ROYAL INSTITUTE
OF TECHNOLOGY

System Level Techniques for Verification and Synchronization after Local Design Refinements

Tarvo Raudvere

Stockholm 2007

Thesis submitted to the Royal Institute of Technology in partial fulfillment of the requirements for the degree of Doctor of Technology

Raudvere, Tarvo
System Level Techniques for Verification and Synchronization after Local Design Refinements

ISBN 978-91-7178-677-7
TRITA-ICT/ECS AVH 07:05
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-07/05--SE

Royal Institute of Technology
School of Information and Communication Technology
Department of Electronic, Computer, and Software Systems
Electrum 229
S-164 40 Kista, Sweden

Abstract

Today's advanced digital devices are enormously complex and incorporate many functions. In order to capture the system functionality and to be able to analyze the needs for a final implementation more efficiently, the entry point of the system development process is pushed to a higher level of abstraction. System level design methodologies describe the initial system model without considering lower level implementation details and the objective of the design development process is to introduce lower level details through design refinement.

In practice this kind of refinement process may entail non-semantic-preserving changes in the system description, and introduce new behaviors in the system functionality. In spite of new behaviors, a model formed by the refinement may still satisfy the design constraints and to realize the expected system. Due to the size of the involved models and the huge abstraction gap, the direct verification of a detailed implementation model against the abstract system model is quite impossible. However, the verification task can be considerably simplified, if each refinement step and its local implications are verified separately. One main idea of the Formal System Design (ForSyDe) methodology is to break the design process into smaller refinement steps that can be individually understood, analyzed and verified.

The topic of this thesis is the verification of refinement steps in ForSyDe and similar methodologies. It proposes verification attributes attached to each non-semantic-preserving transformation. The attributes include critical properties that have to be preserved by transformations. Verification properties are defined as temporal logic expressions and the actual verification is done with the SMV model checker. The mapping rules of ForSyDe models to the SMV language are provided. In addition to properties, the verification attributes include abstraction techniques to reduce the size of the models and to make verification tractable. For computation refinements, the author defines the *polynomial abstraction technique*, that addresses verification of DSP applications at a high abstraction level. Due to the size of models, predefined properties target only the local correctness of refined design blocks and the global influence has to be examined separately. In order to compensate the influence of temporal refinements, the thesis provides two novel *synchronization techniques*. The proposed verification and synchronization techniques have been applied to relevant applications in the computation area and to communication protocols.

To my family

Acknowledgments

There are many colleagues and friends who have advised and helped me during my doctoral studies. I take the opportunity to say: “Thank You!”

I would like to thank my supervisors Axel Jantsch and Ingo Sander for their continuous support and encouragement over the years of my doctoral studies.

I am thankful to Axel for the offered Ph.D. position at KTH and for his expert advice. He has spent a lot of energy to create a good research environment, to define interesting research projects and to find financial support for them. In addition to research, the development and teaching the System Modeling course with Axel was not only a useful experience but also provided enjoyable hours in front of classroom.

It has been a real pleasure to work with Ingo. We have spent countless hours for research and private discussions. Even on very busy days he could find a moment to listen to my questions and to help if possible. Besides the great supervision, he invited me to Swedish rock bands’ concerts, which were really great.

I am thankful to Mads Dam and Dilian Gurov for their advice in my first years at KTH. I also want to express my gratitude to Andres Keevallik, my former supervisor of Diploma and Master’s thesis at Tallinn University of Technology, that he introduced to me the research world. Andres is the person who first suggested that I should continue my academic career with doctoral studies.

I say thanks to the ForSyDe group members Ashish Kumar Singh, Zhonghai Lu and Jun Zhu for interesting research discussion. Endless discussions with Ashish have been a source of many ideas which some of them have been published as conference papers. I also say thanks to the former and current members of the SAM group. I am grateful to Johnny Öberg who did voluntarily the proof reading of the thesis. I have had long discussions about sailing, GPS, fishing and home-brewing with office neighbors Erland

Nilsson and Raimo Haukilahti. Me and my roommate Iyad Al Khatib, we have spent patently incredible hours reading a book, which changed philosophy and, for example, discusses how to decide whether two things are identical or indistinguishable. It was relaxing to visit Gery Einberg and to chat a couple of hours on Friday evenings, despite that it took almost the same amount of time to reach his place.

Thanks to Hans Berggren, Richard Andersson and Peter Magnusson for keeping my computer and servers running, and Lena Beronius, Agneta Herling, Gunnar Johansson and Rita Fjällendahl for helping in all kinds of administrative questions.

I am grateful to Wolfgang Ecker for inviting me to work three months in the spring of 2005 at Infineon Technologies AG in Munich. I am thankful to Volkan Esen for arranging an apartment for me in Munich and for his friendship. I spent a really good time in Munich.

I send greetings to my Estonian friends with whom I have traveled in Europe in last summers and who have attended little song-parties in Laitse.

Finally, I would like to thank my family for their care and support. The greatest thanks go to my mother and father who have always encouraged and supported me and my sister to continue studies. I am very happy that I have my lovely wife Evelin and son Hans Oliver. I am very thankful to Evelin for her care, love and constant support, and for the patience when I was away in Stockholm. I also thank my parents-in-law for supporting my wife, when I could not. My sister Piret and her husband Ursel have regularly visited me in Stockholm and Munich. I feel glad that you pulled me out of my office and invited to opera and to discover historical sites.

The research was partially funded by the Swedish Research Council and the Swedish Foundation for Strategic Research.

Stockholm, August 2007
Tarvo Raudvere

Contents

List of Figures

Chapter 1

Introduction

Researchers in the electronic system design field face continuous challenge to cope with the growing design complexity of the future digital systems. To be competitive in the electronic segment means to release new products on the market with extended services as fast as possible. The growing design complexity from one side and the pressure to shorten the design time from the other side, create a demand for more efficient system design approaches. Since the design refinement from a native language written specification to a hardware and software implementation involves a large number of design decisions, the design development process becomes error-prone. A major part of the design time is indubitably spent for verification, to check whether the final product behaves expectedly or not.

Due to the enormous amount of functionalities that are mixed to the system description, it is not realistic to rely on a short time simulation and to decide if the system contains unexpected behaviors. Therefore, in addition to simulation, formal verification techniques have to be applied in the design process, as model checking is used in the present work. Although model checkers can prove the system correctness concerning a certain property, the time and memory demands scale-up exponentially with the size of the system in the worst case.

The refinement based formal design approach, where the system is developed to a final implementation by using a serial application of predefined design transformation, makes it possible to localize the modified design part and to concentrate on that part when checking whether the refined system part conflicts the expected behavior or not. The present work proposes a verification approach, where critical properties, which due to a particular refinement may become violated, are checked after every single non-semantic-preserving design transformation. In this case the system verification is divided into drastically smaller verification activities that are distributed over the entire system development process from the system level to the RT-level (Register Transfer Level), instead of the fairly impossible trial to verify a very detailed implementation directly against the abstract specification. In order to apply the described verification technique, a good model for system level design is needed.

1.1 System Level Design

Related to the progress in the semiconductors industry [96] that allows integrating more and more components on a single chip, the design complexity of electronic systems is continuously increasing. A single integrated circuit may contain digital and analog parts, several controllers, digital signal processors and microprocessors, which were available as separate components before. The market term for these systems is *System on Chip* (SoC), where all the system's components are integrated to a single chip. The terms MPSoC and NoC denote some complex concepts in the System on Chip design. MPSoC stands for multi-processor approaches [63] that address computation problems and NoC (Network on Chip) [66] targets interconnection issues on a chip. Although SoC platforms involve many advantages, the design process of these complex systems requires much more effort. Due to the enormous amount of different behaviors that system functionality contains, the system design process that transforms the desired functionality to hardware and software is very challenging.

In order to be more efficient in the design development process and in a better way to capture the system's functionality, the entry point of digital system design is moving to higher levels of abstraction. As complement to the former step from the gate level to the RT-level, the current step starts from the RT-level to reach the system level [76]. One of the main ideas of system level design is to avoid the common habit that initial hardware and software descriptions are given in different models using different languages. In order to put system designers to speak the same language [14] and to capture system level design needs, two languages, SystemC [7] from software side and SystemVerilog [88] from the hardware community, were developed. These languages aimed to describe the system's functionality in one model, at a higher level than usual hardware description languages normally do.

One of the motivations for having system level design is to replace a paper based design specification with a concrete high abstraction level system model that is referred to during the rest of the design process. This model expresses the system functionality without disturbance of lower level details, which could be specific to a certain design implementation. This approach allows exploring the system's functionality in an early design phase, before any design decision is taken. From verification point of view, this approach creates many opportunities that were not possible to apply before. Since a system description in high level languages describes hardware and software parts in the same model, the system verification can be done in an earlier design phase, where only one model is considered. Instead of the old scenario, where the design engineers provided hardware and software models to the verification engineers after the design phase, verification is becoming a part of to the design development process, starting from the first system level models.

The system mapping from the RT-level to the gate level is well defined and automated by commercial synthesis tools, but the refinement from the system level to the RT-level is still quite challenging. Obviously there is quite a huge abstraction gap between an RT-level model, which has to contain all implementation details that are required for synthesis, and the initial system level model, which describes only the ideal system functionality. The main task of the system level design process is to refine the system model by introducing lower level implementation details, and in such a way to reduce the abstraction gap between the initial system model and a synthesizable RT-level model. Since the same functionality can be implemented on several architectures, the goal of the refinement process is to analyze each intermediate refined model [55] and to project them to an optimal final implementation, which satisfies the required design constraints. Every refinement can be considered as the introduction of an implementation detail, reducing the abstraction gap and restricting the design space of possible final implementations, as illustrated in Figure 1.1.

The ultimate purpose is to derive an implementation model from the initial model by a step by step refinement process such that the relations between any two consecutive models are explicit. In addition, the same language should be used to describe all models. This makes the further verification drastically simpler, since, as pointed by Abdi and Gajski in [2], "checking equivalence of two independently written high level language programs is not feasible". Due to the large size of the system, different system

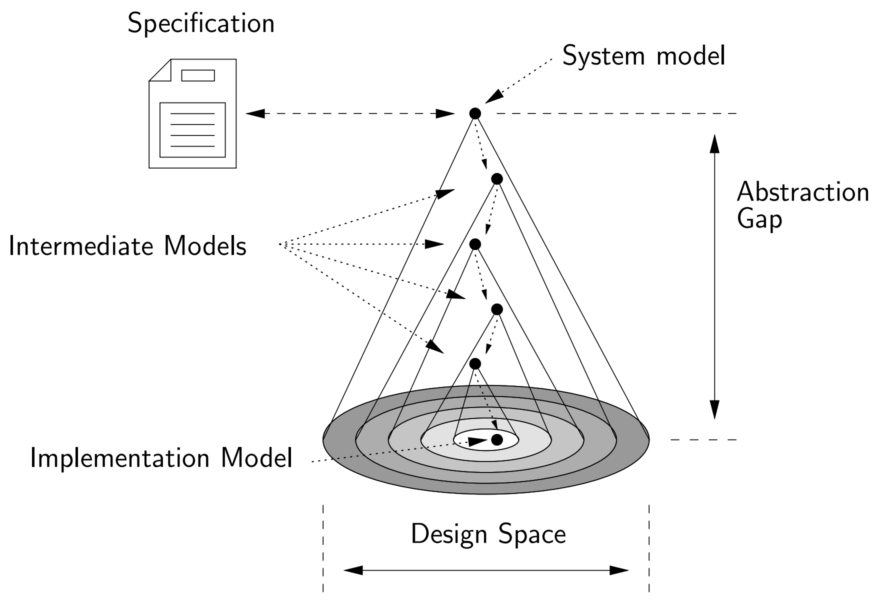


Figure 1.1. Step by step refinements decrease the abstraction gap and the available design space

blocks are developed by several design engineers. Therefore, in order to synchronize the design flow and to assist the reuse of design components, intermediate models at different abstraction levels have to correspond to certain standards, as for instance proposed in [58]. A standard captures the input/output interfaces of design blocks, the expected data rates, etc. The use of a common standard makes it easier to describe the constraints that design blocks have to satisfy, in a systematic way. Systematically defined constraints make it much simpler to verify after a design transformation whether the refined design block satisfies required properties or not.

1.1.1 Formal System Design

Due to the large size of the systems, an efficient design development process should not only start at a high abstraction level, but also involve formal methods, as pointed by Edwards [40]. In addition, formal verification has to be applied at a high abstraction level, and the final implementation has to be derived through correct-by-construction design refinements.

Formal system development is the objective of the ForSyDe [90] (Formal System Design) methodology. The design process in ForSyDe starts with the creation of a functional system model. The system model is defined by using formal semantics [84], which makes it suitable to apply formal methods within the further design development process. The initial model is deterministic and can be classified as a synchronous model of computation [56]. The main idea of synchronous models is to consider systems to produce output values synchronously with the input values [10], and to separate computation from communication. An alternative to the synchronous models are asynchronous models, where the communication delays may cause non-deterministic behaviors in computation blocks. One of the strengths of ForSyDe is the ideal view of the design in the system model, which is described in terms of unlimited computational resources. In addition to the zero communication and computation time, the model uses unlimited storage resources and data types with non-constraint bit-widths. The functional model with unlimited resources is a perfect description of a system to analyze and verify its functionality, without superfluous implementation specific details.

Obviously, there is quite a huge abstraction gap between the ideal system model and an RT-level model, which is suitable for hardware synthesis. In contrast to automatic [73] and semantic-preserving design refinements [95], the ForSyDe methodology addresses manual design refinement, where both semantic-preserving and non-semantic-preserving design transformations are used to develop an RT-level implementation model.

Design transformations are classified according to the following:

Semantic-preserving transformations do not change the meaning and the behavior of the model. Mainly, they are used to optimize the model for synthesis.

Non-semantic-preserving transformations change the meaning of the model. A good example of a non-semantic-preserving transformation is the refinement of a buffer, where the ideal unlimited buffer is replaced with a fixed-size buffer. While such a transformation definitely modifies the system's semantics, the refined model may still behave in the same way as the original model under the circumstances that no buffer overflow occurs.

The design development starting from the system model towards an implementation model is illustrated in Figure 1.2. The ForSyDe methodology provides predefined design transformations in the transformation library. In

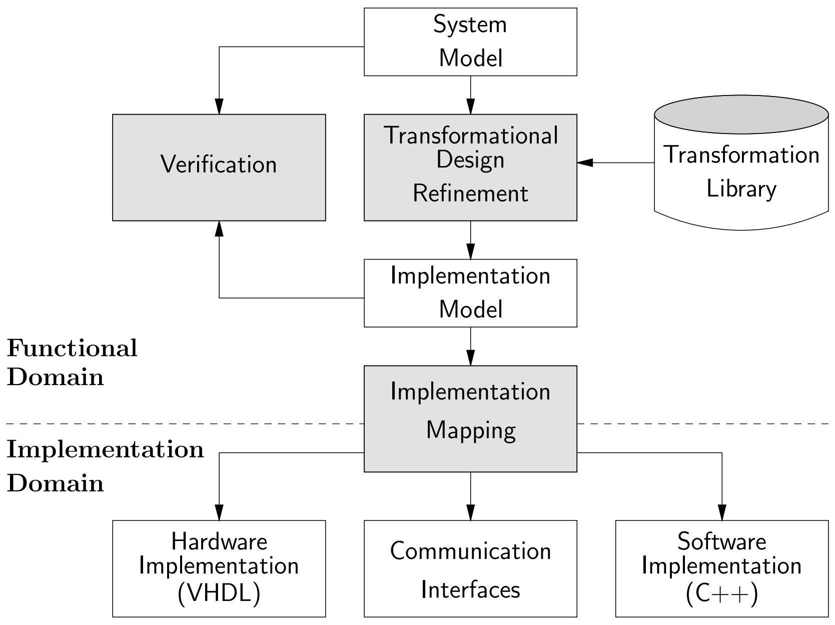


Figure 1.2. The ForSyDe design flow

order to refine the system model, the designer selects a proper design transformation rule from the library according to characteristics and required implications, and applies it to the model. The result of the transformation is an intermediate model, which is more detailed than the original model. The final, implementation model includes all necessary details, which are required for implementation mapping to software [59] (C++) and hardware [90] (VHDL).

1.1.2 Thesis Objective

The research in the context of the ForSyDe methodology was started in the late nineties. To date, clear and elegant concepts for system level modeling, formal design refinement, and implementation mapping have been established. One of the basic ideas, supporting the methodology, was to provide a good platform for design verification. Formerly, only simulation based techniques have been used to verify ForSyDe models, and formal methods have not been applied.

The simulation of models at a high abstraction level has a clear advantage in front of RT-level models. Today's RT-level models include an enormous number of behaviors and it is not possible to verify all of them with a restricted set of input vectors within a limited amount of time. On the other hand, an ideal model at a high level of abstraction do not capture implementation specific details and therefore the verification of the model with a smaller number of behaviors is much more tractable. An implementation model can be considered to be correct-by-construction as far only semantic-preserving transformations are involved to the system development, since they do not change the meaning of the model. However, in practice, a synthesizable RT-level model is not achievable without introducing lower level details by the application of non-semantic-preserving design transformations. Obviously, this kind of refinement process is potentially error-prone and it is not possible to verify by simulation whether an implementation model is equivalent to the initial system model and satisfies the design constraints. In addition, the abstraction gap between the system model and an implementation model is so huge that even by using formal methods it is extremely complex to verify these models directly against each other. The general objective of the thesis is to show how formal verification can be integrated into the design development process, following the design transformations based refinement steps.

1.2 Author's Contribution

In order to have an efficient and capable design verification process, the author has proposed to share verification activities over the entire transformational design refinement. To consider the least number of changes at every verification step, refined intermediate models have to be verified immediately after non-semantic-preserving transformations. Since the change by a single transformation is tiny compared to the whole design refinement, the verification of two consecutive intermediate models concentrates on a certain change and requires much less effort. The basic idea of the proposed verification approach is to associate non-semantic-preserving design transformations with a set of verification properties. According to the fundamental concept of ForSyDe, all design transformations used in the design development process, are predefined in the design library. Thus, it is possible to analyze every single transformation and to find in each of them a set of critical issues, which are relevant to check after the corresponding refinement. Verification properties are defined as temporal logic expressions and refined sub-blocks of intermediate models are checked locally against the properties by using formal verification tools. Throughout the thesis the model checking technique is used to perform formal verification. Hence, the verification properties are specified as CTL* expressions and the thesis gives general rules how to map ForSyDe models to the input language of the SMV [102] model checker.

Model checking is fully automatic and searches design faults exhaustively, but it is sensitive to the size of the model. In order to be more efficient, all unnecessary system behaviors have to be abstracted from the model before verification. In addition to the predefined properties, the design methodology has to provide abstraction techniques, which are accompanied with non-semantic-preserving transformations. The thesis describes the development of verification properties and data abstraction for communication channels and proposes the polynomial abstraction technique for verification of design blocks after computation refinements at a high abstraction level.

The proposed verification approach addresses only local properties of refined blocks since formal methods cannot handle today's large models without an ad hoc simplification. Although a refined design block can be considered to be correct after verification, the changes in the design block may influence the rest of the model. The global influence has to be explored by static analysis techniques. The thesis addresses timing changes by temporal refinements that locally increase the computation time in the refined design blocks. In order to preserve the system's correct functional behavior after temporal refinement, two synchronization techniques are proposed.

According to the Rugby coordinates [50] the thesis addresses refinements in three domains: computation, communication and time. Refinements in the fourth, data domain, are considered as a future research topic.

1.2.1 Author's Publications

1. T. Raudvere, I. Sander, A. K. Singh, D. Gurov, and A. Jantsch. The ForSyDe semantics. In *Proceedings of the Swedish System-on-Chip Conference (SSoCC'02)*, Falkenberg, Sweden, March 2002. [84]

   This paper introduces the operational semantics of the ForSyDe process constructors in the synchronous model of computation.

2. T. Raudvere, I. Sander, A. K. Singh, and A. Jantsch. Verification of design decisions in ForSyDe. In *Proceedings of CODES+ISSS'03*, Newport Beach, California, USA, October 2003. [85]

This paper presents how design blocks can be verified against predefined properties after non-semantic-preserving design transformations. In addition, the paper describes the mapping procedure of the ForSyDe constructions to the input language of the SMV model checker.

3. T. Raudvere, A. K. Singh, I. Sander, and A. Jantsch. Polynomial abstraction for verification of sequentially implemented combinational circuits. In *Proceedings of the conference on Design, automation and test in Europe (DATE'04)*, Paris, France, February 2004. [86]

   This paper introduces the fundamental concept of polynomial abstraction.

4. T. Raudvere, A. K. Singh, I. Sander, and A. Jantsch. System level verification of digital signal processing applications based on the polynomial abstraction technique. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'05)*, San Jose, California, USA, November 2005. [87]

   This paper addresses the verification of refined computation blocks with combinational functionality and presents a more detailed and extended version of the polynomial abstraction technique.

5. T. Raudvere, I. Sander, and A. Jantsch. A synchronization algorithm for local temporal refinements in perfectly synchronous models with nested feedback loops. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI'07)*, Stresa, Italy, March 2007. [83]

   This paper describes the problems caused by local temporal refinement and provides a synchronization algorithm to preserve the system's functionality in the global sense after local temporal refinements.

6. T. Raudvere, I. Sander, and A. Jantsch. Synchronization after design refinements with sensitive delay elements. In *Proceedings of CODES+ISSS'07*, Salzburg, Austria, October 2007. [82]

   The paper introduces the concept of sensitive delay elements used for synchronization after temporal refinements. Sensitive delay elements are special in the sense that they do not delay synchronization events. Transforming carefully selected ordinary delay elements to sensitive delay elements modifies the system so that a significantly smaller number of synchronization delay elements are required to synchronize the system after temporal design refinements.

7. T. Raudvere, I. Sander, and A. Jantsch. Application and verification of local non-semantic-preserving transformations in system design. *submitted to IEEE Transactions on Computer-Aided Design*, 2007. [81]

   The design development strategy, which incorporates non-semantic-preserving transformations, verification properties, abstraction techniques and synchronization techniques is described in this paper. The paper illustrates the refinement and verification in the context of the design of a digital audio equalizer.

The main ideas in all the listed publications are from the author of the thesis, as well as the problem formulations, solutions, algorithms, performed experiments and most parts of the manuscripts. At the same time discussions with the co-authors have been a source of many ideas. The co-authors, especially Axel Jantsch and Ingo Sander, have helped to treat problems in a more general context, to finalize the proposed ideas and to examine the manuscripts. Dilian Gurov and Ashish Kumar Singh have helped in operational semantics issues in [84]. The concept of the transformational design refinement in the ForSyDe methodology and many design transformations used in [81, 82, 83, 85, 87] are from Ingo Sander. The ForSyDe digital equalizer model, used throughout several case studies was also written by him. The proofs in [87] were written by Ashish Kumar Singh.

1.3 Thesis Layout

The related work and background information about design methodologies, verification and abstraction techniques, and synchronization methods are collected in Chapter 2. Chapter 3 gives an overview about the ForSyDe methodology and brings guidelines how to map ForSyDe models to the SMV language [85].

The following chapters can be grouped into two parts. The first part, Chapters 4, 5, and 6 address local verification issues in ForSyDe. In the second part, Chapters 7 and 8 target the synchronization after local temporal refinements in order to preserve the system's correctness in the global sense.

Chapter 4 describes the proposed verification approach [85], which follows the non-semantic-preserving refinements by providing property templates, abstraction techniques and stimuli generators, in order to verify critical details, which a refinement has changed in the model. Chapter 5 addresses communication refinements, by defining general verification properties for communication channels [85]. Chapter 6 elaborates on computation refinements at a high abstraction level, where ideal combinational functions are mapped to sequential implementations [86, 87].

Chapter 7 defines a system level synchronization algorithm, which restores the system's correct behavior after local temporal refinements [83]. The synchronization algorithm is extended in Chapter 8, by introducing sensitive delay elements that help to reduce the number of resources that are required to synchronize refined models [82].

Chapter 9 illustrates the design refinement, verification and synchronization process in the context of a digital audio equalizer [81]. Chapter 10 summarizes the thesis and gives proposals for future research.

Chapter 2

Background and Related Work

2.1 Verification Techniques

The design development process transforms the system specification into an implementation. Since the system specification describes only the system's functionality, there are several implementations that satisfy the specification. Although both the design specification and an implementation express the same functionality, due to the different abstraction levels, the number of details in the implementation is much higher than in the specification. For example, the functional specification does not contain information about computation delays, computation precision or buffer sizes, which again are present in an implementation. Thus a task for verification is to check if the implementation satisfies the design constraints on delays and precision, and behaves according to the expected functionality. In general, the verification process abstracts the implementation model to check whether it corresponds to the functionality defined in the specification that is opposite to the design refinement process, which stepwise adds implementation details to the specification.

The verification techniques can be divided into simulation based and formal verification techniques. Simulation based techniques are straightforward to check if for a given input assignment an implementation responds as described in the system specification. One possible way for simulation is to provide a set of input stimuli vectors and the expected output values against which the simulation results are compared. In many cases predefined stimuli vectors are created to address some narrow set of design faults. Although these vectors work well for dedicated faults, vectors that are generated in a pseudo random fashion may be much more efficient to find arbitrary design faults. In the latter case the same randomly generated vectors are applied in parallel to the implementation and the system specification to compare respective output values. In order to keep track which parts of the design are executed and how many times, verification coverage metrics [51] are used. Based on the coverage metrics specific vectors can be generated, which target uncovered design parts [43]. The design simulation is not limited to the stimuli and response pairs, and may include monitors that observe internal system behavior. In hardware description languages the monitors are described in the assertion format, expressing the expected relations of internal signals. If an assertion is not satisfied, the monitor reacts by reporting an error message in a log file or stopping the simulations. The main advantage of monitors is that the system behavior can be observed in parallel at a large number of points and at every input stimuli vector. Looking at the other point of view, the monitor assertions can be considered as properties that the system has to satisfy [98]. For formal verification, monitors can be treated as a part of system specification that property checkers examine. For example, in [45] a high level abstract system description is verified against properties written in PSL (Property Specification Language) [4] by using model checking, and for simulation both the model and the PSL properties are translated to SystemC.

Due to the high complexity of today's systems and the shortage of the available time for verification, exhaustive simulation is not possible [20]. As complement to simulation, formal verification methods can be applied, which use sophisticated mathematical methods for analyzing the computation paths of the system in order to find mismatches between two different models or between the system specification and an implementation. Instead of assigning input stimuli vectors to the model, formal methods try to find if there exists an input vector or a sequence of vectors, which cause different behaviors in two system description. Although the quality of these techniques is comparable with exhaustive simulation, they can verify only relatively small models. At the same time, it is possible to check that a block of a larger system behaves correctly. Also, one can verify the system behavior on certain conditions, which may touch only a small part of the entire system. Two main classes of formal verification techniques are equivalence checking and property checking.

Equivalence checking is used to determine if two circuits are functionally equivalent, i.e., for any input stimuli the output values of the circuits are always pairwise the same. The method is widely used for verification of Boolean circuits, where BDDs (Binary Decision Diagram) [22] and SAT (Boolean satisfiability) based techniques [62] are applied. Various novel approaches are additionally used, in order to reduce the complexity of the problem. Examples are decomposition of a larger circuit into smaller blocks [13], the use of test generation [21] and random simulation [54], or a combination of the mentioned techniques in elegant ways [25, 71].

Two well known **property checking** techniques are theorem proving and model checking. **Theorem proving** [36, 44] is a formal verification technique where *axioms* and *inference rules* are used in order to prove mathematically that a model satisfies its specification. Compared with model checking the technique has a strong advantage - it can be used for the verification of a system with infinite state space, because the system correctness is proven through formal deduction, instead of applying state space exploration. Also the method can be applied at different abstraction levels. On the other hand, it demands good knowledge and ingenuity to compose a mathematical proof. In order to assist the designer to write specifications and construct proofs supporting tools can be used. For example PVS (Prototype Verification System) [69] is a popular theorem prover.

Model checking [34, 33] is a formal technique for verification of sequential circuits and communication protocols. A model is expressed as a state-transition graph and its correctness is verified against a specification, which is defined in a temporal logic, for example in CTL* (Computation Tree Logic) [31]. An exhaustive state space exploration procedure is used to decide the correctness of the model. The procedure can be fully automated and this makes the technique easier to use compared to theorem proving. In addition to the fully automatic verification, model checkers provide counter examples if the model does not satisfy the specification. A counter example is a trace of transitions from the initial state towards a state where the given specification is violated. This feature is very important in order to assist the designer to find what caused the negative answer. Due to the complex data structures in software models and the sake that model checking can be applied only to a model with a finite state space, it has been mostly used in hardware verification. However, newer model checkers also target software verification problems [29].

The idea to represent transition relations with binary decision diagrams (BDD) [22] was a major step towards the verification of realistic complex designs. The model checking algorithms, which uses the latter type of the presentation of transition relation is named symbolic model checking [23]. SMV (Symbolic Model Verifier) is a model checker that is based on this approach. The Cadence version of SMV [102] is used throughout the thesis as a tool for verification of ForSyDe models. SMV is introduced in Section 2.3. Other popular model checkers are the SPIN tool [49], which is used for verification of asynchronous and distributed systems and communication protocols, and UPPAAL [9] that is developed for verification of systems represented as a timed automata. NuSMV [68] is a reimplementation of SMV, which additionally supports bounded model checking [17]. In bounded model checking only bounded lengths of computation paths are checked, in order to find counter examples. The Mentor Graphics 0-IN formal verification tool [65] combines model checking with simulation. Instead of starting model checking from the initial state, it uses simulation to find critical and more interesting states (for example states, where buffers become full) and starts bounded model checking from those states.

Since model checking suffers from the state space explosion problem, various **abstraction techniques** are proposed for simplification of model checking tasks. Clarke et. al. present in [32] a methodology named *counter example guided abstraction refinement*. In this work an initial abstract model is generated by an automatic analysis of the control structure of the design. If the model checking of the initial abstract model gives an erroneous counter example then iterative refinements will be done according to the counter examples until a valid abstraction is found. Although the proposed technique is fully automated the problem to find a refinement is NP-hard and may not be suitable for larger designs.

The idea of uninterpreted function symbols [24] is used to simplify the verification task. In [12] an out-of-order processor is verified by using model checking and uninterpreted functions. In this treatment the use of symbolic values and instructions allows to show the correctness of the machine, independently of the actual instruction set architecture and the implementation of the functional units. Since this technique is based partly on theorem proving, it needs a remarkable amount of designer contribution. In [6] the verification of optimized microprocessors is addressed by combining counter example guided abstraction refinement and uninterpreted data path functions.

McMillan presents a methodology for system-level hardware verification based on compositional model checking [64]. This methodology relies on a technique of *circular compositional proof*, which allows to assume the correctness of some components when verifying other and vice versa. Although the state space of the model can be reduced significantly, the designer has to construct a complex proof.

Hojati and Brayton [48] describe a methodology for integer combinational/sequential systems. According to the notion of data independence they separate a design into control part and data path. All the data path variables are replaced with the binary variables. The system verification is performed using language containment.

Spatial abstraction combines this idea with the interval propagation theory [47]. The tool ADAbT [72] is an implementation for automatic spatial abstraction and verification of VHDL models. Through this technique the bit widths of data path storage elements can be reduced so that all the possible behaviors of the control part are preserved. The variables of the design can be classified as *data*, *control* or *mixed*. A variable, which does not determine the control flow, is classified as a data variable. All other variables are control or mixed. A variable that belongs to a loop or a branch construct is a control variable and all the variables, which take place in evaluation of control or mixed variables are mixed. Control variables, and mixed variables, which also are input variables, are irreducible. Data variables are initialized with one bit variables. Through interval propagation [47] their bit widths are calculated until a fix-point or the actual range of the variable is found.

The disadvantages of the method are: (1) the domains of all variables must be determined before abstraction, (2) the abstract model is valid only for verification of the properties related to the control part and interconnecting signals between data path and control part. Unfortunately the data properties, which refers to the values on the system input and output ports to express the functionality as described in the specification are not verifiable if the design includes multiplication and division operations. The polynomial abstraction technique in Chapter 6 has some advantages compared to the spatial abstraction technique. Polynomial abstraction allows to verify a refined model, based on the model input/output functionality, against the system specification even if multiplication and division operations are used. In addition, the domains of data signals in the design may be undetermined, i.e., the polynomial abstraction is applicable at a higher level of abstraction.

Polynomial methods have been applied for verification also at lower abstraction levels. In [101] a method for component reuse based on matching an arithmetic specification with bit-level implementations is introduced. They derive word-level polynomials for both specification $S(x)$ and implementation $R(x)$ and equivalence checking is performed through comparing the coefficients of the polynomials or quantifying the difference $D(x)$, where $D(x) = S(x) - R(x)$.

Two polynomial functions with different degrees and coefficients may compute the same result in a limited range of values, which is defined by the word length of variables. The equivalence check of this kind of polynomial arithmetic functions implemented on multiple word length data paths is addressed in [97]. The polynomial abstraction technique targets design descriptions where the word sizes of an implementation are not yet specified.

Compared to the polynomial abstraction where proper ranges of variables for verification are found according to their degrees, in [77] Pnueli et. al. describe a method to analyze the structure of an equality formula, which describes the source and target models in terms of uninterpreted functions. Based on the analysis, they determine the ranges of the variables in the formula to check whether the formula is satisfied or not. Similarly to polynomial abstraction, it is possible to verify systems at an abstraction level, where the domains of variables are unspecified. However, their approach is sensitive to arithmetic optimization and thus may not find that a refined model corresponds to the specification.

A method for verification of combinational circuits, that consists of the functions $\{+, -, *\}$ is introduced in [89]. This study finds the order of a functional implementation and uses the simulation with a restricted set of input vectors to verify the system correctness. Compared to polynomial abstraction, they do not address sequential designs and rational functions.

2.2 The Computation Tree Logic (CTL*)

Properties of a finite state system as traces of transitions between states can be expressed through temporal logic. The Computation Tree Logic (CTL*) [31] is one of these temporal logics, which intuitively expresses the properties of computation trees. A computation tree is an infinite tree, which gives all the possible executions of the system as paths starting from the initial state. CTL* makes it possible to define various properties including liveness, fairness, safety and deadlock freedom, and can describe very complex relations of signals in terms of timing and values.

CTL* formulas contain *path quantifiers* and *temporal operators*. The former describe the branching structure of a computation path and the latter describe the properties of a path through the tree.

The path quantifiers are the following:

- **A** - for all the computation paths;
- **E** - for some computation path.

The basic temporal operators are the following:

- **G** (*globally* or *always*) - a property holds at every state along the computation path;
- **F** (*in the future* or *sometimes*) - a property holds at some state on the path;
- **X** (*next time*) - a property holds at the next state of the path;
- **U** (*until*) - this operator combines two properties. There exists a state where the second property holds and at every preceding state the first property holds.

Some examples of CTL* formulas with explanations are given in order to illustrate the construction of specifications in temporal logic:

- $\mathbf{EF}(Started \wedge \sim\ Finished)$ - there exists a state where a process is started but not finished;
- $\mathbf{AG}(Request \rightarrow \mathbf{AF} Acknowledge)$ - always every request will be eventually acknowledged;
- $\mathbf{AG}(Request \rightarrow \mathbf{A}(Request\ \mathbf{U}\ Acknowledge))$ - always if a request occurs then it will continuously hold until it is eventually acknowledged.

2.3 The SMV System

The Cadence version of the SMV model checker can be used for verification of synchronous or asynchronous concurrent finite state models against a specification given in the temporal logic CTL*. The verification of a property terminates either with a report *true*, if the model satisfies the specification, or with a counter example, which shows why the property does not hold. The SMV language allows a modular hierarchical description of a model, where a module can instantiate another module.

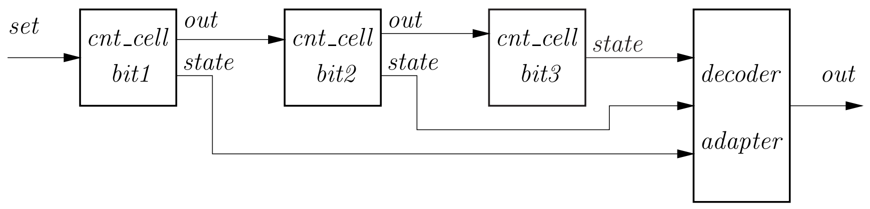


Figure 2.1. A three bit binary counter with decimal output

Let's consider the three bit counter with decimal output in Figure 2.1. The values of the binary input *set* are counted by three counter cells and the output values of the cells are decoded into decimal values by the decoder. The corresponding SMV program is given in Figure 2.2.

