UNIVERSITY OF CALIFORNIA,
IRVINE

Formal Analysis of Electronic System Level Models using Satisfiability Modulo Theories and Automata Checking

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical Engineering and Computer Science

by

Che-Wei Chang

2015

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

It could never be possible for me to finish Ph.D program without the supports from my academic advisor, family, friend, and colleagues in the past 5 years, and I would like to express my gratitude to these great people here.

First I would like to gratefully and sincerely thank my advisor, Professor Rainer Dömer, for his tutoring, advice, and support during my doctorate program in UC Irvine. Being his Ph.D student and teaching assistant is a great learning experience. In the meeting, he always knew when to encourage me to think out of the box and when to stop me from doing something too unrealistic, and also gave me suggestions based on his own research and teaching experience. I believe the research and teaching I experienced in his group will benefit me for my whole life.

I would like to thank Professor Daniel D. Gajski and Professor Pai Chou for serving me on my dissertation committee and giving me valuable feedback and suggestion in the defence.

I would like to thank Lisa Lin, for being such a wonderful girlfriend. I appreciate the happiness she brought to me, the support she gave to me in my sad day, and the great journey we experienced together. Being with her, life is full of joy and surprise.

I would like to thank my two best friends since college, Chien-Hung Yeh and Wei-Cheng Huang, for always encouraging me when I was sad, feeling happy for me when I was happy, and telling what I should do when I was too carried away. Also I would like to thank Heng-I Su and Lu-Yueh Hsu. Though now we are living in different continents, those short-term vacations back to Taiwan every year always gave me strength and courage to continue my work in the United States.

I would like to thank Michelle Lee, Pei-Yuan Hsieh, Mike Su, Willy Fang, and every member in the travelling group. Though they have zero contribution to my research, I really appreciate that I can know these crazy travellers. Their craziness shows me no journey is impossible, and each trip we have experienced together gave me great energy.

Many thanks my colleagues, Weiwei Chen, Yasaman Samei, Xu Han, Guantao Liu and Tim Schmidt, for the support in research and career in the past five years. It is my pleasure to work with them.

Finally, I would like to express my deepest gratitude to my parents and my brothers, for their unconditional and endless love. Their support is the greatest motivation for me to pursue my dream.

CURRICULUM VITAE

Che-Wei Chang

EDUCATION

Doctor of Philosophy in EECS **2011–2015**
University of California, Irvine *Irvine, California*

Master of Science in EECS **2009–2011**
University of California, Irvine *Irvine, California*

Master of Science in Eletrical Engineering **2001–2003**
National Cheng-Kung University *Tainan, Taiwan*

Bachelor in Eletronic Engineering **1996–2001**
Fu-Jen University *HsinChuang, Taiwan*

RESEARCH EXPERIENCE

Graduate Research Assistant **2011–2014**
University of California, Irvine *Irvine, California*

Graduate Research Assistant **2001–2003**
National Cheng-Kung University *Tainan, Taiwan*

TEACHING EXPERIENCE

Teaching Assistant **2013–2014**
University of California, Irvine *Irvine, California*

- Computation Methods
- Advanced C Programming
- Software Engineering Project in C Language

Substitute Lecturer **2014**
University of California, Irvine *Irvine, California*

- Advanced C Programming

INDUSTRY EXPERIENCE

Digital Logic Engineer **2004–2008**
Himark Technology Inc. *HsinChu, Taiwan*

REFEREED JOURNAL PUBLICATIONS

W. Chen, X. Han, C. Chang, G. Liu, and R. Dömer
Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models **2014**
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems
vol. 33, no. 12, pp. 1859-1872, December 2014

W. Chen, X. Han, C. Chang, G. Liu, and R. Dömer
Advances in Parallel Discrete Event Simulation for Electronic System-Level Design **2013**
IEEE Design and Test of Computers, vol. 30, no. 1, pp. 45-54, January-February 2013

REFEREED CONFERENCE PUBLICATIONS

C. Chang and R. Dömer
May-Happen-in-Parallel Analysis of ESL Models using UPPAAL Model Checking **March. 2015**
Proceedings of Design, Automation, and Test in Europe 2015,
Grenoble, France, March 2015

C. Chang and R. Dömer
Communication Protocol Analysis of Transaction-Level Models using Satisfiability Modulo Theories **January. 2015**
Proceedings of Asia and South Pacific Design Automation Conference 2015,
Tokyo, Japan, January 2015

C. Chang and R. Dömer
Formal Deadlock Analysis of SpecC Models Using Satisfiability Modulo Theories **June. 2013**
Proceedings of the International Embedded Systems Symposium 2013
Springer, Paderborn, Germany, June 2013

W. Chen, C. Chang, X. Han, and R. Dömer
Eliminating Race Conditions in System-Level Models by using Parallel Simulation Infrastructure **November. 2012**
Proceedings of the International High Level Design Validation and Test Workshop 2012,
Huntington Beach, California, November 2012

TECHNICAL REPORT

C. Chang and R. Dömer
Abstracting ESL Designs to UPPAAL System Models **November 2014**
Center for Embedded and Cyber-Physical Systems, Technical Report 14-13

C. Chang and R. Dömer
System Level Modeling of a H.264 Video Encoder **June 2011**
Center for Embedded Computer Systems, Technical Report 11-04

ABSTRACT OF THE DISSERTATION

Formal Analysis of Electronic System Level Models
using Satisfiability Modulo Theories and Automata Checking

By

Che-Wei Chang

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Irvine, 2015

Professor Rainer Dömer, Chair

For a system-level design which may be composed of multiple processing elements running in parallel, various kinds of unwanted consequences may happen if the system is constructed carelessly. For example, deadlock may happen if improper execution order and communication between processing elements is used in the system. Another problem which may be caused by the concurrent execution is race condition, as shared variables in the system-level model could be accessed by multiple concurrent threads in parallel. Those unwanted behaviors definitely have negative influence on the functionality of the system. Furthermore, the functionality is not the only concern in system design as timing constraints are critical as well. If the system cannot finish the job within timing constraints, it is still considered an unwanted design. To address these issues, we propose two formal analysis approaches in this dissertation to analyze three types of properties we discussed above, which are

1). liveness,

2). satisfiability of timing constraint, and

3). May-Happen-in-Parallel access.

These two approaches are based on Satisfiability Modulo Theories (SMT) and UPPAAL automaton model respectively. We run these two approaches on our in-house system models, including a JPEG encoder, MP3 decoder, AMBA AHB and CAN bus protocol models. The experimental results show our approaches are capable of analyzing those properties meeting our expectation within reasonable analysis time.

Chapter 1

Introduction

Embedded systems are widely used in our daily life nowadays. From coffee machines to the smarthouse control system, from cars to aircrafts, from smartwatches to smartphones, embedded computer system are pervasive in our modern society. As the size and complexity of the embedded system keep increasing to satisfy requirements from various area, it also imposes great difficulty in the system design [8] [9]. For this reason, or Model-Based Design approach (MBD) [8] or Electronic System Level (ESL)[9] design methodology aiming at a higher level of abstraction and with less detail is proposed to cope with the design challenge imposed by the large size, complexity, heterogeneity of the embedded system nowadays. Except for the design methodology, various validation and verification techniques for the system-level design are also proposed to assure the correctness and detect design errors at early stages of the design [5]. In this dissertation, we aim at the formal verification of the system-level design.

1.1 System-Level Design

Due to the complexity of an embedded system nowadays, it is extremely difficult for designers to consider all details of the intended system at the early stage of the design. However, without the complete picture of the entire system, it is hard to guarantee the system matches the expectation after putting all software and hardware components together and integrating them into a system. A promising approach to address this issue is *System-level Design*. The 2004 edition of the International Technology Roadmap for Semiconductors(ITRS) [4] places System-level as "a level above RTL including both hardware and software". Instead of developing the hardware and then trying to incorporate the software into the the hardware to form a complete system, designers treat the intended system as a whole entity to make sure the hardware and software can be designed jointly at the same time. This is typically referred to as Hardware/Software Co-Design [6] [7]. To achieve this goal, appropriate abstraction is required for designers to increase comprehension about the system. Abstraction allows engineers to handle the complexity of the system since the number of components at the lower level implementation can be greatly reduced to a manageable size at higher abstraction level.

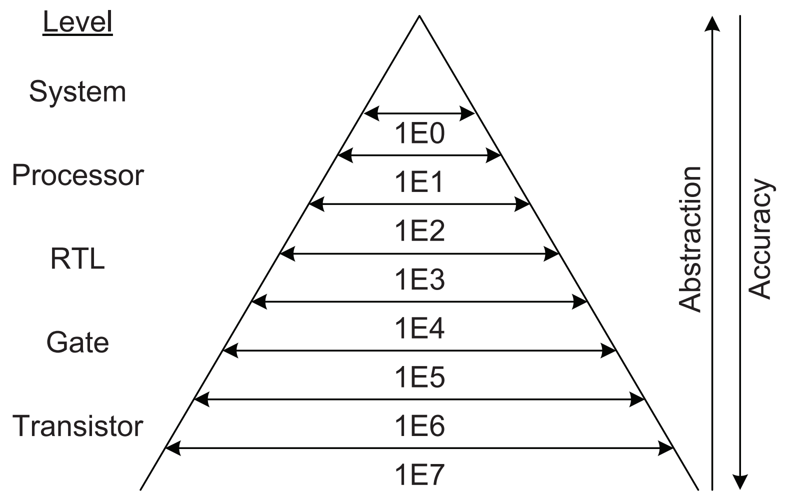


Figure 1.1: Level of abstraction in SoC design (source[10])

Fig. 1.1 illustrates the concept of using different levels of abstraction to manage the size of the design problem. For instance, the number of transistors in a system can be tens of millions. By using logic gates to represent a set of transistors, the number of components can be reduced to one tenth. Register-transfer level (RTL) abstracts the logic gates into modules and reduces the number of components to several thousands. The system can be further simplified and represented by a number of processor elements which construct the system. The approach to realize the abstraction is to using *model* to represent the behavior of the component in the system. For example, it takes tens of gates to implement an adder, but at RTL this operation can be described with a module with arithmetic statements. With this approach, designers can focus on the higher abstraction level concerns such as functional specification and algorithm which are independent from the hardware platform and software implementation, and still keep implementation features such as structure of the system or communication between modules in the design consideration. The trade-off here is that the higher the abstraction level is, the more implementation details is hidden and the less accurate the model is.

1.1.1 The Y-Chart and Top-Down Design Methodology

Gajski proposed a conceptual framework named *Y-Chart* [1] to coordinate the abstraction levels in three design domains. These three domains are *Behavior*, *Structure*, and *Physical*, and they describe the functionality of the system, the architecture of the hardware platform and the implementation detail of structure, respectively. The chart is illustrated in Fig. 1.2(A), in which abstraction levels are represented as dashed concentric circles and three design domains are shown as axes. Starting from the innermost circle representing the transistor level to the outermost circle for the system-level, in between abstraction levels such as gate-level, RTL and processor are placed in the chart in the order of abstraction where the outer circle is at a higher abstraction level than the inner circle.

Except for the level of abstraction and design domain, design flows can be illustrated as arches in the chart as well. Here we use the *Top-Down* design methodology as the example, as the refinement-based design flow we used in later chapter is of this type. As the illustration in Fig. 1.2(B), the top-down design methodology starts at the highest abstraction level and then gradually map the functional description into components supported in each abstraction level. The whole design is further converted to lower level of abstraction with more and more implementation details. Only at the transistor level abstraction, the system layout is required. The advantage of this approach includes that designers can mostly focus on the functionality at high abstraction level and easily customize the system without the distraction of low-level implementation details. However, the disadvantage here is that it is difficult to evaluate the performance accurately without the information only available at lower levels.

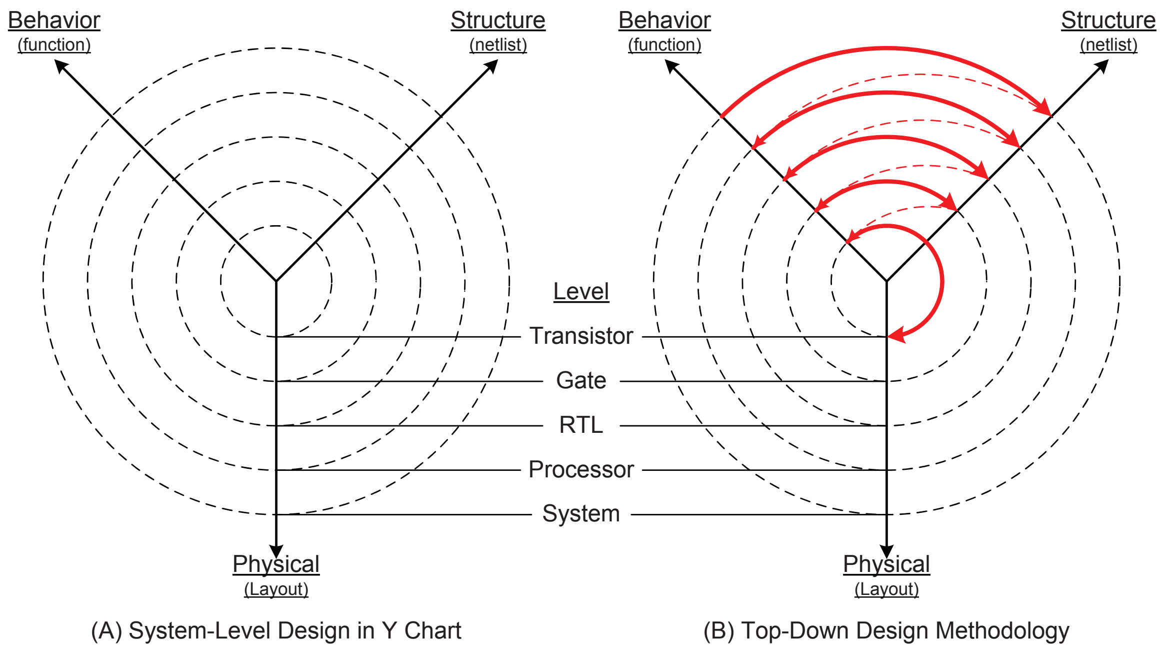


Figure 1.2: System-level design in the Y-Chart(source[11])

1.1.2 SpecC and Refinement-based Methodology

In this dissertation, we use *SpecC* system-level description language (SLDL) [12] to describe our system-level design. SpecC is based on the C language and a superset of ANSI-C, and it has a set of extensions to the constructs in ANSI-C to support the requirements of system-level modeling, which include the support of structural and behavioral hierarchy, concurrency behavior like parallel and pipelined compositions, communication over various channel types, event synchronization and timing information.

