrow F = \bar{F}$ is instantiated by

$$(own\_o = 1 \ \wedge \overline{own\_o} = 1) \Rightarrow rdy\_o = \overline{rdy\_o}$$

And this condition cannot be proven on free variables. This indicates that the original integration condition from section 6.1.5 is not implementable. Hence the complete verification of the peripheral must not be merged with other complete verifications. .

With the improved integration assumption

```
integration_condition of peripheral is

assume:
      if own_o = '1' then rdy_i = rdy_o end if;
      if own_o = '0' then determined(rdy_i) end if;

guarantee:
      …

end integration_condition;
```

the main part of the tests 6-6 and 6-7 becomes

$$\big((own\_o = 0 \vee \ \overline{own\_o} = 0) \wedge own\_o = 1 \ \wedge \overline{own\_o} = 1\big) \Rightarrow rdy\_o = \overline{rdy\_o}$$

This test is satisfied, because the left side of the implication is always violated. Therefore, this improved determination assumption should be used for the verification of the example of section 6.1.5.

6.2.4 Structural Compatibility

The second requirement about the integration assumptions requires structural compatibility with the circuit. To test the structural compatibility a list K^M of true combinatorial dependencies between an input signal and an internal or output signal is computed. To this end, a syntactic analysis of the RTL code identifies candidates $i \rightarrow o$ of a combinatorial dependency between an input signal i and an internal or output signal o. Some of these candidates may only be syntactic dependencies. To identify true combinatorial dependencies the property

```
property check_dependency is
assume:
      at t:       <$v = \bar{v}$> for all input signals except i;
prove:
      at t:       $o = \bar{o}$;
end property;
```

is checked on two copies of the model of the circuit under verification. If the property fails, the syntactic dependency $i \rightarrow o$ is a real one and becomes part of K^M.

The combinatorial dependencies of the equivalent circuit $N^{IA}$of the integration assumption IA are only determined by a syntactical argument. By this approach the graph K^{IA} of combinatorial dependencies of the integration assumption may become a bit larger than necessary. By this the criterion may become somewhat pessimistic, but it is assumed that the integration assumption could be rewritten if a syntactic dependency disturbs the criterion although it is not real.

The approach is as follows: If the ITL of $\widetilde{IA}_l\left(II_l^{(t)}, II_{[0,l-1]}^{(t)}, II^{[0,t-1]}, OO^{[0,t]}\right)$ references an output or internal signal o at time $t,$ a combinatorial dependency $o \rightarrow i$ is included in K^{IA} for every signal i in $II_l^{(t)}$. Further combinatorial dependencies $\hat{\imath} \rightarrow i$ will be included for all inputs $\hat{\imath}$ that are described by $II_{[0,l-1]}^{(t)}$. By this, K^{IA} may contain longer paths.

The combinatorial dependencies in K^{IA} reflect the situation of the equivalent circuit, the creation of which is provided in the Annex to demonstrate the implementability of IA. If K^{IA} is extended by the combinatorial dependencies from K^M, the situation of the combined circuit is reflected. To conclude structural compatibility, $K = K^{IA} \cup K^M$ needs to be cycle free.

It is sufficient to restrict the examination of K onto the variables of only one model of the circuit under verification. This is a consequence of symmetry, because if $a \rightarrow b$ is a combinatorial dependency of K^{IA} or K^M then $\bar{a} \rightarrow \bar{b}$ is also a combinatorial dependency and vice versa. Mixed dependencies of the form $\bar{a} \rightarrow b$ or $a \rightarrow \bar{b}$ are the result of a determination condition which then also leads to $a \rightarrow b$, such that cycles of K that extend over both sets of variables will always appear together with cycles in only one set of variables.

6.2.5 Plausibility Criteria about Constraints and Determination Assumptions in ABFV and Complete Verification

This chapter presents implementability and structural compatibility as the two requirements that ensure that an integration condition is sensible. The implementability of the integration condition is independent of the circuit, whereas the structural compatibility can only be determined if the respective circuit is known.

Plausibility conditions about the constraints of a verification are also important in ABFV. There, the plausibility conditions are intended as a guide against unsuitable constraints that may devalue the verification. By actually small errors in the constraints the assertions are proven for less inputs than intended by the user. A verification under erroneously restrictive constraints becomes simpler: The assertions can be proven more easily, and usually the user weighs himself in false happiness. This is because the devaluation of the verification does not easily become apparent because constraints are the only objects of a verification that are not being questioned by the formal tools. To counter this problem, reviews about the constraints are executed. In addition plausibility tests have been developed to check the constraints. These plausibility tests detect self-contradictory constraints. The test consists of the proof of an implication of the form

$$\prod Constraints \Rightarrow false$$

If this can be proven, the constraints are contradictory in themselves and therefore unsuitable. If the proof leads to counterexamples, the constraints are accepted. For the plausibility tests, the constraints are initially regarded as free variables. In this case the contradictions that are discovered arise solely from the interplay of the constraints themselves. In a second pass of the plausibility test it is executed on the circuit under verification. This might detect incompatibilities between the constraints and the circuit.

If a plausibility test fails, users want diagnostic support. But it is difficult to generate it for the conventional ABFV plausibility test, because the problem arises often from the interaction of multiple conditions which are harmless if examined separately.

However, the hunt for contradictions in the constraints is only a rough test, that highlights only the most egregious cases of bad constraints. This test is already satisfied when the constraints and the circuit allow even a single input trace. Even if a reactive constraint inadvertently restricts the output behavior of a circuit, it is usually not identified by this rough test.

Compared to that, the implementability and structural compatibility tests presented here are much more critical plausibility tests. Their application to integration assumptions is due to the focus of this work. But they can be easily adapted to the needs of ABFV. What needs to be done is to check the implementability of the constraints, and they need to be structurally compatible with an instance of the circuit.

This allows at least identifying all situations, in which there is at least one trace of output and internal signals, for which the constraint does not allow any input trace. Such a trace of output and internal signals can advantageously be returned as a counter example. Consequently, all constraints that satisfy the proposed tests are already free of contradictions. Hence the proposed tests reject all constraints that would also be rejected by the usual plausibility tests.

But it gets even better, because the test for structural compatibility ensures that for every point in time there is an input value that satisfies the constraint. Hence it is secured, that constraints that pass the proposed tests are free of contradictions, even if the circuit is accounted for. The examination of combinatorial paths is sufficient to reject any constraint that would be rejected in combination with the circuit.

All in all, the presented tests are attractive alternatives to todays plausibility tests about constraints. They allow a more detailed examination, and besides that they also allow more detailed diagnosis, because on one side problematic lines of ITL code can be highlighted on the other side an output trace can be calculated, in which the problem arises. If the even more detailed tests from the second part of section 6.2.3 are performed, even more accurate diagnosis is possible.

6.3 Assembling Complete Verifications

6.3.1 Compositional Complete Verification and Assume-Guarantee Reasoning

The interconnection of circuit components is treated in many approaches to model digitial functionality, such as ASMs [Börger / Stärk 2003], TLT [Cuellar / Huber 1995], TLA [Lamport 1994], or UML [Broy / Rumpe 2007]. Related activities are subsumed by the keyword Assume-Guarantee Reasoning [Abadi / Lamport 1995]. They also have implications for reactive constraints derived from the environment of the model [Broy 1994].