The SMV language provides finite data types - Boolean, enumerated and array with a fixed size. The first two lines of the code define new data types *Int_0_1* and *Int_0_7*. The former represents an enumerated definition, where symbolic values of the data type are given in curly braces. The domain of *Int_0_7* is defined through the lowest and the highest values. Although the integers are defined as enumerated, the SMV tool interprets them as integers and provides Boolean and arithmetic operations on them. The considered model defines three modules - *main*, *cnt_cell* and *decoder*. The module *main* is a component of every SMV program and has a special meaning similar to the main module in the C language. The module starts with the declaration of the variable *set* that has type *Int_0_1*. Since there is no assignment to the variable, the model checker can select a value from the domain of that variable in a non-deterministic manner at every execution cycle.

User defined modules are considered as variables. The module *cnt_cell* is instantiated three times and the module *decoder* once, as variables *bit1*, *bit2*, *bit3* and *adapter* respectively in the *main* module. The connections between components are described in these definitions. For instance, the signal *set* is connected with *bit1*, and *bit1* assigns a value to *bit2*. The expression *bit1.out* denotes the output *out* of the component *bit1*.

The model checker verifies the system against properties that are a part of the main module. Properties are called "specification" and their definitions start with *SPEC*. In this example two properties are defined. The first says that the system output gets always eventually the value 7, and the second one says that there exists a possibility that the output value will be 7. The former claim is false, because the input *set* may have always the value 0, and in this case the output value will never be 7.

```
typedef Int_0_1 {0,1};
typedef Int_0_7 0..7;

MODULE main(){
      set : Int_0_1;
      bit1 : cnt_cell(set);
      bit2 : cnt_cell(bit1.out);
      bit3 : cnt_cell(bit2.out);
      adapter : decoder(bit1.state, bit2.state, bit3.state);

      SPEC (AF (adapter.out = 7));
      SPEC (EF (adapter.out = 7));
}

MODULE decoder(i1, i2, i3){
      out : Int_0_7;
      out := 4 * i3 + 2 * i2 + i1;
}

MODULE cnt_cell(i1){
      out, state : Int_0_1;
      init(state) := 0;
      next(state) := case{
                           state = 0 : i1;
                           state = 1 & i1 = 0 : 1;
                           state = 1 & i1 = 1 : 0;
                           };
      out := case{
                  state = 1 & i1 = 1 : 1;
                  default : 0;
                  };
}
```

Figure 2.2. The three bit counter of Figure 2.1 described in the SMV language

The module *decoder* has three binary inputs (i_1, i_2, i_3) and one output (*out*) of type *Int_0_7*. The module consists of the declaration of the output variable *out*, and an arithmetic expression, which evaluates the output.

The last module, *cnt_cell* consists of two binary variables *state* and *out*. The special SMV operators *init* and *next* provide an opportunity to define sequential machines. For example, the initial value of *state* is 0 and through the use of the operator *next*, the *case* expression assigns values at the following execution cycles. The operator *next* may be interpreted as a unit delay. *Case* expressions are given in the form, where each line contains a condition and an arithmetic expression that are separated with a colon. The conditions in a case expression are checked in a top-down manner. For example, if *state* and i_1 are both equal to 1, then the value 1 is assigned to the variable *out*, otherwise the default value is 0.

2.4 Synchronization Techniques

Although most of the design transformations refine only one design block at a time, the change in a single block may influence the entire system on a global perspective. In addition to verifying the local correctness of refined system blocks, the timing problems caused by refinements in computation blocks at a high abstraction level are addressed.

Retiming and pipelining are two well-known techniques that address latency and data arrival problems. In order to reduce the circuit area or a critical path, retiming algorithms [61, 57] relocate already existing memory elements. Although retiming techniques address synchronization problems, these problems are not caused by additional delays inserted into the model. On the other hand, the introduction of additional delays in the model is elaborated in pipelining transformations. Pipelining in software is simpler, since different models from the perfectly synchronous one are used there. In software the synchronization points are not defined by clock events, but according to the instances when certain computation tasks are completed. In hardware the data synchronization is solved by a pipeline controller, derived from a high level system specification [107]. The synchronization techniques in Chapters 7 and 8 make it possible to introduce pipelining at system level in synchronous models with nested feedback loops, without adding controllers or changing computational models.

In order to avoid synchronizations problems caused by refinements that increase the delays of computation blocks, desynchronization [11, 78] or latency insensitive design [26] (LID) techniques can be applied. The former technique can be used to transform a synchronous model to a globally asynchronous locally synchronous (GALS) model, which is less sensitive to delayed data transfer over the asynchronous media. A comparison between synchronous, latency insensitive and asynchronous design approaches is presented in [27]. LID targets the mapping of an IP-block based synchronous model to hardware, where longer wires entail delayed data arrival. The synchronization problem is solved by (1) wrappers around IP-blocks stalling computation if input data is not available, and (2) by handshake channels and relay stations between IP-blocks that replace synchronous communication. The handshake mechanism distributes stalling messages and a relay station buffers data items if the destination process cannot consume them. A bridge-based approach is described in [103]. In this approach the relay stations are replaced with some interface logic in both ends of the communication channels between IP-blocks and instead of single wires the channels contain many parallel wires for the data transfer. In [28] an approach is proposed, which replaces LID protocols with schedulers at every IP-block. Although this method drastically simplifies the implementation of LID, it is still much more complex to refine a model including schedulers at the system level, compared to the refinement in a pure synchronous model.

Although both GALS and LID models are common in practice, they have side effects that the synchronization algorithms in Chapters 7 and 8 avoid. The proposed algorithms avoid unnecessary discontinuities in the design process caused by changes in the computational model. It is impractical to switch the computational model due to a single local refinement. The use of the same computational model makes it much easier to verify refined models against each other. In addition, verification in deterministic synchronous models by using simulation or formal methods is simpler than in other models. On the other hand the proposed synchronization techniques are complementary applicable within refinements in a synchronous island of a GALS model or in an IP-block of an LID model. The latency equivalence of the original model and a refined model after synchronization can be checked by using validation techniques described in [103].

2.5 Design Methodologies and Languages

This section introduces other design methodologies, concerning transformational design refinements and verification approaches.

A general overview about transformations from a formal problem specification to an efficient program is presented in [70], and transformations of functional and logical programs in [74]. Transformational approaches have been used mostly for development of software programs [74] to obtain from an initial specification a final program with the same semantics or with a subset of the initial semantics if the specification is non-deterministic. Transformational approaches are also used in hardware design [95], but they include mainly semantic-preserving transformations.

The **CIP** (computer-aided, intuition-guided programming) project [8] is an example of transformation system, where program development starts with describing the problem as a formal specification, and continues through a gradual development process until an executable program is created for the expected target machine. This process is based on transformation rules, which preserve semantics and thus it is ensured that the final product satisfies the initial specification. As pointed in [8] stepwise refinement by using non-semantic-preserving transformations requires to repeat verification after each step. Therefore the CIP project allows only formal, provably semantic-preserving transitions.

Metropolis [37] is a design environment, which relies on the ideas of platform based design. Platforms, as models in Metropolis, are constructed in the sense of processes, media and schedulers that provide a clear separation of computation, communication and coordination. The design flow starts with creating an abstract specification model, which will be refined in order to achieve an implementation in certain architecture. The refinement verification checks if the behavior of the refined model belongs to the abstract behaviors. The verification is performed with control flow graphs, which are created for both the abstract and refined models. For example in [30] Metropolis meta-models are translated to Promela, the input language of the SPIN model checker, and both system level representations and refined representations are verified against assertions or temporal logic properties.

The **SpecC** [38] methodology addresses the System-On-a-Chip design process, by using the SpecC language. The SpecC language [42] is based on C and allows to describe the specification, and hardware and software components by using the same semantics. The SpecC methodology consists of well-defined design models and well-defined transformations, which refine one model to another according to a top-down approach. The models are described at different levels of abstraction: specification, architecture, communication and implementation levels. In the first phase of the refinement, the system architecture is derived from an abstract high level specification, and hardware components are allocated. In the second phase, the communication protocols are synthesized, and finally the communication model is translated into an implementation model. Models at all abstraction levels are described in the same SpecC language. Hence all models are executable and the same test-benches can be used for validation through the whole design process. Compared to the ForSyDe methodology, SpecC does not include stepwise refinements by non-semantic-preserving transformations and therefore does not address related verification problems.

In [104] the flow equivalence of the initial and refined SpecC models at different levels of abstraction are formally verified by using model checking. In this approach the multi-clocked formal description of models is implemented by using the Polychrony workbench.

Lava [18, 100] is a hardware description language based on Haskell [105]. It is used for structural description of hardware, which can be translated to VHDL. In order to perform the verification of a circuit, Lava links the system description code with verification properties and applies theorem proving to check if the circuit satisfies specified properties. Lately, following the approach used in Lava, a similar compiler that translates the ForSyDe model to a synthesizable VHDL description has been developed [5]. Since the Lava and ForSyDe compilers create similar intermediate representations of the model, it is possible to improve the ForSyDe compiler so that the same theorem proving method can be used for verification of ForSyDe models as well.

Esterel [15, 16] is a synchronous language for programming reactive systems. Based on the formal semantics an Esterel model can be verified through theorem proving or a semantic check method, where the verified property is defined as a syntactic rule. Similarly to ForSyDe, a finite state model can be derived from an Esterel program, which translated into a finite state machine is a source model for verification tools to perform behavior analysis and proofs. Due to many supporting verification tools developed specially for Esterel, it is a popular design language used for safety critical systems and aerospace industry [67].

Lustre [46, 106] is a synchronous data-flow language for programming critical real-time systems. The advantage of the ForSyDe methodology is that a system model may have control and data-flow behaviors at the same time. The Lustre program includes a system description as a set of input/output relations, assumptions about the behavior of the environment as a set of assertions and, finally, a set of properties, which are checked by a verification tool. The verification is similar to the symbolic model checking and it is based on binary decision diagrams for state space exploration.

Chapter 3

The ForSyDe Methodology

The ForSyDe methodology [90] has been developed for system level design. The design development process starts at a high abstraction level and the abstraction gap between the system specification and an implementation is filled by transformational design refinement (Figure 1.2). The design process starts with the creation of the functional *system model* that is synchronous, deterministic and described in the functional language Haskell [105]. The design process continues with the stepwise refinement, which transforms the system model to an *implementation model*. The refinement is done through a series of applications of well-defined design transformations that are given in the design transformation library. The system model, the implementation model and all intermediate models are described in the functional domain that makes it easier to verify them against the design specification considering design constraints, since the same verification techniques can be used for all models. The implementation model as a result of the refinement process has all the lower level details required for mapping to hardware (VHDL) and software (C++).

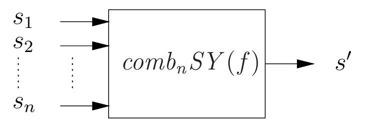
3.1 The System Model

The system model uses a synchronous model of computation, which describes the system functionality at a high abstraction level in terms of unlimited resources without including lower level implementation details. A system is defined as a set of concurrently executed processes that are connected by synchronous signals.

Signal s_i is defined as a sequence of events $\{v_0^i, v_1^i, \ldots, v_j^i, \ldots\}$, where v_j^i is the value of the j-th event (with time tag j) of signal s_i. All signals share the same set of tags for synchronization purposes. The signal direction is from the source process to the destination process, and every process has only one output signal.

There are two kinds of events: (1) *present events* that carry a value and (2) *absent events* that are used only for synchronization. An absent event $\perp_j$ shows that a signal contains no value at a time instant j. The mark $\top$ is used as an abstract value when referring to present values, and $\perp$ denotes absent values. For example, the abstract presentation of the signal of absent extended integers $\{1_1, 4_2, \perp_3, 2_4, \ldots\}$ is $\{\top, \top, \perp, \top, \ldots\}$. To extend a data type T to $T_\perp$ the $\perp$-value is added to its domain.

Processes are defined through *process constructors*, which are higher order functions that take combinational functions and values as arguments and produce processes. This approach allows to separate computation (combinational functions) from communications (process constructors). The semantics of the process constructors is introduced in [84]. The general processes that a system contains are combinational processes P_{comb}, state machine processes P_{FSM} and simple delays P_Δ, and more complex components can be constructed by combining them.



$$
\begin{aligned}
P_{comb}(f, s_1, s_2, \ldots, s_n) &= s' \\
\text{where} & \\
f(v_i^1, v_i^2, \ldots, v_i^n) &= v'_i \\
s_1 &= \{v_0^1, v_1^1, v_2^1, \ldots\} \\
s_2 &= \{v_0^2, v_1^2, v_2^2, \ldots\} \\
&\vdots \\
s_n &= \{v_0^n, v_1^n, v_2^n, \ldots\} \\
s' &= \{v'_0, v'_1, v'_2, \ldots\}
\end{aligned}
\tag{3.1}
$$

Figure 3.1. The combinational process constructor $comb_nSY$

A combinational process takes arguments as a dedicated n-input combinational function $f(x_1, \ldots, x_n)$ and n input signals. For each tag j, a combinational process consumes from its input signals $s_1, \dots, s_n$ events with the tag j carrying values $v_j^1, \dots, v_j^n$ and produces an event with the tag j and a value $v'_j = f(v_j^1, \dots, v_j^n)$ to its output signal s':

$$\begin{aligned} P_{comb}(f(x_1, \dots, x_n), s_1, \dots, s_n) &= s' = \{v'_0, v'_1, \dots, v'_n, \dots\} = \\ &= \{f(v_0^1, \dots, v_0^n), f(v_1^1, \dots, v_1^n), \dots, f(v_j^1, \dots, v_j^n), \dots\} \end{aligned}$$

Combinational processes are defined by using process constructors $comb_n SY$ (Figure 3.1), where n depends on the number of input signals the process is connected with.

A delay process $P_\Delta(st_0, s_1)$ has arguments as an initial state value st_0 and an input signal $s_1 = \{v_0^1, v_1^1, \dots, v_j^1, \dots\}$.

$$P_\Delta(st_0, s_1) = s' = \{st_0, v_0^1, v_1^1, \dots, v_j^1, \dots\}$$

A finite state machine process P_{FSM} (Figure 3.2) with a state function f_{st}, an output function f_{out}, an initial state st_0, n input signals $s_1, \dots, s_n$ and an output signal s' is defined as:

$$\begin{aligned} & P_{FSM}(f_{st}, f_{out}, st_0, s_1, \dots, s_n) = s' \\ & \quad \textit{where} \\ & \quad\quad s' = P_{comb}(f_{out}, s'', s_1, \dots, s_n) \\ & \quad\quad s'' = P_{comb}(f_{st}, (P_\Delta(st_0, s''), s_1, \dots, s_n)) \end{aligned}$$

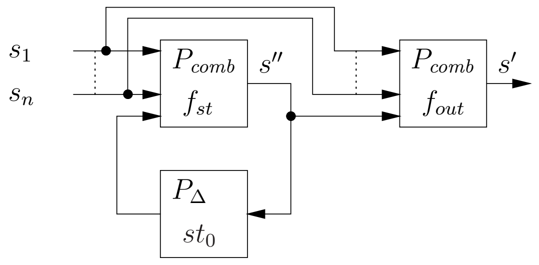
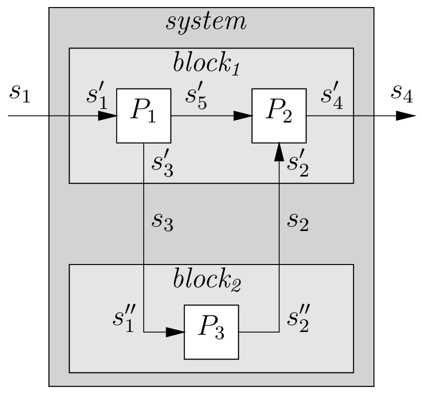


Figure 3.2. Finite state machine

The system is constructed as a network of processes and modeled as a set of equations. For a hierarchical description of the system a set of processes may be composed as a *block*. An example of a process network with the corresponding system description as a netlist is given in Figure 3.3. The system contains two blocks *block₁* and *block₂*. *Block₁* includes two processes P_1 and P_2, and process P_3 belongs to *block₂*.



$$\begin{aligned}
\mathit{block}_1\ (s'_1, s'_2) &= (s'_3, s'_4)\\
&\text{where}\\
(s'_3, s'_5) &= P_1\ (s'_1)\\
(s'_4) &= P_2\ (s'_5, s'_2)
\end{aligned}$$

$$\begin{aligned}
\mathit{block}_2\ (s''_1) &= (s''_2)\\
&\text{where}\\
(s''_2) &= P_3\ (s''_1)
\end{aligned}$$

$$\begin{aligned}
\mathit{system}\ (s_1) &= (s_4)\\
&\text{where}\\
(s_3, s_4) &= \mathit{block}_1\ (s_1, s_2)\\
(s_2) &= \mathit{block}_2\ (s_3)
\end{aligned}$$

Figure 3.3. A network of processes

3.2 Implementation Models

An implementation model is the result of the refinement process. In contrast to the system model, which is a network of concurrent synchronous processes, it may also include synchronous sub-domains with a different signal rate. Synchronous sub-domains violate the synchronous assumption since not all signals share the same set of tags. Thus they are not allowed in the system model, but are introduced by well-defined transformations during the refinement process. Inside a synchronous sub-domain the synchronous assumption is still valid and the same formal techniques can be used as for the system model. The implementation model contains lower level implementation details that are required for hardware (VHDL) or software (C++) synthesis.

3.3 Design Refinement

A main strength of the ForSyDe methodology is the design transformations based design development process, which stepwise refines the initial system model into a final implementation model. The design transformations are well defined and in addition to common semantic-preserving transformations the ForSyDe methodology supports non-semantic-preserving design transformations. The latter allows to start the design process at a significantly higher abstraction level, where the system is described in the sense of ideal resources, that is not possible if only semantic-preserving transformations are applied. On the other hand, if the design refined by semantic-preserving transformations can be considered to be correct-by-construction, it is obligatory to verify that the implementation model derived by the application of non-semantic-preserving transformations satisfies the given specification.

The designer applies transformations to a system model by choosing transformation rules from the transformation library. The transformation rules are characterized by a name, the required format and constraints on the initial process network, the format of the transformed process network and the implication for the design, i.e., the relation between the initial and the transformed process networks is expressed by the characteristic function [93].

3.4 State of the ForSyDe Methodology

Today the ForSyDe methodology supports modeling of concurrent reactive systems at different levels of abstraction. The ForSyDe libraries provide process constructors for describing systems in data flow, synchronous and timed models of computations. An objective of the currently running *ANDRES* project [1] is to integrate these different computation models in the same system description in order to provide more flexibility for analyzing the functionality of embedded systems. In addition, this project addresses functional adaptivity issues in the system description and implementation. For the sake of clarification it has to be mentioned, that the thesis address verification and synchronization only in the context of the synchronous model of computation.

The fundamental and established concept of the design development process based on the design transformation is summarized in [93], though the number of design transformation rules in the ForSyDe design library is rather small. In addition to the design process, the mapping of an implementation model, which is the result of the refinement process, to hardware and software is explained in [94, 5] and [91], respectively. A formal set of synchronization components for refinements into hardware and software implementations are presented in [60].

3.5 ForSyDe Constructions in the SMV Language

In order to formally verify ForSyDe models, the SMV model checker has been used. The following sections give short guidelines for describing ForSyDe models in the SMV language. Text files in both languages have a common structure. Files in both cases include (a) data type definitions, (b) user defined function descriptions, (c) process definitions and (d) a system structural description, which shows how processes are connected with each other. User defined functions, processes, and the system structure that in the ForSyDe language are expressed as Haskell functions, are defined as modules in the SMV language. However, there are only minor differences in terminology between functions and modules.

3.5.1 Example: Process *Receive* in ForSyDe

Let's consider the following definition of the process Receiver in the ForSyDe, explained in more details in Section 5.2, in order to illustrate the mapping from ForSyDe to SMV:

```
data RecState = WaitDataReady | WaitData | OutputData

receive = moore2SY recStateFn recOutFn (WaitDataReady, 0)

recOutFn :: (RecState      , Int )-> (AbstExt RecMsg , AbstExt Int )
recOutFn    (WaitDataReady , _   ) = (Prst Ack       , Abst        )
recOutFn    (WaitData      , _   ) = (Prst Ready     , Abst        )
recOutFn    (OutputData    , v   ) = (Abst           , Prst v      )
```

The process Receiver is an FSM process, with a state function *recStateFn* and an output function *recOutFn*. The state of Receiver may have three values *WaitDataReady*, *WaitData* and *OutputData*, which are defined as an enumerated data type in the first line of the given ForSyDe description. On the following line, the process *receive* is defined as a Moore state machine with the state function *recStateFn*, the output function *recOutFn* and the initial state value (*WaitDataReady*, *0*). The top most line starting with *recOutFn* defines input/output data types of the Receiver output function, and the following lines describe the behavior of the output function. The next state function is not presented since its layout is close to the output function.

3.5.2 Data Types

With some minor modifications, it is possible to define equivalent data types for all ForSyDe data types in SMV. SMV provides Booleans and enumerated data types, structures and arrays.

Enumerated An enumerated data type definition in ForSyDe starts with the word **data** that follows by the data type name and a list of predefined values, as shown below:

$$\text{data } \textit{data\_type\_name} \;=\; \textit{value}_1 \mid \textit{value}_2 \mid \ldots \mid \textit{value}_n$$

An equivalent construction in SMV has the following format:

$$\text{typedef } \textit{data\_type\_name} \; \{\, \textit{value}_1 \;,\; \textit{value}_2 \;,\; \ldots \;,\; \textit{value}_n \};$$

The following is an example of the *RecState* definition in SMV:

$$\text{typedef } \textit{RecState} \;\; \{\textit{WaitDataReady} \;,\; \textit{WaitData} \;,\; \textit{OutputData}\ \};$$

Integers Contrary to the ForSyDe language, integer is not a predefined data type in SMV. Similarly to enumerated data types, the user has to define integers as a bounded set of values. However, SMV treats these enumerated values as integers in arithmetic and logic expressions. Two equivalent constructions that define an integer data type including the values from 0 to 7 are given below:

$$\text{typedef } \textit{Int\_0\_7}\ \{0,1,2,3,4,5,6,7\};$$
$$\text{typedef } \textit{Int\_0\_7}\ \{0..7\};$$

Constructors All *absent extended* data types in ForSyDe are defined by constructors. For example, the following is a data type definition of absent extended integers in ForSyDe:

$$\text{data } \textit{AbstExt a} \;=\; \textit{Abst} \mid \textit{Prst a};$$
$$\text{type } \textit{Abst\_Int} \;=\; \textit{AbstExt Int};$$

The constructor based data types in ForSyDe have equivalent definitions as structures in SMV, which have the following format:

$$\begin{array}{lllll}
\text{typedef} & \textit{data\_type\_name}\ \text{struct} & \{ & \\
& \textit{component\_name}_1 & : & \textit{data\_type}_1; \\
& \textit{component\_name}_2 & : & \textit{data\_type}_2; \\
& \ldots & & \ldots \\
& \textit{component\_name}_n & : & \textit{data\_type}_n; \\
& & \}; &
\end{array}$$

Absent extension *AbstExt* can be defined as an enumerated data type comprising of values *Abst* and *Prst* in SMV. Let the elements of the absent extended integers in SMV be (1) a constructor *Con* carrying a value *Prst* or *Abst* and (2) a component *Val* with the values from 0 to 7. The SMV definitions of these data types are given below:

$$\begin{array}{llll}
\text{typedef} & \textit{AbstExt}\ \{\textit{Prst}, \textit{Abst}\}; & & \\
\text{typedef} & \textit{Abst\_Int\_0\_7}\ \text{struct} & \{ & \\
& \textit{Con} & : & \textit{AbstExt}; \\
& \textit{Val} & : & \textit{Int\_0\_7}; \\
& & \}; &
\end{array}$$

To the i-th component c_i of a variable v can be turned by $v.c_i$ in SMV. For example, to add the constant $y = 2$ to an absent extended integer x and to assign the result to a variable z, the following SMV expressions can be used:

$$\begin{array}{l}
\#\text{define } y\ 2 \\
z.Con := x.Con \\
z.Val := x.Val + y
\end{array}$$

Due to the definition of absent extension in SMV, similarly to present events, all absent events carry a value. However, this does not influence model checking, since the values assigned to absent events are neither used in the system specification nor in the verified properties. In fact, ForSyDe models do not give any value to absent events and the SMV tool allows to leave these values undefined.

Lists The SMV model checker can only verify finite models. Therefore, the lists with unlimited number of elements in ForSyDe models have to be bounded to include only a finite number of elements. In this case a list can be defined as an array with a fixed number of elements in SMV. As a complement, a counter is needed in the list definition, in order to know how many elements are currently stored in the list. When an element is added or removed from the list, the counter value has to be updated. An SMV definition of a list that can include maximum seven Booleans is given below:

```
typedef List_Bool_7 struct   {
counter                      :   Int_0_7;
list                         :   array 0..6 of Boolean;
                             };
```

Tuples n-tuples in ForSyDe can be modeled as n-element structures in SMV, where all structure elements have the same type. The definition of a ForSyDe tuple as a structure in SMV is not necessary if the tuple is used in a function output description. The values of a multi-output function are combined together to a tuple in ForSyDe, since functions definition in ForSyDe may have only one output. SMV does not have this kind of restriction and the function outputs can be defined as separate signals without using tuples.

3.5.3 Arithmetic and Logic Expressions

Boolean and Arithmetic Operators

SMV uses the values 0 and 1 as Boolean values that are equivalent to the values False and True in ForSyDe. Equivalent Boolean operators in both languages are listed in Table 3.1, and arithmetic operators in Table 3.2. Although *exclusive or* and *implication* are not explicitly defined in ForSyDe, these operators are extensively used in temporal logic property definitions.

Table 3.1. Boolean operators

Operation	*ForSyDe Operator*	*SMV Operator*
Logical and	(x `&&` y)	(x `&` y)
Logical or	(x `||` y)	(x `|` y)
Logical not	*not*(x)	(`~`x)
Exclusive or	not defined	(x `^` y)
Implication	not defined	(x `->` y)

Table 3.2. Arithmetic operators

Operation	*ForSyDe Operator*	*SMV Operator*
Addition	$(x + y)$	$(x + y)$
Subtraction	$(x - y)$	$(x - y)$
Multiplication	$(x * y)$	$(x * y)$
Integer Division	*div x y*	(x/y)
Remainder of Division	*mod x y*	*x mod y*

The comparison operators in ForSyDe and SMV are presented in Table 3.3. The operators *equal* and *not equal* can be applied to Boolean and enumerated values, and the rest of the operators to integers as well. In order to avoid confusions with equality operators, it is appropriate to mention that the syntactic element for assignment '=' in ForSyDe corresponds to the syntax ':=' in the SMV language.

Table 3.3. Comparison operators

Operation	*ForSyDe Operator*	*SMV Operator*
Greater than	$(x > y)$	$(x > y)$
Less than	$(x < y)$	$(x < y)$
Greater than or Equal to	$(x >= y)$	$(x >= y)$
Less than or Equal to	$(x <= y)$	$(x <= y)$
Equal to	$(x == y)$	$(x = y)$
Not Equal to	$(x$/=$y)$	$(x$~=$y)$

Conditionals

If-then-else conditional expressions have the following format in ForSyDe:

```
if       condition_1   then   statement_1
else if  condition_2   then   statement_2
...
else if  condition_n   then   statement_n
else                          default statement
```

Equivalent expressions in SMV are defined by using *case* statements as shown below:

$$\begin{array}{lll} \text{case} \ \{ & \mathit{condition}_1 & : \ \mathit{statement}_1; \\ & \mathit{condition}_2 & : \ \mathit{statement}_2; \\ & \ldots & \\ & \mathit{condition}_n & : \ \mathit{statement}_n; \\ & \text{default} & : \ \mathit{default\ statement}; \\ \}; & & \end{array}$$

3.5.4 Functions and Pattern Matching

Pattern matching is a common construction to describe functions in ForSyDe. A general structure of the pattern matching statements is given below[1]:

$$\begin{array}{llllll} -- & \mathit{input}_1 & \mathit{input}_2 & \ldots & \mathit{input}_n & = \ \mathit{output} \\ \mathit{fun} & \mathit{pattern}_{1,1} & \mathit{pattern}_{1,2} & \ldots & \mathit{pattern}_{1,n} & = \ \mathit{statement}_1 \\ \mathit{fun} & \mathit{pattern}_{2,1} & \mathit{pattern}_{2,2} & \ldots & \mathit{pattern}_{2,n} & = \ \mathit{statement}_2 \\ \ldots & & & & & \\ \mathit{fun} & \mathit{pattern}_{m,1} & \mathit{pattern}_{m,2} & \ldots & \mathit{pattern}_{m,n} & = \ \mathit{statement}_m \end{array}$$

Every line in the function definition above, presents one pattern including n components, one for each input variable. A pattern component can be a literal value, a variable, a tuple pattern, a constructor pattern or wild-card (don't care). The evaluation of a pattern matching statement starts from the top. The first pattern ($\mathit{pattern}_i$), where all components ($\mathit{component}_{i,j}$) match input values, evaluates the function output by $\mathit{statement}_i$.

Literal value An argument matches the pattern component if it is equal to the value that is given in the pattern component;

Variable All arguments match variables. A variable can be distinguished from an enumerated value according to the first letter in the string, which represents it. All variable names in ForSyDe start with small letters and enumerated values start with capital letters;

Tuple An argument matches a tuple pattern if all elements in the argument match their respective components in the pattern. For instance, the pattern $(7, x)$ contains two components, where the first one is a literal value and the second is a variable. This two component tuple pattern matches any argument, where the first component is equal to 7;

[1]Lines starting with " $--$" are comments in the ForSyDe language

Constructor Constructor patterns are very similar to the tuple patterns. An argument matches a constructor pattern if all elements in the argument match the respective components in the pattern. In the following example, present values match the first pattern and absent values match the second one.

$$\begin{array}{llcll} \textit{fun} :: & \textit{AbstExt Int} & -> & \textit{Int} & --\ \textit{type definition} \\ \textit{fun} & \textit{Prst } x & = & x+1 & --\ \textit{pattern one} \\ \textit{fun} & \textit{Abst} & = & 0 & --\ \textit{pattern two} \end{array}$$

Wild-card Wild-card "_" is used as a default value, that matches any argument value.

Pattern matching constructions can be expressed by case statements in SMV, where literal value patterns are defined by equality comparisons. Variables in patterns become arguments for the respective arithmetic/logic statements. Patterns, which do not include any literal value, are equivalent to "default" choice in case statements.

The following definition of the function *fn* illustrates the mapping from pattern matching constructions to SMV case statements.

$$\begin{array}{llccl} \textit{fn} :: & \textit{AbstExt Int} & -> & (\textit{Int}, \textit{Int}) \ -> & \textit{Int} \\ \textit{fn} & \textit{Prst } 7 & & (b, c) \ = & c+2 \\ \textit{fn} & \textit{Prst } a & & (b, 3) \ = & a+b \\ \textit{fn} & \textit{Abst} & & (b, c) \ = & c \\ \textit{fn} & \_ & & (b, c) \ = & b \end{array}$$

The function has two inputs with the data types absent extended integer and tuple of integers, respectively. Let o_1 denote the function output, i_1 denote the first input argument, and i_2 and i_3 denote the tuple elements of the second input argument. As defined before the data type *AbstExt* is a structure of two components, which are denoted as *Con* and *Val*. The function *fn* has the following format in SMV:

$$\begin{array}{lll} o_1 := \text{case} & \{ & i_1.Con = Prst \ \texttt{\&} \ i_1.Val = 7 \quad : \ i_3 + 2; \\ & & i_1.Con = Prst \ \texttt{\&} \ i_3 = 3 \quad : \ i_1.Val + i_2; \\ & & i_1.Con = Abst \quad : \ i_3; \\ & & \text{default} \quad : \ i_2; \\ & \}; & \end{array}$$

Since all input values match the last pattern in *fn*, the pattern is defined as the *default* choice in the SMV case statement.

Functions

The general function structure in ForSyDe is given in the following:

$$\begin{array}{llllll} --\,fn\_name & input_1 & -> & input_2 & -> & (output_1, output_2) \\ fn_2 :: & type_1 & -> & type_2 & -> & (type_3, type_4) \\ fn_2 & pattern_{1,1} & & pattern_{1,2} & = & (statm_{1,1}, statm_{1,2}) \\ fn_2 & pattern_{2,1} & & pattern_{2,2} & = & (statm_{2,1}, statm_{2,2}) \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{array}$$

All lines in a function description start with the function name. The first line describes the data types of inputs and an output, separating them by "$->$". In ForSyDe, every function has exactly one output and the right most data type on the first line corresponds to the function output. The rest of components belong to the function inputs. Components of a tuple input or output are considered as separate inputs and outputs in SMV.

After data type descriptions, the following lines define the function behavior in the pattern matching form. In SMV, functions are described as modules. An example of a module that is equivalent to fn_2 is the following:

```
MODULE $fn_2$ $(i_1, i_2)${
  $o_1$ : $type_3$;
  $o_2$ : $type_4$;
  $o_1$ := case {$condition_1$ : $statem_{1,1}(i_1, i_2)$;
                 $condition_2$ : $statem_{2,1}(i_1, i_2)$;
                 ...
            };
  $o_2$ := case {$condition_1$ : $statem_{1,2}(i_1, i_2)$;
                 $condition_2$ : $statem_{2,2}(i_1, i_2)$;
                 ...
            };
}
```

A module starts with the function name and the names of local input variables, given by the designer. The module continues with the data type declaration of outputs and the description how the outputs are evaluated. All components of a tuple output are evaluated separately. $Condition_i$ in the given case statements is an equivalent interpretation of the conditional defined by $pattern_i$.

Function definitions may use function calls, though recursive calls are not allowed in SMV. Therefore, ForSyDe models with recursive function calls cannot be verified by SMV. The following functions fn_3 and fn_4 illustrate the use of function calls in ForSyDe.

$$\begin{array}{llcl}
fn_3 :: & Int & \rightarrow & (Int, Int) \\
fn_3 & a & = & (a-1, a+1) \\
 & & & \\
fn_4 :: & Int & \rightarrow & (Int, Int) \\
fn_4 & 0 & = & (0,0) \\
fn_4 & a & = & (fn_3\ (2*a))
\end{array}$$

The following SMV modules are equivalent to the functions fn_3 and fn_4 in ForSyDe:

```
MODULE $fn_3$ ($i_1$){
  $o_1$ : $Int\_0\_7$;
  $o_2$ : $Int\_0\_7$;
  $o_1$ := $i_1 - 1$;
  $o_2$ := $i_1 + 1$;
}

MODULE $fn_4$ ($i_1$){
  $o_1$ : $Int\_0\_7$;
  $o_2$ : $Int\_0\_7$;
  $s_1$ : $fn_3(2 * i_1)$;
  $o_1$ := case { $i_1 = 0$    : 0;
                  default : $s_1.o_1$;
                };
  $o_2$ := case {$i_1 = 0$    : 0;
                  default : $s_1.o_2$;
                };
}
```

The module fn_4 uses an internal signal s_1 to refer to the function fn_3 with the argument $2 * i_1$. In order to address the output signals of the module fn_3 in the function calls, constructions $s_1.o_1$ and $s_1.o_2$ are used.

An equivalent module to the Receiver output function *recOutFn* (Section 3.5.1) is presented below:

```
MODULE $recOutputF(i_1, i_2)\{$
  $o_1$ : $AbstRecMsg$;
  $o_2$ : $Abst\_Int\_0\_7$;
  $o_1.Con$ := case { $i_1 = WaitDataReady : Prst$;
                      $i_1 = WaitData : Prst$;
                      $i_1 = OutputData : Abst$;
                    };
  $o_1.Val$ := case { $i_1 = WaitDataReady : Ack$;
                      $i_1 = WaitData : Ready$;
                      $i_1 = OutputData : Ack$;
                    };
  $o_2.Con$ := case { $i_1 = WaitDataReady : Abst$;
                      $i_1 = WaitData : Abst$;
                      $i_1 = OutputData : Prst$;
                    };
  $o_2.Val$ := case { $i_1 = WaitDataReady : i_2$;
                      $i_1 = WaitData : i_2$;
                      $i_1 = OutputData : i_2$;
                    };
}
```

The module starts with the data type definitions of the receiver output. The original tuple output is split into two outputs o_1 and o_2. Since the data types of both outputs are absent extended and absent extension is formed as a two element structure in SMV, the elements *Con* and *Val* are evaluated in separate case expressions.

3.5.5 Processes

In ForSyDe, processes are described by using process constructors that are higher order functions. A process constructor takes a set of functions and initial state values as arguments and applies these functions to the values consumed from the input signals. In SMV, processes can be defined as modules that use function calls to modules, which correspond to combinational functions in ForSyDe.

Combinational Processes

In ForSyDe, combinational processes are defined by process constructors $comb_nSY$, where n shows the number of process inputs. $Comb_nSY$ takes one value from all input signals and applies a dedicated n-variable combinational function to the input values at every clock cycle. In SMV, there is no difference in a module definition whether the module receives a single input assignment or a sequence of assignments as a signal. Therefore, an SMV module describing a combinational function is equivalent to the ForSyDe combinational process with the same function.