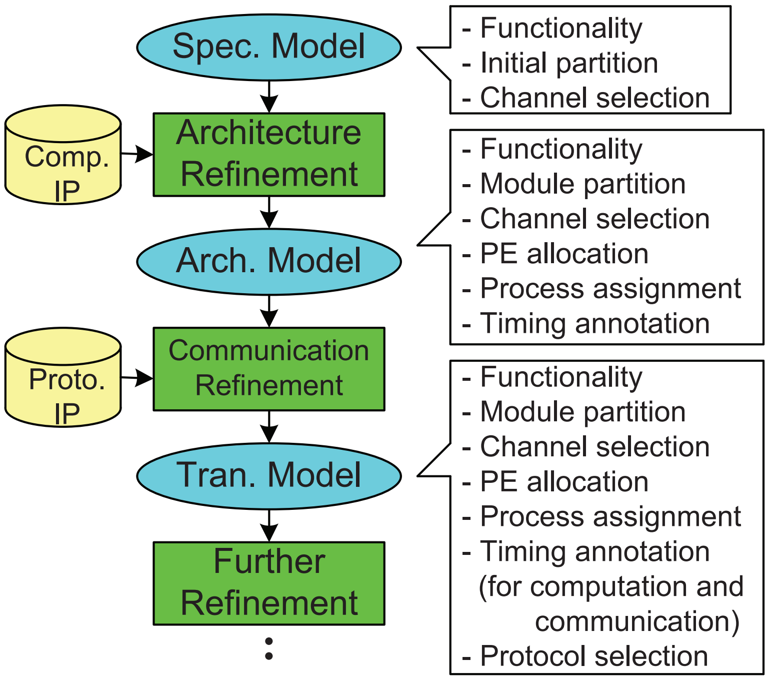


Figure 1.3: SpecC refinement-based design methodology and features of models

The design methodology in SpecC SLDL is a top-down *refinement-based* methodology illustrated as shown in Fig. 1.3. The procedure starts at the specification model in which the functionality, initial module partition and channels between modules are specified. In this model, detailed information such as timing delay caused by computation and on what bus protocol the communication is implemented are omitted so that designer can focus purely on the evaluation of the functionality. Later the specification model is refined into an architecture mode with the architecture refinement tool provided in System-on-Chip Environment (SCE)[3]. In the architecture refinement, designers specify the number and types of processing elements which will be used to implement the system, as well as what modules should be mapped to which processing element. Also, with the computational profiler in the tool, timing delay caused by the computation is also annotated to each instance. The next step is Communication refinement in which communication protocols chosen to implement the channel are inserted to the model. In this step, the delay caused by communication over chosen protocol are annotated to the model therefore the delay for communication is also taken into consideration in the model validation and verification.

More refinement steps down to the Instruction Set Simulate (ISS) are defined in SCE, but in this dissertation the models we aim to verify is at specification, architecture, and transaction level. For more information about the refinement, please refer to SCE [3].

1.2 Verification of ESL Design

1.2.1 Formal Analysis and Verification

As we described in the previous section, our system-level design flow is a top-down design methodology. Starting from the specification model which simply defines the functionality and structure, the design is refined into architecture model, transaction level model, and so on. Before a design is further refined into another model at lower abstraction level, we need to make sure the design meets our expectation. Along with the refinement, more and more real-world implementation details is added to the model, and designers can use the refined model to validate or verify the system design

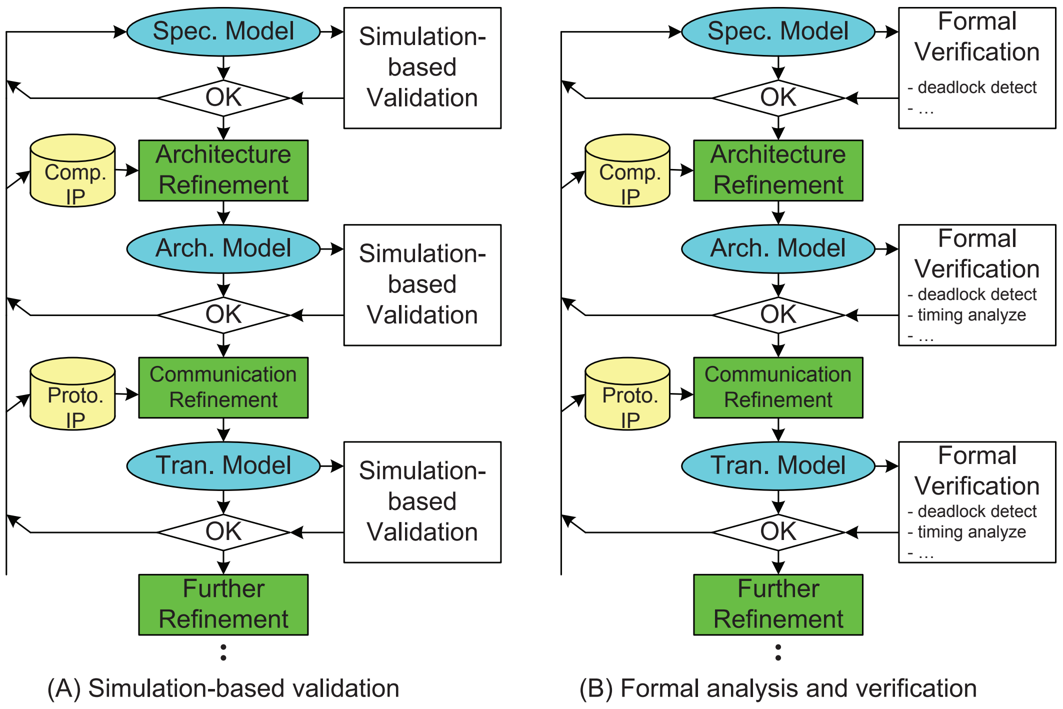


Figure 1.4: Validation and verification approach

from different aspects. We mostly care the correctness of functionality at specification level, but at architecture and transaction level model we might also want to take timing information and channel implementation into account. According to the result, designers decide whether the model should be further refined to a model closer to the real-world implementation or should go into the refinement loop and back to higher abstraction levels. In Fig. 1.4 we show two major categories of validation/verification approach along with our refinement procedure, and the advantage and disadvantage of these two approaches are shown in Fig. 1.5

The common approach to evaluate a system model is *Simulation-based Validation*, in which the system model is compiled into an executable and then run simulations on input test patterns. The advantage of this approach is that it is relatively fast, compared with formal verification. In this approach the designer provides test patterns as input and then carefully examine the output simulation to make sure the behavior of the design match the expectation. However, the advantage of this approach is

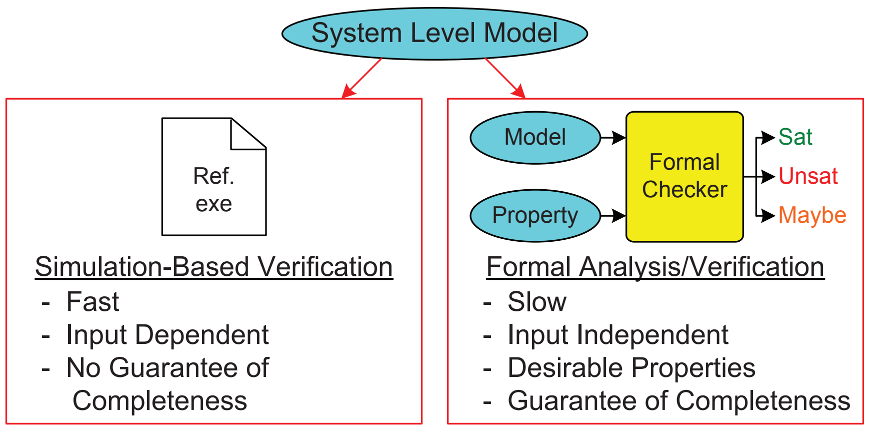


Figure 1.5: Approach comparison

also its disadvantage. The disadvantage of simulation-based verification is that it is input dependent. Though it can validate that for certain input vector the simulation result matches the expectation, simulation-based verification cannot guarantee that if a certain property is always satisfiable or not for all possible input and conditions. Therefore, the major disadvantage of simulation-based verification is that it cannot guarantee the completeness.

Compared to simulation-based verification, formal analysis and verification usually takes longer run time and requires more memory to verify a design [13]. In spite of these shortcomings compared to the simulation-based verification, formal verification still gains more and more attention because of its advantage over the other – completeness. Formal verification is input independent and the designer does not have to provide test vectors and run simulation. Instead, the designer needs to specify the desirable or undesirable property of the system, and then use formal checker to exhaustively search all possible input and conditions to prove or disprove the given property. Depending on the selected formal checker, formal verification not only checks whether the given property is satisfiable but also gives a case in which the property is satisfied, which is very useful for problem identification. For example, the designer can follow the case to find out why an undesirable property is satisfiable and then modify the design to prevent the undesirable case from happening.

1.2.2 Related Work

Functional verification of a system-level design is as important as the system itself as we want to get rid of as much potential error as possible at the early stage of the design. If a bug is found at the implementation level at lower level of abstraction, the current design may need to be abandoned if the bug is from the initial specification. For that reason, we should try to verify the design to the extent possible at higher level of abstractions. To enhance the performance and coverage of the system-level design validation and verification, techniques in both simulation-based validation and formal verification area are widely studied. For example, to accelerate the discrete event simulation for a system composed of multiple concurrent processing elements, parallel simulation techniques are proposed in [64] and [2] to make use of the power of multi-core processors.

In this dissertation, instead of running simulation on the system-level models, our proposed approaches make use of existing formal method tools [17] [18] to statically analyze the design. Formal methods are used in both hardware [19] and software [20] analysis and verification. For example, verifying the equivalence of two combinational circuit can be achieved by converting two circuit into Binary Decision Diagram (BDD) and comparing these two diagrams instead of running simulation for all possible input vectors and comparing the output [21], and model checking [24] technique, which involves of exhaustively searching the state space in finite state systems to check the satisfiability of a given properties, is used in software verification to find safety violations or liveness issue [25] as well as in hardware verification such as sequential circuit and communication protocol analysis [26]. Industries also use formal methods to improve their design methodologies. For example, EDA tool companies Synopsys and Cadence provide formal checking tools such as HECTOR[28], Formality[27], or Encounter Conformal Equivalence Checker [29] for functional verification or equivalence checking.

As for analysis of system-level model using formal verification techniques, according to the properties of interest system level models are abstracted into formal representations such as automata [38] [39] [40], Petri net [42] or Integer Linear Programming [52] problem, and then make use of existing formal solver or model checker to analyze the abstracted model. Except for approaches using formal verification techniques, there are also work relying purely on the static analysis of the model. For example, [65] analyze the model based on the execution semantics to detect potential race condition.

We use the approach converting the system-level design into another formal model for analysis. In this dissertation, we aim to convert the design into Satisfiability Modulo Theories (SMT) problem and concurrent automaton networks and use SMT theorem prover and automaton model checker to formally verify properties of interest in a system-level model. These two approaches belong to two different formal verification approach categories. The SMT-based approach belongs to deductive verification in which the design and the requirement are converted into a collection of mathematical proof obligations and then a SMT solver is used to prove the truth of the obligations. Tools likes Yices[30], CVC4[31], Z3[46], or Alt-Ergo[32] can be used to prove the truth of the assertions, and in this work we use Microsoft Z3 theorem prover. On the other hand, our second approach is in the realm of model checking. In model checking the system is first represented as a finite state systems or automata, and then model checking algorithms may be used to find the reachability of states where properties of interest are satisfied. Details of these two approaches will be described in later chapters.

1.3 Goals

Except for the correctness of the output result, there are other aspects the designer needs to consider in an embedded system design. Since SLDL supports the modeling of implementation details, such as module partitioning and choice of processing elements or communication protocol selection, the verification of a system model should also cover all these aspects.

In this dissertation, we aim at the formal analysis and verification of the following three properties of interest:

1) liveness (deadlock)

2) timing constraint verification, and

3) May-Happen-in-Parallel analysis.

It is difficult to verify these three properties with simulation-based validation due to the limitation of simulation-based approach we described in the previous section. Therefore, the goal of this dissertation is to design analysis and verification approaches which statically examine system level models and guarantee the satisfiability or unsatisfiability of these three properties by using standard formal model checking methods such as SMT theorem prover and UPPAAL model checker. We will utilize existing infrastructure as foundation, in particular, focus on models specified in SpecC SLDL [12].

1.3.1 Formal Deadlock Detection

In this dissertation, we first aim at formal detection of the potential deadlock in the system model. For a system-on-chip design which may be composed of multiple processing elements running in parallel, improper execution order and communication assignment may lead to problematic consequences, and one of the consequences could be deadlock. For example, for a system model composed of two modules running in sequential fashion, using double handshake channel to communicate this two processing elements may lead to deadlock situation. It may be possible to identify the issue manually when the design is simple, but it become difficult when the design is composed of multiple processing elements working in parallel and communicate through various types of channels. Therefore, in this work we want to propose an approach which can formally detect the potential deadlock in the model, and also identify which part of the design leads to deadlock condition. We plan to convert a system level design in SpecC SLDL into a Satisfiability Modulo Theories (SMT) problem and then make use of an existing SMT solver to prove or disprove the existence of potential deadlock.

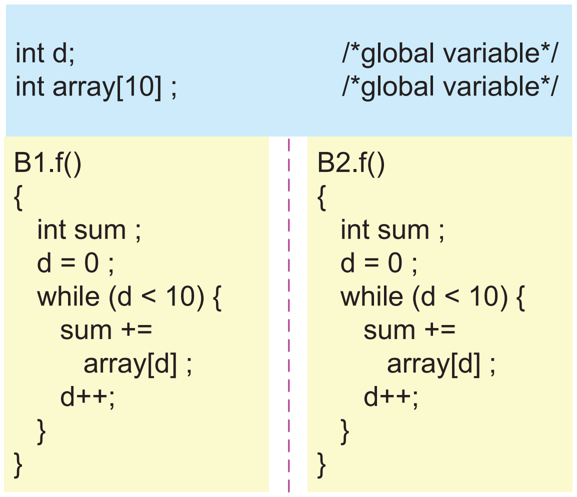
1.3.2 Timing Constraint Verification

The second property we want to formally verify in this work is timing constraint verification. In a system level design, we not only care the correctness of the output results, but also when the output can be obtained. Except for the functionality, another critical aspect in SoC design is the correctness of communication between system blocks, especially for real-time systems and communication protocols. Take dashboard displaying in a vehicle as the example. Except for verifying if the revolutions per minute (RPM) can be read and displayed on the dashboard properly, we also want to know if the reading update of the RPM can be executed within the timing constraints fitting the safety regulation. Simulation-based validation is commonly used to check if the system level model containing timing information can be finished within the giving timing constraint. However, the execution time can be input vector dependent, i.e., the timing constraint is not guarantee unless we run simulation for all possible conditions in the design. Here we plan to propose a approach which takes multiple aspects of the design such as delay caused caused by computation in processing elements and communication protocols into consideration, and analyze the satisfiability of given timing constraint.