Assume-Guarantee Reasoning usually uses Moore automata, i.e. automata without direct dependency between outputs and inputs. To justify, that the combination of properly functioning Moore automata M_1 and M_2 functions correctly, a proof by contradiction is executed. It is assumed that the combination eventually does something wrong. Then there must be a first time at which the malfunction occurs in one of the two components. However, the malfunction can only occur if previously the other component does not behave properly. This is in contradiction to the choice of the earliest time when the malfunction had been observed. So the the composition of M_1 and M_2 works correctly.

In contrast, complete verification considers operation automata, for which it is almost a main feature that the output signals can be controlled by the inputs. The operation and transaction automata are therefore no Moore automata, such that the usual argument of Assume-Guarantee-Reasoning is not directly applicable. Nevertheless, the theory presented below does not impose restrictions on the form of the operation and transaction automata, because the problem could be pushed back to the verification of the underlying circuits. But the usual automaton representation of synchronous digital circuits is also no Moore automaton, because that would prohibit combinatorial dependencies between input and output signals. But taking advantage of the requirement that the circuits must not contain combinatorial loops, a refined time model based on causality could be defined, to which the usual Assume-Guarantee reasoning can be applied. The results can be transferred to the operation automata in turn, because they were proven against the RTL description.

All in all, there are differences to the aforementioned modeling process, because here circuits are to be verified. This requires firstly the modelling of the function. Secondly, there are integration conditions which [Broy/Rumpe 2007] classifies as interface conditions, which provide information about the verification code of the single module.

Another peculiarity of the theory presented here is that it can be applied if a verification task is to be divided into the verification of multiple clusters which are connected to each other cyclically. It is not required that the model has to be decomposed into the individual clusters, as suggested by e.g. [Abadi / Lamport 1995]. Instead it has some practical advantage if the model contains multiple clusters, and only a few individual signals have to be cut, i.e., replaced by primary inputs and outputs. This cutting shall establish a cycle free dependency graph of the clusters. The clusters may remain connected in something like the opposite direction. This can considerably simplify the verification, as illustrated by section 3.5.7.

6.3.2 Preconditions for the Assembly of Complete Verification

The starting point of the investigation is shown in Figure 19, a circuit M consisting of blocks M_j, each block being fully verified separately with local integration commitments IC_j and local integration assumptions IA_j.

M shall have an own global integration commitment GC, as well as an own implementable global integration assumption GA. M and GA shall be structurally compatible in accordance with section 6.2.4 .

This chapter shall provide information on the conditions that ensure that M is completely verified by the union of the complete property sets about the M_j, with the global integration assumption GA and the global integration commitment GC.

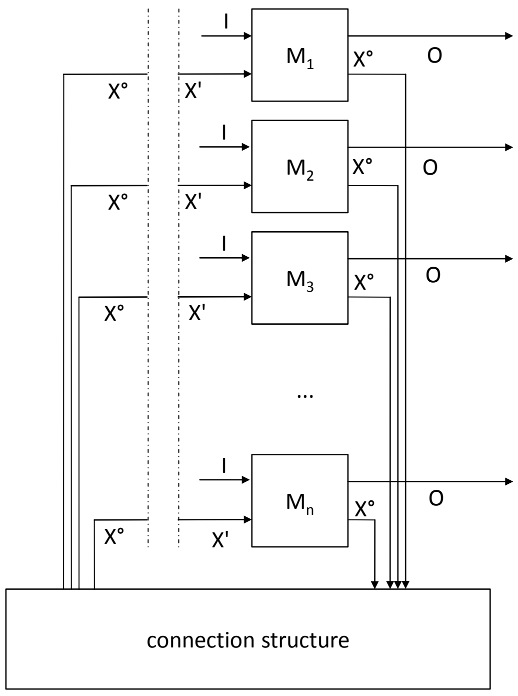


Figure 19: Assembling complete verifications

The input signals of the M_j are divided into primary input signals of the overall circuit that form the trace I and signals that are connected to output signals of other blocks. This connection structure is given by point-to-point signals that are collected in the trace X. Only a selection of them actually leads to the block M_j. If a block is driving several inputs, usually because it is connected to multiple other blocks, X contains multiple signals and the block has multiple copies of the output signal that an integration commitment requires to be equal.

The output signals of the M_j are divided analogously into primary output signals that form the trace O, and signals that drive the input of an other module. These connecting signals are assembled in the trace X. As long as the circuit is cut along the line shown in Figure 19, each M_j is examined separately, and the signals in the connection structure form a trace $X°$ of output signals and a trace X' of input signals. The behavior of M_j in separation is given by a trace predicate $M_j(I, X', U, O, X°)$. Since the full circuit is assumed to be free of combinatorial cycles, the predicate of the full circuit is

$$M(I, U, X, O) = \bigwedge_j M_j(I, X, U, O, X)$$

where U is the trace of all internal signals of all M_j.

In the manner described above the signals in the property sets $\mathcal{P}_j$ of the M_j shall already be renamed such that they reflect the connection structure. The M_j shall be completely verified according to the integration assumptions IA_j and the integration commitments IC_j. To ensure that the full circuit is completely verified by $\bigcup_j \mathcal{P}_j$ with the integration assumption $GA(II, UU, XX, OO)$ and the integration commitment $GC(II, UU, XX, OO)$ the following preconditions must be met:

Precondition 1 requires that the overall graph of all combinatorial dependencies of the overall circuit, the global and local integration conditions $K^{GA} \cup K^M \cup \bigcup K^{IA_j}$ shall be cycle free.

Precondition 2 requires that the global integration assumption $GA(\boldsymbol{II}, UU, XX, OO)$ shall be implementable and restricts only II (the restricted trace shall be highlighted by bold face). In addition, the integration assumptions $IA_j(\boldsymbol{II}, \boldsymbol{XX'}, UU, OO, XX°)$ shall be implementable and shall restrict only II und XX'.

Precondition 3 requires that the global integration assumption $GA(\boldsymbol{II}, UU, XX, OO)$, if evaluated on free variables, may only allow those traces II that are also admitted by the local integration assumptions IA_j, i.e.

$$GA(\boldsymbol{II}, UU, XX, OO) \Rightarrow \bigvee_{XX'} \bigwedge_j IA_j(\boldsymbol{II}, \boldsymbol{XX'}, UU, OO, XX°)$$

Precondition 4 requires that, if evaluated on free variables, the local integration assumptions are justified by local integration commitments and the global integration assumption, i.e.,

$$GA(\boldsymbol{II}, UU, XX°, OO) \wedge GA(\boldsymbol{JJ}, UU, XX', OO) \wedge \bigwedge_j IC_j(II, XX', UU, OO, XX°)$$
$$\Rightarrow \bigwedge_j IA_j(\boldsymbol{JJ}, \boldsymbol{XX}°, UU, OO\ XX')$$

Here, JJ denotes a trace of copies of the variables of II.

Precondition 5 requires that the global integration commitment must be a result of the local integration commitments and the global integration assumption, i.e. that

$$GA(\boldsymbol{II}, UU, XX, OO) \wedge \bigwedge_j IC_j(II, XX, UU, OO, XX) \Rightarrow GC(II, UU, XX, OO)$$

holds on free variables.

If the preconditions 1 to 5 are satisfied, the union $\bigcup_j \mathcal{P}_j$ of the property sets of the partial circuits completely verifies the whole circuit M. The appendix provides the proof.

6.3.3 Examination of the Preconditions

Section 6.2 describes in detail, how the preconditions 1 and 2 are checked. The preconditions 4 and 5 are obvious applications of a SAT prover.