Delay Process

In order to solve the computation in feedback loops in the perfectly synchronous models, the feedback signals have to be initialized. ForSyDe models use *delaySY* processes to give initial values to feedback loops. *DelaySY* sends out its initial state value st_0 at the first clock cycle, and the received input values one clock cycle after arrival. The process *delaySY* is defined in SMV as follows:

```
MODULE $delay(i_1, \ldots, i_{n'})$\{
   $o_1$          :    $data\ type\ of\ the\ output\ component_1$;
   ...
   $o_{n'}$         :    $data\ type\ of\ the\ output\ component_{n'}$;
   init($o_1$)    :=   $st0_1$;
   ...
   init($o_{n'}$)   :=   $st0_{n'}$;
   next($o_1$)    :=   $i_1$;
   ...
   next($o_{n'}$)   :=   $i_{n'}$;
 }
```

The module has n' inputs and outputs since tuple inputs in ForSyDe are split into components in SMV. Similarly, the initial value st_0 of the delay process is decomposed into n' component values.

State Machine Processes

Process constructor $mealySY_n$ takes a next state function f , an output function g and an initial state value v_{init} as arguments and models a finite

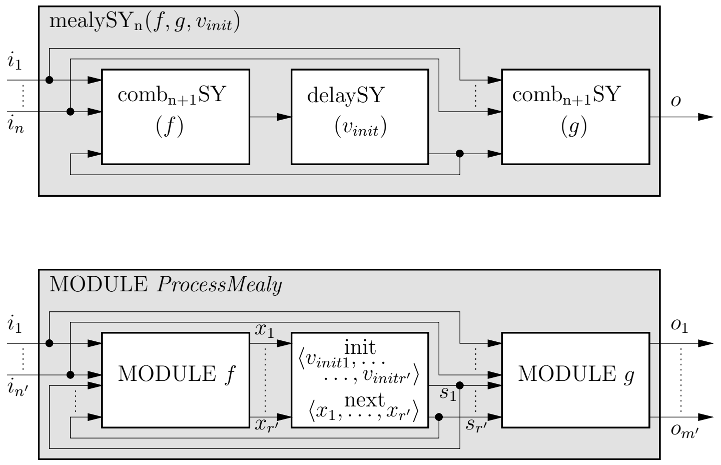


Figure 3.4. Process and module of the Mealy FSM

state machine. The definition of an n-input Mealy finite state machine is the following:

$$\mathit{ProcessMealy} = \text{mealySY}_\text{n}(f, g, v_{init})$$

The layouts of the process *mealySY$_n$* in ForSyDe and the corresponding module in SMV are presented in Figure 3.4. The definition of a Mealy state machine process as a module in SMV, leaving out the next state and output functions, is the following:

```
MODULE $\mathit{ProcessMealy}(i_1, \ldots, i_{n'})\{$
    $s_1$               :    $\mathit{data\ type\ of\ the\ state}$;
    $\ldots$
    $s_{r'}$            :    $\mathit{data\ type\ of\ the\ state}$;
    $o_1$               :    $\mathit{data\ type\ of\ the\ output}$;
    $\ldots$
    $o_{m'}$            :    $\mathit{data\ type\ of\ the\ output}$;
    $\mathit{StateFunction}$  :    $f(s_1, \ldots, s_{r'}, i_1, \ldots, i_{n'})$;
    $\mathit{OutputFunction}$ :    $g(s_1, \ldots, s_{r'}, i_1, \ldots, i_{n'})$;
    init($s_1$)         :=   $v_{init1}$;
    $\ldots$
    init($s_{r'}$)      :=   $v_{initr'}$;
```

$$\begin{array}{lll}
\quad \text{next}(s_1) & := & \textit{StateFunction}.o_1; \\
\quad \ldots & & \\
\quad \text{next}(s_{r'}) & := & \textit{StateFunction}.o_{r'}; \\
\quad o_1 & := & \textit{OutputFunction}.o_1; \\
\quad \ldots & & \\
\quad o_{m'} & := & \textit{OutputFunction}.o_{m'}; \\
\}
\end{array}$$

The definition of a module that corresponds to a Moore finite state machine process is identical to the Mealy one, though in a Moore machine input arguments do not appear in the function call to the state machine output function. For example, the process Receive defined as a Moore state machine has the following coding in SMV:

$$\begin{array}{lll}
\text{MODULE } \textit{Receive}(i_1, i_2)\{ & & \\
\quad s_1 & : & \textit{RecState}; \\
\quad s_2 & : & \textit{Int\_0\_7}; \\
\quad o_1 & : & \textit{AbstRecMsg}; \\
\quad o_2 & : & \textit{Abst\_Int\_0\_7}; \\
\quad \textit{FunRecState} & : & \textit{recStateF}(s_1, s_2, i_1, i_2); \\
\quad \textit{FunRecOutput} & : & \textit{recOutputF}(s_1, s_2); \\
\quad \text{init}(s_1) & := & \textit{WaitDataReady}; \\
\quad \text{init}(s_2) & := & 0; \\
\quad \text{next}(s_1) & := & \textit{FunRecState.o}_1; \\
\quad \text{next}(s_2) & := & \textit{FunRecState.o}_2; \\
\quad o_1 & := & \textit{FunRecOutput.o}_1; \\
\quad o_2 & := & \textit{FunRecOutput.o}_2; \\
\}
\end{array}$$

The input, state and output signals of the Receive process that in the ForSyDe model are defined as tuples are described by separate variable pairs $(i_1, i_2, s_1, s_2, o_1, o_2)$ in SMV. In order to evaluate the state and output variables, the module *Receive* uses function calls to the modules implementing the receiver state and output functions.

3.5.6 Netlist

In ForSyDe a system is modeled as a network of processes that are connected by signals. For example, the connections of a system including three computation processes P_1, P_2, P_3 in Figure 3.5 are described as follows:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textit{system}\ (s_{in}) \ = \ (s_{out})} \\
\multicolumn{3}{l}{\quad \text{where}} \\
\quad\quad s_1 & = & P_1\ (s_{in}) \\
\quad\quad (s_2, s_3) & = & \text{unzipSY}\ (s_1) \\
\quad\quad s_4 & = & P_3\ (s_3) \\
\quad\quad s_{out} & = & P_2\ (s_2, s_4)
\end{array}$$

The signals s_{in} and s_{out} are the input and output signals of the system. The task of the additional process *unzipSY* is to separate the tuple output signal s_1, produced by P_1, to the component signals s_2 and s_3.

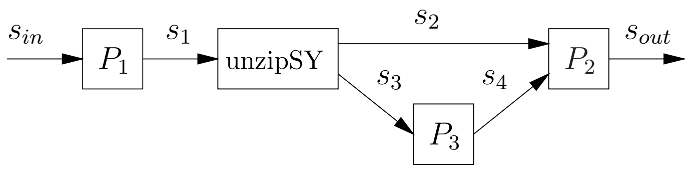


Figure 3.5. A network of processes in ForSyDe

The first line in a netlist defines the system name and specifies the input and output signals. The lines below of *where* describe connecting signals between processes. Since components of a tuple signal in ForSyDe are modeled as separate component signals in SMV, processes *zipSY* and *unzipSY* are excluded from SMV models. Thus the SMV network in Figure 3.6 is equivalent to the network in Figure 3.5.

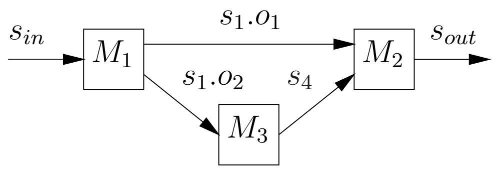
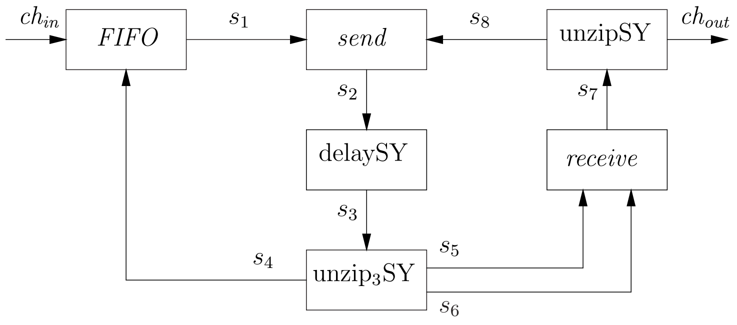


Figure 3.6. A network of processes in SMV

In SMV, the connections between processes are described in the *main module*. The main module of the system in Figure 3.6 has the following definition:

MODULE main(){
s_{in} : *type of* s_{in};
s_1 : M_1 (s_{in});
s_4 : M_3 ($s_1.o_2$);
s_{out} : M_2 ($s_1.o_1, s_4$);
}

The main module starts with the type definitions of the system input signals, with the definition of the signal s_{in} in the given example. On the following lines, the module continues with the instantiation of the modules M_1, M_2, M_3 including the description of connections between modules. Since the *unzipSY* process is not considered in the SMV model, the signals $s_2.o_1$ and $s2.o_2$ are equivalent to the ForSyDe signals s_3 and s_4 respectively.



$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textit{channel}\ (ch_{in}) = (ch_{out})} \\
\multicolumn{3}{l}{\quad \text{where}} \\
s_1 & = & \textit{FIFO}\ (ch_{in}, s_4) \\
s_2 & = & \textit{send}\ (s_1, s_8) \\
s_3 & = & \text{delaySY}\ (s_2) \\
(s_4, s_5, s_6) & = & \text{unzip}_3\text{SY}(s_3) \\
s_7 & = & \textit{receive}\ (s_5, s_6) \\
(s_8, ch_{out}) & = & \text{unzipSY}\ (s_7)
\end{array}
$$

Figure 3.7. The process network and the netlist of an asynchronous channel

Let's consider the netlist definition of a channel, which implements asynchronous data transfer from the channel input ch_{in} to ch_{out}, in Figure 3.7.

The channel includes three finite state machine processes: *FIFO*, *send* and *receive*. *FIFO* stores arrived data items if *send* and *receive* are busy with a handshake protocol based data transfer of a previously arrived data item through the channel. For a more detailed description of the channel, see Section 5.2.

After excluding *zipSY* and *unzipSY* processes from the process network, an equivalent module network in SMV has the structure as shown in Figure 3.8.

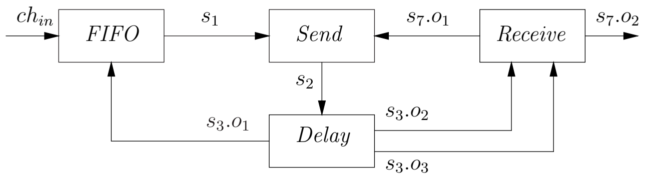


Figure 3.8. The module network of the channel

The *main* module of the asynchronous channel has the following definition in SMV:

MODULE main(){
ch_{in} : *Abst_Int_0_7*;
s_1 : *FIFO* $(ch_{in}, s_3.o_1)$;
s_2 : *Send* $(s_1, s_7.o_1)$;
s_3 : *Delay* (s_2);
s_7 : *Receive* $(s_3.o_2, s_3.o_3)$;
}

3.6 Summary

This chapter defines mapping rules that show how to describe ForSyDe synchronous models in the input language of the SMV model checker. The SMV model checker is used to formally verify the correctness of refined models that is explained in the following chapters. The given rules cover data type, function and process definitions, and internal connections between processes. Based on these rules the mapping from ForSyDe to SMV can be automated by creating a similar compiler as in [5], which translates synchronous ForSyDe models to VHDL. However, it is not possible to map any ForSyDe model to SMV. The SMV model checker does not support recursive function calls and data types with unspecified domains. Therefore, recursive function calls should be forbidden and the ideal data types and infinite lists, used in the high level system description, have to be replaced with finite ones before the mapping to SMV.

Chapter 4

Design, Refinement and Verification

4.1 Design

The system development in the ForSyDe methodology starts with describing the system functionality as an ideal *system model*, that reflects the system specification written in a native language. The model at a high abstraction level is ideal in the sense that it has no side effects caused by lower level implementation details. For example, arithmetic functions are described without applying limits on the operand or output bit-widths and due to that functions are free of overflow behaviors. Similarly, the size of storage elements is not limited, in order to avoid exceptional behaviors caused by buffer overflows, for instance. The model is synchronous and deterministic, which makes it easy to simulate and analyze or to compare it against other models for equivalence check.

The main task of the design refinement process is to extend the system model with details that are necessary for the design implementation in software and hardware. The last stage of the refinement process is an implementation model, which can be mapped to software and hardware models. Obviously, for an implementation the unlimited bit-widths and infinite storage units have to be replaced with realistic counter parts. Thus, the objective of the system level design refinement process is to fill the abstraction gap between the initial system model and an implementation model, which includes all lower level details that are required to continue the design process by today's synthesis tools.

4.2 Refinement

The design refinement from the system model to an implementation model is performed by a stepwise application of design transformations [90], as illustrated in Figure 4.1.

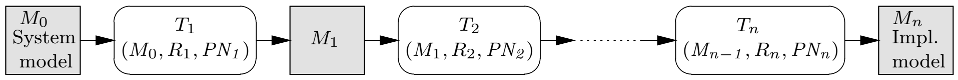


Figure 4.1. Design flow

Starting from the system model M_0 an implementation model M_n is developed by a series of transformations T_i. In every step, for a given model M_i, a process network PN in M_i and a transformation rule R, the transformation $T(M_i, R, PN) \to M_{i+1} = M[R(PN)/PN]$ refines the process network PN. The result of the transformation is an intermediate system model M_{i+1}, where in contrast to the model M_i the process network PN is replaced with $R(PN)$. For example, in Figure 4.2 the model M is refined to M' by the transformation T, which replaces a process network PN with a modified process network PN'.

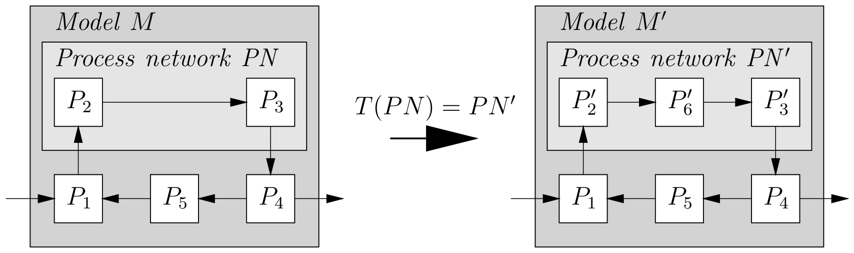


Figure 4.2. Design transformation

A basic assumption of the design and verification strategy is that the designer only uses those design transformations in the refinement process that are available in the design transformation rule library. Hence, ad-hoc design refinements based on the designer intuition are forbidden. Despite that, if there is a need for a special transformation that is not yet included in the design library, the new transformation can be defined with all necessary details and stored in the library within the system development process. All design transformations in the library are specified by name, the requirements to the original process network, the formats of the original and refined process networks, and the implication for the design. The implication is expressed by characteristic functions [94] that describe the relation between the input and output events in a process network.

According to the characteristic function a transformation can be classified either as semantic-preserving or non-semantic-preserving. Semantic-preserving transformations do not change the meaning of the model in the sense that the input and output events of the original and a refined process networks are identical. These transformations mainly change the structure of the model by decomposing and merging of processes, and relocating functions between processes in order to optimize the design. Non-semantic-preserving transformations change the meaning of the model and introduce new behaviors. In spite of the change in semantics, the refined model may still behave equivalently to the original model if it is expected that the refined model has to implement only a subset of the behaviors of the original model. A correctly behaving refined model is not always identical to the original model. For instance, an infinite FIFO buffer may be replaced with a realistic finite one, without any impact on the system functionality if the data rate on the buffer input does not exceed the limit, which causes the buffer to overflow. Whether the buffer size corresponds to the expected data rate or not can be proved formally. Also the increase in computation time and input/output latency may leave the refined model in the range of acceptable designs according to the design constraints. Thus, the verification task is to show that a refined model is either identical to the original model or satisfies the given design constraints.

4.3 Design Transformations and Verification

The initial system model in the ForSyDe methodology is derived from the system specification. Since this model expresses the ideal functionality of the system, the design correctness at this level can be checked by using simulation techniques. The system model that is defined in the sense of ideal and unlimited resources, does not contain corner-case behaviors caused by overflows and out-of-range computation results. In contrast to lower level models, the input stimuli that are used for simulation of the system model, does not have to target exceptional cases. Therefore, simulation based techniques are appropriate at this level. In further design refinements and verification steps it is assumed that the initial system model behaves correctly. Thus, after verifying that the behavior of an implementation model is equivalent to the system model and it satisfies the design constraint, the design is considered to be correct by construction.

Instead of verifying an implementation model directly against the system model, it possible to divide the complex verification task into smaller subtasks if the design development processes involves only predefined design transformations. Since semantic-preserving transformations do not change the meaning of the model, the verification of models that are refined by this kind of transformations is not necessary. Thus, only these intermediate models have to be verified, which are created by non-semantic-preserving transformations. Since a single transformation involves only small changes, the design correctness check has to target only these particular changes.

4.4 Verification after Non-semantic-preserving Refinements

For the sake of efficient verification, every non-semantic-preserving transformation in the design library has to provide a specific verification strategy. This strategy targets exactly those critical properties, which must be checked in the refined model, and leaves the preserved system behaviors unconsidered in this verification step. Therefore, in addition to the design transformation library, the design methodology has to include (1) *the verification property library*, (2) *the stimuli generator library* and (3) *the abstraction technique library* (Figure 4.3). Non-semantic-preserving transformations point to predefined properties in the verification library, which are necessary to verify in the refined model. The abstraction technique library provides abstraction techniques to simplify models before model checking, and stimuli generators generate proper input stimuli to the system block in verification.

4.4.1 Verification Property Library

The verification property library has to contain generic properties for every non-semantic-preserving transformation. After the designer applies a transformation T_i to the model M_i, T_i points to a set of properties that the refined process network PN_j in the model M_{i+1} has to satisfy. The properties in the library are predefined according to the impact of the transformation and target particular details that may violate the system specification and may not satisfy the design constraints. Since the details of a property depend on the actual design constraints, the library includes incomplete property

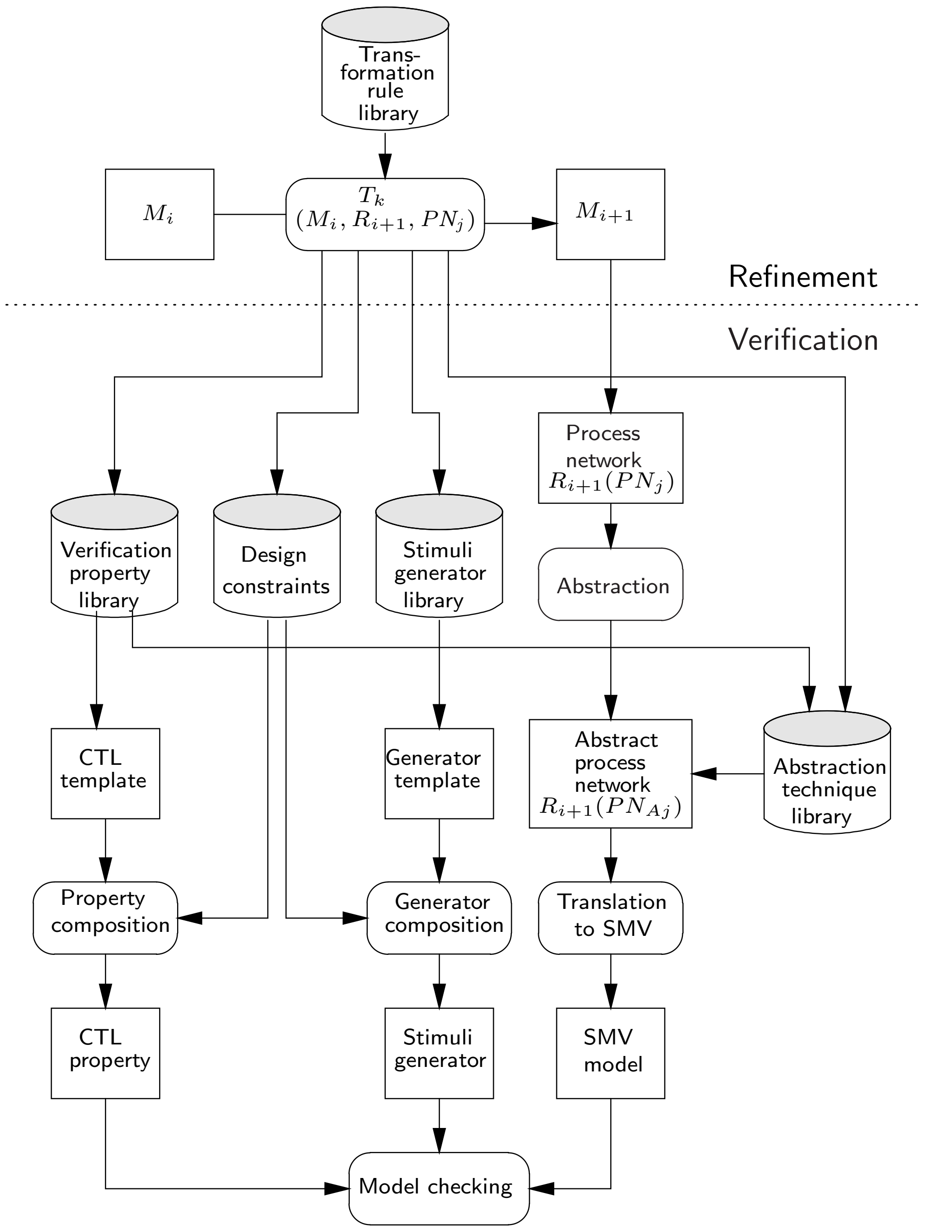

Trans- formation rule library
M_i
T_k (M_i, R_{i+1}, PN_j)
M_{i+1}
Refinement
Verification
Process network $R_{i+1}(PN_j)$
Verification property library
Design constraints
Stimuli generator library
Abstraction
CTL template
Generator template
Abstract process network $R_{i+1}(PN_{Aj})$
Abstraction technique library
Property composition
Generator composition
Translation to SMV
CTL property
Stimuli generator
SMV model
Model checking

Figure 4.3. Verification flow

templates, where parameters are used instead of constant values. It is a designer's task to assign values to template parameters and to finalize properties. Since the Cadence SMV model checker [102] is used for verification of synchronous ForSyDe models, the properties are described as CTL* temporal logic expressions [31].

4.4.2 Abstraction Technique Library

Together with the predefined verification properties the design methodology has to provide abstraction techniques for every specific property in the abstraction technique library. It is important to select proper abstraction methods, since the memory demands and the time spent for verification depends directly on how much the design is simplified. The main idea of the abstraction is to exclude all irrelevant details and to reduce the model's state space. At the same time the relevant system behaviors regarding the verified property have to be preserved in the model. Thus, the application of an unsuitable abstraction technique may remove an erroneous behavior from the model rendering further model checking meaningless.

4.4.3 Stimuli Generator Library

One objective of the abstraction is to limit the number of different input values that are used within the property checking. Since the refined process network is considered separately from the whole system, it is required to provide external input stimuli, which correspond to the verified property [108]. In some cases the stimuli can be expressed in the verification property or be defined as the input data type of the abstract block. For instance, if only a small set of input values is enough to decide the correctness of a block, the abstract input domain can be defined as a new data type in the SMV language. If there is no explicit assignment to this input, the model checker assigns values from the abstract domain to the block's input in a non-deterministic manner. In order to model a sequence of values that varies regularly in time, reconfigurable finite state machine based input stimuli generators have to be provided in the stimuli generator library. The designer can parameterize them to generate specific input patterns according to the verified properties and the design constraints.

4.4.4 Summary

The complete verification flow is presented in Figure 4.3. The transformation rule R_{i+1} refines the process network PN_j of the model M_i into $R_{i+1}(PN_j)$. The transformation rule R_{i+1} points to a set of properties. Each property template refers to an abstraction technique and an input stimuli generator. The property template and the input stimuli generator are completed according to the design constraints and the abstraction technique. The abstraction technique simplifies the refined process network, which is thereafter mapped to the SMV language. Finally, the SMV model checker verifies if the composed property holds in the abstract model, which is connected to the input stimuli generator.

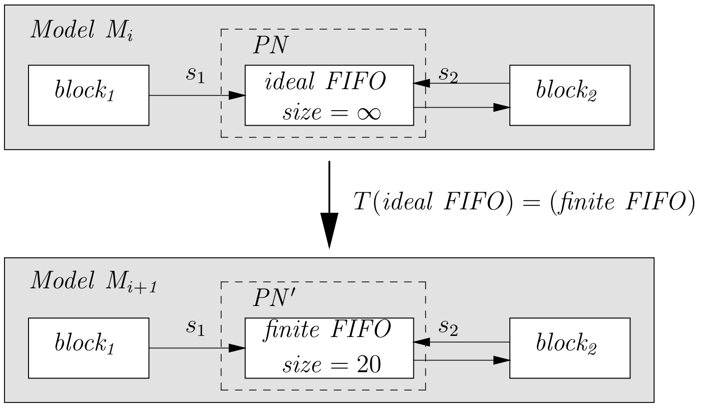


Figure 4.4. Refinement of a FIFO buffer

Figure 4.4 illustrates a non-semantic-preserving design transformation that replaces an ideal FIFO buffer between two design blocks with a finite FIFO. According to the design constraints, *block₁* writes k data items to the buffer within every $k + l$ clock cycles, and *block₂* consumes one item in every second clock cycle. The design transformation points to the property, which has to express that the expected data rate on the FIFO input does not cause the buffer to overflow.

The CTL template of the property is: $\text{AG}(size \leq x)$. In this template the designer has to assign a value to the parameter x, that corresponds to the size of the finite FIFO buffer, and replace the term *size* with the variable name that shows how many elements are stored in the buffer. Let the variable *cnt_FIFO* give the number of elements in the FIFO buffer, and let the maximum number of storage slots in the buffer be equal to 20.

The completed property is: AG($\mathit{cnt\_FIFO} \leq \mathit{20}$). Since the FIFO buffer is not sensitive to the actual values on its input s_1, the abstraction technique replaces the actual input domain with a reduced one that contains only one value. The property points to two input stimuli generators; one produces input data items from the abstract input domain following the input data rate, and the other one sends the read requests every second clock cycle. The abstract model is translated into the SMV language and the SMV model checker verifies if the abstract model connected with the stimuli generators satisfies the given property.

4.5 Discussion

The step by step refinement and verification technique has many advantages. First of all, the designer does not need to invest time to find out critical issues that have to be verified in refined process networks. Instead of formulating the expected system behaviors as complex temporal logic expressions, every non-semantic-preserving transformation points to already defined property templates. These templates can be completed according to the design constraints that requires less designer's effort than specifying and writing the same properties from scratch. In addition, the verification technique points to the suitable abstraction technique that simplifies the model before model checking, and provides stimuli generators to model required input streams from abstract input domains.

Although the creation of the verification property, the abstraction technique and the stimuli generator libraries takes a remarkable amount of time, these libraries have a huge value in the design process. Obviously, it is more efficient if a verification expert creates properties and selects abstraction techniques, than leaving the same work to the designer. Once a transformation is equipped with verification facilities, designers do not need to do the same work over and over again, to figure out after every use of a transformation, how to verify the refined system block. Since one set of properties and abstraction techniques suits many transformations, the introduction of new transformations takes less time if it is possible to adapt already existing properties to the new context.

Due to the limits coming from formal verification tools and a huge variety of possible designs, the proposed verification technique addresses only the local correctness of refined system blocks. In order to find out how a local refinement influences the system in the global sense, static analysis techniques have to be applied. For example, if a refinement decreases the throughput in one block, it has to be ensured that the modification corresponds to the design constraints and the system does not exhibit any deadlock. The global synchronization problems that are caused by local temporal refinements, which increase the latency of refined system blocks, are addressed in Chapters 7 and 8.

The verification technique implies strong requirements on the system specification and constraints. The system specification has to be very detailed, describing the functionality and constraints of each system block. If this information is lacking, the completion of the verification properties is rather impossible, since it is not known what the constraints are that the refined system block has to satisfy.

Chapter 5

Verification of Communication Refinements

5.1 System Level Communication Refinement

The initial high level system description omits all communication caused side effects that may influence the ideal system functionality. Communication between processes and sub-system blocks is performed by synchronous signals and lower level implementation constraints are not taken into account in the system model. These signals are ideal in the sense that no delays nor loss of data occur and thereby they provide reliable data transfer from one computation unit to another. This abstract system view makes it simple to model and analyze the system functionality. Since communication and computation take no time in this model, different system structures may be explored by decomposing and merging computation processes without any changes to the system functional behavior.

Compared to the system model, an implementation model has to be detailed for mapping to software and hardware models and has to satisfy the constraints of the actual communication mechanisms. In general, features such as a noticeable latency, limited bandwidth, loss of data, etc., are apparent parts of actual communication channels. No doubt, these side effects change the system functionality. One possible method is to implement abstract communication channels between system components on a bus architecture. The refinement to a bus architecture, including the selection of a proper bus protocol and the generation of interfaces and drivers according to the constraints on the expected data loads, can be done automatically, as described in [3, 99]. Another approach is to develop the desired communication mechanism by a series of refinement steps at different levels of abstraction. The parameters of the communication mechanism have to be configured according to the expected data rate, so that after refinements an implementation model satisfies the system specification and the design constraints [80, 35]. For example, the ideal and perfect communication channel in the system model can be replaced with a channel, which implies an acceptable latency. However, it is required that the receiver side computation block has some flexibility and does not behave unexpectedly if there are delays in the data arrival. Similarly, an ideal buffer on the sender side of a channel, which stores data items when the channel is busy with a data transfer, can be replaced with a realistic finite buffer. The replacement is valid if the buffer is large enough to operate with the expected data rate from the sender side, which is specified according to the design constraints.

The lower level communication details are introduced by communication refinements in the ForSyDe methodology. According to their effect, the design transformations that increase the latency of communication channels and introduce potential loss of data are classified as non-semantic-preserving transformations. Although the behavior of a refined model may be expected to be equivalent to the initial system model, it is crucial to verify that the model actually satisfies the design constraints.

General properties for verification of refined communication channels are developed in the following sections. The properties target reliability, latency, bandwidth and order preserving issues. The verification of communication refinements is explained in the context of a refinement, described in [92] that replaces a synchronous channel with an asynchronous one, which implements a handshake protocol. This transformation introduces limited bandwidth and possible lossy data transfer due to a limited number of slots in the channel buffer, and a noticeable latency increase.

5.2 Refinement from a Synchronous into an Asynchronous Channel

The refinement of a synchronous channel into an asynchronous channel, which implements a handshake communication protocol, involves four models at different levels of abstraction, as illustrated in Figure 5.1.

In the initial model (Figure 5.1.a) communication between two subsystem blocks is performed over an ideal synchronous channel, which has no

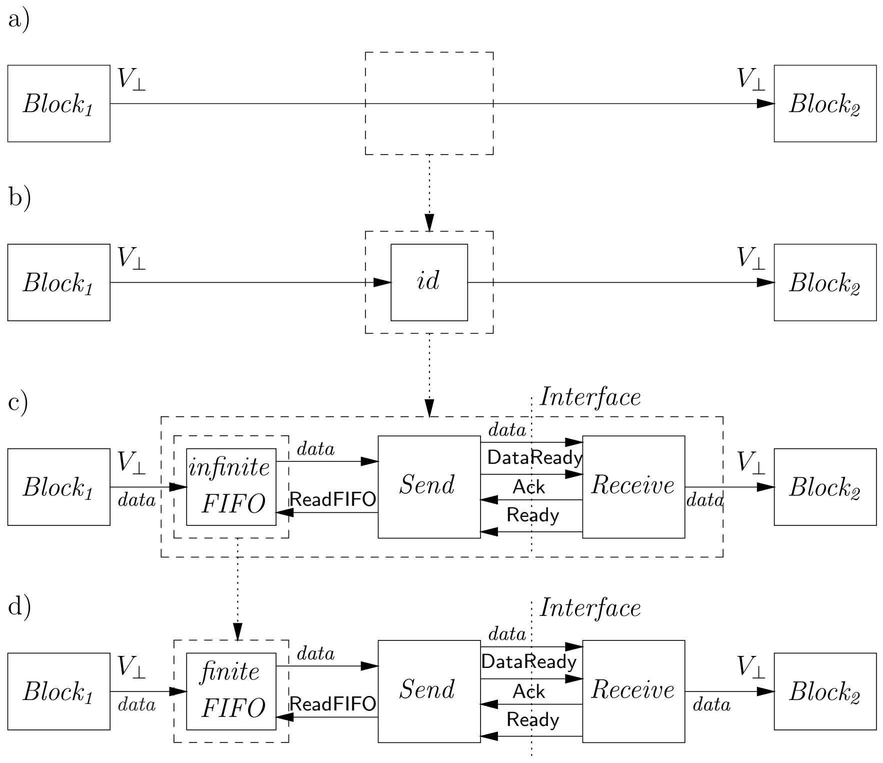


Figure 5.1. Refinement into a handshake protocol

delay and where both blocks operate at the same clock frequency. A prerequisite for this communication refinement is that the data type of the channel must be absent extended ($V_{\perp}$). The refinement is done in three steps. The first step, classified as semantic-preserving, introduces an identity process between *Block₁* and *Block₂*, and the second step, non-semantic-preserving, refines this process into a handshake protocol implemented by three processes: *infinite FIFO buffer*, *Send* and *Receive* (Figure 5.1.c). *Send* and *Receive* are Moore state machine processes, whose state transition diagrams are presented in Figures 5.2 and 5.3, respectively. States are drawn so that the identifier of a state is shown above the horizontal line, and the output values in this state are below the line. A missing output value in a circle means that no present value is assigned to the output signal in this state and the process emits *Abst* value instead. For example let's consider Figure 5.3, when the process *Receive* is in the state *WaitData* it announces the message Ready to *Send* and the rest of the output signals hold the *Abst* values in this clock cycle. The arcs between states are labeled with input values that trigger certain transitions. Similarly, the missing labels on arcs are considered as the *Abst* values in the state transition graphs.

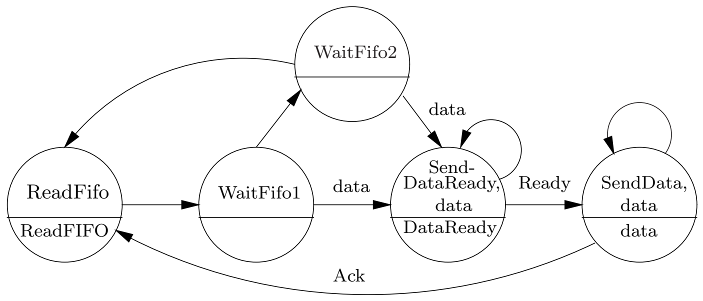


Figure 5.2. State flow graph of the process *Send*

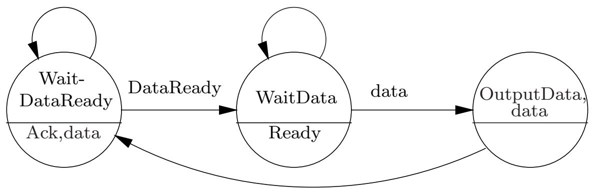


Figure 5.3. State flow graph of the process *Receive*

The refined channel transfers only present values from *Block$_1$* to *Block$_2$*. When *Receive* has no present data item to deliver to *Block$_2$* it sends an *Abst* value. The handshake protocol is defined so that blocks on different sides of the channel can operate at different clock frequencies. The protocol operates as follows. When *Send* is idle it tries to read data from *FIFO* by sending the message ReadFIFO to *FIFO*. If the data arrive from *FIFO*, *Send* starts a data transmission over the interface, and emits the message DataReady to *Receive*. After that *Send* waits for the reply message Ready from *Receive*. If this message arrives it means that *Receive* is ready for the data transfer. When Ready appears, *Send* transmits the data until *Receive* answers with an acknowledgment Ack, about the data arrival.

The behavior of the channel in Figure 5.1(c) is clearly different from the original model, since the communication protocol between *Send* and *Receive* performs a number of steps when transporting a data item. Therefore the latency in the number of clock cycles has increased. In addition, the handshake channel can transport only one data item at a time, which decreases the channel bandwidth. However, due to the ideal FIFO buffer, none of the data items waiting for transfer are lost. On the other hand, at a lower abstraction level this ideal buffer has to be replaced with a finite one, as shown in Figure 5.1(d), which may cause the loss of data.

5.3 Properties of Communication Channels

This section illustrates the development of verification properties for the communication transformation that is presented in Figure 5.1. These are general properties, and they can be used for verification of other communication channels as well. However, in practice, additional properties have to be defined to cover all specifics of each particular transformation.