1.3.3 May-Happen-in-Parallel Analysis

The third property we want to analyze in this work is May-Happen-in-Parallel (MHP) statements in the design. Our definition to this problem is that ”*for two given statements in the design, is it possible that these two statements are executed in parallel?*”. This analysis is important for the verification of system-level design because MHP statements are critical in system models and, among other problem, can lead to race condition. Race condition is a set of problems that could happen in multi-thread programming, and it is caused by parallel accesses to shared variables.
Fig. 1.6 shows an example where May-Happen-in-Parallel statements lead to race condition. In this example, two parallel instances B1 and B2 both call function `f()` in parallel. Function `f()` use a global integer `d` as the index to access an global array with 10 entries. The desirable behavior of this model is that each instance calculates the sum of all 10 elements in the array, but what may really happen is that each instance misses some elements because the index is increased by the other instance. Unfortunately, this issue may not be detected by simulation-based verification. To address this issue, we plan to proposed an approach which converts the system-level design into an UPPAAL model [59] [60] and makes use of the UPPAAL model checker to identify May-Happen-in-Parallel statements in the design.

In the following subsections, we introduce these two formal representations – Satisfiability Modulo Theories (SMT) and UPPAAL automaton model, and the corre-



```
int d;                          /*global variable*/
int array[10] ;                 /*global variable*/

B1.f()                    B2.f()
{                         {
  int sum ;                 int sum ;
  d = 0 ;                   d = 0 ;
  while (d < 10) {          while (d < 10) {
    sum +=                    sum +=
      array[d] ;                array[d] ;
    d++;                      d++;
  }                         }
}                         }
```

Figure 1.6: Importance of May-Happen-in-Parallel statements

sponding tools we plan to in this work to analyze the system-level design. The details of our approaches based on these formal representations and tools are introduced in Chapter 2, 3 and 4.

1.3.4 Satisfiability Modulo Theories

The first formal representation we will use in this work is the model of a Satisfiability Modulo Theories (SMT) problem [45] [47]. Satisfiability (SAT) is the problem of determining if an assignment of the Boolean variables exists, that makes the outcome of a given Boolean formula true. The Boolean formula is purely described with Boolean variables and logical operations such as AND, OR, and NOT. For a given Boolean formula, a SAT solver can find out if there is an assignment for all Boolean variables in the formula making the outcome true. An example is given here to demonstrate a SAT problem. For a Boolean formula `(a|b|c)` $\bigwedge$ `( a| b|c)` $\bigwedge$ `(a| b| c)`, a SAT solver answers satisfiable and also returns a case in which `a=true`, `b=true` and `c=true`. Similar to SAT problem, *Satisfiability Modulo Theories* (SMT) is also the problem of determining the whether there is an assignment of the variables to make the outcome of a formula true. Unlike the formula in SAT, which is purely built with Boolean variables and composed of logical operations in SAT, the satisfiability modulo theories supports richer language such as linear arithmetic or inequality therefore some problem in SMT can be described more naturally [51]. In this dissertation we use a SMT solver to formally analyze the existence of potential deadlock (in Chapter 2 and Chapter 3) and the satisfiability of specified timing constraints (in Chapter 3).

1.3.5 UPPAAL System Model

The second formal model we are going to use in this work is UPPAAL automaton model [59] [60]. UPPAAL system model is composed of a network of concurrent processes and these processes are created by instantiating predefined automaton templates. The whole system can be seen as a set of automata running concurrently and the enabled state transitions in those automata can take place in a non-deterministic order. Processes in a UPPAAL model can communicate and synchronize with each other through their parameters.

An example is provided in Fig. 1.7. In this example, the system is composed of two concurrent processes TA1 and TA2 in which there are four statements respectively (X1 $\sim$ X4 and Y1 $\sim$ Y4), and these two processes communicate with an integer and a channel. To create this system, templates for processes TA1 and TA2 must be defined first and then instantiated in the system definition with proper binding of parameters. After an UPPAAL model is created, designers can use UPPAAL model checker to verify the satisfiability of certain property. For instance, we can convert a question like "*Is it possible that automaton TA2 reaches state Y4 ?*" into a query in UPPAAL requirement specification language and ask the model checker if it is satisfiable. If the query is satisfiable, UPPAAL model checker can also generate a trace recording the transition from the initial state to the state where the property is satisfied. With the trace, designers can replay the transitions to identify in what condition a property can be satisfied. This feature is very useful for debugging since it is capable of showing how an undesirable situation can happen in a system in a graphical interface.

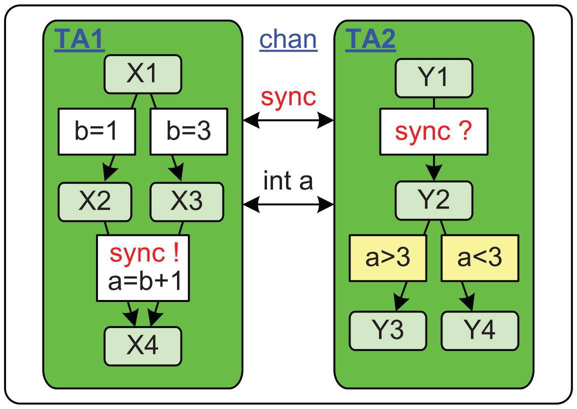


Figure 1.7: An UPPAAL model example

Note that due to the nature of problems behind these two approach (satisfiability and model checking), these two approaches are NP-Complete [33] and PSPACE-Complete(or EXPTIME-complete) [34].

1.4 Overview

In the second chapter, we propose an approach to abstracting SpecC-based system models for formal analysis using satisfiability modulo theories (SMT). Based on the language execution semantics, our approach [35] abstracts the timing relations between the time intervals of the behaviors in the design. We then use SMT solver to check if there are any conflicts among those timing relations. If conflict is detected, our tool will read the unsatisfiable model generated by the SMT solver and report the cause of the conflict to the user. We demonstrate our approach on a JPEG encoder design model.

In Chapter 3, we present an approach evolved from the method proposed in Chapter 2 to formally verify various aspects of communication models, including timing constraints and liveness. Our approach [48] automatically extracts timing relations and constraints from the design and builds a Satisfiability Modulo Theories (SMT) model whose assertions are then formally verified along with properties of interest input by the designer. Our method also addresses the complexity growth with a hierarchical approach. We demonstrate our approach on models communicating over industry standard bus protocol AMBA AHB and CAN bus. Our results show that the generated assertions can be solved within reasonable time.

In Chapter 4, we propose a method to model parallel discrete event simulation (PDES) with hierarchical concurrent automata and formally identify those states that *May-Happen-in-Parallel* (MHP). Our MHP analysis [57] [58] utilizes formal verification by use of the UPPAAL model checker. The proposed approach converts the system model in SpecC SLDL into an UPPAAL model and generates a set of queries that automatically and completely finds all possible MHP pairs. The experimental results show our approach can report more precise MHP analysis results compared to other works at the cost of extended analysis run time.

Finally, Chapter 5 summarizes the contribution of the work presented in this dissertation and concludes with a brief discussion on the future work.

Chapter 2

Formal DeadLock Analysis of SpecC Models

2.1 Introduction

An embedded system design can be implemented in many ways, and a typical design usually consists of hardware and software components running on one or multiple processing elements. In such a design, the partitioned components on different processing elements are executed in parallel. To make sure the data dependency and the execution order is correct, communication between components synchronizes the execution of components on different processing elements. In system-level description languages (SLDLs), like SpecC [12] [10] and SystemC [15] $citesystemc_{b}ook$, the communication between components is implemented as channels, and multiple types of channel are provided in the SLDLs to satisfy different kinds of communication and synchronization requirements.

Channels provide a convenient way to communicate among multiple processing elements. However, misusing the type of channel or setting incorrect buffer size in channel can lead to deadlock situations, and it is difficult to determine the cause of deadlocks when the design is complex. In this chapter [35] we propose a method to perform static analysis and detect deadlocks in the design automatically. Based on the SpecC execution semantics, our approach can extract the timing relations between behaviors in the design, and then analyze these with Satisfiability Modulo Theories (SMT) to detect any conflicts. To accelerate the debugging process, our approach also reports the causes of the deadlock to the user if deadlock situation is found.

This chapter is organized as follows: in Section 2 we review some of the related works in formal validation of SLDLs. In Section 3, we briefly introduce SpecC SLDL and Satisfiability Modulo Theories. In Section 4, we describe our proposed approach in detail, including the assumptions and limitations at this point. Also, we illustrate the conversion from SpecC model to SMT assertions. In the last two sections, we demonstrate our approach with a JPEG encoder model and sum up with a conclusion and future work.

2.2 Related Work

A lot of research has been conducted in the area of verification and validation of system-level design. We can see that many researches convert the semantics or behavioral model of the SLDL into another well-defined representation and make use of existing tools to validate the extracted properties. In [38] and [39], a method to generate a state machine from SystemC and using of existing tools for test case generation is proposed; in [40] and [41], a SystemC design is mapped into semantics of UPPAAL time automata and the resulting model can be checked by using UPPAAL model checker; [42] proposed an approach to translate SystemC models into a Petri-net based representation for embedded systems(PRES+) which can then be used for model checking. In [43], a SystemC design is represented in the form of predictive synchronization dependency graph (PSDG) and extended Petri Net, and an approach combining simulation and static analysis to detect deadlocks is proposed. [44] focuses on translating a SystemC design into a state transition system, i.e., the Kripke structure, so that existing symbolic model checking techniques can be applied on SystemC. In this section, our approach [35] uses SpecC[1] language to create the system level model for deadlock detection. The basic idea of our proposed approach is to abstract the timing relations from the behavior and channel communication in the SpecC model and then convert the relations into a SMT model composed of assertions in the form of inequality expressed in SMT-LIB2 language. If there is any deadlock situation in the system, the SMT model extracted out of the system will not be satisfiable as the circular waiting creates conflicts between timing relations.

2.3 Preliminaries

2.3.1 SpecC SLDL

SpecC [36] is a SLDL and is defined as extension of the ANSI-C programming language. It is a formal notation intended for the specification and design of digital embedded systems, including hardware and software portions. SpecC supports concepts essential for embedded systems design, including behavioral and structural hierarchy, concurrency, communication, synchronization, state transitions, exception handling, and timing. The execution semantics of the SpecC language are defined by use of a time interval formalism [37].

[1]Due to its similarity, our results are equally applicable to SystemC

There are several key concepts in the design of a system model in SpecC Language. The first concept is explicit structure. The design is partitioned modules and communicate with proper connection through ports. The second feature is explicit hierarchy. In a system model, a component is built in hierarchical way and composed of sub-components. In addition, in a system model the designer can explicitly specify the execution order of instantiated sub-components in the design, i.e., the instantiated components can be executed in sequential, parallel, or even pipelined fashion. The last important feature is clear separation of computation and communication. By clearly separate the communication from the computation, we can choose different types of channels or protocols to implement the inter-module communication and evaluate the performance. With these four features, we can not only evaluate the functionality of the design, but also implementation details like module partition or channel selection before further implementation.

2.3.2 Satisfiability Modulo Theories

Satisfiability (SAT) [50] is the problem of determining if an assignment of the Boolean variables exists, that makes the outcome of a given Boolean formula true. Satisfiability Modulo Theories (SMT) [45] [51] checks whether a given logic formula is satisfiable over one or more theories. Unlike the formulas in Boolean SAT which are built from Boolean variables and composed using logical operations, a SMT solver can capture the meaning of a formula described in richer language. For example, with a supporting theory of arithmetic, the meaning of a formula composed of integer variables and linear arithmetic can be captured by the SMT solver. Formula with arithmetic and inequality like $(x + 2y \leq 7) \wedge (2x - y \leq 10)$ can be solved by a SMT solver supporting arithmetic theory, and a satisfiable answer with an assignment to x and y satisfying the formula is given (for example, x=0, y=0). If the formula is unsatisfiable, the SMT solver we choose in this work also reports the assertions causing the unsatisfiability. This feature enables the ability to report the problematic design in the system in our approach.

After the formulas are generated, a SMT solver is then used to solve the set and check if it is satisfiable. Note that the solver may or may not give a firm answer saying the model is satisfiable or not. For a model which is too complicated for the solver to handle or quantified expressions without clear bound, the solver may not be able to determine whether the assertions is satisfiable or not. In this case, instead of giving a SAT or UNSAT answer, the solver returns an UNKNOWN to indicate it is unable to solve the formula. As we briefly described before, for a satisfiable assertion set, dependent on the chosen tool, the SMT solver may return an assignment to all function symbols in the model which satisfies the assertions. If assertions are not satisfiable, the solver may return a set of indices indicating which assertions form the conflict and lead to unsatisfiability.

In our implementation, we use Z3 theorem prover developed at Microsoft Research as our SMT solver. For more detailed information about SMT-LIB2 language and Z3 theorem prover, please refer to [47] and [46].

2.4 From SpecC to SMT assertions

In this section, we first introduce the supported SpecC execution types in our approach and their execution semantics.

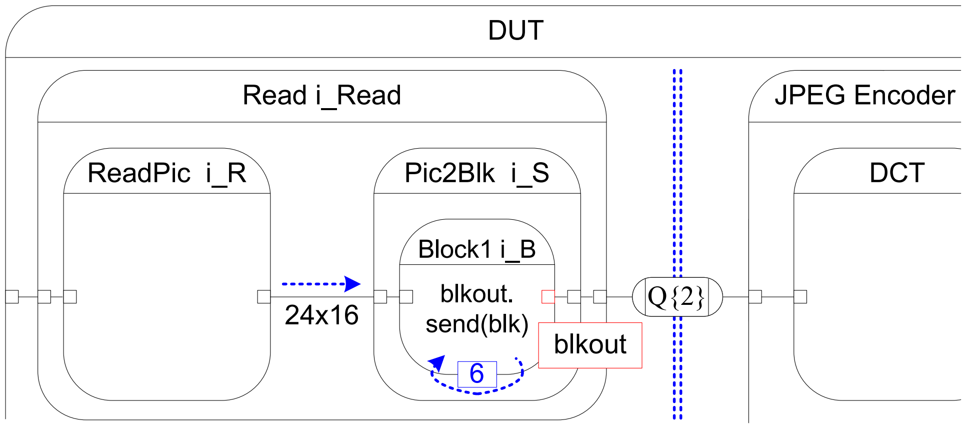


Figure 2.1: Behavior *Read* in the JPEG encoder SpecC model

2.4.1 Execution

The basic structure of a SpecC behavior includes port declaration, a *main* method, local variable and function declaration (optional), and sub-behavior instantiation (optional). The supported SpecC behavior models in our tool can be categorized into the following two types:

Leaf Behavior: A behavior is called leaf-behavior if it is purely composed of local variable(s), local function(s) and a *main* method, and there is no sub-behavior instantiation in the behavior. In the example shown in Figure 2.1, behavior *ReadPic* and *Block*1 are leaf behaviors.