It remains to be clarified, how the existence statement of precondition 3 is to be checked. In practice, the individual interfaces of a block are usually independent, and the signals of II form such an interface. It is therefore likely, that every IA_j can be decomposed into a number of conjuncts, one of which is IA_j^{II} about the signals of II that is independent of XX', i.e.

$$IA_j(\boldsymbol{II}, \boldsymbol{XX'}, UU, OO, XX^\circ) = IA_j^{II}(\boldsymbol{II}, UU, OO, XX^\circ) \wedge IA_j^{XX'}(II, \boldsymbol{XX'}, UU, OO, XX^\circ)$$

In this case the check of precondition 3 can be reduced to the implication

$$GA(\boldsymbol{II}, UU, XX, OO) \Rightarrow \bigwedge_j IA_j^{II}(\boldsymbol{II}, UU, OO, XX^\circ)$$

that can easily be handled by a SAT prover.

6.3.4 Role of the Communication Structure

When selecting IP blocks for a SoC it is usually not known exactly, how the blocks are to be interconnected. Accordingly it is checked only for selected connections, whether the blocks involved can communicate. The selection of the blocks that are examined depends on the intended communication. Point-to-point connections need only two blocks, but if blocks are to communicate via a bus system, then all blocks of the bus system need to be interconnected. If the bus system is to be designed to match the performance requirements of the blocks later on, it may first be replaced with a simpler structure, just to ensure protocol compliance. For example, a bus matrix with several connection layers need not be developed before the protocol compliance of the IP blocks is checked. Nor is it necessary to introduce flip flops for intermediate synchronization of physically long buses.

It is advantageous if the bus infrastructure is an IP block itself. In this case, this IP block is just one of the M_j like any other IP block. In this case the connection structure of section 6.3.2 becomes particularly simple, because it only needs to represent how each IP is connected to the bus.

6.4 Decomposition of a Complete Verification Task

6.4.1 Disassembling a Verification Task into Clusters

This section is about the complete verification of a circuit that was designed and developed as a whole. This could for example be one of the blocks, which are combined into an SoC with the procedures previously presented.

As discussed in section 3.5.6, such circuits can usually not be described by a single operation automaton. Instead, the circuits must be considered in parts called clusters, for which operation automata are created. The overall function of such circuits is described by multiple chains of operation properties. The chains run concurrently one besides the other, and they are connected to each other via signals. Accordingly, the individual clusters have input and output signals that are internal signals in the context of the overall circuit.

If a formal verification engineer shall work on a larger RTL, s/he may decide how much of the RTL shall be processed with the frontend of the assertion checker, which converts it into an automaton. The choice may range between two extremes: The user may choose to convert the whole RTL at once. But then, a verification in multiple clusters may lead to inconsistencies that will be discussed in section 6.4.3. These inconsistencies can be avoided if the clusters are examined in isolation. This is simple, if a cluster coincides with some module of the circuit. In this case the user may indicate to the formal model generation that it shall create the model for that module. This turns the module inputs and outputs into primary signals. If the cluster does not coincide with the module boundaries of the circuit, a larger circuit will be specified for the formal model generation, and then the input and output signals of the cluster will be “cut”. This is an operation of the formal frontend, which turns the driver of the cut signal into a primary output and creates a primary input that feeds all loads of the cut signal. By this, the model may contain superfluous parts besides the cluster, but these parts will not influence the behavior of the cluster because there is no connection between the superfluous parts and the cluster.

Examining the clusters in isolation has the disadvantage, that all restrictions about the behavior of input signals of the cluster must be made explicit in a constraint. This is possible, but sometimes creates more effort than necessary. A potential optimization is discussed in the next section 6.4.2.

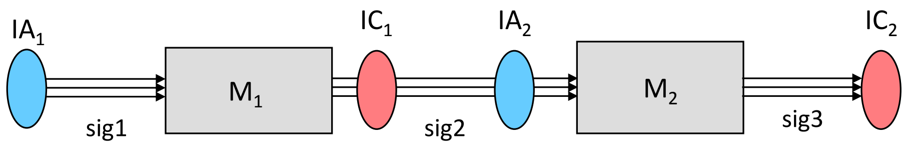


Figure 20: Internal signals being input signals to a cluster

6.4.2 Advantages of Models with Multiple Clusters

Usually it is not necessary to completely isolate a cluster. An example of this is given in Figure 20. In this example, the proof of the operation properties of both clusters can be carried out on the full model of the circuit that contains both clusters, without cutting any signals between the clusters. It only needs to be ensured that $IC_1 \Rightarrow IA_2$. Then the reasoning from section 6.1.4 can be used to prove that the property sets of M_1 und M_2 provide a complete verification of the full circuit.

If the clusters are not completely separated in the model, there are input and output signals of a cluster, that are internal signals of the circuit under verification. Such input and output signals are referred to as internal, as opposed to the primary input and output signals. If the focus is on the interface of a cluster, with no distinction between primary and internal signals, the term local inputs and outputs of the cluster will be used. In Figure 20, the signals $sig2$ are internal inputs of the cluster M_2 and internal outputs of M_1. The local inputs and output of M_2 are $sig2$ and $sig3$.

Conditions shall be called local constraints if they take the role of a constraint during the complete verification of an incompletely separated cluster. For the verification of the full circuit, local constraints are justified by assertions of the neighbouring clusters and global constraints of the full verification.

It is advantageous not to completely separate the clusters of a complete verification, because that saves verification effort. The cluster M_1 in Figure 20, for example, ensures that M_2 obtains only those traces that M_1 can actually produce. Since the verification is carried out with IPC, these traces include those from unreachable states of M_1 but even this still means a limitation of the behavior of $sig2$. This limitation makes the verification of M_2 easier, although it need not be mentioned explicitly in IC_1 and IA_2.

If M_1 and M_2 were examined separately, $sig2$ would become a primary input of the model that contains M_2. The protocol of this internal interface is often not standardized, so that the identification of the protocol conditions can become a time-consuming task. These conditions should not be too weak, because then the proofs of the operation properties M_2 become impossible, but they should also not be too strong, because then they cannot be proven on M_1. To find the right balance can already be difficult if the two circuits are free of errors. But it gets worse, if e.g., M_2 has an error that only occurs if a certain feature of the protocol is used. A user will be tempted to insert and remove a condition about the protocol feature, depending on wether s/he proves M_1 and M_2. If M_1 and M_2 remain connected, many conditions of the protocol need not be fomalized explicitly, and this saves effort.

6.4.3 Necessity of Primary Input and Output Signals

Unfortunately, the reasons for the completeness of the verification of the circuit from Figure 20 cannot easily be transferred to circuits with a cyclic connection of the clusters. An example of the verification gaps that may occur is shown in Figure 21. There are two clusters, which both consist mainly of a flip-flop with enable. When the enable signal has the value '1', the flip-flops store from the value of the data input signal, otherwise, they keep the number that is already stored.