The main objective of the verification properties is to check that the characteristics of the selected handshake protocol and the size of buffers correspond to the data rate on the channel input. The properties are defined as temporal logic expressions that can be verified by model checkers. In order to shorten the verification time and to reduce the memory requirements of the model checker, it is necessary to simplify the model by using appropriate data abstraction techniques. Comparing the models in Figure 5.1(a) and 5.1(d) it can be noticed that in opposite to the synchronous channel, the asynchronous channel is sensitive to the data type of the input values. However, the channel's behavior depends only on the types of input events, and the actual values of present events are processed in the same way. Therefore, the present data values on the channel input can be modeled as abstract data items during verification. However, depending on each actual property, the input domain has to contain a different number of present values.

5.3.1 Property 1: Reliability

Reliability is a highly important property of every communication channel. It requires that all data values, in the current context present values, appearing to the channel input will eventually be transferred to the channel output. Since the asynchronous implementation of the channel includes a sender side FIFO buffer, which has a limited storage capacity, it is obvious that any sufficiently high input data rate will cause buffer overflow and therefore the loss of data. In order to verify that a data item is lost only when the buffer limits are exceeded, the following property has to be checked: *"if there is at least one empty slot in the FIFO buffer when a data item is* *entering the channel, then eventually the data item will be transmitted on the channel output"*. If the property holds, it shows that the data can only be lost by the buffer overflow. The CTL* specification of this property is:

$$\begin{aligned}&\mathrm{AG}((\mathit{fifo\_size} < \mathit{SIZE}-1 \,\&\, \mathit{ch\_input.Con} = \mathit{Prst} \,\&\, \mathit{ch\_input.Val} = 0\,) \\ &\quad -> \mathrm{AF}\ (\mathit{ch\_output.Con} = \mathit{Prst} \,\&\, \mathit{ch\_output.Val} = 0))\end{aligned}$$

The terms *ch_input* and *ch_output* refer to the channel data input and output, respectively. The variable *fifo_size* represents the current number of elements in the FIFO buffer. The constant *SIZE* is defined as the maximum number of elements the FIFO buffer can store. Instead of the actual data type, the property uses integers as an abstract data type. The given property can be read as saying that always, if the FIFO buffer has at least one empty slot and the channel input holds the value *Prst* 0, then always eventually the channel emits the value *Prst* 0. If this property holds for the value *Prst* 0, then it holds for any other present value as well, since the channel behavior does not depend on the integer value on the channel input. In order to distinguish the value *Prst* 0 on the channel input from any other values that may be stored in the buffer, the abstract input domain has to include at least one more present value, for example *Prst* 1. For the verification of this property a model checker can assign non-deterministically any sequence of values from the respective data domain to the channel input. The input stimuli generator in the SMV language for the reliability property is the following:

```
MODULE genPropR(){
  o1 : Abst_Int_0_1;
};
```

The data type *Abst_Int_0_1* includes the absent value *Abst* and the integers *Prst 0*, *Prst 1*. For the data type definitions in SMV, see Section 3.5.2.

5.3.2 Property 2: Latency

As discussed above, the handshake protocol implies a delay. According to the simulation of the channel in the synchronous model, where both the sender and receiver side blocks operate at the same clock frequency, the channel delay is seven clock cycles. More precisely, it takes seven cycles to transport a data item from the channel input to the channel output if the process *Send* is in the initial state at the moment when the data item enters the channel. In order to verify that this is always true, the channel has to be verified against the following property:

$$\begin{aligned}&\text{AG } ((\mathit{input.Con} = \mathit{Prst}\ \&\ \mathit{fifo\_size} = 0\ \&\\ &\quad \mathit{Send\_state} = \mathit{ReadFifo})\\ &\quad -> \ (\text{AX AX AX AX AX AX AX } \mathit{output.Con} = \mathit{Prst}))\end{aligned}$$

In the given statement the variable *Send_state* represents the state of *Send* and the initial state of the *Send* process is *ReadFifo*. The given property says, if a present value appears to the channel input when the FIFO buffer is empty and *Send* is in the state *ReadFifo*, then after seven clock cycles the present value is transferred to the channel output. The same input stimuli generator as for the latency property can be used, although here it is sufficient to have only one present value in the channel input data domain. This is equivalent to the generator below:

```
MODULE genPropL(){
  o1 : Abst_Int_0;
};
```

5.3.3 Property 3: Bandwidth

Bandwidth is a relevant parameter of communication channels that can be defined by the maximum number of data items that a channel is able to transfer from the input to the output within a fixed time interval. For example, the previously defined latency property states that the data transfer through the channel takes a certain number of clock cycles - seven cycles for the particular model in Figure 5.1(c). Apparently, it is valid to predict that if within every seven clock cycles at most one present value arrives at the channel input, then all data are transferred over the channel.

A configurable stimuli generator that produces a sequence of frames, where every frame of length *frame_size* contains at most *max_P* present values, is given below:

```
MODULE $genPropB$ (){
 $data$ : $\{Abst, Prst\}$;
 $cnt\_P : 0 .. max\_P$;
 $cnt\_F : 0..frame\_size$;
 $o\_1 : Abst\_Int\_0$;
 init($cnt\_F$) := 0;
 next($cnt\_F$) := $(cnt\_F + 1)$ $mod$ $(frame\_size + 1)$;
 init($cnt\_P$) := 0;
 next($cnt\_P$) := case{$cnt\_F < frame\_size$ & $data = Abst : cnt\_P$;
                     $cnt\_F < frame\_size$ & $data = Prst : cnt\_P + 1$;
                     $cnt\_F = frame\_size : 0$;
                    };
 $data$ := case{$cnt\_P < max\_P : \{Abst, Prst\}$;
             default : $Abst$;
            };
 $o_1.Con := data$;
 $o_1.Val := 0$;
}
```

The generator includes two counters: *cnt_F* to count the length of a frame, and *cnt_P* to keep track of how many present values the current frame already contains. If the number of present values in a frame has not reached its upper limit the variable *data* may get assignments *Abst* or *Prst*. The value of *data* determines the output values of the generator. The constructor based variable o_1 contains two components: *o_1.Con* for the absent and present extension and *o_1.Val* carrying the abstract integer value 0.

If the input stimuli by *genPropB* causes overflow of stored values that are waiting for transmission in the FIFO buffer, then the input load is too high for the channel or the FIFO size is not properly dimensioned. In order to verify that the channel is able to provide lossless data transfer for a specified input data rate, the following property has to be checked:

$$\text{AG } (\mathit{fifo\_size} \ \leq \ \mathit{SIZE})$$

Coming back to the assumption about the channel in Figure 5.1(d) that the data transfer through the channel takes seven clock cycles, an input signal where each frame with seven events contains at most one present event should not cause the buffer to overflow. In contrast to the assumption, the SMV model checker finds that the proposed property is not true, and gives a trace of transitions, which leads to a state, where the property is violated. After increasing the frame length from seven to nine, the model checker reports that the new property is satisfied. The former property did not hold because after the process *Receive* has received one data item it takes two clock cycles for the process *Send* to ask for the next data item. Thus, the prediction about the channel bandwidth was misleading. The wrong presumption is a typical mistake, which shows that in order to validate a system, it is essential to incorporate formal techniques, instead of using only designer intuition or simulation techniques.

5.3.4 Property 4: Order

Systems in ForSyDe are described by using the synchronous model of computation, where synchronous signals establish the order of events. In the model in Figure 5.1(a), *Block$_2$* receives all data items in the same order as *Block$_1$* sends them to the channel. To ensure that after the communication refinement the *Receive* side subsystems (Figure 5.1.d) process present events in the initial order, present values can be equipped with distinct and ordered indices, that can be used to recover the order of the input values. On the other hand it can be verified whether the asynchronous channel always preserves the order of present values by itself or not. An abstract input stimuli generator for the verification of the order property is the following:

```
MODULE $genPropO()${
  $o1 : Abst\_Int\_0\_2;$
  $val : Int\_0\_2;$
  $con : AbstExt;$
  $init(con) := Abst;$
  $next(con) := \{Abst,\ Prst\};$
  $init(val) := 0;$
  $next(val) := case\{$
                    $con = Prst\ :\ (val + 1)\ mod\ 3;$
                    $con = Abst\ :\ val;$
                    $\};$
  $o1.Con := con;$
  $o1.Val := val;$
}
```

The module generates an input signal to the channel by preserving the order *Prst 0*, *Prst 1*, *Prst 2*, *Prst 0*, ... and adding an arbitrary number of absent values between the present values.

The two latest present values are always stored in the following observer module that has to be connected to the channel output.

```
MODULE $observer(i1)${
  $st1 : Int\_0\_2$;
  $st2 : Int\_0\_2$;
  $init(st1) := 0$;
  $init(st2) := 0$;
  $next(st1) := case${
                     $i1.Con = Prst \;:\; i1.Val$;
                     $i1.Con = Abst \;:\; st1$;
                     };
  $next(st2) := case${
                     $i1.Con = Prst \;:\; st1$;
                     $i1.Con = Abst \;:\; st2$;
                     };
}
```

The order property has the following form:

$$\mathrm{E}((fifo\_size < (SIZE - 2)\mathrm{U}(observer.st1 = 1 \;\&\; observer.st2 = 2))$$

The property says that there exists a computation path, where in each state the FIFO does not store more than $(SIZE - 2)$ values, until the observer has saved an output sequence, where the next present value after *Prst 2* is *Prst 1*. A channel does not preserve the order of input values, if the property is satisfied. *Prst 2* always comes after *Prst 1* in the generated input sequence, and the property cannot hold in a correctly behaving channel. In order to exclude the scenario that a *Prst 2* gets lost due to the FIFO overflow, it is assumed that there are always at least two empty slots in the buffer. Since the channel behavior does not depend on the exact values of the present events on the channel input, this property holds for any input sequence of present values.

5.4 Discussion

Similarly to the Open Verification Library [41] that provides assertions to verify RT-level designs, the collection of properties defined in this chapter can be viewed as a part of a library that targets the verification of communication channels. Although these properties are defined regarding one specific design transformation and an asynchronous channel, they capture general behaviors of communication channels. For any particular transformation that creates some other channel, the given properties can be used as templates to derive new and suitable verification properties. A clear benefit of this kind of library is that the designer does not need to spend time for figuring out which kind of properties of the refined model should be verified and how to express them as temporal logic expressions. After finding that one certain handshake protocol together with the limits on the buffer size cannot satisfy the data rate on the channel input, another protocol may be introduced and the same properties checked again. Although predefined properties simplify the design analysis process, the designer has the task to estimate the data rate on the channel input, if the expected data rate is not given in the design specification. Obviously this estimation is decisive, since wrong assumptions lead to a faulty design implementation.

Chapter 6

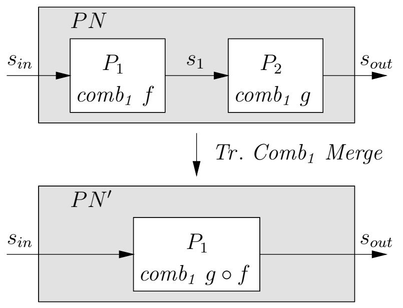
Verification of Computation Refinements

6.1 Introduction

The design of arithmetic computation blocks at a high abstraction level starts with describing their functionality by using ideal arithmetic operators. Since the main focus at this level is on the system functionality, the system structure is secondary. The idea of the design refinement process is to derive a final implementation where arithmetic blocks have an optimal granularity and the computation load is efficiently shared among sub-blocks.

The simplest transformations that change the structure of computation blocks are the *splitting and merging* of combinational processes. For example, a design transformation that merges two combinational processes with one input and one output is presented in Figure 6.1. The transformed process that applies the sequential composition $(g \circ f)$ of combinational functions f and g to input values is semantically equivalent to the composition of processes that apply the same functions sequentially. Therefore, verification in the transformed process is not essential, since the applied design transformation is semantic-preserving. The counter part to combinational merge is a split transformation that decomposes the computation into several processes. A more advanced transformation introduces resource sharing in arithmetic computation blocks as illustrated in Figure 6.2.

The main idea behind resource sharing is to reuse computational resources. Instead of implementing an arithmetic function as a large combinational circuit, it can be mapped to a smaller sequential design, where



Transformation : $Comb_1Merge(PN)$
Original process network :
$s_{out} = PN\ s_{in}$
where
$s_1 = comb_1\ f\ s_{in}$
$s_{out} = comb_1\ g\ s_1$

Transformed process network :
$s_{out} = PN'\ s_{in}$
where
$s_{out} = comb_1\ (g \circ f)\ s_{in}$

Figure 6.1. Transformation: merging of single input combinational processes

computation is distributed in time. In the sequential design a controller executes the computation in a data path and a register file stores intermediate computation results. For example, the functionality of many digital signal processing applications is expressed as a polynomial or a rational function that can be split into smaller data path operations [75].

Figure 6.2 illustrates the resource sharing refinement in an n-th order FIR filter[1] [79]. In the initial system model (Figure 6.2.a) the filter is described as a composition of a shift register and a combinational process, which calculates the polynomial function $f(d,c) = \sum_{i=1}^{n}(c_i * d_i)$ over the filter coefficients c_i and the i-cycles delayed input data values d_i. Instead of the direct form realization in Figure 6.2(b) that includes n multipliers and $n-1$ adders, the polynomial function f can be implemented on a sequential design, which uses only one adder and one multiplier in Figure 6.2(c). The sequential model contains two additional signals *start* and *ready*. *Start* triggers the controller to begin computation and *ready* notifies when the computation result is available on the output. The controller is configured so that it writes the value *zero* to the internal register *reg* at the first clock cycle after high *ready*. At each of the following clock cycles the data path calculates a term $c_i * d_i$, and sums the terms together in the register.

In contrast to the simple semantic-preserving merging and splitting transformations, the resource sharing transformation is clearly non-semantic-

[1]Finite Impulse Response filter

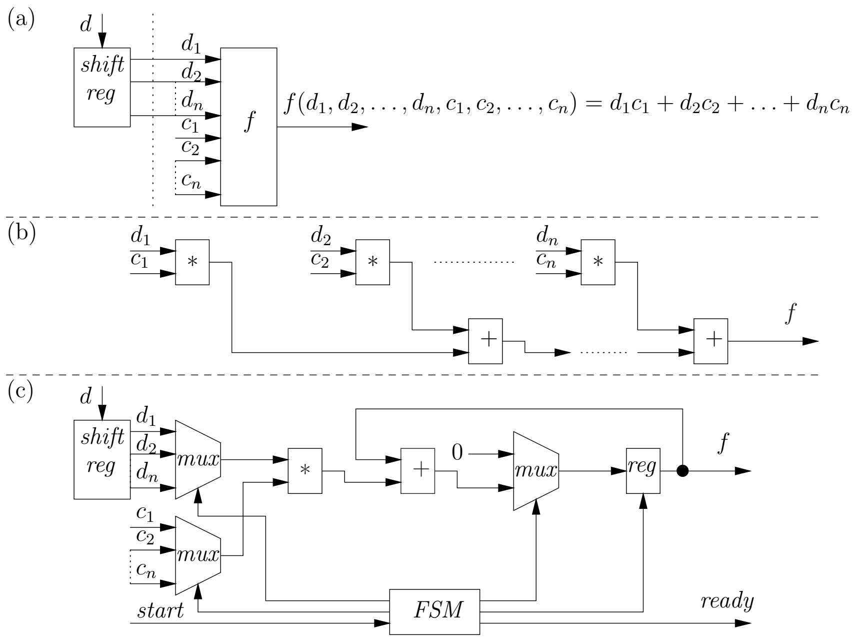


Figure 6.2. Resource sharing in an n-th order FIR filter

preserving. Compared to the original model, where computation takes no time, the sequential design has a noticeable delay due to the feedback loop. Since the additional signals *start* and *ready*, and the functionality of the controller are not defined in the design specification, the correctness of the refined block can be checked only based on the block's input/output functionality.

One method to verify that the sequential block behaves correctly is to apply symbolic execution [52, 53], which analyzes the computation in the data path according to the instructions from the controller and finds the functionality of the block in terms of symbolic input values. In the ideal case the functionality has a polynomial form that can be compared against the polynomial specification of the original block. However, the controller may be configured to run different micro programs, which are selected according to the values on the controller inputs. For example, instead of calculating always multiplication $d_i * c_i$, the controller may perform more optimal operations: $d_i * c_i = 0$ if $d_i == 0$, $d_i * c_i = c_i$ if $d_i == 1$, $d_i * c_i = c_i + c_i$ if $d_i == 2$. Thus, different micro programs may calculate polynomials, which have formats that are very different from the polynomial function of the system specification. An alternative is to use symbolic execution to find the maximum possible degrees of input variables in the output polynomials without finding the exact polynomials. Based on the fundamental theorem of algebra [39] a finite number of input vectors can be used to decide if the sequential implementation calculates the same function as defined by the polynomial specification. The number of required input vectors is determined by the maximum degrees of input variables in the specification polynomial and in the functions calculated by the sequential implementation. This approach, denoted as *polynomial abstraction*, allows to replace the infinite domains of input variables of a sequential design with finite small ones and to use a model checker for verification.

6.2 Polynomial Abstraction

The polynomial abstraction technique is developed for the verification of sequential designs, whose functionalities are expressed as polynomial or rational functions by using model checking. It is also applicable for the verification of two sequential models against each other, if they implement polynomial functions. A sequential implementation (Figure 6.3) may operate on

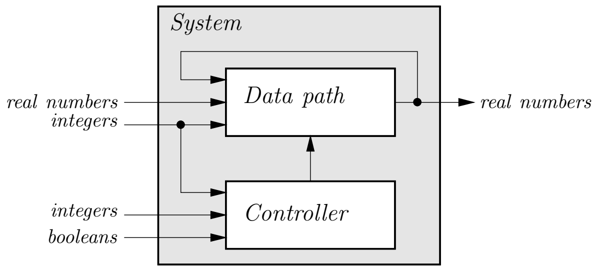


Figure 6.3. System structure

real numbers, integers and Booleans, where the real numbers or integers are the operands in the polynomial specification. Data paths may include operations that are created by combining the ideal functions $+, -, *, /$. Similarly to the polynomial specification, the output values of the sequential design are not limited to any fixed bit-widths. The verification technique requires that data path operations are correct, or their correctness is checked separately.

The objective of the abstraction technique is to minimize the input domains of the data path input signals and thereby to reduce the size of the sequential model. Polynomial abstraction cannot reduce the domains of input signals, whose values are used in computations in the data path if the computation result determines the controller behavior. The domains of this kind of input signals have to be finite and may contain only integer values. In other words, feedback signals directed from the data path to the controller are forbidden. If this kind of signal exists, it is considered as a part of the controller. Despite that, the controller behavior may be determined by integer input signals, if the values on inputs are compared to constants in equality expressions. For example, the data value d_i in the FIR filter can be directly compared to the values 0 and 1 in order to choose between different computation paths, which correspond to different polynomial functions. The abstraction technique requires that the system calculates only a finite number of different polynomial functions. Each combination of input values has to correspond to a certain finite sequence of data path commands executed by the controller.

6.2.1 Theoretical Background

The polynomial abstraction technique relies on the fundamental theorem of algebra. The theorem is used in [87] to show that a finite set of input assignment vectors can be used to find whether two multi-variable polynomials are identical or not. The number of different values assigned to each individual input depends on the maximum degree of the respective input variable in these polynomials. Two polynomials $P_a(x) = A_k x^k + A_{k-1} x^{k-1} + \ldots + A_0$ and $P_b(x) = B_k x^k + B_{k-1} x^{k-1} + \ldots + B_0$ are identical if all their respective coefficients are equal, i.e., $\forall i, (0 \leq i \leq k), A_i = B_i$. In this case the coefficients of their difference polynomial $P_c(x) = (P_a(x) - P_b(x))$ are equal to zero. The fundamental theorem of algebra says that a degree k uni-variable polynomial has exactly $k+1$ complex roots unless all of its coefficients are equal to zero. For clarification, the value v is a root of $P(x)$ if $P(v) = 0$. If $P_a(x)$ and $P_b(x)$ calculate pairwise the same result for $k+1$ input assignments, then according to the fundamental theorem of algebra the coefficients of the difference polynomial $P_c(x)$ are equal to zero and the polynomials $P_a(x)$ and $P_b(x)$ are identical. In [87] it is proven that the same decision mechanism is valid for multi-variable polynomials as well. The identity checking can be performed by using integer values that are special cases of complex numbers. Two multi-variable polynomials $P_a(\overline{x})$ and $P_b(\overline{x})$ are identical if they calculate pairwise the same result for all combinations of input assignments, where x_i ($x_i \in \overline{x}$) gets $k+1$ distinct values and the maximum degree of x_i in $P_a(\overline{x})$ and $P_b(\overline{x})$ is k.

If a sequential design implementation calculates function $I(\overline{x})$ and its specification is defined by function $S(\overline{x})$, a bounded number of different values, determined by the maximum degrees of variables in these polynomials, can be used to verify if $I(\overline{x}) == S(\overline{x})$. The given theoretical background allows to reduce the infinite domains of real number input signals to finite integer ones for model checking. Thus, the behavior of the sequential design block can be checked by following the block's input/output functionality.

Although the set of input values, used for verification is reduced and finite, some values may still grow quite large in the data path. For instance, the degree of x in $P(x) = x^{10}$ is 10 and an implementation of this function can be verified by only eleven different input values $(0, 1, \ldots, 10)$. The greatest value calculated by the function $P(x) = x^{10}$ is $P(10) = 10^{10} \approx 2^{33}$, which means that a model checker has to create 33 binary variables to store the function values. In order to reduce the memory demands, an additional state space reduction has to be performed. Based on the arithmetic identity presented in (6.1) the following Theorem 6.2, allows to verify a set of reduced models instead of the initial model, which calculates larger values. Theorem 6.2 is equivalent to the Chinese remainder theorem used in [33].

Let $\mathbb{Z}$ denote the set of integers and $\mathbb{Z}^+$ denote the set of positive integers.

$$\begin{aligned} ((a\ mod\ n_i)\ OP_j\ (b\ mod\ n_i))\ mod\ n_i\ &== (a\ OP_j\ b)\ mod\ n_i\ | \\ OP_j \in \{+,-,*\},\ n_i \in \mathbb{Z}^+\ and\ a,b \in \mathbb{Z} & \end{aligned} \tag{6.1}$$

Theorems on Modulo Calculation

Let the function $rpn(x,y)$ evaluate to one, if x and y are relatively prime numbers, and to zero otherwise. Two or more numbers are relatively prime if they have no common integer divisor other than one. Let $\mathbb{Z}^i$ consist of all possible i-component tuples from the set $\mathbb{Z}$.

Theorem 6.1 *Two polynomial functions $S(\overline{x})$ and $R(\overline{x})$ calculate pairwise the same results v_j on the input assignments $\overline{v}$ ($\overline{v} \in \mathbb{Z}^i$), if:*

$$\begin{aligned} v_j \in [v_0, v_0 + n_1 n_2[, \\ n_1, n_2 \in \mathbb{Z}^+\ and\ rpn(n_1,n_2) = 1, \\ S(\overline{v})\ mod\ n_1 = R(\overline{v})\ mod\ n_1, \\ S(\overline{v})\ mod\ n_2 = R(\overline{v})\ mod\ n_2 \end{aligned} \tag{6.2}$$

The theorem says that if two polynomial functions $S(\overline{x})$ and $R(\overline{x})$ extended with modulo computations ($mod\ n_1$ and $mod\ n_2$) calculate pairwise the same result on a set of input assignments, then $S(\overline{x})$ and $R(\overline{x})$ calculate pairwise identical values on the same input assignments, if these values v_j belong to the domain $[v_0, v_0 + n_1 n_2[$. In other words, if the conditions in (6.2) holds for the assignment $\overline{v}$, then two different values v_1 and v_2 do not exist in the domain $[v_0, v_0 + n_1 n_2[$ such that $S(\overline{v}) = v_1$, $R(\overline{v}) = v_2$.

Proof

Let's make a contradictory assumption that there exists an assignment $\overline{v}$ such that $S(\overline{v})\ mod\ n_1 = R(\overline{v})\ mod\ n_1$, $S(\overline{v})\ mod\ n_2 = R(\overline{v})\ mod\ n_2$, $S(\overline{v}) = v_1$, $R(\overline{v}) = v_2$, but $v_1 \neq v_2$ and $(abs(v_1 - v_2)) < n_1 n_2$.

According to the conditions in (6.2):

$$\begin{aligned} v_1\ mod\ n_1 &= v_2\ mod\ n_1 \\ v_1\ mod\ n_2 &= v_2\ mod\ n_2 \end{aligned} \tag{6.3}$$

Based on (6.1) the equations in (6.3) can be written in the following form:

$$\begin{aligned} (v_1 - v_2)\ mod\ n_1 &= 0 \\ (v_1 - v_2)\ mod\ n_2 &= 0 \end{aligned} \tag{6.4}$$

Since the results of the modulo computations in (6.4) are equal to zero, there exist integer constants c_1 and c_2 so that:

$$\begin{aligned} (v_1 - v_2) &= c_1 n_1 \\ (v_1 - v_2) &= c_2 n_2 \end{aligned} \tag{6.5}$$

From the equations in (6.5), it can be derived that $c_1 n_1 = c_2 n_2$, which can be written in the following form:

$$\frac{c_1}{c_2} = \frac{n_2}{n_1} \tag{6.6}$$

According to the contradictory assumption, v_1 is not equal to v_2, and referring to (6.5), c_1 and c_2 cannot not be equal to zero. Since n_1 and n_2 are relatively prime numbers and c_1 and c_2 are integers, the least values that satisfy the equation in (6.6) are $c_1 = n_2$ and $c_2 = n_1$. In this case

$v_1 - v_2 = n_1 n_2$ and two different values v_1,v_2 cannot belong to the same domain $[v_0, v_0 + n_1 n_2[$. Based on the latter it is valid to deduce the correctness of Theorem 6.1.

□

Relatively prime numbers have the following property: if $rpn(n_1, n_2) = 1$, $rpn(n_2, n_3) = 1$ and $rpn(n_1, n_3) = 1$, then $rpn((n_1 n_2), n_3) = 1$. Based on the previous property, Theorem 6.1 can be extended in the number of relatively prime numbers.

Theorem 6.2 *Two polynomial functions $S(\overline{x})$ and $R(\overline{x})$ calculate pairwise the same result v_r on the input assignments $\overline{v}$ ($\overline{v} \in \mathbb{Z}^i$), if:*

$$\begin{aligned}\forall i, (1 \leq i \leq m), n_i \in \mathbb{Z}^+,\\ \forall i, j (1 \leq i, j \leq m \ and \ i \neq j), rpn(n_i, n_j) = 1,\\ v_r \in [v_0, v_0 + \prod_{i=1}^{m} n_i[,\\ \forall i, (1 \leq i \leq m), S(\overline{v}) \ mod \ n_i = R(\overline{v}) \ mod \ n_i\end{aligned} \tag{6.7}$$

Proof

The theorem can be proven by induction on the number of relatively prime numbers n_i.

Base Case. The proof of the base case ($i = 2$) follows directly from Theorem 6.1.

Induction Step. In order to prove that Theorem 6.2 is valid for m relatively prime numbers, let's assume that it holds for $m - 1$ relatively prime numbers $(n_1, \ldots, n_{m-1})$:

$$S(\overline{v}) \ mod \ (\prod_{i=1}^{m-1} n_i) = R(\overline{v}) \ mod \ (\prod_{i=1}^{m-1} n_i) \tag{6.8}$$

Let n_m be a positive integer, so that $\forall i, (1 \leq i \leq (m-1)), rpn(n_i, n_m) = 1$. In this case $rpn((\prod_{i=1}^{m-1} n_i), n_m) = 1$. If $S(\overline{v}) \ mod \ n_m = R(\overline{v}) \ mod \ n_m$ and the condition in (6.8) holds then according to Theorem 6.1 $S(\overline{v}) \ mod \ (\prod_{i=1}^{m} n_i) = R(\overline{v}) \ mod \ (\prod_{i=1}^{m} n_i)$.

□

Based on the previous theorem, the verification of the example polynomial $P(x) = x^{10}$ that requires 33 bits to represent the largest value, can be replaced with the verification of eleven much smaller models calculating the functions $P(x) = (x^{10})\ mod\ n_i$, $n_i \in \{2, 3, 5, \ldots, 31\}$. The multiplication of the first eleven prime numbers $\{2, 3, 5, \ldots, 31\}$ is $\prod_{i=1}^{11} n_i \approx 2^{37}$ that is greater than 2^{33}. The largest value $31 = 2^5$ appearing in these models can be represented with only 5 bits compared to the original 33 bits.

6.2.2 Roadmap of the Polynomial Abstraction Technique

The main steps of the abstraction technique are presented in Figure 6.4. A design transformation replaces the combinational block with a sequential design implementation and the task for verification is to find whether the sequential design implements the functionality that is described by the original combinational block. The computation in the data path may include division operations that involve real numbers. Since model checkers can operate only with integer values, the first step of the technique is to map the refined model to a fractional model. Real number data signals are replaced with pairs of integer signals in the fractional model. A similar replacement is done for storage elements. Arithmetic operators in this model are replaced with fractional ones, which instead of calculating division store the result as a fraction of two integers.

The algorithm classifies the input signals of the fractional model as data and control signals. The domains of data signals can be reduced by data abstraction. The algorithm finds the maximum possible degrees of data input variables in polynomial functions, which represent the functionality of the sequential design, and specifies finite domains to the input signals according to these degrees. In the next step the algorithm estimates the maximum values that appear in the data path by using a domain propagation method. In order to decrease the bit-widths of data path registers, a number of smaller abstract models are created by applying the modulo theorem. These abstract models are mapped to the SMV language and the SMV model checker verifies whether they calculate the same results.

6.2.3 Fractional Model

The SMV model checker supports division on integers but the result is always stored as an integer value. According to the scope of the verification technique, a design may have real number data inputs and division operations on data signals. Even if the abstract domains of the input variables are composed so that they contain only integer values, the result of division may still be a real number. In order to avoid real number values calculated in the

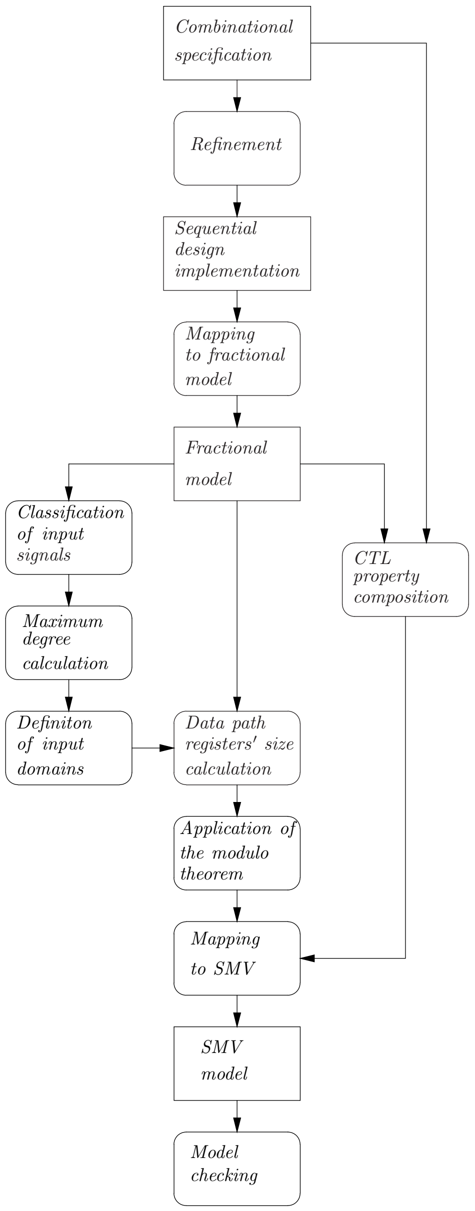
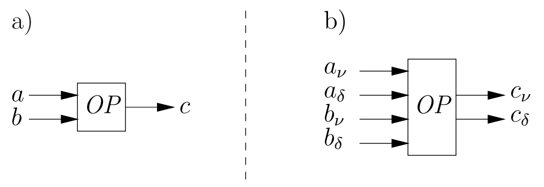

Combinational specification
Refinement
Sequential design implementation
Mapping to fractional model
Fractional model
Classification of input signals
Maximum degree calculation
Definiton of input domains
CTL property composition
Data path registers' size calculation
Application of the modulo theorem
Mapping to SMV
SMV model
Model checking

Figure 6.4. Roadmap of the polynomial abstraction technique

data path, the refined sequential model is mapped to a *fractional model*. In the fractional model all data values v that are results of division operations are modeled as fractions $\frac{v_\nu}{v_\delta}$ of respective numerator and denominator values v_ν and v_δ. Since the model operates with fractional values, all original arithmetic functions have to be replaced with the equivalent fractional functions as shown in Figure 6.5(b).



OP	c_ν	c_δ
$c = a + b$	$a_\nu * b_\delta + b_\nu * a_\delta$	$a_\delta * b_\delta$
$c = a - b$	$a_\nu * b_\delta - b_\nu * a_\delta$	$a_\delta * b_\delta$
$c = a * b$	$a_\nu * b_\nu$	$a_\delta * b_\delta$
$c = a/b$	$a_\nu * b_\delta$	$b_\nu * a_\delta$

Figure 6.5. Operations a) in the original model b) in the fractional model

Assertion 6.1 *If $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$ and $OP \in \{+, -, *\}$ and $c = (a\ OP\ b)$, then $c \in \mathbb{Z}$.*

If all arithmetic functions in the data path are mapped to the functions presented in Figure 6.5 and the input stimuli include only integers, then based on Assertion 6.1, all data signals carry only integer values, since all values are calculated by arithmetic operations $\{+, -, *\}$. The relation between the original and the fractional models are:

1. all data signals s_i in the original model are modeled as pairs of signals $s_{\nu i}$ and $s_{\delta i}$ in the fractional model. The value v_i on signal s_i is equal to the division of $v_{\nu i}$ and $v_{\delta i}$, which are respective values on $s_{\nu i}$ and $s_{\delta i}$;

2. arithmetic operations in the original model are replaced with fractional operations according to Figure 6.5;

3. all registers, multiplexers and other supporting elements connected with data signals s_i, are modeled as pairs of respective elements connected to signals $s_{\nu i}$ and $s_{\delta i}$.

The mapping from the original model to the respective fractional model is illustrated in Figure 6.6.

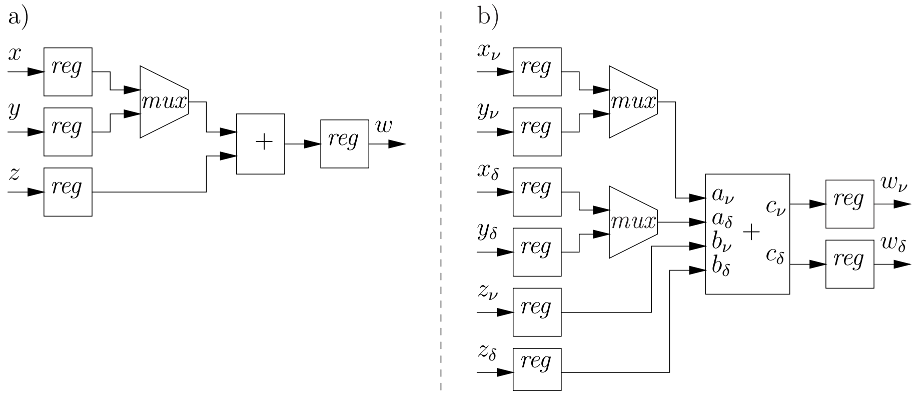


Figure 6.6. Data path a) in the original, and b) in the fractional model

6.2.4 Classification of Input Signals

Input signals of the sequential design are classified as *control*, *data* and *semi-data* signals. The classification procedure is similar to the methods used in [48, 72]. All Boolean signals are control signals. If an internal signal steers the controller to choose between several next states within computation, then the signal is a control signal. In other words, all signals that do not appear in the conditional part of an *if* or *case* expression or as a pattern in a *pattern matching* construction are data signals. If an output of a process is a control signal then all the input signals of the process are regarded as control signals as well. Clearly, the last definition is recursive. All these signals, which are not marked as control signals are data or semi-data signals. Data signals are not used in any conditional expression. However, if the value of an input signal is directly used in a conditional expression and compared against a constant for equality check, the signal is classified as a semi-data signal.

The domains of the control signals have to remain unmodified in order to preserve all behaviors of the controller. If the design contains any control input with an infinite domain, then the designer has to specify a finite domain for that input signal.

The domains of the data and semi-data signals can be reduced by the polynomial abstraction technique. In addition to the values determined by the degree, the abstract domain of a semi-data input signal has to include the constant values from conditional expressions. For example, if in an FIR-filter the data value d_i is compared against the values 0 and 1 and the degree of $d_i = 1$, the abstract domain of d_i has to include the values 0 and 1, and two additional arbitrary values.