Non-Leaf Behavior: A behavior is called non-leaf behavior if it is purely composed of sub-behavior instance(s) and a main method. For non-leaf behaviors, all statements in the *main* method are limited to statements specifying the execution type of the behavior and must be function calls to sub-behavior instances. In the example shown in Figure 2.1, behavior *DUT*, *Read*, *JPEG_encoder* and *Pic*2*Blk* are non-leaf behaviors.

Note that for simplicity our tool does not support models which do not fit into these two categories, and the execution types we are going to describe in this section is for non-leaf behaviors only since sub-behavior instantiation can only occur in non-leaf behavior.

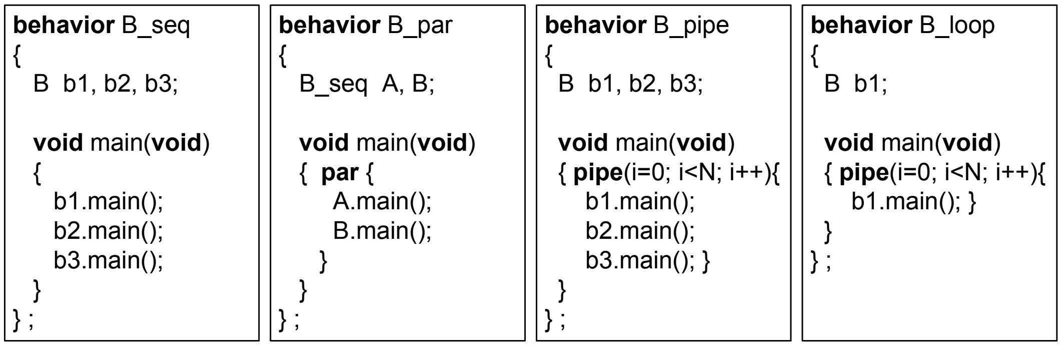
In SpecC, the sub-behavior or sub-channel instantiation is regarded as a statement of function call to method of the sub-behavior or sub-channel. To specify the execution time of a statement, for each statement s in a SpecC program, a time interval $\langle T_{start}(s),\ T_{end}(s)\rangle$ is defined. $T_{start}(s)$ and $T_{end}(s)$ represent the start and end times of the statement execution respectively, and the following condition must hold: $T_{start}(s) < T_{end}(s)$

The execution time of an instantiated behavior s $T_{exe}(s)$ is defined as $T_{exe}(s) = T_{end}(s)$ - $T_{start}(s)$. For a statement S consisting of a set of sub-behavior instances $\langle s_{sub\_1}, s_{sub\_2}, s_{sub\_3}, ...s_{sub\_n}\rangle$, the following condition holds:

$$\begin{aligned} \forall i \in \{1,2,3,...n\}, T_{start}(S) &\leq T_{start}(s_{sub\_i}) \\ T_{end}(S) &\geq T_{end}(s_{sub\_i}) \end{aligned}$$

The type of execution defines the relation between the T_{start} and T_{end} of the behavior instance under the current behavior, and it is specified in the *main* method of the behavior.

In the following, four types of supported execution are described, which are *Sequential*, *Parallel*, *Pipelined*, and *Loop* execution. Figure 2.2 shows an example of specifying *Sequential*, *Parallel* and *Pipelined* execution in a SpecC behavior. *Loop* execution is not a explicitly defined behavioral execution in SpecC, but we can regard it as a special case of *Pipelined* execution with only one instance inside.



```
behavior B_seq
{
  B  b1, b2, b3;

  void main(void)
  {
    b1.main();
    b2.main();
    b3.main();
  }
} ;
```

```
behavior B_par
{
  B_seq  A, B;

  void main(void)
  {  par {
       A.main();
       B.main();
     }
  }
} ;
```

```
behavior B_pipe
{
 B  b1, b2, b3;

 void main(void)
 { pipe(i=0; i<N; i++){
     b1.main();
     b2.main();
     b3.main(); }
 }
} ;
```

```
behavior B_loop
{
 B  b1;

 void main(void)
 { pipe(i=0; i<N; i++){
     b1.main(); }
 }
} ;
```

Figure 2.2: Four Supported Execution Types

Sequential Execution

Sequential execution of statements is defined by ordered time intervals that do not overlap. Formally, for a statement S consisting of a sequence of sub-statements $\langle s_1, s_2, ...s_n \rangle$, the time interval of statement S includes all time intervals of the sub-statements, and the following conditions hold:

$$\forall i \in \{1, 2, ..., n\}, T_{start}(S) \leq T_{start}(s_i)$$

$$T_{start}(s_i) < T_{end}(s_i)$$

$$T_{end}(s_i) \leq T_{end}(S)$$

$$\forall i \in \{1, 2, ..., n-1\}, T_{end}(s_i) \leq T_{start}(s_{i+1})$$

Note that sequential statements are not necessarily executed continuously. Gaps may exist between T_{end} and T_{start} of two consecutive statements, as well as between the T_{start} (T_{end}) of the sub-statement and the T_{start} (T_{end}) of the statement in which the sub-statement is called. Figure 2.3 shows an example of the time interval for the sequential execution in Figure 2.2.

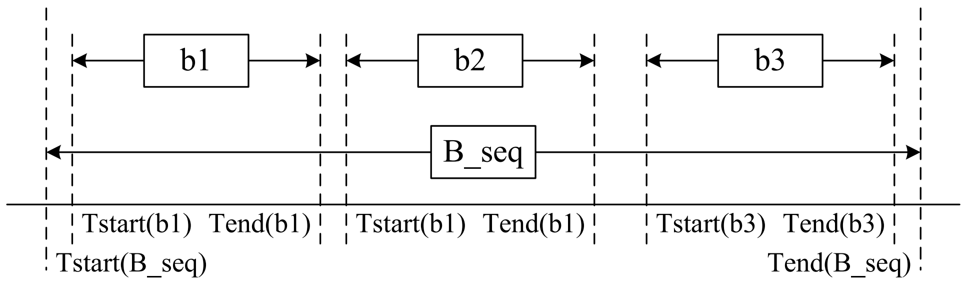


Figure 2.3: Time interval for sequential execution

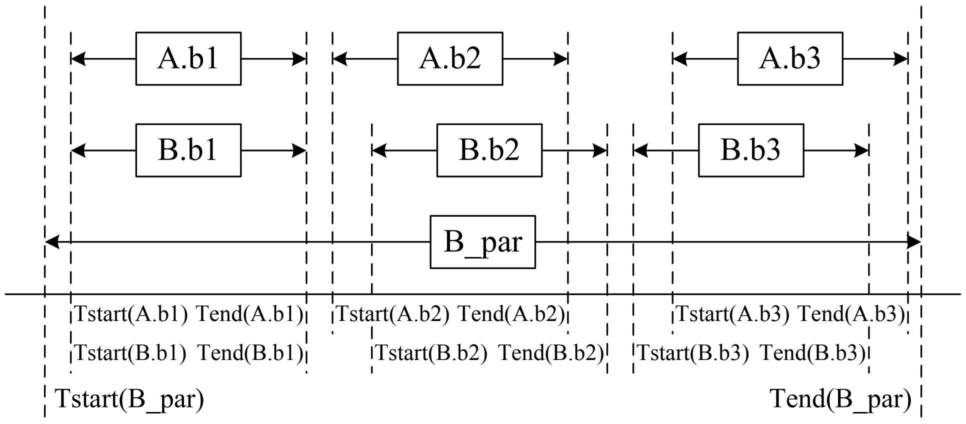


Figure 2.4: Time interval for parallel execution

Parallel Execution

Parallel execution of statements can be specified by *par* or *pipe* statements. In particular, the time intervals of the sub-statements invoked by a *par* statement are the same. Formally, for a statement S consisting of concurrent sub-statements $\langle s_1, s_2, ...s_n\rangle$:

$$\begin{aligned} \forall i \in \{1, 2, ..., n\}, T_{start}(S) &= T_{start}(s_i) \\ T_{end}(S) &= T_{end}(s_i) \\ T_{start}(s_i) &< T_{end}(s_i) \end{aligned}$$

Figure 2.4 shows an example of the time interval for the parallel execution in Figure 2.2.

Pipelined & Loop Execution

Pipelined execution of statements is a special form of concurrent execution. The syntax of *pipe* statement in SpecC is illustrated in Figure 2.2, where N in the example specifies the number of iterations. Formally, for a statement S consisting of sub-statements $\langle s_1, s_2, ...s_n\rangle$ executed for m iterations in pipelined manner, let $s_{i \cdot j}$ represents the j-th iteration of the execution of statement s_i. Then the following conditions hold:

$$\forall i, x \in \{1, 2, ..., n\}\ ,\ j, y \in \{1, 2, ..., m\}$$

$$T_{start}(s_{i \cdot j}) < T_{end}(s_{i \cdot j}),$$

$$T_{start}(s_{i \cdot j}) = T_{start}(s_{x \cdot y}),\ \ \textit{if}\ \ i + j = x + y$$

$$T_{end}(s_{i \cdot j}) = T_{end}(s_{x \cdot y}),\ \ \textit{if}\ \ i + j = x + y$$

$$T_{end}(s_{i \cdot j}) \leq T_{start}(s_{x \cdot y}),\ \ \textit{if}\ \ i + j < x + y$$

Loop execution is not defined explicitly in the behavioral execution semantics of SpecC, but it can be regarded as a special case of *Pipelined* execution with only one sub-statement.

Note that in the definition of *pipelined* statements the iteration number could be infinity if the number is not specified, i.e., no range specification after the statement *pipe*. However, to simplify the static analysis in this proposed method, at this point, the number of iterations has to be a finite integer and explicitly specified in the model.

Please be aware that for now our proposed method does not support all types of execution and communication defined in SpecC. Full support of SpecC is part of our future work.

2.4.2 Communication

In SpecC, the communication between two behaviors can be implemented by port variable, channel communication, or by accessing global variables. Since right now the goal of our approach is to detect deadlocks in the design, the communication implemented with port variables and global variables are not taken into consideration because they will not lead to deadlock situation in the design.

Multiple types of channels are defined in SpecC. These include semaphore, mutex, barrier, token, queue, handshake, and double handshake. In this chapter, we use queue channel with different buffer sizes to model the supported channel communication in our approach. For example, to model the blocking characteristics of handshake channels, we use a queue channel with one element buffer and zero element buffer to implement the handshake and double handshake channel.

To clearly identify the communication between behaviors, we also impose some limitations on the communication between behaviors. First, to make the data dependency between behaviors clear, we limit the communication between behaviors to point-to-point, i.e. every instantiated channel in the design is dedicated to the communication of a pair of sender and receiver. Second, to abstract the channel activity without looking into too much detail of the behavior model, the function call of the sending (receiving) function to (from) a certain channel can only be executed once in the *main* method of a behavior, i.e. function call to channel communication in any type of iteration (for or while loop) in the *main* method of a behavior model is not supported. For the case that the output of a behavior has to be separated into multiple parts and sent to another behavior, the sending (receiving) function calls have to be wrapped in a behavior and executed in loop execution by using *pipe* statements.

Figure 2.1 shows an example of the situation described above. In this example, a small picture of size 24-by-16 pixels is read and encoded into a JPEG file. Since the input image block size for a JPEG encoding process is eight-by-eight pixels, the picture has to be separated into 6 sub-blocks. The raw picture is read into the topmost behavior DUT by sub-behavior $Read$, then behavior $Pic2Blk$ divides the picture into six 8-by-8-pixel blocks and sends the blocks to JPEG encoder model. Inside behavior $Pic2Blk$, behavior $Block1$ is instantiated in a loop execution. Behavor $Block1$ fetches the block from the raw picture according to the current iteration number, and calls the sending function to send the data to JPEG encoder through channel Q. In this example, channel Q is a queue channel with two buffers and each buffer is an integer array of size 64.

Similar to the time interval $\langle T_{start}, T_{end}\rangle$ defined for the execution of a statement, a time stamp set $\langle T_{sent}(Q), T_{rcvd}(Q)\rangle$ is also defined for each channel communication activity between behaviors, where T_{sent} represents the time stamp when the the execution of sending a data to the channel finishes, and T_{rcvd} represents the time stamp when the execution of receiving data from the channel finishes. Based on the definition of T_{sent} and T_{rcvd}, for a queue channel Q communication through which m data items are transferred, the relation between time stamps $T_{sent}(Q_i)$ and $T_{rcvd}(Q_i)$, where Q_i represents the i-th data transfer through channel Q, should hold:

$$\begin{aligned}
\forall i \in \{0,1,2,...,m-1\}, T_{sent}(Q_i) &\leq T_{rcvd}(Q_i)\\
\forall i \in \{0,1,2,...,m-2\}, T_{sent}(Q_i) &< T_{sent}(Q_{i+1})\\
T_{rcvd}(Q_i) &< T_{rcvd}(Q_{i+1})\\
\forall i \in \{0,1,2,...,m-n-1\}, T_{rcvd}(Q_i) &\leq T_{sent}(Q_{i+n})
\end{aligned}$$

where n is the buffer depth of channel Q.