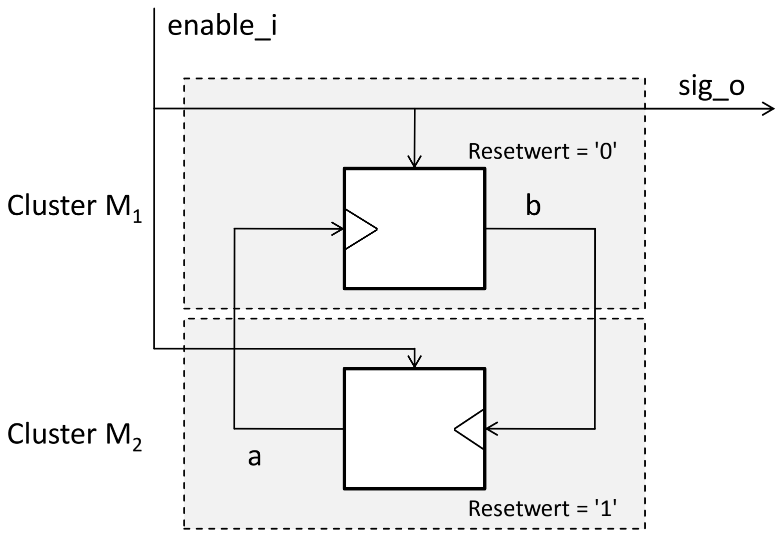


Figure 21: Cyclic Cluster Dependency

Cluster M_1 can be completely verified with the Integration Condition

```
integration_condition of M1 is

assume:
      determined(a);
      determined(enable_i);
      prev(reset) or prev(a) /= a;

guarantee:
      determined(b);
      prev(reset) or prev(b) /= b;
      determined(sig_o);
      sig_o = '1';

end integration_condition;
```

The corresponding operation property is

```
property m1_op is
dependencies: constraints_from_integration_assumption, no_reset;
assume:
      at t:                   b = '1';
prove:
      at t+1:                 b = '0';
      at t+2:                 b = '1';
      during [t+1, t+2]:      sig_o = '1';
end property;
```

Surprisingly, this property can be proven on the full model. Since the constraint restricts the output a of the flip-flop from M_2 it implicitly requests that the enable signal yields ‘1’, as soon as the reset becomes inactive. Therefore, the register values are pushed forward such that the change of the flip-flop values actually happens, that is claimed by the property. This proves the assertion of the integration commitment.

Analogously, M_2 can be completely verified under the integration condition

```
intergration condition of M2 is

assume:
      determined(b);
      determined(enable_i);
      prev(reset) or prev(b) /= b;

guarantee:
      determined(a);
      prev(reset) or prev(a) /= a;

end integration_condition;
```

But from these two complete verifications is must not be concluded that the full circuit satisfies the integration condition

```
integration_condition of M is

assume:
      determined(enable_i);
```

```
guarantee:
      determined(sig_o);
      sig_o = '1';

end integration_condition;
```

Because this would obviously be wrong: sig_o is directly connected to $enable\_i$, and there is no constraint that restricts it. Hence, sig_o can of course become ‘0’.

This happens, although the integration conditions of M_1 and M_2 satisfy almost all preconditions for the assembly of complete verifications according to section 6.3.2. The only exception is that it was required that IP blocks are completely verified on separate models. In fact it is not possible to verify the clusters M_1 and M_2 on separate formal models with a and b as primary inputs and outputs. Consequently, the integration conditions of M_1 and M_2 are no longer correct and the global integration condition can no longer be concluded.

6.4.4 Adaptation of the Model on the Base of the Cluster Graph

The theory of compositional complete verification that was developed for this thesis balances between the desire to save effort by keeping the clusters connected as much as possible, and the requirements arising from the soundness of the proof procedure that requires to cut signals between the clusters, as demonstrated by the counter example from section 6.4.3. The compromise is to allow models in which multiple clusters may be connected by internal signals, but where connecting signals are cut to the extent necessary to avoid cyclic dependencies between the clusters.

The first activity to set such a verification up is the extraction of the cluster graph. In this graph, the clusters are nodes that are connected by directed edges. The edges lead from one cluster to another, if there is a local output signal of the first cluster that is a local input signal of the second cluster. In general, this cluster graph contains cycles. These cycles must be broken. This is done by elimination of a number of edges. These edges correspond to signals that need to be cut. The cutting of signals was introduced in section 6.4.1.

By this a model with multiple connected clusters is created, that has a number of additional primary inputs and outputs. The complete verification of the clusters is executed on this model. If during these verifications reactive constraints become necessary about one or the other local cluster input, these inputs need to be cut as well, and the verification may need to be suitably adapted.

Hence, if there are clusters that interact with each other linearly, their interaction need not be cut and this might save effort. But in practice there are also reasons to cut more than the minimum number of signals, e.g. to reduce the size of the model if it leads to high proof times.

6.4.5 Preconditions about the Integration Assumptions

Following the previous section, the original task of formal verification of some larger circuit M is decomposed into the subtasks to completely verify the clusters M_j, but on a model M' that contains multiple clusters and many, but not all, connections between the clusters. The effort for a verification on M' are higher than the effort for the inconsistent verification on M, but the effort increase can usually be limited if the cut signals are chosen wisely, and it remains less than the additional effort that arises if separate models were used for every M_j.

Like in section 6.3.2, the integration conditions of the subtasks and the global integration conditions must also meet certain preconditions to ensure that the overall circuit is completely verified by the set of all properties of all clusters.

To formalize these preconditions, it must be noted that there are now three categories of local input signals for each cluster. The first category are the primary signals of the overall circuit. Their trace shall be denoted with I as usual. A second category are the primary inputs created by cutting internal signals. Their trace shall be denoted with X'. The third category are the uncut internal input signals with the denotation Y. Analogously there are primary output signals of the overall circuit with the trace O, new primary outputs with the trace $X°$ created by cutting, and internal output signals with the trace Y.

The following preconditions are modified versions of those in section 6.3.2, primarily supplemented by the treatment of the traces Y of internal inputs and internal outputs of the cluster:

Precondition 1 requires that the integration assumption IA_j of each cluster can be written as a conjunction of a condition IA_j^{int} about the internal inputs and outputs, and a condition IA_j^{prim} about the primary signals, i.e. the primary inputs and output of the overall circuit and all cut signals:

$$IA_j(II, XX', YY, UU, OO, XX°) = IA_j^{int}(YY, UU, OO) \wedge IA_j^{prim}(\boldsymbol{II}, \boldsymbol{XX'}, YY, UU, OO, XX°)$$

The combinatorial dependencies of M, of the integration assumptions IA_j^{prim} and the global integration assumption GA must again form a cycle free graph $K^M \cup K^{GA} \bigcup K^{IA_j^{prim}}$.

Precondition 2 requires that the global integration assumption $GA(\boldsymbol{II}, UU, XX, YY, OO)$ is implementable and that only the input trace II of MM' is restricted by GA. Moreover, the integration assmptions $IA_j^{prim}(\boldsymbol{II}, \boldsymbol{XX'}, YY, UU, OO, XX°)$ must be implementable and must constrain only II and XX'.

Precondition 3 requires that $GA(\boldsymbol{II}, UU, XX, YY, OO)$ allows on free variables only those traces II that are also admitted by the integration assumptions, i.e.

$$GA(\boldsymbol{II}, UU, XX, YY, OO) \Rightarrow \bigvee_{XX'} \bigwedge_j IA_j^{prim}(\boldsymbol{II}, \boldsymbol{XX'}, YY, UU, OO, XX°)$$

Precondition 4 requires that the local integration commitments of the clusters and the global integration assumption satisfy the local integration assumptions of the clusters. Consequently,

$$GA(\boldsymbol{II}, UU, XX°, YY, OO) \wedge GA(\boldsymbol{JJ}, UU, XX', YY, OO) \wedge \bigwedge_j IC_j(II, XX', YY, UU, OO, XX°)$$
$$\Rightarrow \bigwedge_j IA_j(JJ, XX°, YY, UU, OO\ XX')$$

should hold on free variables. Again, JJ denotes a trace of copies of the variables of II.