For example, all input signals d_i and c_i of the data path in Figure 6.2(c) are data signals since their values do not determine the controller's behavior. Thus, the domains of these signals can be reduced according to their degrees.

6.2.5 Calculation of the Maximum Degrees

In order to find the degrees of data and semi-data variables in the system output function, calculated by the sequential design, the degree calculation method analyzes the computation in the data path by using an approach that is similar to symbolic execution. Instead of actual values and arithmetic operations, this method uses symbolic values, presenting the degrees of input variables in the calculated functions, and special degree propagation operations.

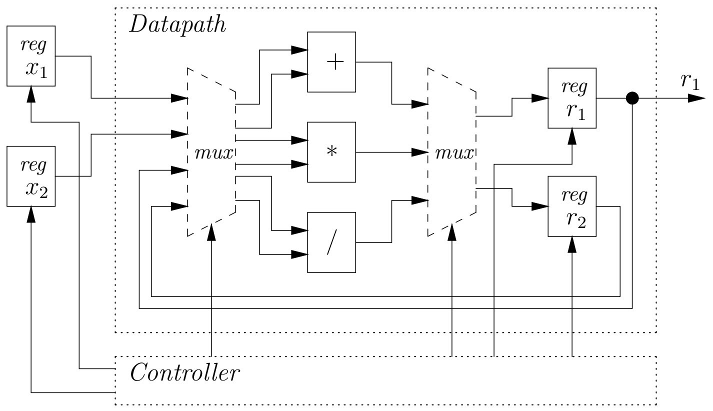


Figure 6.7. A data path example

Let's consider the sequential design in Figure 6.7 with a data path and a controller to explain the maximum degree calculation. In this system the data path can compute three arithmetic operations on the values that are stored in the input registers x_1, x_2 and in the internal registers r_1, r_2. Data path operations, source registers from where operands are read and destination registers where computation results are stored are determined by the controller's state.

Let the controller execute operations $r_1 = x_1 + x_2$ in the first state, $r_2 = r_1 * x_2$ in the second state and $r_1 = r_2 * r_2$ in the third state. Symbolic execution follows the data path operations executed by the controller, applies the same operations to the symbolic input values and the functions stored in the registers and in such a way finds exact functions for each register in every state. The function that evaluates the register r_1 after the third state is $r_1 = (((v_1 + v_2) * v_2) * ((v_1 + v_2) * v_2)) = v_1^2 v_2^2 + 2v_1 v_2^3 + v_2^4$, where v_1 and v_2 are symbolic values that correspond to the inputs x_1 and x_2, respectively. The input variable v_1 has the degree equal to two and the degree of v_2 is equal to four in the found function. Instead of finding the degrees by composing the exact functions, the maximum possible degrees of the input variables can be derived through the degree propagation technique.

Degree Propagation

The maximum degree calculation differs from symbolic execution in some extent. The degree propagation technique assigns to every data input a degree vector instead of symbolic values. Instead of composing a function in symbolic input values to each data path register in every controller's state, the degree propagation technique calculates the maximum possible degrees of input variables in the functions that evaluate registers.

Considering the previous example, the degrees of variables corresponding to the inputs x_1 and x_2 are constant ones. The data path calculates $r_1 = x_1 + x_2$ in the first cycle, which means that the maximum degree of x_1 in the register r_1 is one, and the degree of x_2 in r_1 is one too. In the second state the register r_2 stores the result of the function $r_2 = r_1 * x_2$, where the degree of x_1 stays equal to the degree of x_1 in r_1, but the degree of x_2 is increased by one. In the third state, the degrees in r_1 are twice higher than they were in r_2. Thus instead of constructing the respective functions for each register by symbolic execution, the maximum degree calculation requires simple integer arithmetic.

Due to the model, where functions and values are modeled as fractions, the maximum degrees are represented in the fractional format as well. For a function, expressed as a fraction of polynomials $\frac{P(\overline{x})}{Q(\overline{x})}$ the maximum degree fraction has the format $\frac{\langle \nu_{x_1},\ldots,\nu_{x_n}\rangle}{\langle \delta_{x_1},\ldots,\delta_{x_n}\rangle}$, where ν_{x_i} and δ_{x_i} are the maximum degrees of the input variable x_i in the numerator and denominator polynomials, respectively. For example, if a system calculates function $\mathcal{G} = \frac{x_1^3x_2^2+2x_1}{x_1x_2+7}$, the maximum degrees of x_1 and x_2 in the numerator polynomial are three and two, respectively, the maximum degrees of x_1 and x_2 in the denominator polynomial are both one, and the maximum degree fraction of the system output is $\frac{\langle 3,2\rangle}{\langle 1,1\rangle}$.

Since the functionality of a system input does not have any functional dependence from other inputs, the degree of an input variable is equal to one, and the maximum degree fraction of an input variable x_i is: $\frac{\langle \nu_{x_1},\ldots,\nu_{x_{i-1}},\nu_{x_i},\nu_{x_{i+1}},\ldots,\nu_{x_n}\rangle}{\langle \delta_{x_1},\ldots,\delta_{x_{i-1}},\delta_{x_i},\delta_{x_{i+1}},\ldots,\delta_{x_n}\rangle} = \frac{\langle 0,\ldots,0,1,0,\ldots,0\rangle}{\langle 0,\ldots,0,0,0,\ldots,0\rangle}$. The notation $\frac{\langle \nu_{x_i}\rangle}{\langle \delta_{x_i}\rangle}$ is used to refer to the degree of a variable x_i. In this case the degree of an input variable x_i is denoted as $\frac{\langle 1\rangle}{\langle 0\rangle}$. The maximum degree calculation rules for the basic arithmetic operations, which are in the format $z = x\ OP\ y$, are given in (6.9).

$$\begin{gathered}
\frac{\langle \nu_1^z,\nu_2^z,\ldots,\nu_n^z\rangle}{\langle \delta_1^z,\delta_2^z,\ldots,\delta_n^z\rangle} = \frac{\langle \nu_1^x,\nu_2^x,\ldots,\nu_n^x\rangle}{\langle \delta_1^x,\delta_2^x,\ldots,\delta_n^x\rangle} OP \frac{\langle \nu_1^y,\nu_2^y,\ldots,\nu_n^y\rangle}{\langle \delta_1^y,\delta_2^y,\ldots,\delta_n^y\rangle}, where \\
\\
if\ \ OP == ('+'\ or\ '-')\ then \\
\nu_i^z = max((\nu_i^x + \delta_i^y),(\delta_i^x + \nu_i^y))\ and\ \delta_i^z = \delta_i^x + \delta_i^y\ ; \\
\\
if\ \ OP ==' *'\ then \\
\nu_i^z = \nu_i^x + \nu_i^y\ and\ \delta_i^z = \delta_i^x + \delta_i^y \\
\\
if\ \ OP ==' /'\ then \\
\nu_i^z = \nu_i^x + \delta_i^y\ and\ \delta_i^z = \delta_i^x + \nu_i^y
\end{gathered} \tag{6.9}$$

The maximum degree calculation starts from the state where the controller turns when the control signal *start* goes high, and follows the state trace that is determined by a combination of values on the control and semi-data inputs. In every state, the degree calculations technique updates the degree vectors of all these registers, which are destinations for data path operations in the current state. The degrees of the register that evaluates the data path output when the signal *ready* goes high determine the abstract domains of the input values for model checking.

According to the values on the control and semi-data inputs, the controller may select between different functions to perform the computation, but the degrees of these functions may not be identical. Therefore, the degree calculation has to be performed for all different functions, which are determined by the values on the control and semi-data inputs. The abstract domain of a data or semi-data input signal x_i is defined based on the maximum degree of the input x_i over all of these calculations.

The following illustrates the degree analysis for the output signal r_1 of the system in Figure 6.7 that calculates the function $\mathcal{G}$. The controller's states and corresponding data path instructions are shown in Table 6.1. Let the function $\mathcal{D}(P(\overline{x}), x_i)$ give the maximum degree of x_i in $P(\overline{x})$, and $\mathcal{D}(r_i(\overline{x}))$ denote the degree vectors of r_i.

Table 6.1. A maximum degree computation example

state	r_1	r_2	$\mathcal{D}(r_1(\overline{x}))$	$\mathcal{D}(r_2(\overline{x}))$
1	$x_1 * x_2$	$x_1 + x_1$	$\langle\frac{1,1}{0,0}\rangle$	$\langle\frac{1,0}{0,0}\rangle$
2	$r_1 * r_1$		$\langle\frac{2,2}{0,0}\rangle$	$\langle\frac{1,0}{0,0}\rangle$
3	$x_1 * r_1$		$\langle\frac{3,2}{0,0}\rangle$	$\langle\frac{1,0}{0,0}\rangle$
4	$r_1 + r_2$	$x_1 * x_2$	$\langle\frac{3,2}{0,0}\rangle$	$\langle\frac{1,1}{0,0}\rangle$
5		$r_2 + 7$	$\langle\frac{3,2}{0,0}\rangle$	$\langle\frac{1,1}{0,0}\rangle$
6	r_1/r_2		$\langle\frac{3,2}{1,1}\rangle$	$\langle\frac{1,1}{0,0}\rangle$

The degree fractions of r_i have the format $\frac{\langle\nu_{x_1},\nu_{x_2}\rangle}{\langle\delta_{x_1},\delta_{x_2}\rangle}$. In the initial state the degree fractions are filled with zeros. In the first state the data path calculates two operations: $r_1 = x_1 * x_2$ and $r_2 = x_1 + x_1$. The degree of x_1 in r_1 after the computation in the first state, according to the degree calculation rules is:

$$\begin{aligned}
\mathcal{D}(r_1(\overline{x}), x_1) &= \mathcal{D}((\tfrac{x_1}{1} * \tfrac{x_2}{1}), x_1) \\
&= \frac{\mathcal{D}((x_1 * x_2), x_1)}{\mathcal{D}((1*1), x_1)} \\
&= \frac{\mathcal{D}((x_1), x_1) + \mathcal{D}((x_2), x_1)}{\mathcal{D}((1), x_1) + \mathcal{D}((1), x_1)} \\
&= \frac{\langle 1+0\rangle}{\langle 0+0\rangle} \\
&= \frac{\langle 1\rangle}{\langle 0\rangle}
\end{aligned}$$

Similarly, the degree of x_2 is $\frac{\langle 1\rangle}{\langle 0\rangle}$. Thus, the degrees of r_1 are $\frac{\langle 1,1\rangle}{\langle 0,0\rangle}$. The degree of x_1 in $r_2 = x_1 + x_1$ is:

$$\begin{aligned}
\mathcal{D}(r_2(\overline{x}), x_1) &= \mathcal{D}((\tfrac{x_1}{1} + \tfrac{x_1}{1}), x_1) \\
&= \tfrac{max(\mathcal{D}((x_1*1),x_1),\mathcal{D}((x_1*1),x_1))}{\mathcal{D}((1*1),x_1)} \\
&= \tfrac{max(\mathcal{D}((x_1),x_1),\mathcal{D}((x_1),x_1))}{\mathcal{D}((1),x_1)+\mathcal{D}((1),x_1)} \\
&= \tfrac{max(1,1)}{0+0} \\
&= \tfrac{\langle 1\rangle}{\langle 0\rangle}
\end{aligned}$$

The degree of x_2 in r_2 is $\frac{\langle 0\rangle}{\langle 0\rangle}$, since this input variable is not an operand of the function that evaluates r_2. The maximum degree calculation proceeds as shown in Table 6.1, and terminates with the degree fraction $\frac{\langle 3,2\rangle}{\langle 1,1\rangle}$ for r_1.

Since the degree calculation in every state is based on the degrees in previous states and does not consider the actual functions, the simplification of a degree fraction by reducing the respective degrees ν_i and δ_i is not allowed. For example, it is not valid to change the fraction $\frac{\langle 3,2\rangle}{\langle 1,1\rangle}$ to $\frac{\langle 2,1\rangle}{\langle 0,0\rangle}$. This kind of simplification is correct for the function $\frac{x_1^3 x_2^2}{x_1 x_2}$, but does not hold for the function $\mathcal{G} = \frac{x_1^3 x_2^2 + 2x_1}{x_1 x_2 + 7}$. Thus, actually the technique finds the upper bounds of degrees.

If the output polynomials according to the specification are $\frac{P(\overline{x})}{Q(\overline{x})}$ and the implementation functionality is expressed as $\frac{R(\overline{x})}{S(\overline{x})}$, then based on the mathematical rule $\frac{a}{b} = \frac{c}{d} \Rightarrow ad = bc$, the final maximum degree for the input variable x_i is calculated as:

$$\max(\mathcal{D}(P(\overline{x})S(\overline{x}), x_i), \mathcal{D}(Q(\overline{x})R(\overline{x}), x_i))$$

The final maximum degrees of x_1 and x_2 for the function $\mathcal{G}$ are 4 and 3, respectively. If the degree of a data input variable x is k, then a model checker can assign values from n to $n + k$ to the input x, where n is an arbitrarily chosen integer value. The input domain of a semi-data input is extended with the constant values from the equality comparison expressions against constants.

6.2.6 Domain Propagation

In addition to specifying the domains for the data input signals, the SMV model has to contain the domain declarations of all internal and output signals that are used in a function description. The declaration of a data path register has to cover all the possible values, which may be stored there, regarding the input values and the behavior of the controller. The simplest way is to specify a domain by the least and the greatest values. If the least value of x is x^- and the greatest value is x^+, the domain of the variable x can be described as $[x^-, x^+]$, which means that x may have any value between x^- and x^+.

The domain propagation rules for the basic arithmetic operations on integer values are given in Table 6.2. The function $\mathcal{P}(x, y)$ calculates pairwise multiplications on the least and the greatest values of operands x and y from their respective domains. The result of $\mathcal{P}$ is a four element set, which contains values: $\mathcal{P}(x, y) = \{x^- * y^-, x^- * y^+, x^+ * y^-, x^+ * y^+\}$.

Table 6.2. Domain propagation rules for arithmetic operations

OP	*Domain Calculation*
$x + y$	$[x^-, x^+] + [y^-, y^+] = [x^- + y^-, x^+ + y^+]$
$x - y$	$[x^-, x^+] - [y^-, y^+] = [x^- - y^+, x^+ - y^-]$
$x * y$	$[x^-, x^+] * [y^-, y^+] = [min(\mathcal{P}(x, y)), max(\mathcal{P}(x, y))]$

According to the domain propagation rules, two operands a and b that have domains $[-5, 2]$ and $[-3, 4]$, respectively, give $\mathcal{P}(a, b) = \{15, -20, -6, 8\}$. Thus, the result of multiplication of a and b is bounded by the minimum value -20 and the maximum value 15, which form the domain $[-20, 15]$.

The domain propagation rules for the fractional operations in Figure 6.5 are presented in Table 6.3.

According to the rules in Table 6.3, the output domain of a division operation applied to the fraction of signals s and t, with domains $\frac{[-2,4]}{[3,5]}$ and $\frac{[6,8]}{[-7,9]}$, respectively, is:

$$\frac{[min(-2 * -7, -2 * 9, 4 * -7, 4 * 9), max(-2 * -7, -2 * 9, 4 * -7, 4 * 9)]}{[min(3 * 6, 3 * 8, 5 * 6, 5 * 8), max(3 * 6, 3 * 8, 5 * 6, 5 * 8)]} = \frac{[-28, 36]}{[18, 40]}$$

The domain propagation analysis is similar to the degree propagation. Considering the domains that are specified by the degrees of input variables in the system output functions, the domains of data path registers are found according to the sequence of instructions executed by the controller. If the domain of an input signal x_i in the sequential model is $[x_i^-, x_i^+]$, the

Table 6.3. Domain propagation rules for fractions of signals

$\frac{x_\nu}{x_\delta} + \frac{y_\nu}{y_\delta}$	$\frac{[x_\nu^-, x_\nu^+]}{[x_\delta^-, x_\delta^+]} + \frac{[y_\nu^-, y_\nu^+]}{[y_\delta^-, y_\delta^+]} = \\ = \frac{[min(P(x_\nu, y_\delta)) + min(P(x_\delta, y_\nu)), max(P(x_\nu, y_\delta)) + max(P(x_\delta, y_\nu))]}{[min(\mathcal{P}(x_\delta, y_\delta)), max(\mathcal{P}(x_\delta, y_\delta))]}$
$\frac{x_\nu}{x_\delta} - \frac{y_\nu}{y_\delta}$	$\frac{[x_\nu^-, x_\nu^+]}{[x_\delta^-, x_\delta^+]} - \frac{[y_\nu^-, y_\nu^+]}{[y_\delta^-, y_\delta^+]} = \\ = \frac{[min(P(x_\nu, y_\delta)) - max(P(x_\delta, y_\nu)), max(P(x_\nu, y_\delta)) - min(P(x_\delta, y_\nu))]}{[min(\mathcal{P}(x_\delta, y_\delta)), max(\mathcal{P}(x_\delta, y_\delta))]}$
$\frac{x_\nu}{x_\delta} * \frac{y_\nu}{y_\delta}$	$\frac{[x_\nu^-, x_\nu^+]}{[x_\delta^-, x_\delta^+]} * \frac{[y_\nu^-, y_\nu^+]}{[y_\delta^-, y_\delta^+]} = \frac{[min(\mathcal{P}(x_\nu, y_\nu)), max(\mathcal{P}(x_\nu, y_\nu))]}{[min(\mathcal{P}(x_\delta, y_\delta)), max(\mathcal{P}(x_\delta, y_\delta))]}$
$\frac{x_\nu}{x_\delta} / \frac{y_\nu}{y_\delta}$	$\frac{[x_\nu^-, x_\nu^+]}{[x_\delta^-, x_\delta^+]} / \frac{[y_\nu^-, y_\nu^+]}{[y_\delta^-, y_\delta^+]} = \frac{[min(\mathcal{P}(x_\nu, y_\delta)), max(\mathcal{P}(x_\nu, y_\delta))]}{[min(\mathcal{P}(x_\delta, y_\nu)), max(\mathcal{P}(x_\delta, y_\nu))]}$

respective domain of $\frac{x_i}{1}$ in the fractional model becomes $\frac{[x_i^-, x_i^+]}{[1,1]}$. The analysis starts from the state where the controller turns after the input signal *start* goes high, follows the order of state transitions, and finishes in the state where the output signal *ready* goes high. It is assumed that all state traces lead to states where the signal *ready* is high. For every controller state the domain propagation technique finds the domains of intermediate computation results stored in the registers. The domain of a modified register file element depends in the data path operation in a controller's state, and the domains of selected operands. If based on the control and semi-data input values the controller may choose between several state traces to execute instructions in the data path, the domain propagation analysis has to be performed for all state traces. The declaration of a register r_i in the abstract model is determined by the least and the greatest values that were stored through all state traces in r_i.

Let the abstract domains of x_1 and x_2 of the function $\mathcal{G}$ be $0, \ldots, 4$ and $0, \ldots, 3$. The domain propagation for the function $\mathcal{G}$, following the sequence of the data path operations in Table 6.1, is illustrated in Table 6.4.

Let the specification and implementation polynomials be $\frac{P(\overline{x})}{Q(\overline{x})}$ and $\frac{R(\overline{x})}{S(\overline{x})}$, respectively. The model checker verifies that $P(\overline{x})S(\overline{x})) == Q(\overline{x})R(\overline{x})$.

Table 6.4. Domain propagation in the computation of the function $\mathcal{G}$

state	r_1	r_2	*Domain* r_1	*Domain* r_2
1	$x_1 * x_2$	$x_1 + x_1$	$\frac{[0,12]}{[1,1]}$	$\frac{[0,8]}{[1,1]}$
2	$r_1 * r_1$		$\frac{[0,144]}{[1,1]}$	$\frac{[0,8]}{[1,1]}$
3	$x_1 * r_1$		$\frac{[0,576]}{[1,1]}$	$\frac{[0,8]}{[1,1]}$
4	$r_1 + r_2$	$x_1 * x_2$	$\frac{[0,584]}{[1,1]}$	$\frac{[0,12]}{[1,1]}$
5		$r_2 + 7$	$\frac{[0,584]}{[1,1]}$	$\frac{[0,19]}{[1,1]}$
6	r_1/r_2		$\frac{[0,584]}{[0,19]}$	$\frac{[0,19]}{[1,1]}$

The value zero in the denominator domains does not cause division by zero, since the fractional model does not use division. However, the final domain has to be calculated as the multiplication of the numerator and denominator domains. The domain for the function $\mathcal{G}$ is $[0 * 0, 584 * 19] = [0, 11096]$.

6.2.7 Application of the Modulo Theorem

The theoretical background of the modulo theorem in Section 6.2.1 allows to reduce the domains of data inputs and data path variables and thereby to reduce the time and memory demands in model checking. The correctness of the abstract fractional model is deduced from the verifications of a number of drastically smaller models. In these smaller models all data path arithmetic operations are extended with modulo computation, i.e., an assignment statement $r = x\ OP\ y$ in the original model is transformed to $r = (x\ OP\ y)\ mod\ n_i$. Constants n_i belong to the set of relatively prime numbers $M = \{n_1, \ldots, n_m\}$. Every constant in M defines one model that has to be verified. The amount of prime numbers in M depends on the least and greatest values (v^- and v^+ respectively) computed in the data path in the abstract model. The multiplication of the relatively prime numbers in M has to give a greater value than is the difference between v^- and v^+, i.e., $\prod_{i=1}^{m} n_i > (v^+ - v^-)$.

The domain of the function $\mathcal{G}$ is $[0, 11096]$. In order to encode 11096 values, the model checker has to create 14 binary variables, $2^{14} = 16384$ and $11096 < 16384$. The multiplication of the relatively prime numbers 3,5,7,11,13 is greater than the largest value calculated by $\mathcal{G}$. Thus, the largest value, which is calculated by the models after extending them with the modulo calculations, requires only 4 bits. Since in the worst case the computer resource demands in model checking grows exponentially with the number of binary variables, the reduction from 14 bits to 4 bits is a significant improvement, despite that five different models have to be verified.

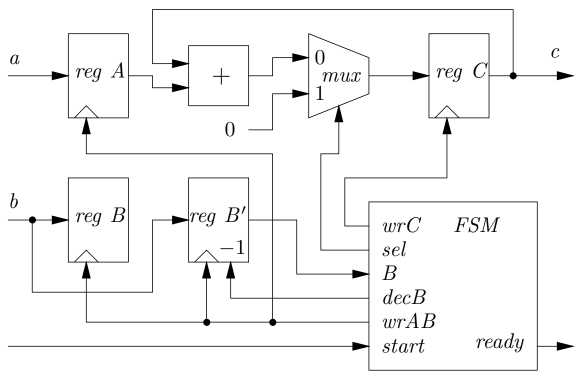
6.2.8 Model Checking

In order to verify the refined sequential design against the polynomial specification, the abstract models extended with modulo computation are mapped to the SMV language, as explained in Chapter 3. According to the maximum degrees of the input variables in the output polynomials, new domains are defined for all the data signals. If the maximum degree of a data signal s_i is k then the model checker can assign values from n to $n+k$ to that input. In addition, the domain of a semi-data value includes constants from the equality comparison expressions. The input domain of a semi-data variable is composed so that the abstract values determined by the degree analysis do not overlap with the values from the equality comparison expressions. The domains of the control signals stay unchanged. The SMV model checker verifies the following property. If the input values ($\overline{v}$) on the combinational specification ($\frac{S_\nu(\overline{x})}{S_\delta(\overline{x})}$) and a sequential implementation ($\frac{I_\nu(\overline{x})}{I_\delta(\overline{x})}$) are equal, then also the output values are equal, i.e., $S_\nu(\overline{v})I_\delta(\overline{v}) == S_\delta(\overline{v})I_\nu(\overline{v})$, in the state where the signal *ready* goes high.

6.3 Verification of Loop-back Structures

6.3.1 Comparison between Spatial and Polynomial Abstractions

The spatial abstraction technique is illustrated in [72] by a verification case study, where the correctness of a sequentially implemented multiplier is checked. The sequential design multiplies two input values by using repetitive addition. The internal structure of the multiplier, equivalent to the model in [72] is depicted in Figure 6.8. The system can be divided into (a) a data path, which contains a couple of registers, a multiplexer and an adder, and (b) a controller composed by a finite state machine and a register that is extended with a decrement operation. The multiplier decrements the value in the input register B' by one, until the value in B' is equal to zero. The value in the other input register A is added to the value in the output register C in parallel to every decrement.



FSM next state function

current state	*input*	*next state*
0	*start = 0*	0
0	*start = 1*	1
1	-	2
2	$B = 0$	4
2	$B \neq 0$	3
3	-	2
4	-	0

FSM output function

output \ *state*	0	1	2	3	4
wrAB	0	1	0	0	0
wrC	0	1	0	1	0
sel	1	1	0	0	0
decB	0	0	0	1	0
ready	0	0	0	0	1

Figure 6.8. Internal structure of a repetitive multiplier

The task for model checking is to verify if the value in the register C is always equal to the multiplication of the values in the registers A and B when the signal *ready* goes high. The property expressed as a temporal logic expression has the following form:

$$AG((FSM.finish = 1) - >(C.out = A.out * B.out))$$

Since the value in the register B is changed within the computation and at the same time the original value is required to check the property after the computation, the register B is duplicated by B'.

The description of the original model of a multiplier with 12-bit input and 24-bit output words includes 64 binary state variables: three 12-bit registers A, B, B', and one 24-bit register C, three bits to encode the FSM states, and one bit for the signal *start*. Both abstraction techniques classify the signal a as a data input, and the signals b and *start* as control inputs. In order to preserve all controller behaviors, the domains of the input signal b and the register B have to stay unchanged. Since the signal a is a data input, spatial abstraction reduces the domain of the signal a and the 12-bit register A to a 1-bit signal and a 1-bit register, respectively. A 12-bit register is sufficient to store the greatest value of the multiplication of a 12-bit and a 1-bit value. Consequently, the 24-bit register C can be replaced with a 12-bit one. Altogether, the model reduced by the spatial abstraction technique contains 41 binary state variables.

According to polynomial abstraction, the degree of the input variable a in the output polynomials is equal to one. Thus, the domain of a can be reduced to include two values 0 and 1. Based on the domain propagation analysis, 0 and 4095 are the least and the greatest values, respectively, in the abstract model. The application of the modulo theorem allows to extend all arithmetic operations with modulo computations ($mod\ n_i$) and consequently to reduce the size of registers that store data values, but not B'. The multiplication of all values from the set of relatively prime numbers $\{5, 7, 11, 13\}$ is equal to 5005 that is greater than 4095. In the largest model, where $n_i = 13$, the greatest value stored in the register C requires 4 bits instead of the original 24 bits or 12 bits after spatial abstraction. After the application of the modulo theorem the model checker verifies the property:

$$AG((FSM.finish = 1) - >(C.out\ mod\ n_i = ((A.out\ mod\ n_i) * (B.out\ mod\ n_i))\ mod\ n_i)$$

The size of the reference register B can be reduced according to the value n_i as well. Thus the size of the largest model after polynomial abstraction includes 25 binary state variables: one 1-bit register A, two 4-bit registers B and C, one 12-bit register B', and four bits for FSM and the signal *start*.

Although polynomial abstraction generates four models for verification, compared to spatial abstraction the amount of time and the number of BDD-nodes required for model checking are drastically reduced. The number of state variables and resource demands for model checking are presented in Table 6.5. The experiments were done on a Sun machine with 900MHz CPU and 16GB RAM, by using the Cadence SMV tool [102].

Table 6.5. Polynomial abstraction versus spatial abstraction

	The number of states variables	*Reduction in states*	*Time*	*BDD−nodes*
Original model	64	−	10 *days* (*estimated*)	−
Model after spatial abstr.	41	36%	4.4 *hours*	118M
Model after polynomial abstr.	25	61%	4.2 *minutes*	540k

6.3.2 Verification of FIR and IIR Filters

A possible internal structure of a 16th order FIR filter is depicted in Figure 6.2(c). The filter description contains a shift register that stores the input data values for fifteen clock cycles. In the verification of the filter implementation, it is considered that the shift register and the data path components are correct. Thus, assuming that the shift register behaves correctly, the verification task is to check that the data path calculates correct output values on the delayed input values d_i and the filter coefficients c_i, according to the filter specification in (6.10).

$$y = \sum_{k=0}^{15} c_i * d_i \tag{6.10}$$

The refined sequential design, which calculates the function in (6.10) has 33 inputs - 16 data inputs d_i, 16 coefficients c_i that may vary in time, and an additional input signal *start*. The input signals d_i and c_i are classified as data signals since they do not appear in any conditional expression. According to the proposed degree calculation algorithm, the degrees of all data signals are equal to one. Thus, for the verification it is sufficient to assign the values 0 and 1 to the data inputs. According to the domain analysis, the least and the greatest values calculated in the data path are 0 and 16. Since the multiplication of the relatively prime numbers 3 and 7 is greater than 16, the actual verification is done in two models where the data path operations are extended with modulo 3 and 7 calculations. The size of the data path internal register is reduced according to the maximum values from the modulo operations as well. The total verification time of these two models and the maximum number of allocated BDD nodes are shown in Table 6.6.

Table 6.6. Verification Time and BDD Nodes

Design	*Function*	*Time sec.*	*BDD−nodes*
FIR filter	$\sum_{i=0}^{15} c_i * d_i$	29.4	$1.4M$
IIR filter	$d_7 * b_0 + \sum_{i=1}^{7}(w_{7-i} * b_i - w_{7-i} * a_i * b_0)$	2.5	$458k$
DFT	*8 point FFT*	1.5	$91k$
Cosines	$\sum_{i=0}^{4} \frac{x^{2*i}}{(2*i)!}$	9.5	$153k$

The main difference between FIR and IIR[2] filters is that the FIR filter specification, defined in (6.10), is a function on delayed input values, but the IIR filter specification in (6.11) comprises delayed output values as well.

$$y_n = \sum_{k=0}^{N} b_k d_{n-k} - \sum_{k=1}^{N} a_k y_{n-k} \tag{6.11}$$

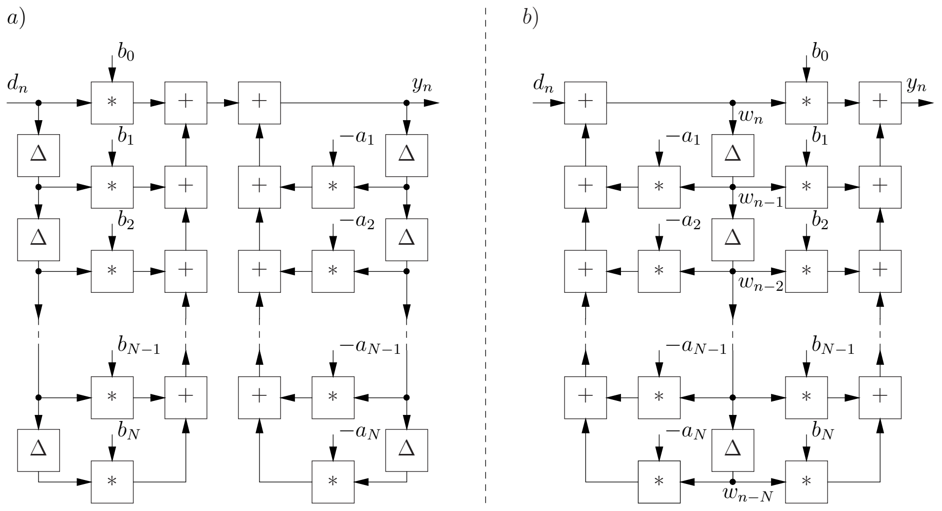


Figure 6.9. IIR filter structures: (a) direct form I and (b) regular direct form II

The IIR filter structure in Figure 6.9(a) corresponding to (6.11) can be viewed as a sequential composition of two systems. The left hand part, calculating a polynomial on the delayed input values d_k and the coefficients b_k, is equivalent to the general FIR filter structure. The right hand part stores the output values y_k for N clock cycles, multiplies them with the coefficients a_k and adds the results to the output value from the left side FIR structure. In order to obtain a more compact structure, the two parts can switch their positions that makes it possible to merge the respective delay elements. The filter structure after the last transformation is depicted in Figure 6.9(b), and its specification is described by the equations in (6.12).

$$w_n = -\sum_{k=1}^{N} a_k w_{n-k} + d_n$$

[2]Infinite Impulse Response

$$y_n = \sum_{k=0}^{N} b_k w_{n-k} \tag{6.12}$$

The system, which calculates the IIR function (6.12) when $N = 7$, has been implemented as a sequential design including a data path and a controller. The controller was configured so that the data path calculates one of the following terms $d_7 * b_0$, $w_{7-i} * b_i$, $w_{7-i} * a_i * b_0$ at each clock cycle. The variables d_7, a, b_i, w_i are classified as data signals and their degrees are equal to one. The abstract domains of these variables have to contain two values for model checking, according to the degrees. A model checker requires only four one bit signals for d_7, a, b_i, w_i in the abstract model. The verification of this implementation took less than 3 seconds and the SMV model checker created less than $500k$ BDD-nodes.

Although both systems in Figure 6.9 give the same result for a given input signal, the polynomial abstraction based verification technique cannot verify directly that the direct form I implements the function in (6.12) and the direct form II implements the function in (6.11). The limitation is caused by the different input arguments in the functions in (6.12) and (6.11) that makes it difficult to define reference points between two different implementation models. For example, w_n is not only a function on the input values d_n but also on the values $w_{n-1}, \ldots, w_{n-N}$. At the same time w_{n-N} is a function on the values $w_{n-(N+1)}, \ldots, w_{n-(N+N)}$ and the input value N clock cycles before. Thus, all past input values starting from the reset state have to be considered in order to verify that these two systems calculate the same result at a time instant t.

In addition to the FIR and IIR filters a cosines function and an FFT application implemented as a data path and a controller have been verified. The time and the number of BDD nodes required for the verification are shown in Table 6.6. All example functions listed in Table 6.6 are present in today's audio, video and communication devices that involve digital signal processing.

Chapter 7

Synchronization after Temporal Refinements

The basic concept of the synchronous model of computation is the synchronous hypothesis that states that computation in processes and communication between them take no time. According to the synchronous hypothesis a combinational process has no delay and the response to input values appears on the process output in the same moment the input values arrive. Compared to combinational processes, pipelined and resource shared computation blocks contain additional delay elements that store intermediate computation results. It is obvious that additional delay elements cause delayed data arrival at the outputs of pipelined and resource shared blocks. Since the introduction of pipelining or resource sharing in combinational processes involves delays, these refinements cause local temporal changes in the corresponding refined models.

A refinement is classified as a *local temporal refinement* if it replaces a combinational block with an equivalent block, which has a delay. The original and replaced blocks compute the same results but because of the delay the refined block produces one or more extra values on the output signal. The longer delay causes mismatched data arrival at multi-input destination processes, which are connected to the refined block. Due to this mismatched data arrival, multi-input processes of the refined model operate with different input assignments than the original model. The change in the local temporal behavior of a sub-system is a potential source of errors in the system. In order to avoid computation errors, concurrently processed data items in the original model have to be processed concurrently in the refined model as well. The data items processed in the original model are denoted as *actual data items* and considered as *informative events*. The objective of the synchronization algorithm described in this chapter, is to insert synchronization delays into the model in addition to the refinement added delays. The initial values of synchronization delays and the initial values of refinement added delays are denoted as *synchronization values* and these values are distinguishable from the actual data items. Synchronization delays are located so that the actual data do not interfere with the synchronization values. Although the additional synchronization values appear on the system output, the actual data values are issued in the same order as in the original model. In other words, the refined and synchronized model is latency equivalent to the original one. Two signals are latency equivalent if they have the same order of informative events. Two models are latency equivalent if they produce latency equivalent output signals on latency equivalent input signals.