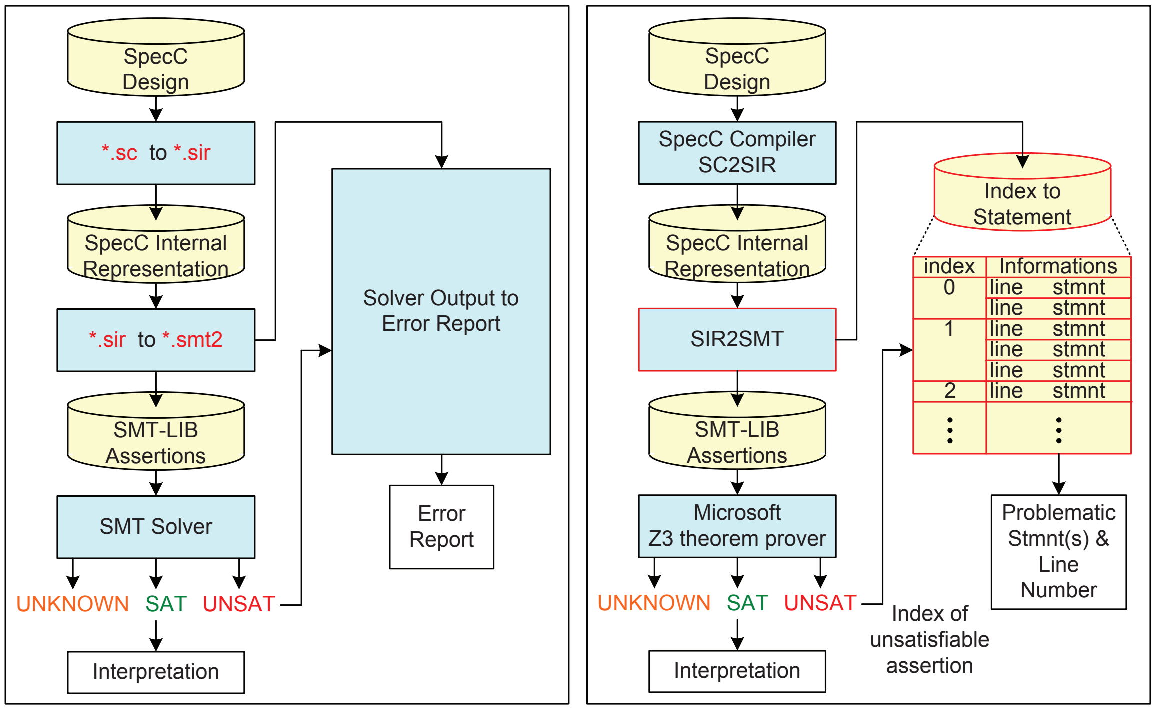


Figure 2.5: The flow of converting a SpecC model into SMT assertions and deadlock analysis with the Z3 SMT solver

2.4.3 From Time Stamp to SMT Assertions

Figure 2.5 shows the flow of our proposed method. First, the SpecC model is converted into a design representation called SpecC internal representation(SIR). The next step is to traverse the internal representation structure and generate the assertions corresponding to the statements in the design. At the same time, an index-to-statement record is created which links the generated assertions to the statements in the design. After the assertions and records are generated, we use the Z3 theorem prover to check if there is any conflict in the set that makes the equations unsatisfiable. If there are any, Z3 will report the indices of assertions leading to the conflict, and our tool can use the indices to access the record and report the problem information to the user. In the following part of this section, we use the model shown in Figure 2.1 as an example, and illustrate the corresponding assertions for the model.

Execution to SMT assertions:

In our proposed method, we use uninterpreted functions in SMT-LIB2 language to represent every time stamp in the model, and convert the timing relations between those stamps into assertions. For an uninterpreted function, the user can define the number of arguments, the data type of argument, the data type of return value, and its interpretation. In our method, the return value of an uninterpreted function is seen as the value of a time stamp, and the argument(s) of the function is (are) used to specify the number of times a behavior instance is executed in a pipelined structure or a loop. For a behavior instance, which is not in a pipelined or loop execution, the time stamps of this instance are represented as uninterpreted functions with no argument since the behavior will only be executed once.

For example, for instance *i_S* in behavior *Read* in Figure 2.1, the following assertions will be generated:

$$(declare-fun\ T_{start}DUT.i\_Read.i\_S\ ()\ Int)$$

$$(declare-fun\ T_{end}DUT.i\_Read.i\_S\ ()\ Int)$$

$$(assert\ (<=\ T_{start}DUT.i\_Read\ T_{start}DUT.i\_Read.i\_S))$$

$$(assert\ (<=\ T_{end}DUT.i\_Read.i\_S\ T_{end}DUT.i\_Read))$$

$$(assert\ (<\ T_{start}DUT.i\_Read.i\_S\ T_{end}DUT.i\_Read.i\_S))$$

$$(assert\ (<=\ T_{end}DUT.i\_Read.i\_R\ T_{start}DUT.i\_Read.i\_S))$$

For a behavior instance, which is executed in a pipelined or loop for multiple times, the time stamps of this instance are represented as uninterpreted functions with one or multiple arguments. The input value of the argument is the number of execution times of this instance.

For example, for instance $S1$ in behavior $Sender$ in Figure 2.1, the following assertions will be generated:

$$(declare - fun\ \ T_{start}DUT.i\_Read.i\_S.i\_B\ \ (Int)\ \ Int)$$
$$(declare - fun\ \ T_{end}DUT.i\_Reae.i\_S.i\_B\ \ (Int)\ \ Int)$$
$$(assert\ \ (forall\ \ ((I0\ \ Int))\ \ (=>\ \ (and\ \ (>=\ \ I0\ \ 0)\ \ (<=\ \ I0\ \ 5))$$
$$(<=\ \ T_{start}DUT.i\_Read.i\_S$$
$$(T_{start}DUT.i\_Read.i\_S.i\_B\ \ I0)))))$$
$$(assert\ \ (forall\ \ ((I0\ \ Int))\ \ (=>\ \ (and\ \ (>=\ \ I0\ \ 0)\ \ (<=\ \ I0\ \ 4))$$
$$(<=\ \ (T_{end}DUT.i\_Read.i\_S.i\_B\ \ I0)$$
$$(T_{start}DUT.i\_Read.i\_S.i\_B\ \ (+\,I0\ \ 1))))))$$

Communication to SMT assertions:

In our approach, the time stamp of every channel activity is represented as an uninterpreted function with one argument, and the input value of the argument is the number of execution times of channel activity. For example, for channel Q in behavior DUT in Figure 2.1, the following assertions will be generated:

$$\forall i \in \{0, 1, ..., 5\}, \quad T_{sent}DUT.Q(i) \leq T_{rcvd}DUT.Q(i)$$
$$\forall i \in \{0, 1, ..., 3\}, \quad T_{rcvd}DUT.Q(i) \leq T_{sent}DUT.Q(i+2)$$

Our tool will also generate the equality for the time stamp of the channel activity and the time stamp of the function call to the interface of the corresponding channel. For example, the following assertion will be generated for the channel accessing function call *blkout* in Figure 2.1:

$$\forall i \in \{0, 1, ..., 5\},$$

$$T_{sent}DUT.q(i) = T_{sent}DUT.i\_Read.i\_S.i\_B.blkout(i)$$

$$T_{start}DUT.i\_Read.i\_S.i\_B(i) \leq T_{sent}DUT.i\_Read.i\_S.i\_B.blkout(i)$$

$$T_{sent}DUT.i\_Read.i\_S.i\_B.blkout(i) \leq T_{end}DUT.i\_Read.i\_S.i\_B(i)$$

Assertions for other modules in the model are generated based on the timing relations we described in Section 2.4.1 and Section 2.4.2 accordingly.

During the assertion creation, a table named *index-to-statement* will also be generated. For every assertion generated by our tool, an identical index is given to the assertion and the information about the corresponding statement that is stored in the entry addressed by that index. Take assertion $T_{rcvd}DUT.Q(i) \leq T_{sent}DUT.Q(i+2)$ listed above as an example. This assertion is generated because channel Q is instantiated in behavior DUT and its depth is set to two. Therefore, in the entry addressed by the index of this assertion, the information of the statement specifying the depth of the channel is stored.

2.5 Experiments

In this section, we demonstrate our proposed method with a JPEG encoder SpecC model. In this example, the JPEG encoder is asked to encode five sub-frames of size eight-by-eight pixels from a raw picture. *Figure* 2.6 shows two different implementations of the SpecC JPEG encoder model.

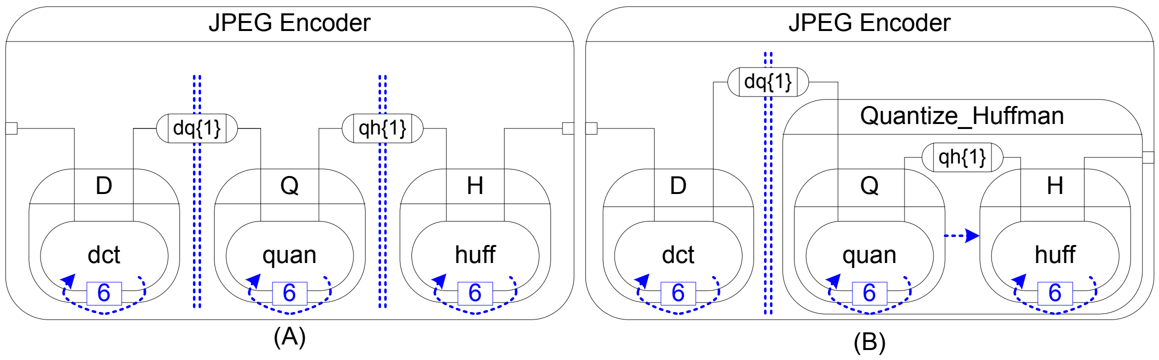


Figure 2.6: Two examples of JPEG encoder SpecC model

In the JPEG encoder, every subframe will be encoded in three steps, two-dimensional discrete cosine transform (DCT), quantization, and Huffman encoding. For every subframe, these three encoding steps have to be executed in order. In our SpecC model, three behaviors D, Q, and H are implemented to perform the discrete cosine transform, quantization, and Huffman coding of JPEG encoding, respectively.

Example(A) is the design without deadlock, while for a SpecC model like Example(B) will incur deadlock situation. As shown in $Figure$ 2.6(A), behavior D, Q, and H are executed in parallel fashion. To make sure these three steps are executed in correct order, two queue channels are used to transfer the intermediate encoding data between these three behaviors, instead of using port variable connections. In model (B), sub-behavior Q and H are wrapped into a behavior $Quantize\_Huff$ and executed in sequential manner. The problem in model (B) is that behavior Q will halt forever after the first two iterations of its sub-behavior $quan$. In this composition, behavior $quan$ will be executed six times before the execution of behavior Q finishes, but the execution will stop because the queue channel between behavior $quan$ and behavior $huff$ becomes full after the first two data sets are generated. Since behavior H can only be executed after the execution of behavior Q finishes, the sub-behavior $huff$ can not be executed to empty the queue channel qh.

We have used our tool to analyse both models. Table 2.1 shows the analysis results

Table 2.1: Static SMT analysis results for model (A) and (B)

Design	*#ofAssertions*	*Time*	*Satisfiability*	Error Report
Model-(A)	187	4.94s	SAT	N/A
Model-(B)	192	1.39s	UNSAT	Type: QUEUE Line[16]: Channel[qh] Type: SEQ Line[23]: Instance[Q] Line[24]: Instance[H] Type: LOOP Line[58]: Behavior[Q] Line[60]: Instance[quan] ...

of the two models.

In Table 2.1, the value in *Line* represents the line number of the statement in the SpecC model, *Type* shows the type of information stored in the entry. For example, the *Type* : *SEQ* in this table shows that behavior instance Q and H are executed in sequential manner, and Q is executed before H. Though for now the error report might not be intuitive for the unfamiliar user to understand what led to the deadlock, the model designer who developed the model will easily recognize the deadlock situation.

2.6 Summary

In this chapter we have proposed an approach to statically analyze deadlocks in SpecC models using a SMT solver. After the introduction of four supported execution types and queue channel communication in our tool, we have described our approach in detail by showing how to extract timing relations between time stamps according to SpecC execution semantics, and have illustrated the conversion from timing relations to SMT-LIB2 assertions. Finally we demonstrated our implementation with a JPEG encoder model, and showed that our approach is capable of detecting the deadlock in the model and reporting useful diagnostic information to the user.

The approach proposed in this section can detect the potential deadlock caused by improper composition and channel selection, but it requires further improvements to support wider range of SpecC model so that we can use this model to verify more properties of interest. The required improvement includes expanding the support for larger models and extending the support for SpecC language execution semantics so that it can cover more design verification problems. At this stage our implementation only supports a confined set of SpecC model and leaves some important features of SpecC unsupported, such as FSM composition and the communication between processing elements. The communication between processing elements are limited to pre-defined channels and user-defined channels are not supported. In the next chapter we introduced the improved tool which is capable of extracting the timing relations from a do-timing construct into assertions and also generating SMT-LIB2 assertions to model the behavior of event delivery. We explored the possibility of using SMT solver to formally prove or disprove the satisfiability of timing constraint in addition to deadlock detection in the following chapter.

Chapter 3

Communication Protocol Analysis of Transaction-Level Models

3.1 Introduction

In system-level design, a transaction level model (TLM) describes the system components, their abstract computation behavior, and in particular the system communication over buses at an abstract functional level. Typically, the functionality and timing of a TLM is validated through simulation. In this chapter [48], we formally verify the transaction level model and propose an improved method over the approach in the previous chapter to statically analyze the TLM and verify features of interest. In particular, our main focus here is on the timing constraints in the communication protocols. As illustrated in Figure 3.1, we perform multiple rounds of verification using SMT, following a designer-in-the-loop methodology.

Based on the given execution semantics of Discrete Event Simulation (DES), our proposed approach extracts the timing relations specified in the design model and

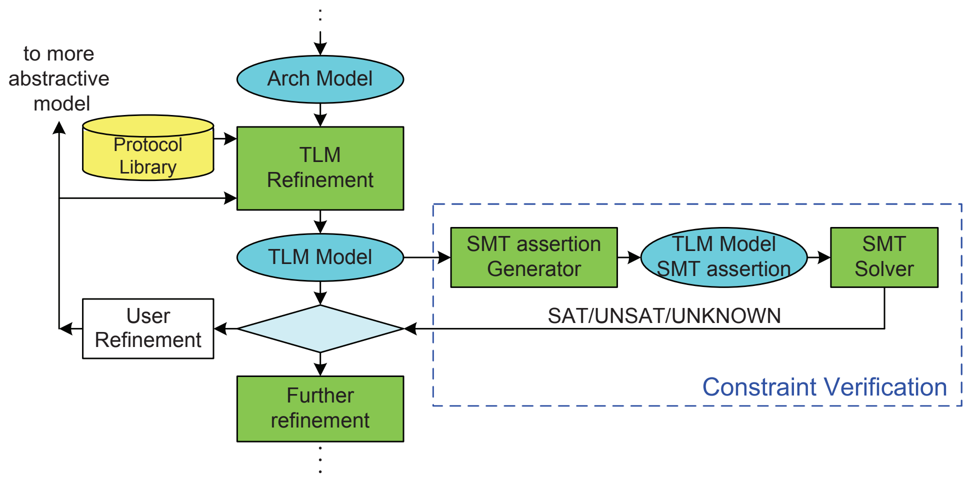


Figure 3.1: SpecC Methodology with static constraint analysis

converts them into assertions as input for the SMT solver[1]. The SMT solver checks the satisfiability of the assertions and reports the result to the system designer. If the assertions are satisfiable, the SMT solver can provide a detailed report of the symbol assignments which make the assertions true. On the other hand, if the model is found unsatisfiable, the SMT solver reports the conflicting assertions leading to the unsatisfiability. Based on the result, the system designer can determine whether or not the TLM satisfies the desired design requirements, as well as where the design fails the requirements, if so.