Precondition 5 requires that the global integration commitment must be a consequence of the local integration commitments and the global integration assumption, i.e.

$$GA(\boldsymbol{II}, UU, XX, YY, OO) \wedge \bigwedge_j IC_j(II, XX, YY, UU, OO, XX) \Rightarrow GC(II, UU, XX, YY, OO)$$

shall hold on free variables.

If the preconditions 1 to 5 hold, the union $\bigcup_j \mathcal{P}_j$ of the properties is complete and verifies M under the integration conditions GA and GC. A detailed proof will be provided in the Appendix.

The process to examine the preconditions is analogous to what was said in section 6.3.3.

6.4.6 Example

At the end of this thesis the decomposition of a complete verification shall be demonstrated at a circuit that consists of the processor and the SDRAM interface. The scenario shall be that both were developed together, and the SDRAM Interface is an exclusive interface to the instruction memory. Moreover, it shall be assumed that no verification was executed so far. The protocol between SDRAM Interface and processor shall represent an ad-hoc protocol. This means that initially no protocol description is available, neither informally by some specification, nor by the integration conditions from section 3.5.2

SDRAM memory blocks are to be used that are provided by a third party. These are characterized by data sheets from which, by consideration of the clock frequencies, the integration conditions are deduced:

```
integration_condition of sdram is
assume:
      determined(sd_ctrl);
      if sd_ctrl = read or sd_ctrl = write or sd_ctrl = activate then
            determined(sd_addr);
      end if;
      if sd_ctrl = precharge or sd_ctrl = activate then
            next(sd_ctrl = nop) end if;
      end if;
      if sd_ctrl = write then determined(sd_wdata) end if;

guarantee:
      if prev(sd_ctrl, 2) = read then determined(sd_rdata) end if;
end integration_condition;
```

The cluster graph of the processor and the SDRAM interface contains cycles. To select the signals that should be cut, it is observed that the SDRAM interface requires that request and address signals are kept stable during a request, i.e., until activation of the ready signal. This requires a reactive constraint that restricts the signals request and address and therefore they are cut, which has the nice side effect that this already removes the cycles from the cluster graph. The verification is therefore performed on a model with signal cuts according to Figure 22. This block diagram corresponds to Figure 3, but with a few differences. In particular, the rw signal is treated as if it is always set to '1', because the SDRAM interface should only read instructions. Therefore, the write data signal is superfluous[4].

A further complication shall be that the processor only works correctly if the ready signal provides pulses and is never active for two consecutive clock cycles. If processor and SDRAM interface were verified separately, this condition needed to be an explicit part of the integration conditions. But since the two modules are interconnected in one model, this con-

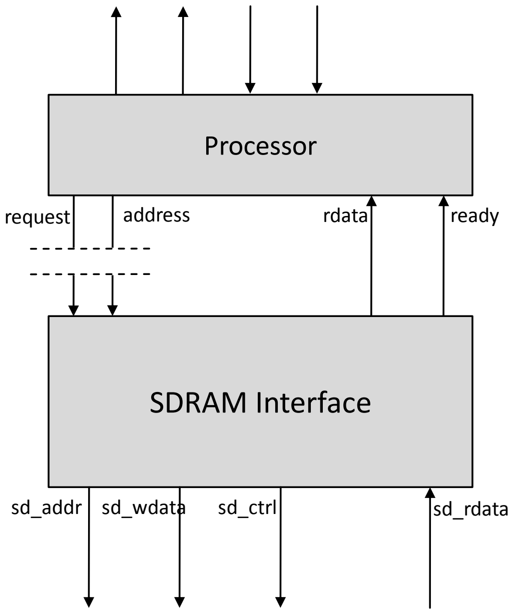


Figure 22: Verification if a circuit consisting of processor and SDRAM interface

dition can be left out. The resultant integration conditions are

```
integration_condition of processor is
assume:
      determined(ready);
      if ready = '1' then determined(rdata); end if;

      <assumption about the data interface>
guarantee:
      determined(request);
      if request = '1' then determined(address); end if;
      if prev(request) = 1 and prev(ready) = 0 then
            request = '1' and
            address = prev(address);
      end if;

      <commitment about the data interface>
end integration_condition;
```

and

```
integration_condition of sdram_if is
```

[4] For simplification it shall be assumed that the program code is already in the memory. The mechanisms to load the binary are ignored.

```
assume:
      determined(request);
      if request = '1' then determined(address); end if;
      if prev(request) = 1 and prev(ready) = 0 then
            request = '1' and
            address = prev(address);
      end if;

      if prev(sd_ctrl, 2) = read then determined(sd_rdata) end if;
guarantee:
      determined(ready);
      if ready = 1 then
            determined(rdata);
      end if;

      determined(sd_ctrl);
      if sd_ctrl = precharge or sd_ctrl = activate then
            next(sd_ctrl = nop) end if;
      if sd_ctrl = read or sd_ctrl = activate then
            determined(sd_addr)
      end if;
      sd_ctrl /= write;
end integration_condition;
```

The integration conditions do not differentiate between signals that are created by cutting internal inputs or outputs, because this can be sorted out automatically.

It will now be examined whether these two integration conditions verify that the entire circuit is verified under the global integration condition

```
integration_condition of full_circuit is
assume:
      if prev(sd_ctrl, 2) = read then determined(sd_rdata) end if;
      <integration assumptions of data interface>
guarantee:
      determined(sd_ctrl);
      if sd_ctrl = precharge or sd_ctrl = activate then
            next(sd_ctrl = nop) end if;
      if sd_ctrl = read or sd_ctrl = activate then
            determined(sd_addr)
      end if;
      sd_ctrl /= write;
      <integration commitment about data interface>
end integration_condition;
```

To this end, the preconditions of section 6.4.5 get tested.

Firstly, the cluster graph of Figure 22 is cycle-free. The SDRAM cluster has no local cluster inputs, therefore $IA_{sd}^{int} = true$. Moreover, IA_{sd} has no combinatorial dependencies.

For the processor interface, IA_p^{int} is given by the integration assumption with the exception of the provisions for the data interface. It is $IA_p^{prim} = true$ and therefore no combinatorial dependency involved.

There are also no combinatorial dependencies in GA. This satisfies precondition 1.

IA_{sd}^{prim}, IA_{p}^{prim} and GA are implementable. Therefore precondition 2 is satisfied.

By recalculating it can be seen that the preconditions 3, 4 and 5 are also met. This shows that the property sets of the processor and of the SDRAM interface together completely verify the overall circuit.

7 Epilogue

This thesis describes a breakthrough in the functional verification of digital circuit designs. It first transfers the paradigm of transaction-based verification from simulation to formal verification. One result of this transfer is a particular form of formal properties, called operation properties. Circuits are examined with operation properties by Interval Property Checking, a particularly powerful SAT-based functional verification. This allows examination of circuits, which are otherwise considered too complex for formal verification. Furthermore, this thesis describes a tool that is to be applied to sets of operation properties, and identifies all verification gaps, keeps pace with the complexity that IPC-based formal verification can handle and is called the completeness checker. The methodology of the operation properties and the technology of the IPC-based property checker and the completeness checker form a beneficial symbiosis for functional verification of digital circuits. Based on this, a procedure is developed to completely verify also the connection of completely verified modules, which is derived from the theories about modeling of digital systems.

The approach presented has demonstrated in many commercial application projects that it rightly bears the name "complete functional verification", because in these application projects no errors were found after reaching a termination criterion that is well-defined by the completeness checker.