7.1 Introduction

Let's consider the introduction of resource sharing in a combinational computation block, as explained in Chapter 6, to illustrate the impact of a local temporal refinement on the entire system. The block B, in Figure 7.1, contains one process with a combinational function fn_1. In the block B' the function fn_1 is mapped to basic data path operations, which are executed by the controller following an m-step schedule. The block B and the block B' can be viewed as two different implementations of the same combinational function fn_1. Due to the reuse of operations in the data path, the controller stores intermediate computation results in the register file and uses them in further computation steps. The feedback loop through the data path and the register file has to include at least one delay process, since the data path and the register file have zero-computation time according to the synchronous hypothesis and zero-delay feedback loops are forbidden in the used synchronous model. Even if the block is surrounded by clock domain interfaces, so that it operates at m-times higher clock frequency, the block B' still has one clock cycle delay. Therefore, the behavior of the block B' is identical to the behavior of B'', which composes a delay process and the process P_{comb} with the combinational function fn_1. The initial value of the additional delay process is not described in the system specification, and processing of this unexpected value by other processes causes the entire

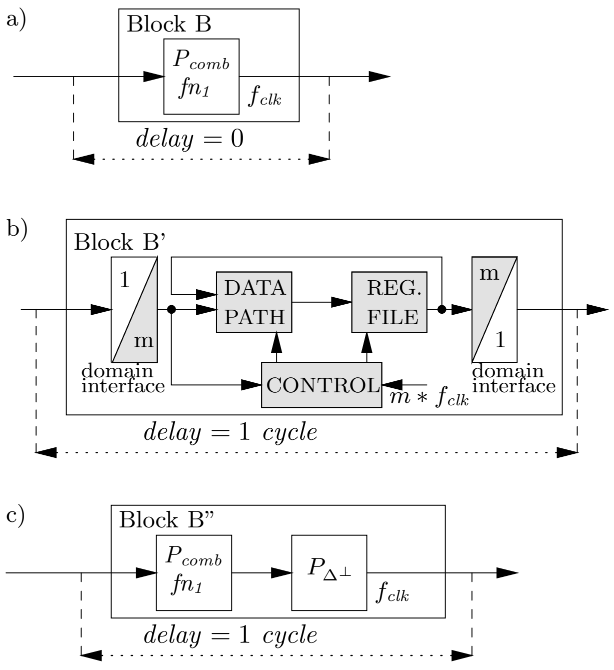


Figure 7.1. Resource sharing adds an extra delay to the system.

system to behave different from the original model with the block B. The unexpected values are reflected on the output signal as well and the reaction of the refined model to an input signal is erroneous.

In order to ensure that the initial values (synchronization values) from the refinement added delay processes and from the synchronization delays do not influence the system behavior, (1) the initial values have to be distinct from the values used in the original model. In addition, (2) combinational and finite state machine processes have to react to synchronization values by emitting synchronization values, and (3) finite state machine processes have to preserve their current states at clock cycles when synchronization values appear on their inputs.

Let the notation of present ($\top$) and absent ($\perp$) values be used in this chapter to distinguish between the actual data values and the synchronization values, respectively. Let $P_{\Delta}^{\perp}$ denote a delay process initialized with the synchronization value. If the original model is already described in the sense of present and absent values, then all values of the original model are considered as present values in the context of synchronization in this chapter and in Chapter 8. The absence of a value in the original model is considered as an informative event in the context of the proposed synchronization techniques. Hence, two different types of absent values simply have to get different encodings in the system implementation.

In a synchronous model, each process consumes exactly one event from all of its inputs at every clock cycle. It is not possible that a multi-input process consumes only synchronization values from its inputs and drops them until all inputs hold present values. In order to ensure that multi-input processes can apply their functions to the same set of input values as in the original model, the system structure has to avoid that multi-input processes receive values of different types ($\perp$ and $\top$) concurrently. The algorithm under discussion synchronizes the system by inserting synchronization delays into the model in addition to the temporal refinement added delays. Similarly to the added refinement delays, the synchronization delays are initialized with $\perp$-values. The algorithm locates the synchronization delays such that the synchronization values from the refinement added delays arrive at the inputs of multi-input processes in the same clock cycles with the synchronization values from the synchronization delays.

7.2 Preparing for Synchronization

For the sake of simplicity, let the system model contain only combinational processes P_{comb}, finite state machine processes P_{FSM} and delay processes P_{Δ}, the latter two are initialized with actual data values ($\top$). Every process has exactly one output that can be connected to the inputs of more than one process. All processes belong to the same clock domain and none of the processes operates with synchronization values ($\perp$) before temporal refinements. In order to prepare processes to operate with synchronization values, the following extensions are necessary:

1. In a combinational process P_{comb} the function $fn(x_1, \ldots, x_n)$ applied to the input values $v_1, \ldots, v_n$ in the original model has to be replaced with a new extended function $\overline{fn}(x_1, \ldots, x_n)$ that is defined as follows:

$$\overline{fn}(v_1, \ldots, v_n) = \begin{cases} fn(v_1, \ldots, v_n) & , \textit{ if } \forall i, (1 \leq i \leq n), v_i \neq \perp \\ \perp & , \textit{ if } \exists i, (1 \leq i \leq n), v_i == \perp \end{cases}$$

2. The next state function fn_{nst} and the output function fn_{out} of a finite state machine process P_{FSM} in the original model have to obtain the following extended functionality, where st denotes the current state and $v_1, \ldots, v_n$ are the input values:

$$\overline{fn_{nst}}(st, v_1, \ldots, v_n) = \begin{cases} fn_{nst}(st, v_1, \ldots, v_n) , & if\ \forall i, (1 \leq i \leq n), v_i \neq \perp \\ st , & if\ \exists i, (1 \leq i \leq n), v_i == \perp \end{cases}$$

$$\overline{fn_{out}}(st, v_1, \ldots, v_n) = \begin{cases} fn_{out}(st, v_1, \ldots, v_n) , & if\ \forall i, (1 \leq i \leq n), v_i \neq \perp \\ \perp , & if\ \exists i, (1 \leq i \leq n), v_i == \perp \end{cases}$$

A finite state machine with the extended functionality preserves its current state and emits a synchronization value, if its inputs hold synchronization values in the current clock cycle.

7.2.1 Definitions

In order to analyze the system structure before temporal refinements in the sense of delays, the synchronization algorithm constructs a *delay graph.* This graph abstracts all combinational and finite state machine processes, and presents only delay processes. The graph is used to analyze the delays of all loops and acyclic paths that the system contains.

Definition 7.1 (Path) *A path in the system model is a sequence $\{P_1, P_2, \ldots, P_n\}$ of processes P_i connected by signals s_i, such that $\forall i, (1 \leq i \leq (n-1)), \exists s_i$ connecting the output of P_i and an input of P_{i+1}, and the path contains each process P_i only once, i.e., $\forall i, (1 \leq i \leq n)$ and $\forall j, (1 \leq j \leq n), P_i \neq P_j$.*

$Path_k(P_i, P_j)$ denotes a path from process P_i to process P_j.

Definition 7.2 (Loop) *$Path_k(P_i, P_j)$ is a loop if there exists a signal s_j connecting the output of P_j and an input of P_i, i.e., loop is a cyclic path.*

Definition 7.3 (Pair of paths) *Two paths, $path_1(P_i, P_j)$ and $path_2(P_k, P_l)$, form a pair of paths if $P_i == P_k$, $P_j == P_l$, $\forall P_x, (P_x \in path_1 \wedge \wedge P_x \neq P_i \wedge P_x \neq P_j)$ and $\forall P_y, (P_y \in path_2 \wedge P_y \neq P_k \wedge P_y \neq P_l)$, $P_x \neq P_y$.*

Two paths form a pair if they have the same first process and the same last process, and the paths do not share any other process.

System inputs and outputs can be viewed as shift registers that produce or consume one event at every clock cycle. Therefore, before creating the delay graph, one delay process is added to every system input and output. This extension makes it possible to express not only the delays between internal processes in the delay graph, but also the delays related to the inputs and outputs. For example, without this extension it is not possible to analyze the delays of paths between an input and some internal process.

Definition 7.4 (Delay graph) *The delay graph $G(W, E)$ contains one vertex w_i ($w_i \in W$) for each delay process $P_{\Delta i}$ in the system model. The graph contains an edge $e_{i,j}$ ($e_{i,j} \in E$) from vertex w_i to vertex w_j, if there is at least one path from delay process $P_{\Delta i}$ to delay process $P_{\Delta j}$ and the path does not include any other delay process.*

Similarly to Definitions 7.1 and 7.2 that are given for the system model, a path is a sequence of vertices and a loop is a cyclic path in the delay graph. If a $loop_i$ runs through processes $P_1, \ldots, P_n$ then the graph G contains a loop with the same number of vertices as many delay processes are in $loop_i$. The latter holds for pairs of paths as well.

Let the function $|path_k(P_i, P_j)|_\Delta$ calculate the number of delay processes in $path_k$ from process P_i to process P_j. In other words, the function gives the delay of a path in terms of clock cycles.

7.3 Synchronization Requirements

The synchronization algorithm considers two facts:

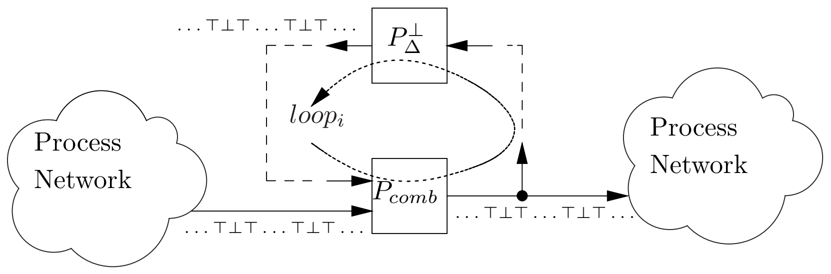
Statement 7.1 *Loops reproduce synchronization events.*

After extending the functions of combinational and finite state machines processes they append $\perp$-values if $\perp$-values appear on their inputs. Therefore, after adding one $P_\Delta^\perp$ delay process to $loop_i$, all processes in this loop regularly operate with $\perp$-values at every n-th clock cycle, if $|loop_i|_\Delta = n$.

The regularity of $\perp$-values on a process inputs can be denoted as a pattern.

Assumption 7.1 *The system does not include any disjoint subpart, i.e., dividing the system into two arbitrarily chosen subparts, there is always at least one path between the subparts.*

A delay process $P^{\perp}_{\Delta}$, added to a loop, causes all processes in the system to operate with $\perp$-values, as illustrated in Figure 7.2. All signals, which start from the loop carry $\perp$-values, and all signals directed to multi-input processes in the loop have to regularly deliver $\perp$-values, such that each multi-input process receives only values of the same type at every clock cycle.



$\ldots\top\perp\top\ldots\top\perp\top\ldots$
$P^{\perp}_{\Delta}$
$loop_i$
Process Network
P_{comb}
$\ldots\top\perp\top\ldots\top\perp\top\ldots$
$\ldots\top\perp\top\ldots\top\perp\top\ldots$
Process Network

Figure 7.2. All processes operate with $\perp$-values after temporal refinements.

Definition 7.5 (Pattern) *A pattern is a minimal sequence $\mathcal{P} = p_0p_1\ldots p_i\ldots p_{n-1}$ of values $\perp$ and $\top$ ($p_i \in \{\perp,\top\}$), which cannot be constructed by a single repetitive subpart $p_0\ldots p_i$.*

Definition 7.6 (Pattern equivalence) *Two patterns $\mathcal{P}' = p'_0\ldots p'_{n-1}$ and $\mathcal{P}'' = p''_0\ldots p''_{n-1}$ are equivalent, if they have the same length and there exists a constant k such that $\forall i, (0 \le i \le (n-1)),\ p'_i == p''_j$ and $j = (i+k)$ mod n.*

For example, the sequence $\perp\top\perp\top$ is not a pattern, since it can be presented by the repetitive subpart $\perp\top$. At the same time $\perp\top$ is a pattern according to the definition. The pattern $\mathcal{P}_1 = \top\perp\perp\top\top$ is equivalent to the pattern $\mathcal{P}_2 = \top\top\perp\perp\top$, but not to $\mathcal{P}_3 = \top\perp\top\perp\top$. In other words, patterns stay equivalent after rotating elements in them, but not after shuffling. Patterns show in which order processes receive $\perp$- and $\top$-values. For example, if combinational processes P_1, P_2 and delay process P_Δ in $path_i = \{P_1, P_2, P_\Delta, P_3, P_4\}$ operate with the pattern $\mathcal{P}_1$, then combinational processes P_3 and P_4 operate with the equivalent pattern $\mathcal{P}_2$, as illustrated in Figure 7.3.

Statement 7.2 *Based on the Assumption 7.1, to ensure that all processes receive only values of the same type at every clock cycle, all processes in the system have to operate with equivalent patterns.*

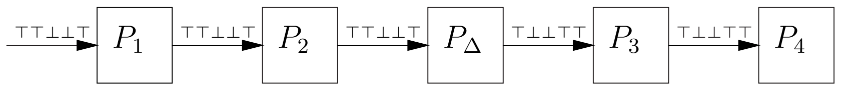


Figure 7.3. Equivalent patterns on a path before and after a delay process

The length of patterns has to correspond to the delays of all loops and the delay differences in all pairs of paths.

The original model, before temporal refinements, does not include $P_\Delta^\bot$ delay processes. According to Definition 7.5, processes operate with one element patterns $\top$ in this model. Let N denote the pattern length. Although the model does not process $\bot$-values, it is possible to say what the pattern length $(N+1)$ is for the system after the first temporal refinement. In order to calculate the pattern length, the delays of loops and the delay differences in pairs of paths have to be found. A pattern fits to a loop, if the number of delay processes in the loop is a multiple of the pattern length. If the system does not contain any pair of paths, the greatest common integer divisor (*gcd*) of the delays of loops determines N. For example, the system in Figure 7.4 contains two loops with two and four delay processes, respectively, and $N = gcd(2, 4) = 2$. After a temporal refinement in this system, all processes have to operate with three element patterns that are equivalent to $\bot\top\top$.

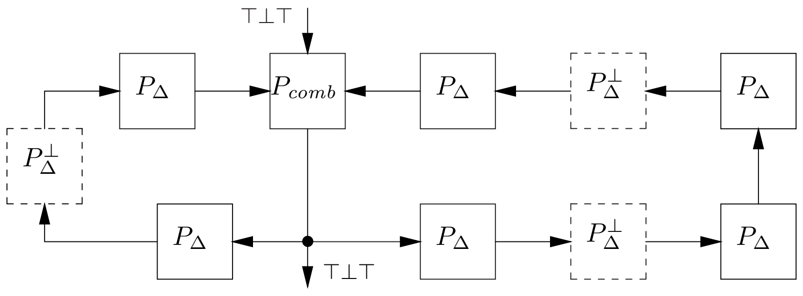


Figure 7.4. Possible locations of synchronization delays in a system with two loops

Similarly to the delays of loops, the pattern length has to match the delay differences in pairs of paths. The idea of the synchronization is to avoid that values of different types arrive at a multi-input process in the same clock cycle. For example, in Figure 7.5 combinational processes P_1 and P_2 are connected by two paths. Since these paths have different delays, one

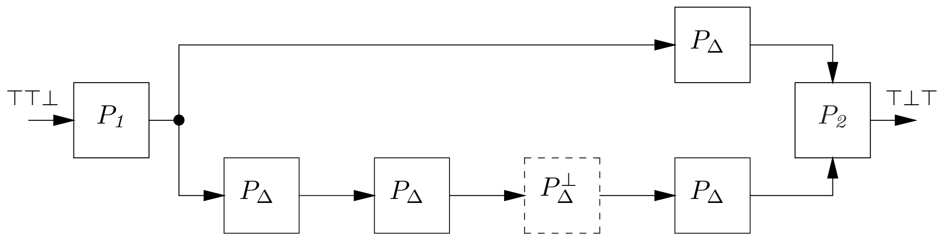


Figure 7.5. Synchronization in a pair of paths

and three, respectively, $\perp$-values sent by P_1 arrive in different clock cycles at P_2. However, P_2 may receive $\perp$-values concurrently on both inputs, if the system operates with patterns that are equivalent to $\top\top\perp$, and one synchronization delay process $P_\Delta^\perp$ is added on the lower path. Hence, the pattern length calculation based on the original model has to take into account the delay differences in pairs of paths as well. The initial pattern length before temporal refinements is calculated as the greatest common divisor of the delays of loops and the delay differences in pairs of paths.

7.4 The Synchronization Algorithm

7.4.1 Outline of the Algorithm

The algorithm is divided into preparation and synchronization phases. In the preparation phase, before any temporal refinements, Algorithm 7.1 analyzes the structure of the original model. After finding all loops and pairs of paths, the algorithm calculates the initial pattern length N, forms an ordered N-element set of labels and gives a label to every delay process in the model. The algorithm performs the labeling so that delay processes, which have the same delay to a multi-input process (or the delay difference is equal to $k * N$ $(k \in \mathbb{N})$ [1]) get identical labels. In the second phase, Algorithm 7.2 synchronizes the system by adding synchronization delay processes $P_\Delta^\perp$ to the model. Synchronization delays get places according to the labels, which were assigned by Algorithm 7.1.

[1]$\mathbb{N}$ is the set of natural numbers

7.4.2 Preparation before Temporal Refinements

Algorithm 7.1 *Labeling of Delay Processes*

Step 1 Find all loops in the delay graph.

Step 2 Find all pairs of paths in the delay graph.

Step 3 Calculate the delays of loops $\Delta_i = |loop_i|_\Delta$.

Step 4 Calculate the delay differences in all pairs of paths $\Delta_j = abs(|path_a|_\Delta - |path_b|_\Delta)$.

Step 5 Calculate the greatest common integer divisor N of all values Δ_i and Δ_j that were found in the previous steps. N is the initial length for the patterns.

Step 6 Create an ordered N-element set *Label* that contains distinct labels - $\{L_0, L_1, \ldots, L_{N-1}\}$. In the following, capital letters $A, B, C, \ldots$ are used as labels.

Step 7 Select an arbitrary vertex in the delay graph and give the first label L_0 to the vertex and to the corresponding delay process in the model.

Step 8 Label all vertices and delay process according to the following rules. If vertex w_i has got label L_j and there is an edge from w_i to w_l give the label L_{j+1} (L_0 if $j == (N-1)$) to w_l. Similarly, label w_k with L_{j-1} (L_{N-1} if $j == 0$) if there is an edge from w_k to w_l and w_l has got the label L_j.

Step 9 Give labels to input events. If a vertex, which corresponds to a system input has got label L_i, give L_k to the j-th input event, where $k = (i - j)\ mod\ N$.

The following techniques find loops and pairs of paths that correspond to Definitions 7.2 and 7.3 by analyzing the delay graph.

Finding loops (Step 1, Algorithm 7.1)

The technique models vertices w_i in the graph G as processes D_i and edges e_{ij} between vertices as signals between processes D_i and D_j. Each process D_i gets a distinct stamp z_i. At every step all processes receive and emit a set of vectors containing stamps. If a vector on the input of process D_i does not contain stamp z_i, D_i adds z_i to the end of the vector and forwards the extended vector to the process output. Process D_i stores received vectors that already contain stamp z_i and does not forward them. A stored vector with the first stamp equal to z_i and received by D_i, contains an ordered sequence of stamps of all processes in a loop. In order to decrease the number of vectors, only those processes emit vectors with their stamps at the first step, whose outputs are connected to more than one process. This restriction does not leave any loop uncovered, under assumption that at least one such a process can be found in each loop. Since a vector can pass no process twice, the maximum number of steps the technique has to run is bounded by the number of delay processes in the model.

Finding pairs of paths (Step 2, Algorithm 7.1)

The pair finding technique models the graph G as a set of processes and signals, and gives a distinct stamp z_i to each process D_i, as it was done in the finding loop technique. At the first step only those processes whose outputs are connected to more than one process emit vectors with their stamps. At every further step all processes add their stamps to the end of received vectors and forward them. Multi-input process D_i discards a received vector if the vector already contains its own stamp z_i. All other vectors received by D_i get extended with the stamp z_i, are stored in the process, and after that forwarded to the process output. Again, it takes a fewer number of steps than the number of processes D_i in the model. In the end of the run, each pair of stored vectors, which have identical stamps only in the first position and in the last position, contains stamps of pairs of paths.

Example 7.1

Let's consider the system in Figure 7.6 that illustrates the preparation phase of the synchronization algorithm. The original model in Figure 7.6(a) is constructed as a process network that includes eight combinational processes $P_1, \ldots, P_8$ and four delay processes $P_{\Delta 9}, \ldots, P_{\Delta 12}$. The delay graph presenting connections between the input, the output and the delay processes in the original model is shown in Figure 7.7(a). Algorithm 7.1 performs the following steps:

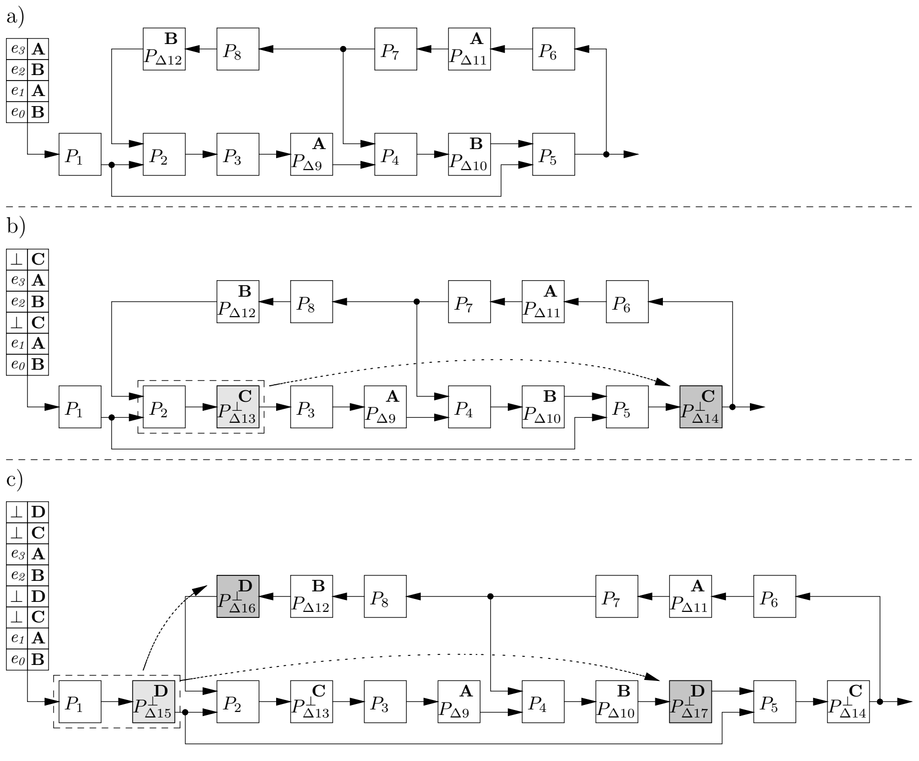


Figure 7.6. Synchronization example: (a) model after labeling, (b) after the first refinement, (c) after the second refinement

Step 1 According to the delay graph, delay processes form two loops: $loop_1 = \{w_{10}, w_{11}, w_{12}, w_9\}$ and $loop_2 = \{w_{10}, w_{11}\}$.

Step 2 Pairs of paths starting from w_{in} are:
$(path_1 = \{w_{in}, w_9\}, path_2 = \{w_{in}, w_{11}, w_{12}, w_9\})$,
$(path_3 = \{w_{in}, w_9, w_{10}\}, path_4 = \{w_{in}, w_{11}, w_{10}\})$,
$(path_5 = \{w_{in}, w_{11}\}, path_6 = \{w_{in}, w_9, w_{10}, w_{11}\})$,
$(path_7 = \{w_{in}, w_{out}\}, path_8 = \{w_{in}, w_9, w_{10}, w_{out}\})$,
$(path_7 = \{w_{in}, w_{out}\}, path_9 = \{w_{in}, w_{11}, w_{10}, w_{out}\})$,
$(path_7 = \{w_{in}, w_{out}\}, path_{10} = \{w_{in}, w_{11}, w_{12}, w_9, w_{10}, w_{out}\})$,

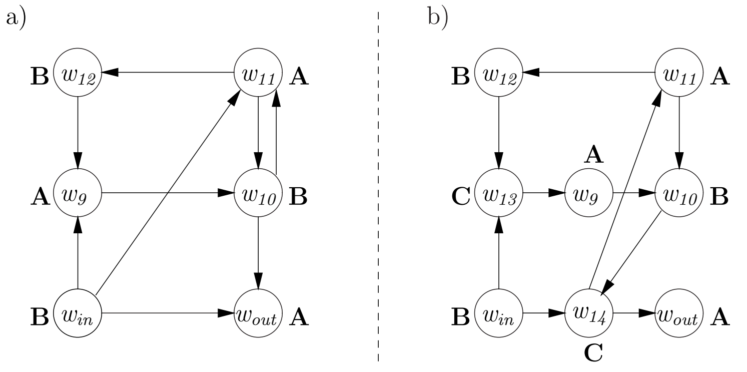


Figure 7.7. Delay graph (a) before and (b) after the first temporal refinement

and a pair starting from w_{11} is:
$(path_{11} = \{w_{11}, w_{10}\}, path_{12} = \{w_{11}, w_{12}, w_9, w_{10}\})$.

Step 3 The delays of the loops are: $\Delta_1 = |loop_1|_\Delta = 4$, $\Delta_2 = |loop_2|_\Delta = 2$.

Step 4 The delay differences in the pairs of paths are:
$\Delta_3 = |path_2|_\Delta - |path_1|_\Delta = 4 - 2 = 2,$
$\Delta_4 = |path_4|_\Delta - |path_3|_\Delta = 3 - 3 = 0,$
$\Delta_5 = |path_6|_\Delta - |path_5|_\Delta = 4 - 2 = 2.$
$\Delta_6 = |path_8|_\Delta - |path_7|_\Delta = 4 - 2 = 2,$
$\Delta_7 = |path_9|_\Delta - |path_7|_\Delta = 4 - 2 = 2,$
$\Delta_8 = |path_{10}|_\Delta - |path_7|_\Delta = 6 - 2 = 4.$
$\Delta_9 = |path_{12}|_\Delta - |path_{11}|_\Delta = 4 - 2 = 2.$

Step 5 The greatest common divisor of the previously found values Δ_i, $(1 \leq i \leq 9)$ is $N = gcd(\Delta_i) = 2$.

Step 6 An N-element ordered set of labels is $Label = \{L_0 = A, L_1 = B\}$.

Step 7 Let w_9 be the first labeled vertex and let the assigned label be A.

Step 8 The vertices w_{11} and w_{out} have label A, and the vertices w_{in}, w_{10} and w_{12} have label B. The respective delay processes in Figure 7.6 obtain the same labels.

Step 9 The input events with the even indices get label B and the events with the odd indices get label A.

The preparation algorithm gives labels so that delay processes, which have the same delay to a multi-input process have identical labels. If the delay processes with identical labels store at each clock cycle only values of the same type ($\top$ or $\bot$), then multi-input processes receive only values of the same type. The synchronization algorithm ensures that the latter behavior is preserved after temporal refinements.

7.4.3 Synchronization

The impact of a temporal refinement in a combinational process P_i can be viewed as the insertion of a delay process $P^{\bot}_{\Delta x}$ at the output of the original process P_i. In order to balance the delays of paths after temporal refinements Algorithm 7.2 follows the given labels and adds synchronization delay processes $P^{\bot}_{\Delta}$ to the refined model.

Algorithm 7.2 *Balancing the Delays of Paths*

Step 10 Add the temporal refinement produced delay $P^{\bot}_{\Delta x}$ to the model and the respective vertex w_x to the delay graph.

Step 11 Find the closest vertices w_a and w_b having edges from w_a to w_x and from w_x to w_b, respectively.

Step 12 Take the labels of $P_{\Delta a}$ and $P_{\Delta b}$. These labels locate in neighbor positions L_{i-1} and L_i (or L_{N-1} and L_0) in the set *Label*.

Step 13 Associate a new label with $P^{\bot}_{\Delta x}$.

Step 14 Preserving the order, shift all labels from positions $L_i, \ldots, L_{N-1}$ to $L_{i+1}, \ldots, L_N$.

Step 15 Insert the new label, associated with $P^{\bot}_{\Delta x}$ into the position L_i and increase the value of N by one ($N_{new} = N_{old} + 1$).

Step 16 Insert a synchronization delay process $P^{\bot}_{\Delta}$ with the label L_i into every path between processes with labels L_{i-1} and L_{i+1}, if these paths do not include any other P_{Δ} or $P^{\bot}_{\Delta}$ processes. Update the delay graph.

Step 17 Extend all input signals with $\bot$-events, so that a new $\bot$-event (with label L_i) enters to the system after every event that is labeled with L_{i+1}.

Example 7.1 continues

Let's consider two temporal refinements in the process P_2 in Figure 7.6(b) and in P_1 in Figure 7.6(c). The original model in Figure 7.6(a) is already labeled by Algorithm 7.1. The additional refinement produced delay process $P^{\perp}_{\Delta 13}$ expresses the increase of the delay in the process P_2. Algorithm 7.2 performs the following steps to synchronize the model after the first refinement.

Step 10 The first temporal refinement adds the process $P^{\perp}_{\Delta 13}$ at the output of P_2, and the vertex w_{13} to the delay graph in Figure 7.7(b).

Steps 11 and 12 The closest vertices to w_{13} are w_{in}, w_{12} and w_9 with labels $L_1 = B$, $L_1 = B$ and $L_0 = A$, respectively.

Step 13 Let C be the new label associated with $P^{\perp}_{\Delta 13}$.

Steps 14 and 15 C gets the position L_0 and the updated content of the set of labels is $\mathit{Label} = \{L_0 = C, L_1 = A, L_2 = B\}$.

Step 16 The algorithm inserts a synchronization delay process into the paths between processes with labels B and A, i.e., it adds $P^{\perp}_{\Delta 14}$ between $P_{\Delta 10}$ and $P_{\Delta 11}$ that is also between w_{in} and w_{out}.

Step 17 In the synchronized model, the input signal is extended so that an additional synchronization value $\perp$ with label C comes after every actual data item with label A.

The second refinement in the process P_1 adds the process $P^{\perp}_{\Delta 15}$ to the model, as shown in Figure 7.6(c). The algorithm adds synchronization delay processes $P^{\perp}_{\Delta 15}$ and $P^{\perp}_{\Delta 16}$ and input events with label D to ensure that values of the same type arrive at multi-input processes at each clock cycle.

7.4.4 Complexity of the Algorithm

In the worst case the number of paths grows exponentially with the number of delay processes in the model, which gives the finding loops and pairs of paths sub-algorithms exponential complexity. Let the number of vertices in the delay graph be n_w. Let the number of vertices with multiple output edges be n_m. Let the average number of output edges of all vertices be n_k. The number of vectors grows n_k times after every step in the finding loops and pairs of paths algorithms. The model contains n_m vectors in the beginning, and after x steps the number of vectors grows up to $(n_m(n_k^x))$. The maximum value of x is n_w since the number of delay processes in a loop is bounded by the number of delay processes in the model. Thus, the upper bound of vectors and paths is $(n_m(n_k^{n_w}))$. Hereby, it is important to note that delay processes only are presented in the delay graph. It is assumed that memories and buffers, similarly to finite state machine processes react to a synchronization value by sending a synchronization value to the output in the same moment, and those processes do not have delay for synchronization values. In addition, not all delay processes belong to each path.

The rest of steps in Algorithms 7.1 and 7.2 have linear complexity. Once Algorithm 7.1 has analyzed the model and given labels, Algorithm 7.2 has only to be applied after every temporal refinement.

7.4.5 Reuse of Synchronization Delays

The introduction of new delay processes that model the delays caused by temporal refinements is not always necessary, since it may be possible to reuse previously added synchronization delay processes for this purpose. For example, the synchronization delay process $P_{\Delta 14}^{\perp}$ in Figure 7.6(b) can be combined with the process P_5 in order to model the refinement added delay in P_5. In addition, it is allowed to shift $P_{\Delta}^{\perp}$ delay processes to proper places in a non-branching structure between other delay processes. The shifting is valid since the reactions of $(P_{comb} \circ P_{\Delta}^{\perp})$ and $(P_{\Delta}^{\perp} \circ P_{comb})$ are identical[2]. In both cases the first computation result is a $\perp$-value, which is followed by the reaction of P_{comb} to the input signal.

Figure 7.8 presents two choices how to model the impact of a temporal refinement in the combinational process P_1. In the model in Figure 7.8(a) a synchronization delay process $P_{\Delta 6}^{\perp}$ is added to the system and combined with P_1. In the alternative case in Figure 7.8(b) an already existing synchronization delay process $P_{\Delta 5}^{\perp}$ is shifted to the left and thereafter combined with P_1. Clearly, the latter solution is more reasonable since it performs the same temporal refinement without increasing the system's delay.

The shifting of delay processes is not limited to non-branching structures. A couple of simple examples of relocating synchronization delay processes in branching structures, known from *retiming* techniques, are presented in Figure 7.9. As shown in Figure 7.9(a), in order to model the impact of a temporal refinement in P_1, the delay processes have be moved from both

[2]Sequential composition ($\circ$) of processes $P_1(x)$ and $P_2(x)$ is defined as $P_1(x) \circ P_2(x) = P_1(P_2(x))$

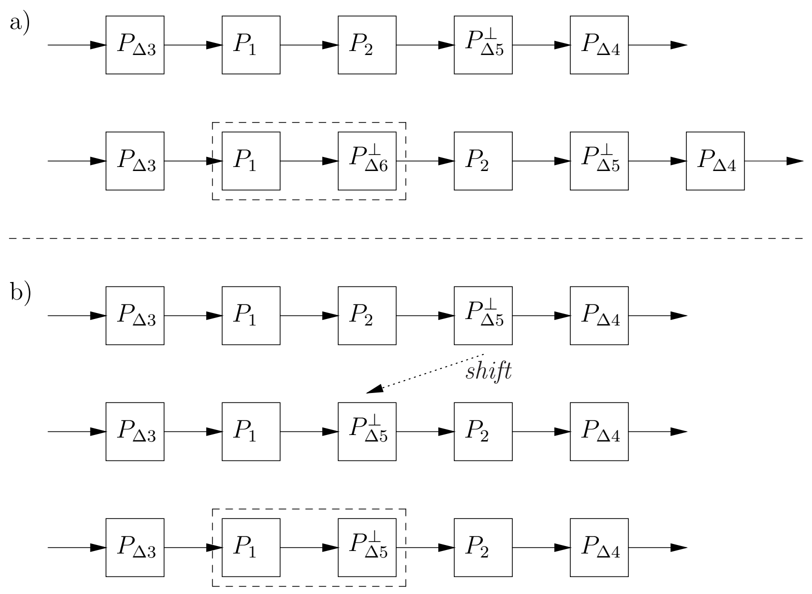


Figure 7.8. Relocation of synchronization delay processes

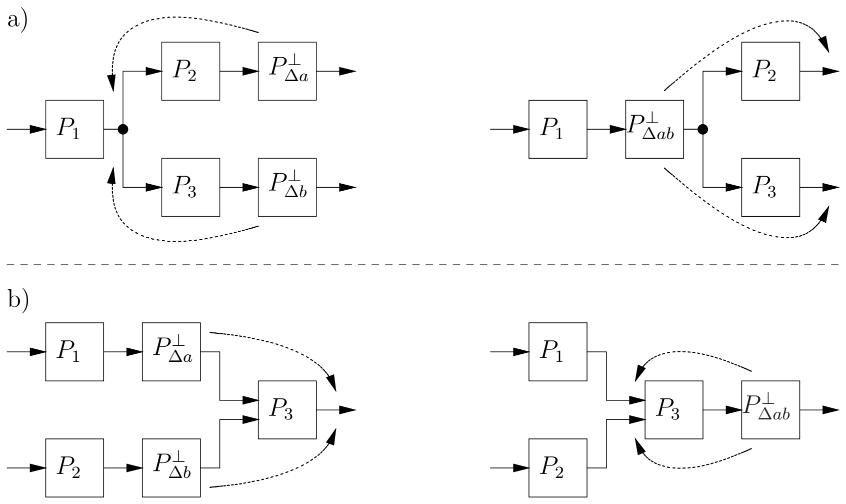


Figure 7.9. Relocation of synchronization delay processes in branching structures

branches next to P_1. In the opposite case, refining P_2 or P_3, the delay process $P_{\Delta ab}$ has to be duplicated and shifted to both branches. A similar approach is used in join structures, as shown in Figure 7.9(b).

7.4.6 Applicability of the Algorithm

The proposed synchronization algorithm is applicable to any system. The proofs of the following two assertions show that (1) for any system there exist equivalent patterns that fit, and (2) all multi-input processes in a system synchronized according to these patterns, receive only one type of values at each clock cycle.

Assertion 7.1 *For any given system there exists a pattern that fits.*

Proof : In order to find the initial pattern length, Algorithm 7.1 calculates the greatest common integer divisor of the delays of all loops and the delay differences in all pairs of paths. Since the integer value *one* is a common divisor of any set of integers, the respective one element pattern $\top$ fits to any system. After a temporal refinement all processes in this kind of model operate with patterns equivalent to $\perp\top$.

□

Assertion 7.2 *In a system that is synchronized according to patterns equivalent to* $\perp\top$ *none of the multi-input processes receive values of different types at any clock cycle.*

Proof : Let's replace all delay processes P_Δ in the original model with sequential compositions of delay processes $P_\Delta^\perp \circ P_\Delta$. In the modified model all processes, including multi-input ones, operate with the actual data ($\top$) values at every odd clock cycle and with synchronization ($\perp$) values at every even clock cycle, i.e., processes operate with patterns equivalent to $\perp\top$. All combinational processes locate between P_Δ and $P_\Delta^\perp$ processes. As explained in Section 7.4.5, it is valid to shift $P_\Delta^\perp$ processes between P_Δ processes. Thus, there is a $P_\Delta^\perp$ process for any combinational process P_i that can be shifted at the output of P_i in order to model the impact of a temporal refinement. Since such a sifting does not change the order of delay processes, all multi-input processes receive values of the same type at every clock cycle. The described model is equivalent to the model synchronized by the synchronization algorithms after a temporal refinement in an arbitrary combinational process P_i.