3.1.1 Designer Augmented Assertions

Our proposed methodology in this section is to automatically extract timing relations and constraints from a giving design and build a corresponding SMT model as verification framework. Then the designer can verify the properties of interest on the framework by augmenting the SMT model with assertions reflecting his points of interest. For example, to verify that the execution time of the application is always less than 100 time units, the designer can augment the SMT model with an asser-

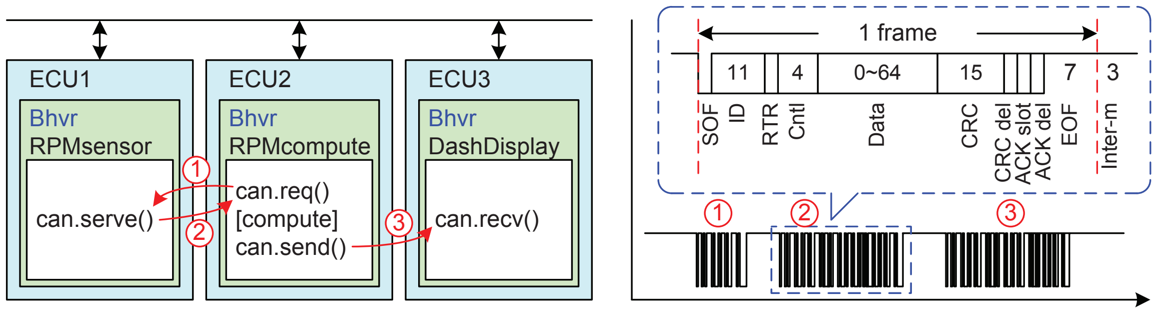
[1]We use Z3 theorem prover [49] developed by Microsoft Research.

tion asking *"Can the execution time be more than 100 time units?"*. If this is found unsatisfiable, the application *will* execute in 100 time units or less, taking *all* conditions into account. In other words, the execution time is proven to meet the timing constraint. On the other hand, if it is found satisfiable, the tool will also list the conditions satisfying the assertions so that the designer can examine the situation.

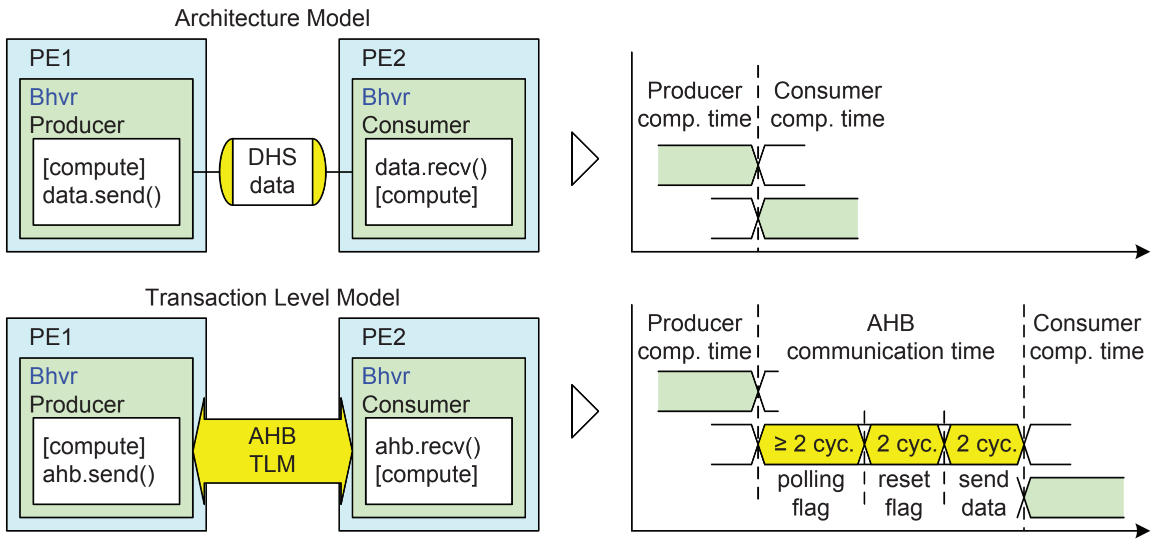
3.1.2 TLM with Communication Timing

Before an application is further implemented in a system, we design a system model to verify the functionality as well as certain design constraints at early stage of the implementation. In a top-down system design flow, we first design a specification model and then refine it into an architecture model in which the structure and the delay introduced by the computation is specified but the communication is assumed to take no time. In the real world scenario, the delay caused by communication cannot be neglected. To take the communication delay into consideration in design verification, the system architecture model is further refined into a TLM. The main objective of TLM refinement is to choose and parameterize a bus protocol to implement the communication between the processing elements in the system. The communication protocol is specified by the inserted transaction level bus model which specifies the detailed communication approach and timing, including synchronization and delays compliant to the chosen protocol. Compared with the previous model, TLM with communication timing better represents the real-world design in which communication does take time.

Figure 3.2 shows two TLM examples and their corresponding timing diagrams. In both designs, the corresponding timing information for AMBA Advanced High-performance Bus (AHB) [53] and controller area network (CAN) bus protocol [54] is specified in the



(A) Timing diagram for CAN bus TLM of a RPM Display example



(B) Timing diagrams for architecture model and TLM of a Producer-Consumer example

Figure 3.2: TLMs with detailed communication timing

TLM channel. The communication delay has great influence on the execution time and even liveness of the design. The delay is highly dependent both on how the communication channel is implemented as well as on what protocol the communication is performed.

Take the Producer-Consumer model illustrated in Figure 3.2 (B) as the example. In this example, producer and consumer communicate with a double handshake channel. The double handshake communication takes no time in architecture refined model, but in TLM the delay is introduced. In our model the double handshake channel is implemented as a three-step communication. The producer first polls the ready flag in the consumer and then clears the flag after it makes sure the consumer is ready to receive the data. After the completeness of the first two, the data is sent from the producer to consumer. With this implementation, the delay caused by these three steps should be considered together. Also, in our producer consumer model the chosen communication protocol is AMBA AHB. As the illustration shown in Figure 3.3, each communication takes at least two cycles to transfer in AMBA AHB protocol. Therefore, for a data transfer using double handshake over AMBA AHB protocol, the delay for the communication is at least 6 cycles.

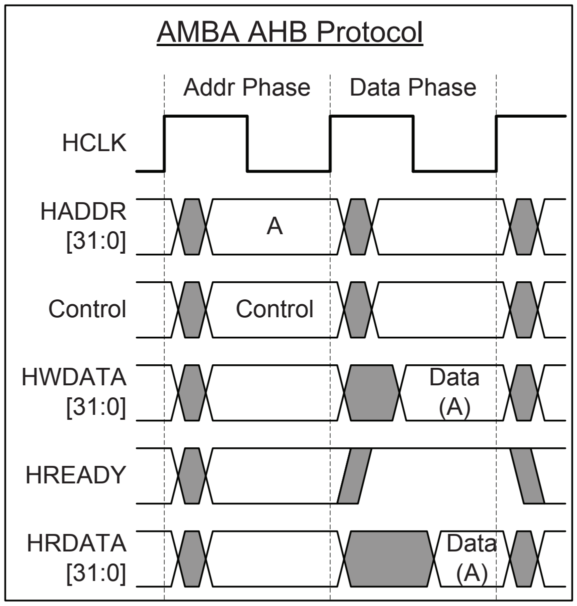


Figure 3.3: AMBA AHB Protocol (source[53])

By formally extracting the timing relations from the TLM and checking these with the SMT solver, our proposed method can *verify* the meeting of the timing constraints with the selected architecture and bus protocol.

3.1.3 Related Work

There is significant work in the realm of formal verification of system-level design, and one research method in this area is to convert the semantics of a behavioral model into another well-defined representation and make use of existing tools to validate the properties of interest. In [40] designs in SystemC are transformed into UPPAAL time automata and verified by UPPAAL model checker; in [39] and [38] a method to convert SystemC into state machines for verification is proposed; [42] proposed to translate SystemC models into a Petri-net based representation for embedded systems (PRES+) for model checking; [43] proposes a multi-layer modeling to represent SystemC design in a predictive synchronization dependency graph (PSDG) and extended Petri net is proposed for formal deadlock checking. [44] translates SystemC to Kripke structure and applies symbolic model checking for verification. In contrast, Our approach [48] acts as an interactive property checking tool which brings uncertainties about critical properties and corner cases to the attention of the designer. In our method, the designer verifies points of interest by adding corresponding assertions to the extracted SMT model. [52] uses similar time interval model for verification of synchronization. In comparison, our approach does not support multiple event synchronization which is not supported in this related work, but also support the verification of timing constraint. The satisfiability report obtained through our method highlights often special situations, such as missed acknowledge signals or unsatisfied condition, and assists the designer in verification of the model for all cases.

3.2 Time Interval Models

In system design, functionality is not the only concern. Timing constraints are critical as well, especially for real-time systems and communication protocols. Therefore, the notion of time is an important aspect of the model. In this section, we propose a time interval model to represent the timing aspect of the target SpecC model. By formally analyzing the time interval model, we prove or disprove the satisfiability of properties of interest in the model. The proposed time interval model can be seen as the combination of timing stamps and timing relations, and the model can be visualized as a graph with timing stamps as vertices in the graph and timing relations as edges between vertices. For each statement of interest s, a set of timing stamps $\langle T_{start}$, $T_{end}\rangle$ [37] are given to represent the start and end time of the execution of the statement in the model. To properly reflect the discrete event semantics with delta cycles, we make every time stamp a 3-tuple $\langle$*Time*(t), *Delta*(d), *Order*(o)$\rangle$, where *Time* and *Delta* represent the simulation time and delta cycle of the time stamps respectively. Note that we use the third member, called *order*, to distinguish statements that otherwise happen at the same time and delta cycle. The ordering is determined based on the timing relation between statements and assigned automatically by the solver. For such time stamps, we define a set of operations as listed in Table 3.1, describing the relations equality and greater-than, as well as time advance by wait-for-time.

Table 3.1: Operations of time stamp

Operation	*Definition*
$T_A = T_B$	$T_{A.t} = T_{B.t}$, $T_{A.d} = T_{B.d}$, $T_{A.o} = T_{B.o}$
$T_A > T_B$	$T_{A.t} > T_{B.t}$ or $T_{A.t} = T_{B.t}$, $T_{A.d} > T_{B.d}$ or $T_{A.t} = T_{B.t}$, $T_{A.d} = T_{B.d}$, $T_{A.o} > T_{B.o}$
T_A waitfor N	$T_{A.t} = T_{A.t} + \text{N}$, $T_{A.d} = 0$

Exact timing, such as delay or execution time of computation and communication, can be specified by using wait-for-time statements that carry a time argument of integral constant type. When a wait-for-time statement is executed, the current behavior is suspended from execution for the specified time. Figure 3.5(A) shows an example with `waitfor`, `wait`, and `notify` statements. Here, statement A is executed *val* time units before statement B due to the `waitfor val;` statement. For this example, the following conditions hold:

$$T_{start.t}(stmnt\_B) = T_{end.t}(stmnt\_A) + val$$

$$T_{start.d}(stmnt\_B) = 0$$

In the following sections, we will describe for what statements our approach generates timing stamps for them and how the timing relations between timing stamps are extracted from the system model.

3.2.1 Timing Constraints

Minimum or maximum bounds on the time between two statements in the model are called timing constraints. To meet real-time constraints imposed on the application by the environment, e.g. for communication, such constraints need to be specified with the design model so that it can be implemented accordingly.

In the SpecC language, timing constraints can be specified in the model with a special `do-timing` construct, with which the timing constraints can be checked during simulation or, in our case, be extracted to assertions for formal verification. The syntax of timing constraints contains two parts: the `do` block specifies a set of labeled statements, whereas the `timing` block contains the actual constraints. In the `do` block, the statements whose timing the designer wants to check are given a unique label and in the following `timing` block the labels are used to set the constraints. Constraints are specified with the `range` construct, which takes four arguments. The first two arguments specify the labels and the last two the lower and upper bounds of the timing constraint, respectively. A `do-timing` example is shown in Figure 3.5(B).

There are three labels in the `do` block, and two constraints are specified with `range` constructs in the `timing` block. Note that label *L2* is attached to a compound statement which contains two child behavior calls. The following condition must hold for the constraints in this example:

$$0 \le T_{start.t}(L2) - T_{start.t}(L1) \le 100$$

$$0 \le T_{start.t}(L3) - T_{start.t}(L2) \le 300$$

Since $T_{start}(L1) = T_{start}(i\_A)$ and $T_{start}(L2) = T_{end}(i\_A)$, the first constraint limits the execution time of *i_A* to less than or equal to 100 time units. The second constraint sets the upper bound for the execution time of the statement labeled *L2*, which is also the sum of the execution time of *i_B* and *i_C*. This way, specific timing constraints on the execution time of any child behavior can be specified.

3.3 Timing Relation Extraction

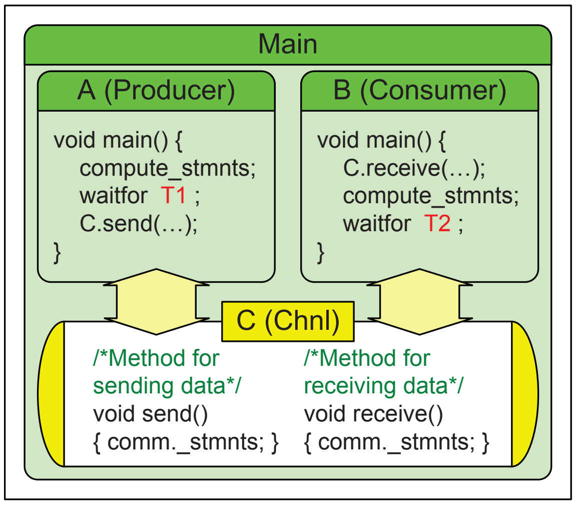


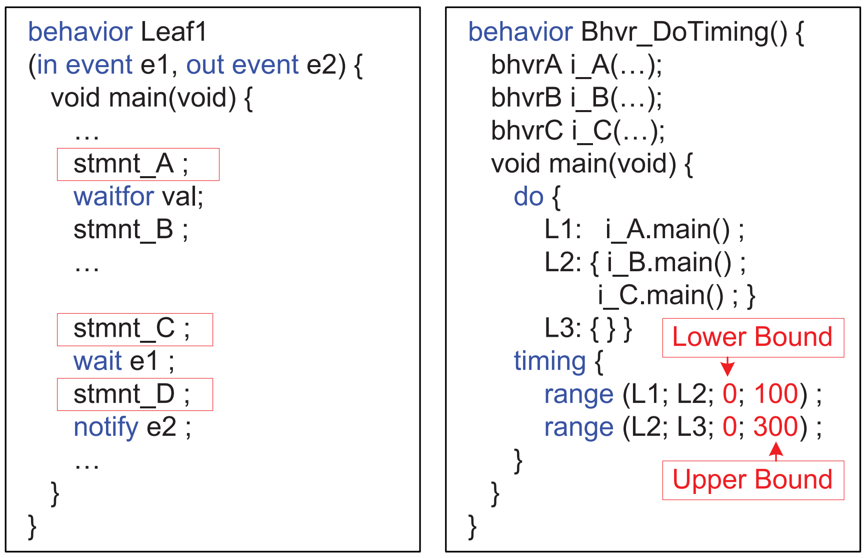
Figure 3.4: Producer-Consumer SpecC Model

A system model is composed of multiple computation blocks (modules, behaviors) with communication (channel) between them. The approach proposed in this chapter distinguishes three types of modules, and here we use a Producer-Consumer model showed in Figure 3.4 to illustrate these three types of modules:

1) **Leaf behavior**, which is purely composed of local variables, local methods, and a main method which implements the computation. In this example, behavior Producer and Consumer are of this type of module.