8 Appendix: The Mathematics of Compositional Complete Verification

Remark: To save effort I will not translate this section until somebody expresses dedicated interest and is not capable of the German language. Please get in touch with me: joerg.d.bormann@web.de.

9 Literature

Abadi / Lamport 1995
M. Abadi, L. Lamport: Conjoining Specifications, *ACM Toplas 17, 3 (May, 1995) 507-534*

Achterberg / Heinz / Koch 2008
T. Achterberg, S. Heinz, T. Koch: Counting solutions of integer programs using unrestricted subtree detection: The Fifth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems CP-AI-OR 2008, http://opus.kobv.de/zib/volltexte/2008/1092

Achterberg / Wedler / Brinkmann 2008
Tobias Achterberg, Markus Wedler, Raik Brinkmann: Property Checking with Constraint Integer Programming, ICCAD 2008

AHB 1999
AMBA™ Specification Rev 2.0, ARM, Document number ARM IHI 0011A, 1999

Bailey 2008
Constrained random test struggles to live up to promises, www.scdsource.com, March 2008.

Bailey et al. 2007
Brian Bailey, Grant Martin, Andrew Piziali: ESL Design and Verification, Elsevier 2007.

Bailey 2007
B. Bailey: The Great EDA Cover-up http://electronicsystemlevel.com/EDA-Cover-up.pdf, 2007

Bergeron et al 2005
Janick Bergeron, Eduard Cerny, Alan Hunter, Andrew Nightingale: Verification Methodology Manual for System Verilog, Springer 2005.

Beizer 1990
B. Beizer: Software Testing Techniques 2nd edition, International Thomson Publishing, 1990

Beuer 2005
T. Beuer: Semiformales Property Checking mit GateProp/GateMon ITL und Modelsim PSL: Vergleich zweier Methoden zur Verifikation digitaler Systeme, Hagenberg 2005.

Beyer / Bormann 2008:
J. Bormann, S. Beyer: Erfindungsmeldung an OneSpin Solutions zur kompositionalen vollständigen Verifikation, 2008

Beyer 2008:
S. Beyer: Completeness Checker overlooks gap. Personal communication.

Beyer 2005
S. Beyer: Putting it all together – Formal Verification of the VAMP, Dissertation, Saarland University, Saarbrücken, 2005

Biere 1999
A. Biere, A. Cimatti, E. Clarke: Symbolic model checking without BDDs, TACAS, 1999

Bombana et al. 1995:
M. Bombana, P. Cavalloro, S. Conigliaro, R. B. Hughes, G. Musgrave, G. Zaza: Design-Flow and Synthesis for ASICs: A Case Study, DAC, 1995

Börger / Stärk 2003
E. Börger and R. Stärk, Abstract State Machines: A Method for High-Level System Design and Analysis, Springer-Verlag, 2003

Bormann 2007:
SVOPs – Slide show to define operation properties in SVA 2007. OneSpin Company confidential.

Bormann et al. 2007:
J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, F. Bruno: Complete Formal Verification of TriCore2 and Other Processors, DVCon 2007

Bormann / Beyer / Skalberg 2006:
J. Bormann, S. Beyer, S. Skalberg: Equivalence Verification between Transaction Level Models and RTL at the Example of Processors, European Patent Application, issued Dec. 22nd, 2006, Publication number EP1933245

Bormann / Blank / Winkelmann 2005
J. Bormann, C. Blank, K. Winkelmann: Technical and Managerial Data About Property Checking With Complete Functional Coverage, Best Paper Candidate, Euro DesignCon, München 2005

Bormann / Busch 2005:
J. Bormann, H. Busch: Verfahren zur Bestimmung der Güte einer Menge von Eigenschaften, verwendbar zur Verifikation und Spezifikation von Schaltungen (Method for determining the quality of a set of properties, applicable for the verification and specification of circuits). European Patent Application issued 2005. Publication number EP1764715.

Bormann / Winkelmann 2003
J. Bormann, K. Winkelmann: Verfahren und Vorrichtung zum Erstellen eines Modells einer Schaltung zur formalen Verifikation. (Method and Tool to Create a Circuit Model for Formal Verification) German Patent DE10325513

Bormann 2003:
J. Bormann: Productivity Figures for Complete Formal Block Verification, edacentrum, DATE 2003

Bormann 2003B
J. Bormann: GateProp-Spezifikation, 2003

Bormann 2001

J. Bormann: Formale Verifikation wird zum Handwerk (Formal verification becomes a craft): GI/ITG/GMM-Workshop“Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", Meissen, 2001

Bormann / Spalinger 2001
J. Bormann, C. Spalinger: Formale Verifikation für Nicht-Formalisten (Formal Verification for Non Formalists) in: IT+TI 1/2001, Oldenbourg Verlag.

Bormann 2000
J. Bormann: From Art to Craft:: Formal Methods in Industrial ASIC Projects. Medea Conference, Munich 2000.

Bormann / Warkentin 1999
J. Bormann und P. Warkentin: Verfahren zum Vergleich elektrischer Schaltungen (Method for the comparison of electric circuits). European Patent 1068580, issued 1999

Bormann et al. 1995
J. Bormann, J.Lohse, M. Payer and G.Venzl: Model Checking in Industrial Hardware Design, DAC 1995.

Brand 199
D. Brand: Verification of Large Synthesized Designs, Proc. IEEE International Conference on Computer-Aided Design (ICCAD), Santa Clara, pp. 534-537, Nov. 1993.

Brinkmann 2003
Preprocessing for Property Checking of Sequential Circuits on the Register Transfer Level, Dissertation, Kaiserslautern 2003

Broy / Rumpe 2007
M. Broy, B. Rumpe: Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. Informatik-Spektrum. Springer Verlag. Band 30, Heft 1, Februar 2007.

Broy 1994:
M. Broy: A Functional Rephrasing of the Assumption/Commitment Specification Style Technical Report number TUM-I9417, Technische Universität München, 1994

Buckow / Bormann 1993
O. Buckow, J. Bormann. Formale Spezifikation und Verifikation eines SPARC-kompatiblen Prozessors mit einem interaktiven Beweissystem (Formal Specification and Verification of a SPARC Compatible Processor with an Interactive Theorem Prover), Diplomarbeit & Siemens-Report, Munich 1993.

Busch 2005:
H. Busch: TPROP - Checking Completeness and other Meta Properties, internal report, OneSpin and Infineon confidential, 2005

Busch 1991
H. Busch: Hardware Design by Proven Transformations, Dissertation, Brunel University, 1991

Büttner 2007

Wolfram Büttner: Functional Verification of IP: Quo Vadis? IP07, Keynote Talk, Grenoble 2007

Büttner / Siegel 2007
Wolfram Büttner, Michael Siegel: 'Achieving completeness in IP functional verification' posted on Feb. 12, 2007, at http://www.eetimes.com/showArticle.jhtml?articleID=197005268

Büttner 2005
Wolfram Büttner: Is Formal Verification Bound to Remain a Junior Partner of Simulation? CHARME 2005:1

Cai / Gajski 2003
Lukai Cai, Daniel Gajski: Transaction Level Modeling in System Level Design
Technical Report 2-10, Center for Embedded Computer Systems, University of California, Irvine, 2003

Carter / Hemmady 2007
Hamilton B. Carter, Shankar G. Hemmady: Metric Driven Design Verification, Springer 2007.