□

7.5 Implementation Options

The synchronization algorithm extends a system after a temporal refinement so that the system operates with $\frac{N}{N_{init}}$ times more input events than the original model, where N_{init} is the initial and N the final pattern length. Due to the extra $\perp$-values the system has to operate at a higher clock frequency than the original model, to obtain the same performance. In order to adapt the refined and synchronized model to the original environment and to add additional $\perp$-values to input signals, finite state machine based interfaces can be used. As illustrated in Figure 7.10, an interface consumes input values from the environment and forwards them according to the pattern, which is defined for the input. Similarly, additional $\perp$-values have to be removed from output signals.

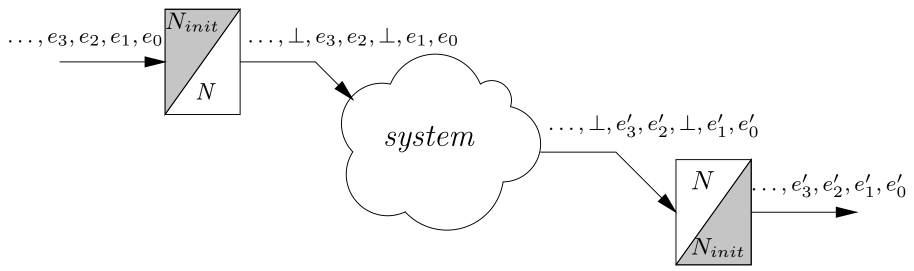

$\ldots, e_3, e_2, e_1, e_0$
N_{init}
N
$\ldots, \perp, e_3, e_2, \perp, e_1, e_0$
system
$\ldots, \perp, e'_3, e'_2, \perp, e'_1, e'_0$
N
N_{init}
$\ldots, e'_3, e'_2, e'_1, e'_0$

Figure 7.10. System with input and output interfaces.

The extension with synchronization values in data types can be implemented by a one-bit signal. In Figure 7.11 a data type $V_\perp$ is divided into a pair of signals - one carrying the actual data values V and the other indicating if the current value is a synchronization value (0) or an actual data item (1). In order to preserve the current state of a finite state machine, when a synchronization value appears on its input, the clock signal driving the internal register in the state machine is gated by an AND-gate. An alternative is to add the one-bit signal to the inputs of the next state and output functions, and to extend the functions as described in Section 7.2.

The synchronization delay processes can be implemented as registers in hardware. Since the algorithm leaves some freedom where to place synchronization delays $P_\Delta^\perp$ on a path containing only combinational and finite state machine processes, retiming techniques can be used to find an efficient solution. For example, after switching the positions of the processes P_2 and

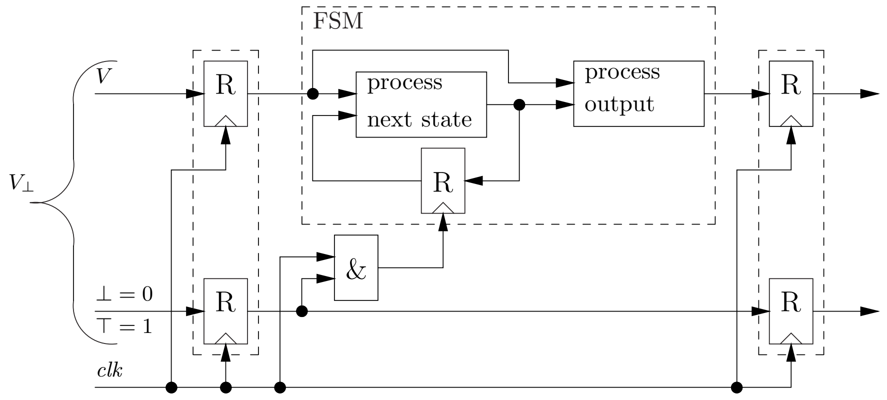


Figure 7.11. Implementation of extended data types in finite state machines.

$P^{\perp}_{\Delta 4}$ in Figure 7.12(a), it may be possible to increase the clock frequency of the system twice, if the latencies of P_1 and P_2 are equal.

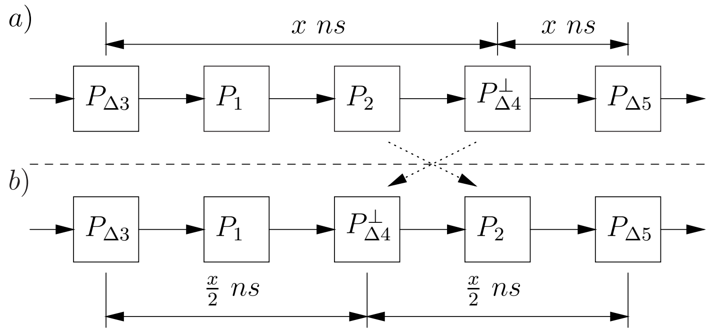


Figure 7.12. Retiming on hardware implementation.

Chapter 8

Synchronization with Sensitive Delay Elements

The synchronization algorithm developed for temporal refinements in Chapter 7 extends a system with additional synchronization delay processes. After synchronization the system acts so that the initial values from synchronization delay processes and computation results on these values do not interfere with the actual data. The main idea of the algorithm is to place synchronization delay processes so that all paths to any multi-input process deliver synchronization events concurrently. The number of additional delay processes required for synchronization and the ratio between the actual data and synchronization events depend on the system structure. In the worst case, if the delays of loops and the delay differences in pairs of paths are not multiples as in Example 7.1, all delay processes become duplicated and the number of events grows twice after a single temporal refinement. However, it may be possible to improve the ratio between the actual data and synchronization values so that a system with $N = 1$ according to Algorithm 7.1, does not necessarily process twice as much values after a temporal refinement as the original model.

The delay of a path is defined as the number of delay processes on this path. Thus, it takes as many clock cycles to transport a synchronization value through a path as many delay processes the path contains. In order to deliver synchronization values faster to multi-input processes, a shorter delay for synchronization values can be achieved through a particular modification in some delay processes. The modification makes a delay process not to cause any delay for synchronization values, and these values simply pass by the actual data values. In other words, a synchronization value overtakes the actual data, and therefore arrives at an input of a multi-input process at one clock cycle earlier. Thus, the modification makes the delay process sensitive to the types of input events. For example, an ordinary delay process with an initial value v_0 and an input sequence of values $v_1, v_2, \perp, v_3, v_4$ produces the output sequence $v_0, v_1, v_2, \perp, v_3, v_4$. A sensitive delay process ($\overline{P_\Delta}$) with the same initial value and input sequence produces the sequence $v_0, v_1, \perp, v_2, v_3, v_4$.

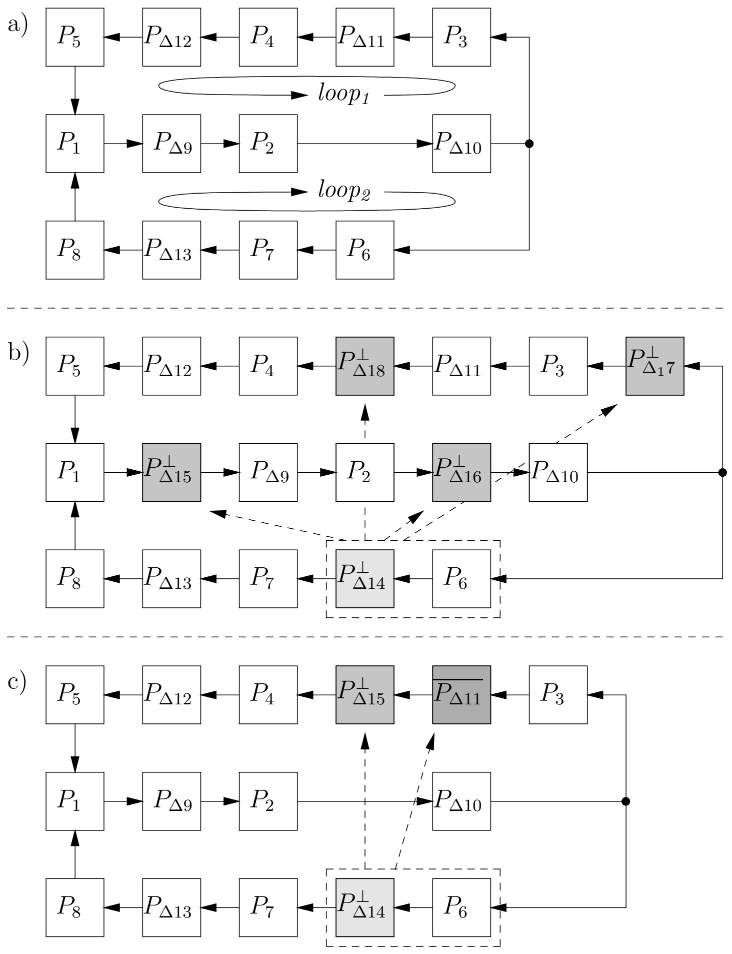
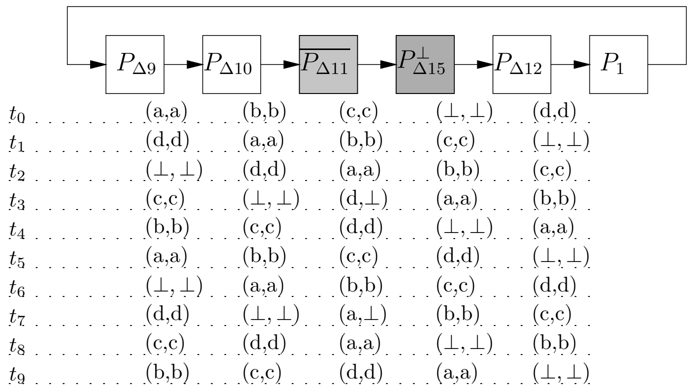


Figure 8.1. Two synchronized loops, which do not have the same number of delay elements

Figure 8.1 indicates the general difference between models synchronized by the algorithm in Chapter 7 and the algorithm, which uses sensitive delay elements that is elaborated in this chapter. Two loops in the model in Figure 8.1(a) have four and three clock cycle delays, respectively, and the greatest common divisor of these values is $N = gcd(4, 3) = 1$. After a temporal refinement in P_6 that introduces $P^{\perp}_{\Delta 14}$, the former algorithm adds four synchronization delay processes to the model, as shown in Figure 8.1(b). The refined and synchronized model has to process twice as much events as the original model to compute the same result. An improvement in performance can be achieved through the introduction of sensitivity in one of the delay processes in $loop_1$, for example in $P_{\Delta 11}$. Since the delays for $\perp$-values in $loop_1$ and $loop_2$ are equal after this modification, one additional delay process $P^{\perp}_{\Delta 15}$ is needed to synchronize the model in Figure 8.1(c) after the same temporal refinement in P_6. Also the ratio between the actual data and synchronization values is significantly improved.



	$P_{\Delta 9}$	$P_{\Delta 10}$	$\overline{P_{\Delta 11}}$	$P^{\perp}_{\Delta 15}$	$P_{\Delta 12}$
t_0	(a,a)	(b,b)	(c,c)	$(\perp,\perp)$	(d,d)
t_1	(d,d)	(a,a)	(b,b)	(c,c)	$(\perp,\perp)$
t_2	$(\perp,\perp)$	(d,d)	(a,a)	(b,b)	(c,c)
t_3	(c,c)	$(\perp,\perp)$	$(d,\perp)$	(a,a)	(b,b)
t_4	(b,b)	(c,c)	(d,d)	$(\perp,\perp)$	(a,a)
t_5	(a,a)	(b,b)	(c,c)	(d,d)	$(\perp,\perp)$
t_6	$(\perp,\perp)$	(a,a)	(b,b)	(c,c)	(d,d)
t_7	(d,d)	$(\perp,\perp)$	$(a,\perp)$	(b,b)	(c,c)
t_8	(c,c)	(d,d)	(a,a)	$(\perp,\perp)$	(b,b)
t_9	(b,b)	(c,c)	(d,d)	(a,a)	$(\perp,\perp)$

Figure 8.2. Rotating tokens in $loop_1$ with a sensitive delay process

Figure 8.2 illustrates the flow of abstract tokens in $loop_1$ within the first ten clock cycles. The first element in a pair under each delay process expresses the value stored in the delay process and the second element shows the value on the output at time instants t_i. In contrast to the ordinary delay processes $P_{\Delta 9}, P_{\Delta 10}, P_{\Delta 12}, P^{\perp}_{\Delta 15}$ whose output values are equal to the values they store at any clock cycle, the sensitive delay process $\overline{P_{\Delta 11}}$ forwards received $\perp$-values to the next delay process immediately. Instead of storing the received $\perp$-value, $\overline{P_{\Delta 11}}$ keeps the currently stored value. Therefore, the process P_1 receives synchronization values after every three clock cycles from both loops, although $loop_1$ contains one more delay process than $loop_2$.

After synchronization, the model in Figure 8.1(c) is latency equivalent to the model in Figure 8.1(a), and P_1 operates with the same pairs of the actual data values in both models.

8.1 Definitions

A few more definitions are needed in addition to those in Sections 7.2 and 7.3.

Definition 8.1 (Transformation *SyncSens***)** *Transformation* $SyncSens(P_\Delta \rightarrow P_{\overline{\Delta}})$ *replaces a delay process* P_Δ *with a sensitive delay process* $P_{\overline{\Delta}}$*, whose reactions to input values are described by the following functions* $\mathcal{F}_\Delta$ *and* $\mathcal{F}_{\overline{\Delta}}$*, respectively, (*$st_0$ *is the initial value in the delay processes,* u_i *is an input and* v_i *is an output value at a time instant* $i, (i = 0, 1, 2, \ldots)$*):*

$$\begin{aligned}
\mathcal{F}_\Delta(u_i, st_i) &= (v_i, st_{i+1}) \\
\textit{where} & \\
v_i &= st_i \\
st_{i+1} &= u_i
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_{\overline{\Delta}}(u_i, st_i) &= (v_i, st_{i+1}) \\
\textit{where} & \\
v_i &= \textit{if } (u_i == \perp) \textit{ then } \perp \textit{ else } st_i \\
st_{i+1} &= \textit{if } (u_i == \perp) \textit{ then } st_i \textit{ else } u_i
\end{aligned}$$

The function $|path_k(P_i, P_j)|_{\overline{\Delta}}$ finds the number of delay processes in $path_k$ between processes P_i and P_j excluding modified sensitive delay processes $P_{\overline{\Delta}}$. The function value is equal to the latency of the path to move a synchronization value ($\perp$) from P_i to P_j.

8.2 Synchronization Requirements

Similarly to the synchronization algorithm in Chapter 7, based on Assumption 7.1 all processes have to operate with equivalent patterns. Hence, each loop gets extended with at least one synchronization delay process $P_\Delta^\perp$ after a temporal refinement. If the shortest loop in the original model includes $\mathbf{r}$ delay processes then after a temporal refinement, all processes in the model operate with equivalent patterns containing one synchronization value ($\perp$) and at most $\mathbf{r}$ actual data values ($\top$). Let $\mathcal{R} = (1\perp : \mathbf{r}\top)$ denote the ratio between the synchronization and actual data values in a pattern. The main aim of the algorithm is to introduce sensitivity in delay processes such that the delay for $\perp$-values in all loops and the delay difference in pairs of paths correspond to the best possible ratio $\mathcal{R}$ that is defined by the delay $\mathbf{r}$ of the shortest loop in the original model. The algorithm attempts to synchronize the system to operate with $\mathcal{R} = (1\perp : \mathbf{r}\top)$.

There are two cases to consider for loops. The algorithm extends a loop, which originally contains $k * \mathbf{r}$ delay processes P_Δ, $(k \in \mathbb{N})$, with k synchronization delay processes $P_\Delta^\perp$, locating them so that the delay between any two $P_\Delta^\perp$-s is equal to $\mathbf{r}$. If the number of delay processes in a loop is not a multiple of $\mathbf{r}$, but equal to $k * \mathbf{r} + \delta$, $(\delta < \mathbf{r}$ *and* $k \in \mathbb{N})$, the algorithm selects δ proper delay processes and modifies them by the transformation *SyncSens*. The result of the modification is a loop, where the delay for synchronization values ($\perp$-values) is equal to $k * \mathbf{r}$, i.e., $|loop|_{\overline{\Delta}} = k * \mathbf{r}$.

After a temporal refinement the model can operate with the ratio $\mathcal{R} = (1\perp : \mathbf{r}\top)$ if all loops satisfy the following condition.

Condition 8.1

$$|loop_i|_{\overline{\Delta}} \ mod \ \mathbf{r} == 0 \ and \ |loop_i|_{\overline{\Delta}} \geq \mathbf{r}, \ \ if \ \mathcal{R} = (1\perp : \mathbf{r}\top)$$

Similar to loops, also pairs of paths have to have a delay differences that correspond to $\mathcal{R}$. The delays for $\perp$-values in $path_1(P_a, P_b)$ and $path_2(P_a, P_b)$ that form a pair of paths have to be equal or have to differ by $k * \mathbf{r}$ clock cycles. Only in this case values of the same type ($\perp$ or $\top$) arrive concurrently at P_b at every clock cycle.

The model can operate with the ratio $\mathcal{R}$ if all pairs of paths fulfill the following condition.

Condition 8.2

$$(|path_1(P_a, P_b)|_{\overline{\Delta}} - |path_2(P_a, P_b)|_{\overline{\Delta}}) \ mod \ \mathbf{r} == 0, \ \ if \ \mathcal{R} = (1\perp : \mathbf{r}\top)$$

The number of delay processes that have to be modified by the transformation *SyncSens* in a $loop_i$ or in a $pair_i$ of paths, in order to satisfy the previous Conditions 8.1 and 8.2, can be expressed as an offset δ_i.

The offset of a $loop_i$ is:

$$\delta_j = |loop_j|_{\overline{\Delta}} \ mod \ \mathbf{r} \tag{8.1}$$

The offset of a $pair_i$ of $path_a$ and $path_b$ is:

$$\begin{aligned} &\delta_i = min(\delta_{ia}, \delta_{ib}) \\ &\quad where \\ &\quad\quad \delta_{ia} = (|path_a|_{\overline{\Delta}} - |path_b|_{\overline{\Delta}}) \ mod \ \mathbf{r} \\ &\quad\quad \delta_{ib} = (|path_b|_{\overline{\Delta}} - |path_a|_{\overline{\Delta}}) \ mod \ \mathbf{r} \end{aligned} \tag{8.2}$$

Since a pair is formed by two paths, the offset of a pair depends on in which path the delay processes are modified by *SyncSens*. For this reason, the offset computation in the paths finds the minimum value. The model is *balanced* for a given ratio $\mathcal{R}$ if the offsets of all loops and pairs of paths are equal to zero.

The offset δ_i of any $loop_i$ is always lower than the number of delay processes in $loop_i$ and the offset δ_j of $pair_j$ is always less or equal to the number of delay processes in the paths that belong to $pair_j$. This leaves several options for the selection of the delay processes to be modified. Since loops and pairs share delay processes, the modification in one loop or pair, to turn its offset to zero, may cause an offset growth in another loop or pair. Therefore, it is important to find a proper *combination* of delay processes, which satisfies all loops and pairs. The combination can be viewed as a binary vector $\vec{b} = \langle b_0, b_1, b_2, \ldots \rangle$ with the length equal to the number of delay processes in the original model. If b_i is equal to one, the delay process $P_{\Delta i}$ is not sensitive and has one clock cycle delay for any input value. If b_i is equal to zero, $P_{\Delta i}$ is replaced with $\overline{P_{\Delta i}}$ and implies zero delay on $\perp$-values.

Due to the system structure it may turn out that it is not possible to select the vector $\vec{b}$ such that all loops and pairs of paths satisfy Conditions 8.1 and 8.2 at the given ratio $\mathcal{R} = (1\perp : \mathbf{r}\top)$. In this case the algorithm is allowed to replace one P_Δ process with $\overline{P_\Delta}$ in the shortest loop, which determines the ratio $\mathcal{R}$. The new ratio what the algorithm follows is $\mathcal{R} = (1\perp : (\mathbf{r} - 1)\top)$. The introduction of sensitive delays in the shortest loop continues until the ratio $\mathcal{R} = (1\perp : 1\top)$. As proven in Section 7.4.6 it is possible to synchronize any system to operate with the latter ratio.

8.3 Algorithm

8.3.1 Outline of the Algorithm

In the preparation phase the algorithm analyzes the system structure by using the delay graph and finds the delays of loops and the delay differences in pairs of paths. Based on the ratio $\mathcal{R}$ that is specified by the shortest loop, the algorithm calculates an offset for each loop and pair of paths. According to the offsets, a set of delay processes are selected, which have to be sensitive in order to make the system operate with the ratio $\mathcal{R}$. The sensitivity to the types of input values is introduced by the transformation *SyncSens*. All unmodified delay processes get labels according to their location. After a temporal refinement, Algorithm 7.2 follows the given labels and inserts synchronization delays $P_\Delta^\perp$ so that the system becomes synchronized.

8.3.2 Preparation Phase

Algorithm 8.1 *Labeling of Delay Processes*

Step 1 Find all loops in the delay graph.

Step 2 Calculate the delays of loops $\Delta_i = |loop_i|_\Delta$.

Step 3 Find a loop with the smallest delay $\mathbf{r}$ that determines the initial ratio $\mathcal{R} = (1\perp : \mathbf{r}\top)$.

Step 4 Find all pairs of paths in the delay graph.

Step 5 Calculate the delay differences in all pairs of paths $\Delta_j = abs(|path_a|_\Delta - |path_b|_\Delta)$.

Step 6 Calculate an offset δ for each loop and pair of paths.

Step 7 Based on the offsets find a combination $\overrightarrow{b}$ of delay processes that after modifications by the transformation *SyncSens* allows the system to operate with the ratio $\mathcal{R}$. If for the given ratio does not exist such a combination, repeat this step with a new ratio, where $\mathbf{r} = (\mathbf{r_{old}} - 1)$, until a proper combination is found.

Step 8 Apply the transformation *SyncSens* to the delay processes indicated by the found combination $\overrightarrow{b}$.

Step 9 Create an $\mathbf{r}$-element ordered set *Label* that contains distinct labels - $\{L_0, L_1, \ldots, L_{\mathbf{r}-1}\}$.

Step 10 Select an arbitrary vertex in the delay graph that represents an unmodified delay process. Give the first label L_0 to the selected vertex and to the respective delay process.

Step 11 Label all unmodified delay process and corresponding vertices, according to the following rules. If vertex w_i has got label L_j and there is an edge from w_i to w_l give the label L_{j+1} (L_0 if $j == (\mathbf{r}-1)$) to the vertex w_l. Similarly, label w_k with L_{j-1} ($L_{\mathbf{r}-1}$ if $j == 0$) if there is an edge from w_k to w_l and w_l has got the label L_j.

Step 12 Give labels to input events. If a vertex that represents a system input has got label L_i, give label L_k to the j-th input event, where $k = (i - j)\ mod\ \mathbf{r}$.

The finding loops and pairs of paths techniques and the labeling issues were elaborated in Chapter 7, and the following discussion concentrates mainly on the search of vector $\vec{b}$.

8.3.3 Finding a Combination Vector $\vec{b}$

In order to analyze the system structure in the sense of delay processes and to find a vector $\vec{b}$, the information about all loops and pairs of paths is collected into a matrix $\mathcal{M}$. Columns c_i in $\mathcal{M}$ correspond to delay processes $P_{\Delta i}$ and rows r_j correspond to loops $loop_j$ and pairs $pair_j$; each loop and pair has its own row in $\mathcal{M}$. Matrix element $m(r_j, c_i)$ has the value one if $P_{\Delta i}$ belongs to $loop_j$, otherwise $m(r_j, c_i) = 0$. For paths, $path_a$ and $path_b$ ($|path_a|_\Delta \geq |path_b|_\Delta$) in $pair_j$ between processes $P_{\Delta k}$ and $P_{\Delta l}$, $m(r_j, c_i) = 1$ if $P_{\Delta i} \in path_a$, $m(r_j, c_i) = -1$ if $P_{\Delta i} \in path_b$ and all other matrix elements in r_j, including $m(r_j, c_k)$ and $m(r_j, c_l)$, are equal to zero.

The delay of $loop_j$ can be calculated by summing up the values in r_j (Σr_j) and the delay difference in $pair_j$ is equal to the absolute value of the sum of the values in r_j ($abs\Sigma r_j$). Referring to the left hand side of Conditions 8.1 and 8.2, $\Sigma r_j = |loop_j|_{\overline{\Delta}}$ and $abs\Sigma r_j = abs(|path_a|_{\overline{\Delta}} - |path_b|_{\overline{\Delta}})$.

The offset δ_j of $loop_j$ calculated on the values in r_j is:

$$\delta_j = (\Sigma r_j)\ mod\ \mathbf{r} \tag{8.3}$$

The offset δ_j of $pair_j$ formed by $path_a$ and $path_b$ in r_j is:

$$\begin{aligned} &\delta_j = min(\delta_{ja}, \delta_{jb}) \\ &\quad where \\ &\qquad \delta_{ja} = (\Sigma r_j)\ mod\ \mathbf{r} \\ &\qquad \delta_{jb} = (-\Sigma r_j)\ mod\ \mathbf{r} \end{aligned} \tag{8.4}$$

The content of every row r_j in the matrix $\mathcal{M}$ determines one offset δ_j. The application of the transformation *SyncSens* to delay process $P_{\Delta i}$ turns all values in column c_i in $\mathcal{M}$ equal to zero, and thereby changes the values of offsets as well. Thus, the application of the transformation *SyncSens* to delay processes that are indicated by the vector $\overrightarrow{b}$ is equivalent to update the content of the matrix $\mathcal{M}$. New values in every row r_j can be calculated by multiplying the current values in r_j with the respective values in $\overrightarrow{b}$, $m_{new}(r_j, c_i) = m_{old}(r_j, c_i) * b_i$.

A vector $\overrightarrow{b}$ makes the model balanced if the following condition holds.

Condition 8.3

$$\forall j, (\sum_i (m(r_j, c_i) * b_i))\ mod\ \mathbf{r} == 0$$

Utilization of a Model Checker

A proper vector $\overrightarrow{b}$ that satisfies Condition 8.3 can be found by using a model checker. In the model checking problem the vector $\overrightarrow{b}$ is modeled as a binary vector and the model checker can nondeterministically assign the values 0 and 1 to each element b_i, which corresponds to the delay process $P_{\Delta i}$. The specification that has to be checked is composed by a set of logic expressions that are derived from Condition 8.3. The expression for $loop_j$ is:

$$\begin{aligned} exp_j = ((\textstyle\sum_i (m(r_j, c_i) * b_i))\ mod\ \mathbf{r} == 0) \bigwedge \\ \bigwedge((\textstyle\sum_i (m(r_j, c_i) * b_i)) \geq \mathbf{r}) \end{aligned} \tag{8.5}$$

The expression for $pair_j$ of $path_a$ and $path_b$ is:

$$exp_j = ((\textstyle\sum_i (m(r_j, c_i) * b_i))\ mod\ \mathbf{r} == 0) \tag{8.6}$$

The task for the model checker is to verify if there exists a content of $\overrightarrow{b}$ such that the following specification is satisfied:

$$SPEC \ = \ \neg\mathbf{E}(exp_1 \& exp_2 \& exp_3 \& \cdots) \tag{8.7}$$

The specification says that there does not exist a case (a vector $\overrightarrow{b}$), where all expressions exp_j are true at the same time. If the model checker finds that the given specification is not satisfied, it reports a counter example. The counter example is a combination of values assigned to b_i-s, which violates the specification. Since the specification is defined through negation the values in the counter example satisfy all expressions exp_j.

If b_i is equal to zero in the counter example, delay process $P_{\Delta i}$ has to be replaced with $\overline{P_{\Delta_i}}$, otherwise $P_{\Delta i}$ stays unmodified. The found vector $\overrightarrow{b}$ may not be optimal because it corresponds to one arbitrary counter example. In order to limit the number of sensitive delays, the *SPEC* in model checking may include an additional expression: $exp_k = \sum_i(b_i) \geq n_u$, where n_u is the least number of unmodified delay processes that are allowed in the model. The expression $exp_l = b_i == 1$ requires that the delay process $P_{\Delta i}$ has to stay unmodified, and the expression $exp_m = b_i \vee b_j$ says that only one of $P_{\Delta i}$ and $P_{\Delta j}$ may be modified.

Algorithm 8.1 labels all unmodified delay processes in the original model, and Algorithm 7.2 inserts synchronization delays between unmodified delay processes according to the given labels after temporal refinements.

8.4 Example

The following example illustrates the synchronization of a model by using sensitive delay elements after a temporal refinement. The model contains fifteen computation blocks that all contain one delay process $P_{\Delta j}$. Hence the delay graph of the model in Figure 8.3 is identical to the structure of the system abstracting from all combinational and finite state machine processes. The model contains six loops and fourteen pairs of paths. Prototype tools of the finding loop and pairs algorithms (Section 7.4.2) that were implemented in the ForSyDe modeling environment spent less than 0.1 seconds on a Sun Ultra 80 machine (450MHz CPU and 4GB RAM) to find all loops and all pairs of paths. Examples of loops are: $loop_1$ with vertices $\{P_{\Delta 9}, P_{\Delta 10}, P_{\Delta 11}\}$, $loop_2$ with vertices $\{P_{\Delta 9}, P_{\Delta 10}, P_{\Delta 12}, P_{\Delta 11}\}$, $loop_3$ with vertices $\{P_{\Delta 6}, P_{\Delta 13}, P_{\Delta 14}, P_{\Delta 15}\}$. In the matrix $\mathcal{M}$, shown in Figure 8.4, the first six rows correspond to the loops. Matrix element $m(r_j, c_i)$ has the value one if P_{Δ_i} belongs to $loop_j$. The rest of the rows in the matrix $\mathcal{M}$ represent pairs of paths, where the delay processes of the shorter path are marked with -1 and the longer path with 1.

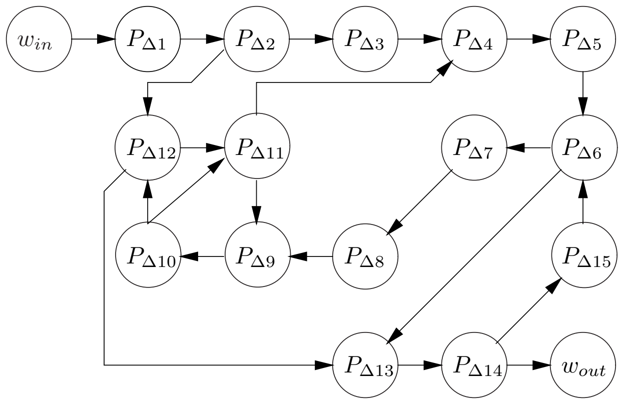


Figure 8.3. Delay graph of a system before temporal refinements

The shortest loop in the model that determines the ratio $\mathcal{R}$ is formed by three delay processes. Therefore after a temporal refinement the model should operate with the ratio $\mathcal{R} = (1\bot : 3\top)$. The sums of elements Σr_j and the offsets δ_j according to the ratio $\mathcal{R}$ are given in the right hand side columns in Figure 8.4.

For the SMV model checker [102] it took 0.11 seconds to find which delay processes have to be sensitive in order to make the model balanced and to operate with the given ratio, i.e., all loops and pairs satisfy Conditions 8.1 and 8.2, respectively. The model checker found a counter example, where b_6, b_7, b_{12} in $\overrightarrow{b}$ are equal to zero and the rest of the elements are equal to one. The application of the transformation *SyncSens* to the delay processes $P_{\Delta 6}$, $P_{\Delta 7}$, $P_{\Delta 12}$ makes them sensitive, and replaces all values in the columns c_6, c_7, c_{12} with zeros (the shadowed columns in $\mathcal{M}$). The updated values of Σr_i and δ_i are presented as $\overline{\Sigma r_i}$ and $\overline{\delta_i}$, respectively, in Figure 8.4. After the transformations, all rows in the modified matrix satisfy Condition 8.3, and all loops and pairs of paths satisfy Conditions 8.1 and 8.2.

According to the value of $\mathbf{r}$, the set of labels contains three labels: $\{L_0 = A, L_1 = B, L_2 = C\}$, which are used to label all unmodified delay process P_Δ as shown in Figure 8.5.

Let a temporal refinement increase the delay in the block the delay process $P_{\Delta 11}$ belongs. The impact of the transformation is equivalent to inserting an additional delay process $P^{\bot}_{\Delta 16}$ as shown in Figure 8.5. Since the refinement places $P^{\bot}_{\Delta 16}$ between delay processes with labels B and C, the new label D of $P^{\bot}_{\Delta 16}$ gets the position L_2 in *Label*, i.e.,

	$r_j \backslash c_i$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Σr_j	δ_j	$\overline{\Sigma r_j}$	$\overline{\delta_j}$
$loop_1$	1									1	1	1					3	0	3	0
$loop_2$	2									1	1	1	1				4	1	3	0
$loop_3$	3						1							1	1	1	4	1	3	0
$loop_4$	4						1	1	1	1	1		1	1	1	1	9	0	6	0
$loop_5$	5				1	1	1	1	1	1	1	1					8	2	6	0
$loop_6$	6				1	1	1	1	1	1	1	1	1				9	0	6	0
$pair_{2,4}$	7			-1								1	1				1	1	0	0
$pair_{2,4}$	8			-1			1	1	1	1	1	1	1	1	1	1	9	0	6	0
$pair_{2,6}$	9			-1	-1	-1							1	1	1	1	1	1	0	0
$pair_{6,13}$	10							1	1	1	1		1				5	2	3	0
$pair_{10,6}$	11				-1	-1						-1	1	1	1	1	1	1	3	0
$pair_{10,9}$	12						1	1	1			-1	1	1	1	1	6	0	3	0
$pair_{10,11}$	13												1				1	1	0	0
$pair_{10,13}$	14				1	1	1					1	-1				3	0	3	0
$pair_{11,6}$	15				-1	-1				1	1		1	1	1	1	4	1	3	0
$pair_{11,9}$	16				1	1	1	1	1								5	2	3	0
$pair_{11,13}$	17				-1	-1	-1			1	1		1				0	0	0	0
$pair_{12,6}$	18				-1	-1						-1		1	1	1	0	0	0	0
$pair_{12,11}$	19						1	1	1	1	1			1	1	1	8	2	6	0
$pair_{12,13}$	20				1	1	1					1					4	1	3	0

Figure 8.4. Matrix $\mathcal{M}$

Label $= \{L_0 = A, L_1 = B, L_2 = D, L_3 = C\}$. The input signal $\{e_0, e_1, e_2, e_3, e_4, e_5, e_6, ...\}$ gets labels $\{e_0^B, e_1^A, e_2^C, e_3^B, e_4^A, e_5^C, e_6^B, ...\}$. Algorithm 7.2 synchronizes the model by adding the synchronization delay processes $P_{\Delta 17}^{\perp}$, $P_{\Delta 18}^{\perp}$ and $P_{\Delta 19}^{\perp}$ to the model. After extending the input signal with regular $\perp$-events, the signal obtains the following format: $\{\perp^D, e_0^B, e_1^A, e_2^C, \perp^D, e_3^B, e_4^A, e_5^C, \perp^D, e_6^B, ...\}$. All processes in the synchronized model operate with the ratio $\mathcal{R} = (1\perp : 3\top)$. The refined and synchronized model and the original model are modeled in the ForSyDe environment, and they produced latency equivalent signals. Without using sensitive delay processes for synchronization the model after the same temporal refinement should operate with the ratio $\mathcal{R} = (1\perp : 1\top)$.