2) **Hierarchical behavior**, which is purely composed of child behavior instances and a main method which specifies the composition of the instances. Behavior Main in the Producer-Consumer model is of this type of behavior.

3) **Channel**, where methods called by the leaf behaviors to implement the communication is defined (channel C in this example).



```
behavior Leaf1
(in event e1, out event e2) {
  void main(void) {
    …
    stmnt_A ;
    waitfor val;
    stmnt_B ;
    …

    stmnt_C ;
    wait e1 ;
    stmnt_D ;
    notify e2 ;
    …
  }
}
```

```
behavior Bhvr_DoTiming() {
  bhvrA i_A(…);
  bhvrB i_B(…);
  bhvrC i_C(…);
  void main(void) {
    do {
       L1:   i_A.main() ;
       L2: { i_B.main() ;
             i_C.main() ; }
       L3: { } }
    timing {
       range (L1; L2; 0; 100) ;
       range (L2; L3; 0; 300) ;
    }
  }
}
```

(A) **waitfor**, **wait**, and **notify** (B) **do-timing** constraint

Figure 3.5: Two types of timing specification in SpecC language

In our proposed method, we utilize the logic of uninterpreted functions with linear arithmetic (QF_LIA) which incorporates the `Core` and `Ints` theories to generate the assertions. We use a function symbol (in SMT-LIB2 language) to represent each time stamp in the model and convert the timing relations between those stamps into assertions. For a newly introduced function symbol, the user can define the number of arguments, and the data type of the argument and the return value. In our method, the return value of an uninterpreted function is seen as the value of a time stamp, and the arguments of the function are used to specify the number of iterations a block is executed (if in a pipelined or loop structure). Take the `waitfor` statement in Figure 3.5(A) as an example. In the example, stmnt_A and stmnt_B will be executed once only and no argument is needed in the function symbol declaration for these two statements, and there is a delay of *val* time units between the execution of stmnt_A and stmnt_B inserted by the `waitfor val` statement. Thus, the assertions below are generated for the timing relation above. Our tool will name the symbol with the full hierarchy path to ensure the uniqueness of each function symbol.

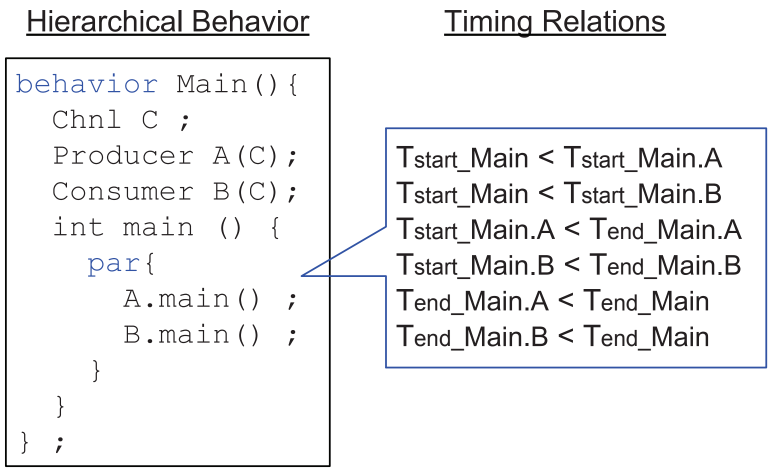
```
(declare − fun Tend.t_stmntA () Int)
(declare − fun Tstart.t_stmntB () Int)
(assert (= Tstart.t_stmntB (+ Tend.t_stmntA val)))
```

3.3.1 Timing Relation for Hierarchical Behaviors

In SpecC, the child behavior instantiation implies a function call to the child behavior. For a behavior S consisting of a set of child behavior instances $\langle s_1, s_2, s_3, ...s_n\rangle$, the following condition holds:

$$\forall i \in \{1, 2, 3, ...n\}, T_{start}(S) \leq T_{start}(s_i),$$
$$T_{end}(S) \geq T_{end}(s_i)$$

The timing relation between the child behaviors is dependent on the execution type specified in the parent behavior. The proposed approach identifies the composition statement in the SpecC hierarchical behavior, and extracts corresponding timing relations for the composition. Take the Main behavior in the producer-consumer model as the example. Our tool first identifies the `par` statement and the instances specified in the compound statement, and then generates a set of assertions for the concurrent composition according to the SpecC execution semantics as shown in Figure 3.6.



```
behavior Main(){
  Chnl C ;
  Producer A(C);
  Consumer B(C);
  int main () {
    par{
      A.main() ;
      B.main() ;
    }
  }
} ;
```

```
Tstart_Main < Tstart_Main.A
Tstart_Main < Tstart_Main.B
Tstart_Main.A < Tend_Main.A
Tstart_Main.B < Tend_Main.B
Tend_Main.A < Tend_Main
Tend_Main.B < Tend_Main
```

Figure 3.6: Hierarchical behavior in Producer-Consumer model

In this work, we support *sequential*, *parallel*, *pipelined*, and *loop* behaviors. SpecC language also defines *FSM* and *exception* behaviors, which we do not support at this time.

1) *Sequential Execution* of statements is defined by ordered time intervals that do not overlap. Formally, for a statement S consisting of a sequence of sub-statements $\langle s_1, s_2, ...s_n \rangle$, the following conditions hold:

$$\forall i \in \{1, 2, ..., n\}, T_{start}(S) \leq T_{start}(s_i),$$
$$T_{end}(s_i) \leq T_{end}(S)$$
$$T_{start}(s_i) < T_{end}(s_i)$$
$$\forall i \in \{1, 2, ..., n-1\}, T_{end}(s_i) \leq T_{start}(s_{i+1})$$

2) *Parallel Execution* can be specified by *par* or *pipe* statements. Formally, for a

`par` statement S consisting of concurrent child statements $\langle s_1, s_2, ...s_n \rangle$, the following conditions hold:

$$\begin{aligned}\forall i \in \{1,2,...,n\}, T_{start}(S) &\leq T_{start}(s_i),\\ T_{end}(S) &\geq T_{end}(s_i)\\ T_{start}(s_i) &< T_{end}(s_i)\end{aligned}$$

3) *Pipelined Execution* of statements is a special form of concurrent execution. Formally, for a `pipe` statement S executed for m iterations, let $s_{i \cdot j}$ represents the j-th iteration of the execution of statement s_i. Then, the following conditions hold:

$$\begin{aligned}&\forall i, x \in \{1,2,...,n\},\ j, y \in \{1,2,...,m\}:\\ &T_{start}(s_{i \cdot j}) < T_{end}(s_{i \cdot j}),\\ &T_{start}(s_{i \cdot j}) = T_{start}(s_{x \cdot y}),\ \ \textit{if}\ \ i + j = x + y\\ &T_{end}(s_{i \cdot j}) = T_{end}(s_{x \cdot y}),\ \ \textit{if}\ \ i + j = x + y\\ &T_{end}(s_{i \cdot j}) \leq T_{start}(s_{x \cdot y}),\ \ \textit{if}\ \ i + j < x + y\end{aligned}$$

A limitation of our approach is that the number of iterations m has to be a *known* integer. If it is statically unknown (i.e. a variable), our tool will prompt the designer to input an upper bound for the loop.

4) *Loop Execution* can be regarded as a special case of *pipelined* execution with only one stage. As above, we assume that the number of iterations is a finite constant.

Assertions for hierarchical behaviors in this chapter looks the same with assertions shown in the previous chapter, but note that each time stamp in this chapter is a 3-tuple as we defined above and we use operations defined in Table 3.1 to specify the timing relations between stamps in the time interval model.

3.3.2 Timing Relation Extraction for Leaf Behaviors

In contrast to the work presented in the previous chapter, we pay significant attention here to analyze the timing information specified in leaf behaviors and channels, which is critical in order to capture communication timing in TLMs. Figure 3.7(A) highlights the statements which are analyzed in the source code as well as the rules to extract the corresponding timing relations for the static analysis. The rule for the `waitfor` statement has been introduced already. We now describe the others.

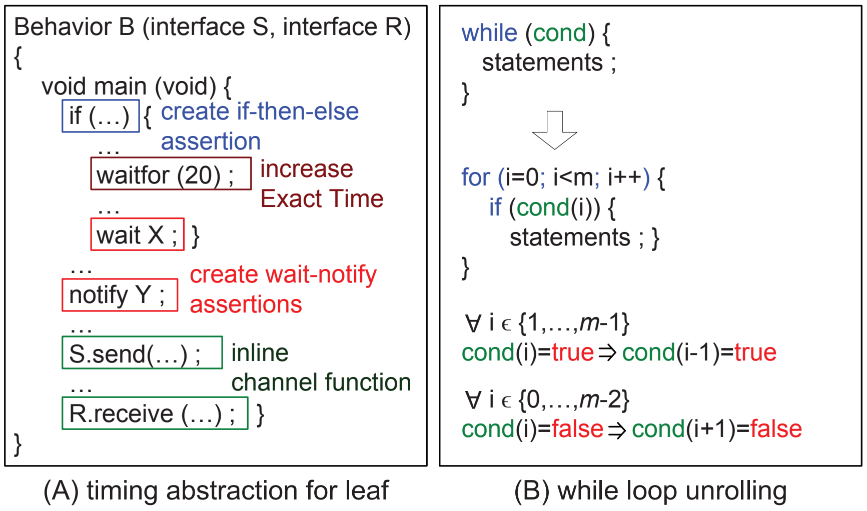

Behavior B (interface S, interface R)
{
void main (void) {
if (…) { create if-then-else assertion
…
waitfor (20) ; increase Exact Time
…
wait X ; }
…
notify Y ; create wait-notify assertions
…
S.send(…) ; inline channel function
…
R.receive (…) ; }
}
(A) timing abstraction for leaf
while (cond) {
statements ;
}
for (i=0; i<m; i++) {
if (cond(i)) {
statements ; }
}
∀ i ∊ {1,…,m-1}
cond(i)=true ⇒ cond(i-1)=true
∀ i ∊ {0,…,m-2}
cond(i)=false ⇒ cond(i+1)=false
(B) while loop unrolling

Figure 3.7: Timing relation extraction for a leaf behavior

1) *Conditional Execution*: When conditional execution, such as a `if` statement or `if-else` statement, is used in the model, we create a time interval $\langle T_{if\_start}, T_{if\_end} \rangle$ and a logic stamp C_{if} which represents the logic condition (for if-else, we also create a 2-tuple $\langle T_{else\_start}, T_{else\_end} \rangle$). Figure 3.8 illustrates the timing relations for the conditional execution. Here, T_{prev} and T_{next} represent the time stamps before and after the conditional execution. As shown with the selection structure in Figure 3.8, the value of T_{next} is dependent on the binary value of C_{if}.

Note that T_{never} is a time stamp with a very large value representing infinity. We represent the situation that a statement will never be executed by giving the corresponding time stamp this large value, as there is no way to represent infinity in the SMT-LIB language. Any time stamp greater or equal to T_{never} means that the corresponding situation will never happen.

Note that our tool will not analyze the specified condition in an if-statement, but only create the conditional assertions as listed in the illustration. It is the SMT solver's job to find an assignment for the condition and time stamps that satisfy the assertions.

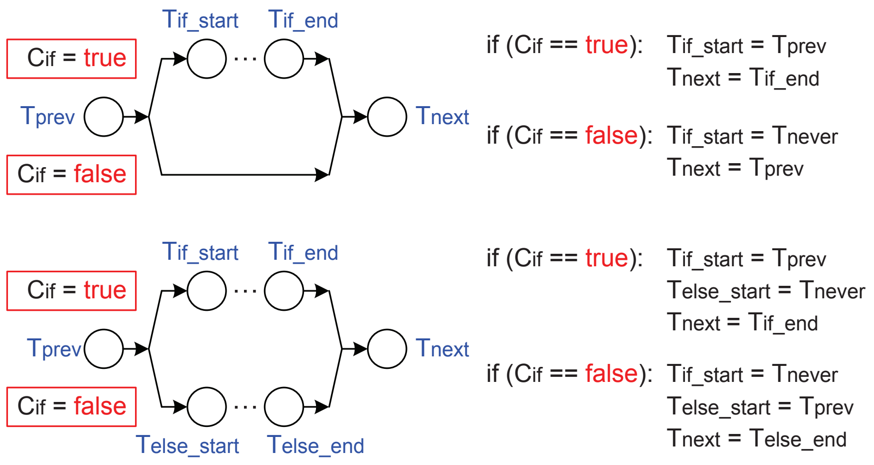


Figure 3.8: Timing relation extraction for the conditional execution

2) *Loop Unrolling*: To limit the verification space and the execution time of the solver, for each loop with undefined iteration count (i.e., the condition is variable), our tool will prompt the designer to provide an the upper bound for the loop, and then unroll the loop to multiple `if` statements. Figure 3.7(B) illustrates the loop unrolling performed by our tool. It also shows that the tool creates implication assertions for the conditions generated by loop unrolling.

3) *wait-notify synchronization*: In order to analyze a TLM with synchronization among multiple concurrent behaviors, we support events and the corresponding `wait-notify` synchronization. When a `wait` statement is executed, it suspends the current thread from execution until the event is triggered by a `notify`. A time interval $\langle T_{start}$, $T_{end}\rangle$ is generated as for other statements. For a `wait` statement W triggered by a

`notify` statement N, the following conditions hold:

$$T_{start}(W) \leq T_{start}(N),$$

$$T_{end.t}(W) = T_{end.t}(N),$$

$$T_{end.d}(W) = T_{end.d}(N) + 1$$

Note that T_{start} equals T_{end} for a `notify` statement. Also, to analyze the satisfiability of the specified timing constraints, we have to determine the mapping between `wait` and `notify` statements, i.e. which `notify` wakes up which `wait`. Our proposed method to generate the assertions for the wait-notify mapping is illustrated in Figure 3.9.