Case / Mishchenko / Brayton 2006
M. L.Case, A. Mishchenko, and R. K. Brayton: Inductively finding a reachable state space over-approximation, Proc. IWLS '06, pp. 172-179.

Certess undated
Certitude Data Sheet – www.certess.com/product/certitude_datasheet.pdf

Chauhan et al. 2002
P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang: Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis, in Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD), 2002.

Claessen 2006
Koen Lindström Claessen - A Coverage Analysis for Safety Property Lists - Workshop on Designing Correct Circuits (DCC), Vienna March 25-26, 2006

Claessen 2007
Koen Claessen - A Coverage Analysis for Safety Property Lists - FMCAD, November 2007.

Clarke / Emerson / Sistla 1986
Clarke, E. M.; Emerson, E. A.; Sistla, A. P., "Automatic verification of finite-state concurrent systems using temporal logic specifications", ACM Transactions on Programming Languages and Systems 8: 244, 1986

Clarke et al. 2000
E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith: Counterexample-guided abstraction refinement in Proc. International Conference on Computer Aided Verification (CAV), 2000, pp. 154–169.

Conquest undated
Conquest – Advanced Verification Environment, Data Sheet, Real Intent

Cuéllar / Huber 1995
Jorge Cuéllar, Martin Huber: TLT. Formal Development of Reactive Systems 1995: 151-169

Dempster / Stuart 2001
D. Dempster, M. Stuart: "Verification Methodology Manual - Techniques for Verifying HDL Designs", Teamwork International, 2nd edition, 2001;

Drechsler / Hoereth 2002
R. Drechsler, S. Hoereth: Gatecomp: Equivalence Checking of Digital Circuits in an Industrial Environment, International Workshop on Boolean Problems, p. 195, Freiberg, 2002

Een / Sörensson 2003
N. Een, N. Sörensson: An Extensible SAT-Solver, SAT 2003.

EN61508 2001
Europäische Norm 61508: Funktionale Sicherheit elektrischer / elektronischer / programmierbarer elektronischer Systeme, siehe insbesondere Teil 2, 2001

Filkorn 1992
T. Filkorn: Symbolische Methoden für die Verifikation endlicher Zustandssysteme, Dissertation, München 1992

Foster et al 2003:
H. foster, A. Krolnik, D. Lacey: Assertion-Based Design, Kluwer Academic Publishers, 2003

Foster et al 2007
H. Foster, Lawrence Hoh, Bahman Rabii, Vigyan Singhal: Guidelines for Creating a Formal Verification Testplan, DVCon 2007

Foster / Krolnik 2008:
Foster, Krolnik: Creating Assertion Based IP, Springer 2008

Freibothe 2009:
Martin Freibothe: "Ein Ansatz für die verifikationsgerechte Verhaltensmodellierung für die semi-formale Verifikation von Mixed-Signal-Schaltungen", Dissertation, TU-Dresden 2009

Ganai / Gupta 2007:
SAT-based Scalable Formal Verification Solutions, Springer 2007.

Goldberg / Novikov 2002:
E. Goldberg, Y. Novikov: BerkMin: A Fast and Robust Sat-Solver. DATE 2002

Grosse / Hampton 2005
Grosse, Joerg and Hampton, Mark: Method for Evaluating the Quality of a Test Program, European Patent Application, publication number EP1771799, Patent applied 2005.

Große et al. 2008
D. Große, U. Kühne, R. Drechsler: **Analyzing Functional Coverage in Bounded Model Checking:** IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Volume 27, Number 7, pp. 1305-1314, July 2008

Gupta et al. 2003
A. Gupta, M. Ganai, Z. Yang, and P. Ashar: Iterative abstraction using SAT-based BMC with proof analysis, in Proc. International Conference on Computer-Aided Design (ICCAD). Washington, DC, USA: IEEE Computer Society, 2003, p. 416.

Gurevich 1995
Y. Gurevich, Evolving Algebras 1993: Lipari Guide, E. Börger (ed.), Specification and Validation Methods, Oxford University Press, 1995

Hirsch 1967
I. N. Hirsch: MEMMAP/360. Report TR-P 1168, IBM Systems Development Division, Product Test Laboratories, 1967, (zitiert nach [Beizer 1990])

Hoereth / Müller-Brahms / Rudlof 2002:
S. Hoereth, M. Müller-Brahms, T. Rudlof: Method and Device for Verifying Digital Circuits, Europäisches Patent EP1546949, eingereicht 2002

Hoereth 2003:
S. Hoereth: Method and apparatus for determining the minimum or maximum switching activity of a digital circuit, Amerikanisches Patent 6960930, eingereicht 2003

Hojati 2003
Hojati, R.: "Determining Verification Coverage Using Circuit Properties" US Patent 6594804, granted July 15th, 2003;

Hoskote et al. 1999
Hoskote, Kam, Ho, and Zhao: "Coverage Estimation for Symbolic Model Checking" Proc. of 36th Design Automation Conference, 1999;

IFV 2005
Incisive Formal Verifier, Data Sheet, Cadence 2005

Jain et al. 2008:
H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke: Word-level predicate-abstraction and refinement techniques for verifying RTL Verilog, IEEE Transactions on Computer-Aided Design, vol. 27, no. 2, pp. 366–379, 2008.

Sheeran / Singh / Stalmarck 2000
M. Sheeran, S. Singh, and G. Stalmarck: Checking safety properties using induction and a SAT-solver, in Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD), 2000.

Jain et al. 2007
Alok Jain, J. Baumgarnter, R. S. Mitra, P. Dasgupta: Formal Assertion-Based Verification in Industrial Setting, Slide Show of Tutorial at DAC 2007

Jasper 2007
Jasper Gold Verification System Familiy v4.3, Data Sheet, Jasper 2007

Katz et al 1999

S. Katz, O. Grumberg, D. Geist: Have I written Enough Properties? CHARME 1999

Kranen 2008
Kathryn Kranen, Time to reconcile the design / verification divorce. www.scdsource.com, May 2008

Kroening / Seshia 2007
D. Kroening, S. A. Seshia: Formal verification at higher levels of abstraction, ICCAD 2007.

Kuehlmann / Krohm 1997
A. Kuehlmann, F. Krohm: Abstract Equivalence Checking Using Cuts and Heaps, DAC 1997

Kühne / Beyer / Pichler 2008
Paper über Prozessorsimulation. Internal report at OneSpin Solutions.

Kunz 1993
W. Kunz: HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning, Proc. 1993 ACM / IEEE International Conference on Computer-Aided Design (ICCAD), pp. 538 - 543, Santa Clara, November 1993

Lamport 1994
L. Lamport: Introduction to TLA, http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-1994-001.htm, 1994

Lohse / Warkentin 1998
J. Lohse, P. Warkentin: Method and Device for Comparing Technical Systems With Each Other, European Patent application, Publication number WO0026825, application issued 1998

Lohse / Warkentin 2001
J. Lohse, P. Warkentin: Method and configuration for comparing a first characteristic with predetermined characteristics of a technical system, US patent 6581026, filed 2001

Loitz et al. 2008
S. Loitz, M. Wedler, C. Brehm, T. Vogt, N. Wehn, W. Kunz: **Proving Functional Correctness of Weakly Programmable IPs - A Case Study with Formal Property Checking, SASP 2008**

Magellan undated
Magellan – Hybrid RTL Formal Verification, Data Sheet, Synopsys undated

Malik et. al 2001
S. Malik, M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, Chaff: Engineering an Efficient SAT Solver, DAC 2001