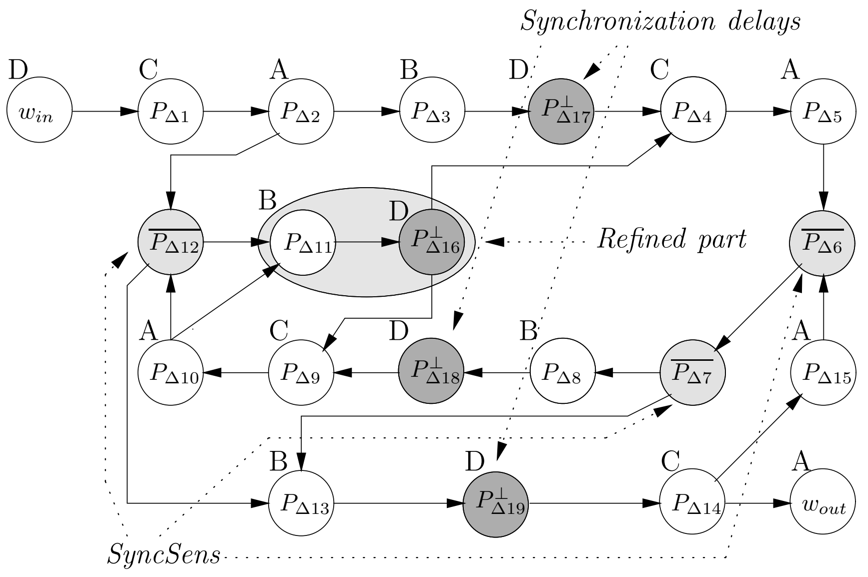


Figure 8.5. Delay graph after temporal refinement

8.5 Hardware Implementation of Sensitive Delays

Sensitive delays are implemented as Mealy state machines in the ForSyDe synchronous model. A possible hardware implementation of a sensitive delay process ($\overline{P_{\Delta 2}}$) with an additional *and*-gate is depicted in Figure 8.6. Again, a one-bit signal is used to notify processes if the value on a process input belongs to the actual data or is used for synchronization. The *and*-gate ensures that the sensitive delay process does not update its content if a synchronization value ($\perp$) appears at its input. Since synchronization values overtake the actual data stored in sensitive delays, synchronization values on the one-bit signal have to reach the next unmodified delay element within one clock cycle, as it is with the signal s_1 in Figure 8.6. Therefore the given solution requires that the communication delays are marginal compared to the computation delays.

8.6 Discussion

The described synchronization algorithms in Chapters 7 and 8 use simple delay elements to synchronize models after temporal refinements. The given approaches allow to keep the synchronous computational models through

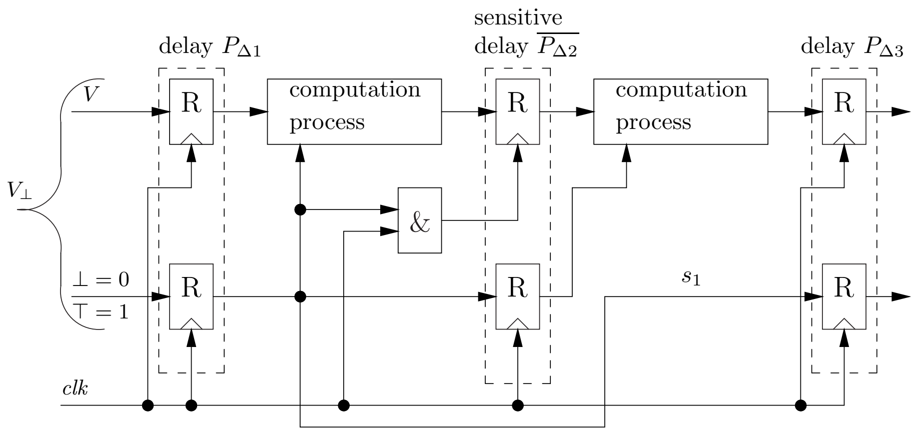


Figure 8.6. Hardware implementation of a sensitive delay

the whole system level design development process if necessary. Hence, it avoids discontinuities in the design flow that are caused by switching to other models, for instance to the LID or GALS models. In general, it is impractical to change the computational model due to a single temporal refinement, since it obliges the designer to reconsider the issues that are specific to every particular model of computation. Staying at the same computational model does not limit the choice for final implementations, since a synchronous model can be mapped to the GALS and LID models and implemented in software or hardware. On the other hand, after moving to the latter models, no additional synchronization is required after further temporal changes.

In the proposed synchronization techniques, after every temporal refinement, the application of Algorithm 7.2 increases the number of synchronization delay processes unless already existing synchronization delays are reused. In the worst case the number of additional delay processes in the model grows to the same number as the original delay processes. In addition, a one-bit signal is added to the actual data signals to distinguish between the actual data and synchronization values. Compared to the proposed solution, an LID implementation uses one-bit signals in both directions to distribute stalling messages. In addition, the handshake communication channels and wrappers contain additional buffers and combinational logic blocks. The input/output latency grows in both solutions, since loops reproduce both stalling events in LID and synchronization values in the proposed techniques.

The LID and GALS approaches are common practice in IP-block based designs. However, they have a disadvantage at system level. It is much simpler to refine a synchronous functional models through formal design refinements or to verify a synchronous deterministic design by using formal verification techniques like model checking. Although LID models by definition are synchronous, the pure synchronous communication mechanism is replaced with handshake communication channels. The handshake protocol with the stalling mechanism is a source of additional system behaviors that makes model checking more involved.

The proposed algorithms add synchronization delay processes that can be implemented as pipeline registers between combinational blocks at the RT-level. The given techniques leave some flexibility to relocate synchronization delays between other delay elements by using retiming techniques and in such a way to fulfill the latency constraints.

Chapter 9

Refinement and Verification of Digital Audio Equalizer

In several research projects that address design development issues in the ForSyDe methodology, a digital audio equalizer has been used as an example system for case studies. According to this custom, this chapter illustrates how the proposed verification and synchronization techniques are applied within the design development process of the equalizer. The digital equalizer model was described for the first time in [19]. The model considered in this chapter has some slight differences from the original, but these changes do not cause any conceptual difference. A detailed ForSyDe model of the equalizer with coding in the Haskell language is available in [90].

The input/output interfaces of the equalizer, depicted in Figure 9.1, are the audio input, the button settings input and the audio output. The listener tunes the sound through pushbuttons, whose settings drive the amplifications levels of different sound frequency bands of the input signal. In addition to the user settings, the equalizer interferes to adjusts the amplification levels, if the strength of the bass frequencies of the output signal is threatening to destroy the loudspeakers.

The functionality of the equalizer is divided into four major blocks. *Audio Filter* uses three *FIR*-filters to separate the high and low frequency bands from the middle ones. Two amplifiers follow the amplification levels provided by the button control block and either turn down or up the bass and treble bands. The output signals from the amplifiers are summed together with the middle band to give the audio output signal of the equalizer. *Audio Analyzer* applies an FFT technique to observe the bass level in the

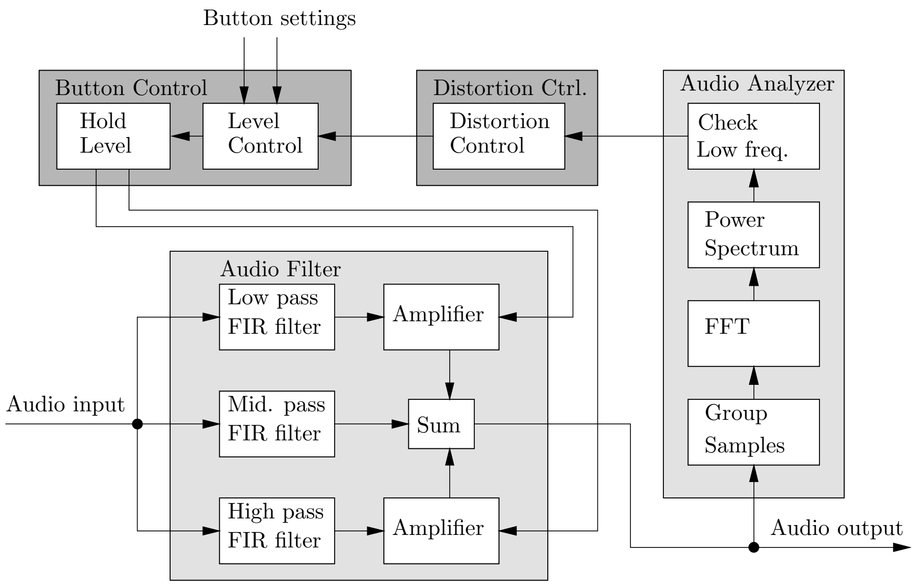


Figure 9.1. Digital audio equalizer

output signal. *Distortion Control* decides whether it is necessary to adjust the current amplification level or not. The *Button Control* block holds the button settings and adjust the amplification levels according to the data from the distortion control block.

9.1 Refinement of the FIR-filters and the FFT unit

A common style is to describe the polynomial functions of signal processing blocks in the initial system model as combinational processes by using unlimited resources as data types with unbounded domains of values and ideal arithmetic functions. Instead of implementing the FFT and FIR-filter blocks as large combinational circuits, resource sharing can be introduced in them. Following the approach described in Chapter 6, their combinational functions can be implemented as sequential computation blocks, which include a controller to execute operations in a data path, and a register file to store intermediate computation results. The sequential blocks are separated from the rest of the design by clock domain interfaces, since the computations in the refined blocks take more clock cycles than in the original model.

In order to verify that the data paths compute the same results as the combinational blocks in the original model, the polynomial abstraction technique can be used. The number of data input signals of the 16th order FIR-filter and the 8-point FFT are 32 and 16, respectively. The combinational functions of the FIR-filters and FFT can be described as polynomials, where the degrees of variables are equal to one. The degree equal to one means that the input domain of a variable has to contain two values in model checking. Therefore, all input variables that are used to describe the sequential behaviors of the FIR-filter and FFT units can be replaced with one-bit variables in the SMV models. In fact, the least possible number of bits to represent a variable is one. The SMV model checker requires less than two minutes and creates 1.5 million BDD nodes to verify that these sequential blocks implement the expected combinational functions. The verification was done on a Sun machine with 900MHz CPU and 16GB RAM.

9.2 Synchronization

The design transformation that introduces resource sharing in the FIR-filter computation blocks is classified as a temporal refinement, according to the description in Section 7.1. Although it is verified that the original and refined blocks compute the same results on input values, the sequential block has a longer latency. The longer latency causes a block to emit values in the first clock cycles that were not defined in the design specification or in the initial system model. In order to ensure that these unexpected values do not change the actual data values, the technique proposed in Chapter 7 synchronizes the system after these local refinements by inserting additional synchronization delay elements into the model. Let the clock domain interfaces, surrounding the refined FIR-filters, be configured so that the refined sequential blocks have externally only one clock cycle delay. Let the refinements started from the process $P_{FIR.L}$ in Figure 9.2.

The model has two loops, which both include only one delay process, $P_{\Delta 1}$ and $P_{\Delta 2}$, respectively. Hence, the delays of $loop_1$ and $loop_2$ are equal to one. All pairs of paths either do not include any delay process, or the delays of paths are equal. Thus, the delay differences in all pairs of paths are equal to zero. The greatest common divisor N of these values is one. According to Algorithm 7.1, all delay processes and input events have the same label A from the set of labels: $\{L_0 = A\}$.

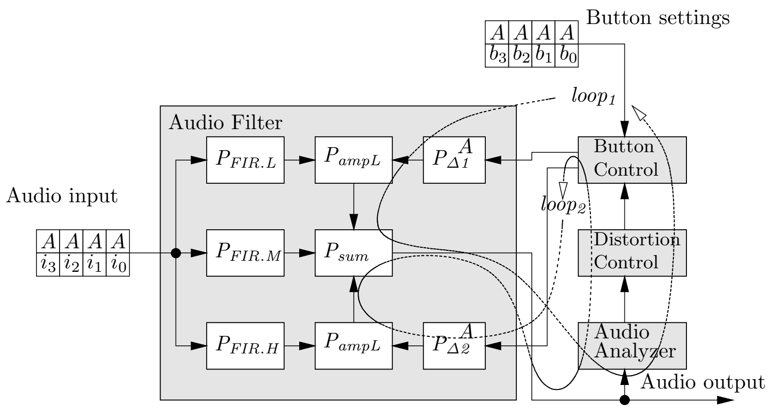


Figure 9.2. The audio equalizer after labeling

The first refinement increases the delay in the process $P_{FIR.L}$ by one clock cycle that is equivalent to add the delay process $P_{\Delta 3}^{\perp}$ at the output of $P_{FIR.L}$, as illustrated in Figure 9.3. Algorithm 7.2 labels $P_{\Delta 3}^{\perp}$ with a new

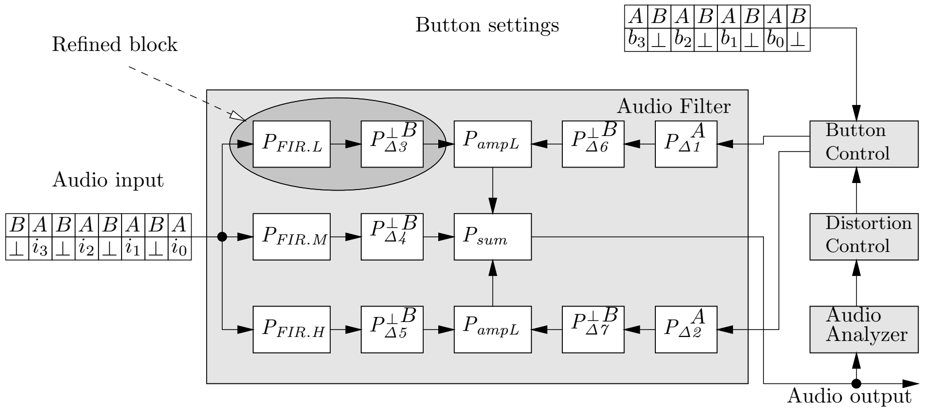


Figure 9.3. The audio equalizer after refinement in $P_{FIR.L}$

label B and the updated set of labels is $\{L_0 = A, L_1 = B\}$. Since the label B is associated with the synchronization value $\perp$, all processes have to operate with patterns equivalent to $\perp\top$. In order to synchronize the refined model, the algorithm adds four synchronization delay processes $P_{\Delta 4}^{\perp}, \ldots, P_{\Delta 7}^{\perp}$. The synchronization processes are placed so that on every path from the system input to the delay process $P_{\Delta 1}$ and $P_{\Delta 2}$ is one synchronization delay process, and on every path from $P_{\Delta 1}$ and $P_{\Delta 2}$ to $P_{\Delta 1}$ and $P_{\Delta 2}$ is one synchronization delay process as well. Two of the added synchronization processes $P^{\perp}_{\Delta 5}$ and $P^{\perp}_{\Delta 6}$ are reused, when $P_{FIR.M}$ and $P_{FIR.H}$ are refined in the same way as $P_{FIR.L}$. In addition, the input signal contains regular $\perp$-values after the synchronization. Although, the refined and synchronized model has to process twice as much input events as the original model, however, these two models are latency equivalent.

Alternatives to the applied synchronization technique are to use GALS or LID design approaches. A possible system structure of the equalizer that is implemented as an LID model is presented in Figure 9.4. In these models

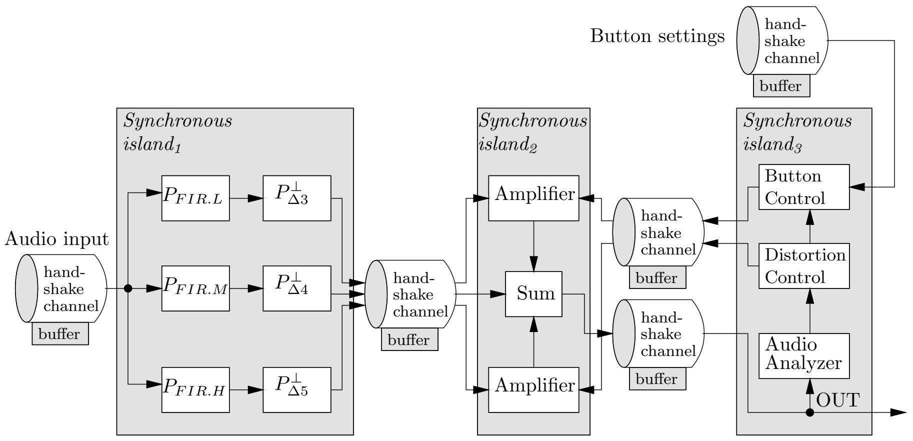


Figure 9.4. Implementation of the audio equalizer as an LID model

the equalizer is divided into (a) three synchronous islands communicating through the asynchronous media or (b) three stallable synchronous blocks surrounded by wrappers and communicating over channels by using a handshake protocol. The equalizer is divided into three islands, since after the delay growth in the FIR-filters, the actual data items do not arrive anymore concurrently at the amplifiers. A wrapper or an interface between the asynchronous media and a synchronous island triggers computation on $island_2$ as soon as all necessary input data have received. Either the interface or the buffers and wrappers store the available data until the delayed data from $island_1$ are received. Although the GALS and LID methods do not add any explicit delay process for synchronization, there are buffers in the interfaces, channels and wrappers that increase the circuit area. The synchronization value ($\perp$) extension in all data types is equivalent to the one-bit stalling signals in the LID model. Similarly to the pattern $\perp\top$, the LID model stalls computation in every second clock cycle, since feedback loops reproduce stalling events [27].

The GALS and LID approaches are very popular and common in today's design implementations. However, they may not be the best candidates for the synchronization after local temporal refinements in system level models. Due to the complex communication mechanism between synchronous islands, formal verification and formal refinements in these kinds of models are considerably more complex.

9.3 Refinement of Communication Channels

The initial equalizer model is described by means of synchronous processes that communicate via synchronous signals. Instead of implementing the whole equalizer as a synchronous circuit in hardware, a part of the equalizer's functionality can be mapped to software. Due to the lower computation load in the distortion and button control blocks than in the rest of the system, these two blocks can be implemented in software. In addition, the amplification levels are not so often modified by the distortion control block or according to changes in the button settings. The delay processes $P_{\Delta 1}$ and $P_{\Delta 2}$ can be extended with finite state machines, which store the current amplification level, and the button control block be modified so that it sends only information about changes in the amplification levels. Thereafter the synchronous signals from audio analyzer to distortion control and from button control to audio filter can be replaced with the asynchronous channels, which implement the handshake protocol, described in Section 5.2. In order to transport data over the channel, the handshake protocol requires more time than the initial synchronous signal. Clearly, the behavior of the latter model is not latency equivalent to the original model. On the other hand, if the delay from a change in buttons settings to the impact on the audio output is not noticeable to the listener, the model may be implemented in software and hardware by using the asynchronous channels.

The asynchronous channel includes a FIFO buffer, which has to have a bounded number of slots for a final implementation. Let's replace the ideal FIFO buffer with a realistic finite one, which contains eight slots. It is important to verify that the expected data rate does not cause the buffer to overflow, i.e., the channel is reliable. Let assume that the data rate is at most four data items within 36 clock cycles. The properties in Section 5.3, including the reliability property, were configured according to the given buffer size and the expected data rate, and verified by the SMV model checker on a SUN machine with 900MHz CPU and 16GB RAM. The amount of time and the number of BDD-nodes created by the model checker to prove that these properties hold, are presented in Table 9.1.

Table 9.1. Verification time and number of BDD nodes

Property	CPU time (sec.)	BDD nodes
Property 1 : Reliability	0.37	32490
Property 2 : Latency	0.11	3111
Property 3 : Bandwidth	0.38	34748
Property 4 : Order	2.56	163219

If the buffer size is too small for the expected data rate or the latency is too high, the designer has to modify the buffer size or to define a new handshake protocol. In order to verify that the new protocol is reliable, provides a certain bandwidth or satisfies the expected data rate, the same properties have to be checked. However, an arbitrary asynchronous channel may include an unexpected behavior, which is not covered by these four properties. Therefore, it is very important that after the development of a new design transformation, the verification engineer approves the use of existing verification properties or defines a new complete set of properties.

Chapter 10

Conclusion

10.1 Summary

The design process of digital systems is becoming more and more complex due to the continuous growth in the number of different functions that such systems have to perform. It is far from trivial to develop an optimal implementation from a given system specification in a limited amount of time. The objective of system level design methodologies is to start the system design at a higher level of abstraction by describing the system functionality without lower level implementation details nor targeting a certain architecture. This kind of abstract system model makes it possible to capture the system functionality more efficiently and to analyze what are the requirements for an optimal implementation. Due to the large number of different functions and the huge abstraction gap it is not realistic to verify a detailed implementation model directly against the abstract specification model.

The thesis addresses verification issues in system design methodologies like ForSyDe. In ForSyDe, the abstract system model is the entry point of the design refinement process, which introduces lower level details and in such a way develops a final implementation model. The refinement process is divided into refinement steps, which are performed by using well-defined design transformations. The proposed verification strategy takes into account the characteristics of design transformations and based on them either (a) considers that the refined model is correct by construction or (b) provides attributes to verify the refined model immediately after applying the transformation. Semantic-preserving transformations do not change the meaning of a model, hence it is not necessary to verify the model after refinement. On the contrary, non-semantic-preserving transformations change the meaning of a model, and for them, the verification attributes include critical properties that have to be checked after refinement. The properties are defined as temporal logic expression and refined models can be formally verified against them, by using model checkers. The rules for mapping of ForSyDe models to the input language of the SMV model checker are provided.

In addition to the verification properties, the attributes include abstraction techniques to reduce the size of refined models and, hence, to reduce the computer resource requirements for model checking. For computation refinements the polynomial abstraction technique is developed. This technique analyzes the computation steps in sequential design blocks that implement combinational functions, and defines abstract finite domains of input values for model checking. The domains are defined according to the degrees of input variables in the calculated output polynomials. The abstraction technique is efficiently applied to DSP applications like FIR and IIR filters at a high level of abstraction where the actual word lengths of input variables are not defined.

The verification attributes have to be provided to all non-semantic-preserving transformations in the design library and the attributes of new transformations have to be defined during the design process. Due to the size of models, the properties address only local correctness of refined design blocks and the global influence has to be analyzed separately. Two synchronization algorithms for local temporal refinements in the synchronous models of computation at system level are developed. The refined and synchronized model stays latency equivalent to the original one that is achieved by using simple delay elements and/or introducing sensitivity in delay elements. The advantage of these techniques is that they do not switch the synchronous computation model to another and do not introduce complex communication mechanisms or schedulers.

The main objective of the thesis is to show how formal verification techniques, like model checking, can be integrated into the formal system development process. The verification approach takes into account the concrete design refinements and provides assistance to the designer through predefined verification attributes.

10.2 Future Works

Design transformations The number of design transformation rules in the ForSyDe design library is rather small today. The design library has to be extended, to make it possible to apply the methodology to larger case studies. New transformations have to be developed, targeting a certain application domain since it is beyond a small research group's ability to address all kinds of systems. Together with new transformation rules, verification properties have to be defined so that the designer can easily check whether the refined models satisfy the system constraints.

Data type refinements Today, the ForSyDe methodology includes design transformations in communication, computation and time domains, but refinement and verification in the data type domain is not yet handled. Refinements, which replace ideal operations with fixed word length ones, are especially interesting. The introduction of fixed-point or saturation arithmetic involves new behaviors in the refined block and has a global influence to the model. Hence, it is necessary to verify that the refined model satisfies the design constraints.

Polynomial abstraction As already said, refinements to fixed-point and saturation arithmetic operations should be included in the methodology. At the moment, the polynomial abstraction technique is applicable to models where computation is performed on values from unlimited input domains. In order to verify the same computation blocks at lower levels of abstraction, where overflow behaviors are included, the polynomial abstraction technique should be extended to handle also modulo calculation and saturation arithmetic operations.

Verification techniques Model checking is the only formal technique that is used for verification in the ForSyDe methodology but also other techniques should be incorporated if they can perform the verification of refined models more efficiently. One possible technique is theorem proving.

Mapping to SMV So far the mapping of ForSyDe models to the input language of the SMV model checker is done manually. An automatic translation tool should be developed, based on the defined mapping rules.

References

[1] ANDRES (Analysis and Design of run-time Reconfigurable, heterogeneous Systems) Project. online: http://andres.offis.de/.

[2] S. Abdi and D. Gajski. On deriving equivalent architecture model from system specification. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC'04)*, 2004.

[3] S. Abdi, D. Shin, and D. Gajski. Automatic communication refinement for system level design. In *Proceedings of the 40th Annual Conference on Design Automation (DAC'03)*, 2003.

[4] Accellera Organization. Accellera property specification language reference manual, version 1.01., 2004.

[5] A. Acosta. Hardware synthesis in ForSyDe. Master's thesis, School for Information and Communication Technology, Royal Institute of Technology (KTH), Stockholm, Sweden, June 2007.

[6] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC'06)*, 2006.

[7] G. Arnout. SystemC standard. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC'00)*, 2000.

[8] F. L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations – computer-aided, intuition guided programming. *IEEE Transactions on Software Engineering*, 15(2), February 1989.

[9] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proceedings of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science. Springer–Verlag, October 1995.

[10] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[11] A. Benveniste, B. Caillaud, and P. L. Guernic. From synchrony to asynchrony. In *Proceedings of the International Conference on Concurrency Theory*, 1999.

[12] S. Berezin, A. Biere, E. M. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, 1998.

[13] C. L. Berman and L. H. Trevillyan. Functional comparison of logic designs for VLSI circuits. In *Proceedings of the International Conference on Computer Aided Design (ICCAD'89)*, 1989.

[14] V. Berman. System-level design language standard needed. *IEEE Design and Test of Computers*, 21(6):592–593, 2004.

[15] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[16] G. Berry, M. Kishinevsky, and S. Singh. System level design and verification using a synchronous language. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'03)*, Washington, DC, USA, 2003. IEEE Computer Society.

[17] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th Annual Conference on Design automation (DAC'99)*, 1999.

[18] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming (ICFP'98)*, 1998.

[19] P. Bjuréus and A. Jantsch. MASCOT: A specification and cosimulation method integrating data and control flow. In *Proceedings of the conference on Design, automation and test in Europe (DATE'00)*, 2000.

[20] D. Brand. Exhaustive simulation need not require an exponential number of tests. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'92)*, 1992.

[21] D. Brand. Verification of large synthesized designs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'93)*. IEEE Computer Society Press, 1993.

[22] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[23] J. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.

[24] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In *Proceedings of the sixth International Conference on Computer-Aided Verification (CAV'94)*, volume 818, Stanford, California, USA, 1994. Springer-Verlag.

[25] J. R. Burch and V. Singhal. Tight integration of combinational verification methods. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'98)*. ACM Press, 1998.

[26] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *Proceedings of International Conference on Computer Aided Design (ICCAD'99)*, 1999.

[27] L. P. Carloni and A. L. Sangiovanni-Vincentelli. A framework for modeling the distributed deployment of synchronous designs. *Formal Methods in System Design*, 28(2):93–110, 2006.

[28] M. R. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *Proceedings of the 41st Annual Conference on Design Automation (DAC '04)*. ACM Press, 2004.

[29] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, 2003.

[30] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Case studies of model checking for embedded system designs. In *proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)*, Guimarães, Portugal, June 2003.

[31] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[32] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, 2000.

[33] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[34] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 2000.

[35] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia. IPSIM: SystemC 3.0 enhancements for communication refinement. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'03)*, 2003.

[36] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In *Proceedings of the Second International Conference on Theorem Provers in Circuit Design - Theory, Practice and Experience (TPCD'94)*, London, UK, 1994. Springer-Verlag.

[37] D. Densmore, S. Rekhi, and A. Sangiovanni-Vincentelli. Microarchitecture development via Metropolis successive platform refinement. In *Proceedings of the conference on Design, automation and test in Europe (DATE'04)*, Paris, France, February 2004.

[38] R. Dmer, A. Gerstlauer, and D. Gajski. SpecC methodology for high-level modeling. In *Proceedings of the 9th IEEE/DATC Electronic Design Processes Workshop*, Monterey, April 2002.

[39] J. E. Eaton. The fundamental theorem of algebra. *American Mathematical Monthly*, 67(6):578–579, 1960.

[40] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[41] H. Foster, K. Larsen, and M. Turpin. Introducing the new Accellera Open Verification Library standard. In *Proceedings of the Design and Verification Conference (DVCoc'06)*, San Jose, CA, USA, Feb. 2006.

[42] M. Fujita and H. Nakamura. The standard SpecC language. In *Proceedings of the 14th international symposium on systems synthesis (ISSS'01)*, 2001.

[43] A. Gluska. Coverage-oriented verification of banias. In *Proceedings of the 40th Annual Conference on Design Automation (DAC'03)*, 2003.

[44] M. J. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[45] A. Habibi, A. I. Ahmed, O. A. Mohamed, and S. Tahar. On the design and verification methodology of the look-aside interface. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'05)*, 2005.

[46] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *Software Engineering*, 18(9):785–793, 1992.

[47] E. Hansen. A generalized interval arithmetic. *Lecture Notes in Computer Science 29*, 1975.

[48] R. Hojati and R. K. Brayton. Automatic datapath abstraction in hardware systems. *Lecture Notes in Computer Science 939*, 1995.

[49] G. J. Holzmann. The model checker SPIN. *IEEE Transaction on Software Engineering*, 23(5):279–295, 1997.

[50] A. Jantsch, S. Kumar, and A. Hemani. The Rugby model: A framework for the study of modeling, analysis, and synthesis concepts in electronic systems. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'99)*, 1999.

[51] V. Jerinić, J. Langer, U. Heinkel, and D. Müller. New methods and coverage metrics for functional verification. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'06)*, 2006.

[52] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Warsaw, Poland, 2003.

[53] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[54] F. Krohm, A. Kuehlmann, and A. Mets. The use of random simulation in formal verification. In *Proceedings of the International Conference on Computer Design (ICCD'96)*, 1996.

[55] J. R. L. Guerra, M. Potkonjak. System-level design guidance using algorithm properties. In *Proceedings of IEEE Workshop on VLSI Signal Processing*, volume VII, 1994.

[56] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[57] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[58] C. K. Lennard, P. Schaumont, G. de Jong, A. Haverinen, and P. Hardee. Standards for system-level design: practical reality or solution in search of a question? In *Proceedings of the conference on Design, automation and test in Europe (DATE'00)*, 2000.

[59] Z. Lu, I. Sander, and A. Jantsch. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS'02)*, Kyoto, Japan, October 2002.

[60] Z. Lu, J. Sicking, I. Sander, and A. Jantsch. Using synchronizers for refining synchronous communication onto hardware/software architectures. In *Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP'07)*, Porto Alegre, Brazil, May 2007.

[61] N. Maheshwari and S. S. Sapatnekar. Minimum area retiming with equivalent initial states. In *Proceedings of the International Conference on Computer Aided Design (ICCAD'97)*, 1997.

[62] J. P. Marques-Silva and K. A. Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Conference on Design Automation (DAC'00)*. ACM Press, 2000.

[63] G. Martin. Overview of the MPSoC design challenge. In *Proceedings of the 43rd Annual Conference on Design Automation (DAC'06)*, 2006.

[64] K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In Hu and Vardi, editors, *Conference on Computer-aided Verification (CAV'98)*. Springer-Verlag, 1998.

[65] Mentor Graphics. 0-In Formal Verification. available: http://www.mentor.com/products/fv/abv/0-in_fv/index.cfm.

[66] G. Micheli and L. Benini. Networks on chip: A new paradigm for Systems on Chip design. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'02)*, 2002.

[67] S. Nadjm-Tehrani and J.-E. Strömberg. Formal verification of dynamic properties in an aerospace application. *Formal Methods in System Design*, 14(2):135–169, March 1999.

[68] NuSMV: a new symbolic model checker. online [available] http://nusmv.irst.itc.it/.

[69] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, Saratoga, NY, June 1992. Springer-Verlag.

[70] H. A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[71] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation. In *International Conference on Computer Design (ICCD'00)*, Austin, Texas , United States, 2000.

[72] V. Paruthi, N. Mansouri, and R. Vemuri. Automatic data path abstraction for verification of large scale designs. In *International Conference on Computer Design (ICCD'98) Topic : Verification and Test*, 1998.

[73] J. Peng, S. Abdi, and D. Gajski. Automatic model refinement for fast architecture exploration. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC'02)*, 2002.

[74] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):361–414, June 1996.

[75] A. Peymandoust and G. D. Micheli. Using symbolic algebra in algorithmic level DSP synthesis. In *Proceedings of the 38th Annual Conference on Design Automation (DAC'01)*, 2001.

[76] J. Plantin and E. Stoy. Aspects of system-level design. In *Proceedings of the seventh international workshop on Hardware/software codesign (CODES'99)*, 1999.

[77] A. Pnueli, Y. Rodeh, O. Strichmann, and M. Siegel. The small model property: how small can it be? *Information and Computation*, 178(1):279–293, 2002.

[78] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Proceedings of the International Conference on Application of Concurrency to System Design*, St Malo, France, 2005.

[79] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing.* Prentice Hall, 3 edition, 1996.

[80] N. H. Z. Radu Marculescu, Umit Y. Ogras. Computation and communication refinement for multiprocessor SoC design: A system-level perspective. *ACM Transactions on Design Automation of Electronic Systems Special Issue on Novel Paradigms in System-Level Design*, 11(3), July 2006.

[81] T. Raudvere, I. Sander, and A. Jantsch. Application and verification of local non-semantic-preserving transformations in system design. *submitted to IEEE Transactions on Computer-Aided Design*, 2007.

[82] T. Raudvere, I. Sander, and A. Jantsch. Synchronization after design refinements with sensitive delay elements. In *Proceedings of CODES+ISSS'07*, Salzburg, Austria, October 2007.

[83] T. Raudvere, I. Sander, and A. Jantsch. A synchronization algorithm for local temporal refinements in perfectly synchronous models with nested feedback loops. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI'07)*, Stresa, Italy, March 2007.

[84] T. Raudvere, I. Sander, A. K. Singh, D. Gurov, and A. Jantsch. The ForSyDe semantics. In *Proceedings of the Swedish System-on-Chip Conference (SSoCC'02)*, Falkenberg, Sweden, March 2002.

[85] T. Raudvere, I. Sander, A. K. Singh, and A. Jantsch. Verification of design decisions in ForSyDe. In *Proceedings of CODES+ISSS'03*, Newport Beach, California, USA, October 2003.

[86] T. Raudvere, A. K. Singh, I. Sander, and A. Jantsch. Polynomial abstraction for verification of sequentially implemented combinational circuits. In *Proceedings of the conference on Design, automation and test in Europe (DATE'04)*, Paris, France, February 2004.

[87] T. Raudvere, A. K. Singh, I. Sander, and A. Jantsch. System level verification of digital signal processing applications based on the polynomial abstraction technique. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'05)*, San Jose, California, USA, November 2005.

[88] D. I. Rich. The evolution of SystemVerilog. *IEEE Design and Test*, 20(04):82–84, 2003.

[89] P. Sanchez and S. Dey. Simulation-based system-level verification using polynomials. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT'99)*, November 1999.

[90] I. Sander. *System Modeling and Design Refinement in ForSyDe*. PhD thesis, Royal Institute of Technology, 2003.

[91] I. Sander and A. Jantsch. System synthesis based on a formal computational model and skeletons. In *Proceedings IEEE Workshop on VLSI'99*, pages 32–39, Orlando, Florida, April 1999. IEEE Computer Society.

[92] I. Sander and A. Jantsch. Transformation based communication and clock domain refinement for system design. In *Proceedings of the 39th Annual Conference on Design Automation (DAC'02)*, New Orleans, USA, June 2002.

[93] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, January 2004.

[94] I. Sander, A. Jantsch, and Z. Lu. Development and application of design transformations in ForSyDe. In *Proceedings of the conference on Design, automation and test in Europe (DATE'03)*, Munich, Germany, March 2003.

[95] T. Seceleanu. *Systematic Design of Synchronous Digital Circuits*. PhD thesis, University of Turku, Finland, 2001.

[96] Semiconductor Industry Association. International technology roadmap for semiconductors. available: http://www.itrs.net/Links/2005ITRS/ExecSum2005.pdf.

[97] N. Shekhar, P. Kalla, and F. Enescu. Equivalence verification of arithmetic datapaths with multiple word-length operands. In *Proceedings of the conference on Design, automation and test in Europe (DATE'06)*, 2006.

[98] K. Shimizu, D. L. Dill, and A. J. Hu. Monitor-based formal specification of PCI. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, Austin, Texas, November 2000.

[99] D. Shin, S. Abdi, and D. D. Gajski. Automatic generation of bus functional models from transaction level models. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC'04)*, 2004.

[100] S. Singh. System level specification in Lava. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'03)*, 2003.

[101] J. Smith and G. D. Micheli. Polynomial methods for component matching and verification. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'98)*, San Jose, California, USA, 1998.

[102] The SMV model checker. online [available] http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/.

[103] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla. Validating families of latency insensitive protocols. *IEEE Transactions on Computers*, pages 1391 – 1401, 2006.

[104] J.-P. Talpin, P. L. Guernic, S. K. Shukla, F. Doucet, and R. Gupta. Formal refinement checking in a system-level design methodology. *Fundamental Informatica*, 62(2):243–273, 2004.

[105] S. Thompson. *Haskell - The Craft of Functional Programming.* Addison-Wesley, 2 edition, 1999.

[106] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems*, 4(4), 2005.

[107] M. Weinhardt and W. Luk. Pipeline vectorization. *In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 234–248, Feb 2001.

[108] Y. Zhu and J. H. Kukula. Generator-based verification. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'03)*, 2003.