Marques-Silva / Sakallah 1996
J. P. Marques-Silva, K. A. Sakallah: GRASP: A New Search Algorithm for Satisfiability. In Proceedings of International Conference on Computer-Aided Design, pp. 220-227, 1996

Mayger / Harris 1991
E. Mayger, R. L. Harris: User Guide for LAMBDA version 4.1, 1991

McMillan / Amla 2003
K. L. McMillan and N. Amla: Automatic abstraction without counterexamples, in Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2003, pp. 2–17

McMillan 1992
Symbolic Model Checking, Dissertation, CMU, Pittsburgh 1992

Meredith 2004
M. Meredith: A look into Behavioral Synthesis, 2004
http://www.eetimes.com/news/design/showArticle.jhtml?articleID=18900783

Mitra 2008
Mitra, R.: Strategies for Mainstream Usage of Formal Verification, Design Automation Conference 2008

Moore 1965
G. E. Moore, Cramming more components onto integrated circuits, Electronics, Volume 38, Number 8, April 1965

Nguyen et al. 2008:
M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, W. Kunz: Unbounded Protocol Compliance Verification Using Interval Property Checking With Invariants: IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol. 27, issue 11, Nov. 2008, p 2068-2082.

Nguyen et al. 2005
Minh D. Nguyen, Dominik Stoffel, Wolfgang Kunz: Enhancing BMC-based Protocol Verification Using Transition-By-Transition FSM Traversal. GI Jahrestagung (1) 2005: 303-307

Nguyen et al. 2005a
Minh D. Nguyen, Dominik Stoffel, Markus Wedler, Wolfgang Kunz: Transition-by-transition FSM traversal for reachability analysis in bounded model checking. ICCAD 2005: 1068-1075

Novikov / Brinkmann 2005
Y. Novikov and R. Brinkmann: Foundations of Hierarchical SAT Solving. ZIB-Report 05-38 (August 2005), Zuse Institut Berlin, 2005

Novikov 2003
Y. Novikov: Local Search for Boolean Relations on the Basis of Unit Propagation. DATE 2003.

Novikov / Goldberg 2001
Y. Novikov and E. Goldberg: An efficient learning procedure for multiple implication checks. DATE 2001.

OneSpin undated
Data Sheet for OneSpin 360 MV, OneSpin Solutions GmbH, www.onespin-solutions.com/download.php

Offutt / Untch 2000

Offutt, Jeff and Untch, Roland: Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, 45-55, San Jose, CA, October 2000.

Paucke 2003
Paucke, Marc: Formal Specification of Properties using Symbolic Timing Diagrams for Bounded Model Checking (gemeinsam mit der Infineon AG München) Diplomarbeit an AG Softwaretechnik des Instituts für Verteilte Systeme an der Otto-von-Guericke-Universität, Magdeburg, 2003

Peukert et al. 2001
R.Peukert, K. Henke, H.-D. Wuttke: A Graphical Input Method for Model Checking. Technical Report, TU Ilmenau, 2001.

PPv2 2008
Infineon Protocol Processor, Success Story, http://www.onespin-solutions.com/download.php, 2008

PSL 2004
Property Specification Language Reference Manual, Version 1.1, Accellera, 2004

Schickel et al. 2007
M. Schickel, V. Nimbler, M. Braun, Hans Eveking: CandoGen – A Property-Based Model Generator. University Booth at DATE 2007, Nice, France, 2007.

Schickel et al. 2006
M. Schickel, V. Nimbler, M. Braun, H. Eveking: An Efficient Synthesis Method for Property-Based Design in Formal Verification. In: Sorin Huss (Ed.): Advances in Design and Specification Languages for Embedded Systems, p. 163-182, Kluwer Acad. Publishers, Boston/Dordrecht/London, 2007

Schlör 2001
R. Schlör: Symbolic Timing Diagrams: A Visual Formalism for Model Verification, Dissertation, Carl-von-Ossietzky-Universität, Oldenburg, 2001.

Schönherr et al 2008:
J. Schönherr, M. Freibothe, B. Straube, J. Bormann. Semi-formal verification of the steady state behavior of mixed-signal circuits by SAT-based property checking: Theoretical Computer Science, Elsevier Science Publishers Ltd., 2008, 404, 293-307

Shimizu et al. 2006
Shimizu, Gupta, Koyama, Omizo, Abdulhafiz, McConville, Swanson: Verification of the Cell Broadband EngineTM Processor: DAC 2006

Stoffel et al. 2004
Dominik Stoffel, Markus Wedler, Peter Warkentin, Wolfgang Kunz: Structural FSM traversal. IEEE Trans. on CAD of Integrated Circuits and Systems 23(5): 598-619 (2004)

Stuart / Dempster 2000:
Verification Methodology Manual – For Code Coverage in HDL Designs, Teamwork International, Yateley, 2000.

Siegel et al. 1999
Michael Siegel, J. Bormann, H.Busch, M. Turpin: Definition of the Interval Temporal Logic (ITL), personal communication, 1999

Solidify undated
Solidify Data Sheet, Averant

System Verilog 2005
IEEE Standard for System Verilog – Unified Hardware Design, Specification, and Verification Language, IEEE Computer Society, 2005

Turpin 2003
M. Turpin: The Dangers of Living with an X (bugs hidden in your Verilog): Synopsys Users Group, Boston 2003

Tarisan / Keutzer 2001:
S. Tarisan, K. Keutzer: "Coverage Metrics for Functional Validation of Hardware Designs", IEEE Design & Test of Computers. 2001.

Thomas et al. 2004:
A. Thomas, J. Becker, U. Heinkel, K. Winkelmann, J. Bormann: Formale Verifikation eines Sonet/SDH Framers. (Formal Verification of a Sonet/SDH framer) GI/ITG/GMM-Workshop “Methoden und Beschreibungssprachen zur Modellierung und Verifikation von SChaltungen und Systemen, Kaiserslautern, 2004

Wang et al. 2006
C. Wang, B. Li, H. Jin, G. Hachtel, and F. Somenzi: Improving Ariadne’s bundle by following multiple threads in abstraction refinement, IEEE Transactions on Computer-Aided Design, vol. 25, pp. 2297 – 2316, 2006.

Wedler et al. 2007
Markus Wedler, Dominik Stoffel, Raik Brinkmann, Wolfgang Kunz: A Normalization Method for Arithmetic Data-Path Verification. IEEE Trans. on CAD of Integrated Circuits and Systems 26(11): 1909-1922 (2007)

Wedler et al. 2005
Markus Wedler, Dominik Stoffel, Wolfgang Kunz: Normalization at the arithmetic bit level. DAC 2005: 457-462

Wedler et al. 2004:
Markus Wedler, Dominik Stoffel, Wolfgang Kunz: Exploiting state encoding for invariant generation in induction-based property checking. ASP-DAC 2004: 424-429

Wedler et al. 2003:
Markus Wedler, Dominik Stoffel, Wolfgang Kunz: Using RTL Statespace Information and State Encoding for Induction Based Property Checking. DATE 2003: 11156-11157

Winkelmann et al. 2004:
Klaus Winkelmann, Hans-Joachim Trylus, Dominik Stoffel, Görschwin Fey: Cost-Efficient Block Verification for a UMTS Up-Link Chip-Rate Coprocessor. DATE 2004: 162-167

Wishbone 2002
WISHBONE, Revision B.3 Specification, 2002
http://www.opencores.org/projects.cgi/web/wishbone/wishbone

0-in 2005
Assertion Based Verification, Datasheet, Mentor 2005