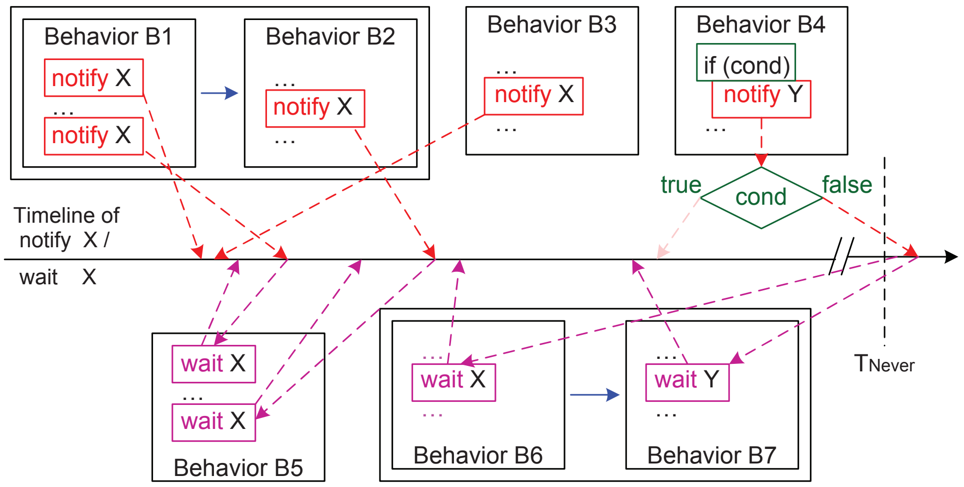


Figure 3.9: Timing relation extraction for the `wait-notify` statement

In this example, all behaviors are executed in parallel except for the two behavior pairs ⟨B1, B2⟩ and ⟨B6, B7⟩ executed sequentially. Our method consists of two steps: (1) for every event in the model, our approach collects all start times of all event notifying statements and generates the assertions to sort the time stamps of the `notify` statements which trigger the event. This step is illustrated in the upper part of Figure 3.9. Note that if the `notify` statement is inside a conditional statement, the value of its time stamp is dependent on the condition. For example, $T_{start.t}$ for

the `notify` statement in behavior B4 in Figure 3.9 will be greater than T_{never} if the logic condition is false.
(2) for every `wait` in the model, we generate the assertions to "search" the sorted time stamps of the `notify` statements and find one that is greater than and the closest to T_{start} of the `wait`, and set the time and delta cycle of the `wait` using the condition we listed above. This step is illustrated in the lower part of Figure 3.9.

4) *Channel Interface Function Call*: In a TLM, the timing information of the target bus protocol and the synchronization mechanism between communicating parties are specified in the interface methods defined in the channel. The communication between the behaviors takes place by calling those interface functions. To generate assertions for the SMT solver, our approach traverses down to the interface method in the channel when it is called. Consequently, the timing information specified in the channel model is taken into consideration during the timing analysis of the behavior.

3.3.3 Liveness and Deadlock

For a multi-PE system model, improper execution order or communication may lead to problems, including deadlock. In our method, a deadlock caused by circular waiting in the model will be reported to the designer in the form of unsatisfiable assertions since there are conflicts in the timing relations. Another potential deadlock would be a `wait` statement missing the wake-up signal. Figure 3.9 also shows examples for two cases. Behavior B6 shows one case in which `wait X` misses all notification for `X` therefore it will never be waken up. Behavior B7 illustrates another case. `wait Y` can not wake up if the condition for `notify Y` in behavior B4 is not true. Both situations are covered by our tool and reported to the designer.

3.3.4 Hierarchical Timing Analysis

The number of assertions generated by our method increases with the complexity of the model. To keep the number of assertions manageable and limit the run time of the SMT solver, our method addresses the complexity growth by analyzing the timing constraints in a hierarchical manner. Timing constraints verified at a lower hierarchy level are regarded as the prerequisite conditions for the verification of the higher level. Verified timing constraints can be specified by use of the do-timing construct in the model. When our method finds a do-timing construct during the design traversal, it will take the constraints as they are and not traverse further down the hierarchy. Thus, the assertions needed for model verification at the higher hierarchical level are greatly reduced.

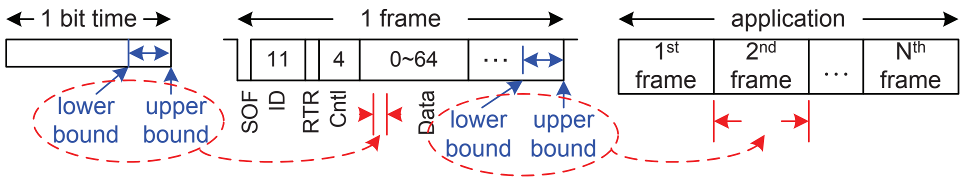


Figure 3.10: Hierarchical timing analysis of CAN bus protocol

Take the CAN bus protocol as an example. The bit time generated by the bit time logic for each engine control unit (ECU) can vary due to different local operating frequencies. Thus, the time needed for transmission can differ from one frame to another. To verify the timing constraint of the frame transmission, we use the pre-verified lower and upper bound of the bit time as prerequisite conditions. Figure 3.10 illustrates the hierarchical timing analysis of CAN protocol from the bit time via frame time up to the application.

3.4 Case Study and Experiments

We use two standard bus protocols widely used in industry to demonstrate our approach. As shown in Table 3.2, both models are of reasonable size with practical analysis times.

3.4.1 Case Study on AMBA AHB

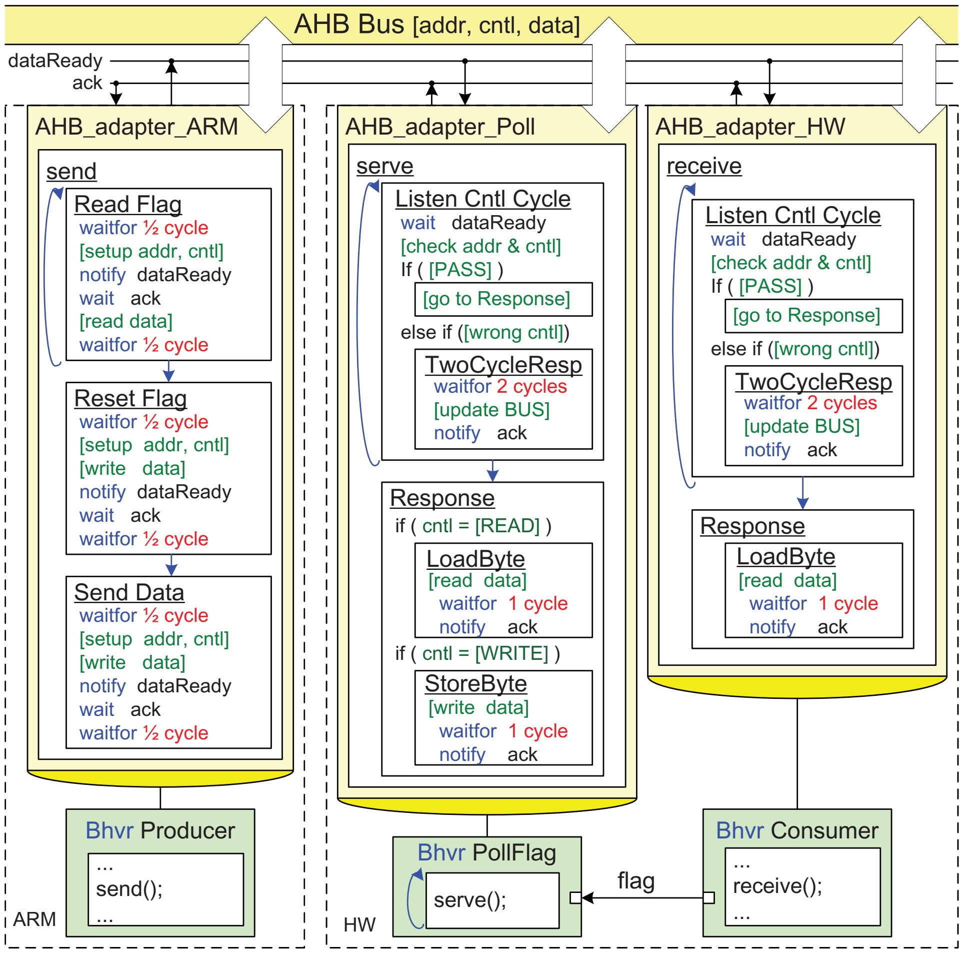


Figure 3.11: TLM of Producer-Consumer example using AMBA AHB

Our first example is a producer-consumer model communicating over an AMBA AHB protocol [55]. Here, the producer and consumer call interface functions `send` and `receive`, respectively, to transfer data through an AMBA AHB channel specified at

TLM abstraction [55]. In this example, the producer acts as the bus master and sends data to the consumer (slave). Figure 3.11 illustrates the TLM with the detailed bus model.

The ARM producer calls the interface function `send` defined in the AHB channel adapter to send the data, and the consumer calls `receive` to service the requests from the master. The procedure contains three steps as the timing diagram shown in Figure 3.2(B). The bus model use polling in the implementation of double handshake channel (no interrupts). Note that a parallel behavior PollFlag is created and instantiated in the HW slave. The functionality of the PollFlag block is to respond for the slave to all polling requests from the master before data is transferred. The data transfer includes three stages in our AMBA AHB TLM:
(1) To check whether the slave is ready, the master reads a ready flag from the slave. The interface function keeps executing the ReadFlag function until the ready flag asserted by the slave and the request to read the flag is successful. When the address and control are valid for the behavior PollFlag, the master reads the flag and proceeds to the ResetFlag function for full synchronization with the slave.
(2) The master and slave synchronize in double handshake fashion. For this, the flag in behavior PollFlag has to be reset after the master has read it. Here, the master requests the reset operation by writing 0 to the address of the flag in PollFlag.
(3) After the synchronization steps, the master proceeds to the SendData function. In this stage, the master sets the address and data signals on the bus with the slave's address and the data content. After the slave checks the address and the control signals, it proceeds to the Response function to receive the data from the bus.
Figure 3.12 shows the time line of the three steps. Note that this example assumes the best case, i.e., successful requests for polling and data transfer. Note that according to the first stage of synchronization, the number of polling iterations (ReadFlag) in the send function depends on the iterations for the ListenCntlCycle in the serve function

as well as the condition in the if statement in the Response function. However, this complex relationship cannot be generated automatically into assertions. In this case, the system designer augments the assertions manually so that the SMT solver can take this relation into account when computing the satisfiability. The 6 augmented assertions in Exp1 in Table 3.2 represent the six correlation shown in Figure 3.12: ① and ② specify that the ReadFlag function ends after the behavior PollFlag responds to the polling request with writing flag to the bus, and the Response function starts after the address and control are valid in the ListenCntlCycle stage; ResetFlag function in the producer ends after the behavior PollFlag responds to the reset request to the flag are specified with ③ and ④; and assertions for correlations ⑤ and ⑥ specify that Response function starts after the producer proceeds to the SendData function, and the SendData function ends after the consumer has received the data .

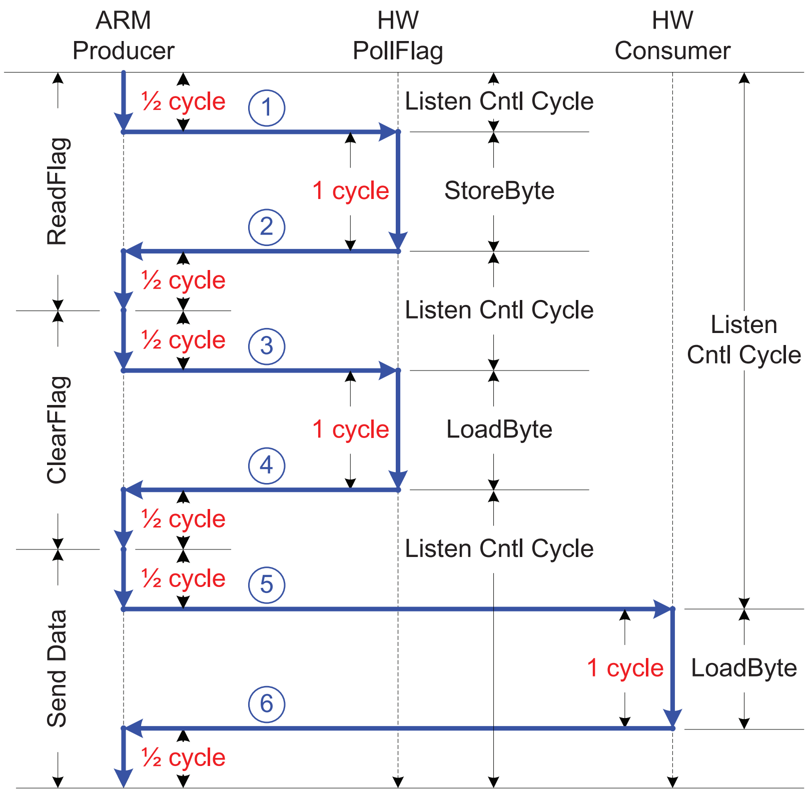


Figure 3.12: The procedure of a data transfer from master to slave on AHB

3.4.2 Case Study on CAN Bus Protocol

The second example is a three-ECU communication over a CAN bus protocol[56]. In this automotive example, the RPMcompute ECU issues a request to an RPMsensor using a remote frame. Upon the reception of the request, the sensor initiates an operation to read revolutions per minute (RPM) from the engine and sends it back to RPMcompute using a data frame. After receiving the raw RPM from the sensor, the RPMcompute ECU calculates the average RPM and sends that to Dashboard ECU for displaying. The procedure is illustrated in Figure 3.13, and the detailed bus TLM is shown in Figure 3.14. Similar to the previous example, the designer has to specify the scenario described above with 3 augmented assertions. These three assertions represent the following timing relations: the end time of remote frame transmission in RPMcompute equals the end time of the remote frame reception in RPMsensor; the end of frame transmission in RPMsensor equals the end of the frame reception in RPMcompute; and the end of the frame transmission in RPMcompute equals the end of the frame reception in Dashboard.

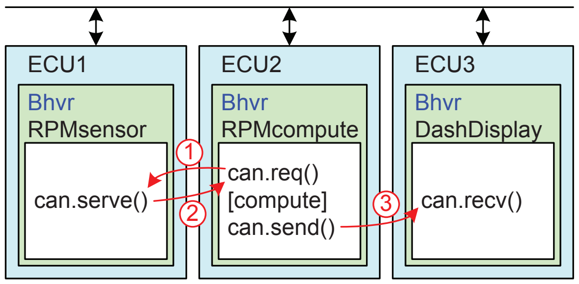


Figure 3.13: Automotive example using CAN bus

According to the CAN bus protocol, each frame transfered with CAN bus protocol is composed of following fields: Start-of-frame (SOF), Identifier, Control(including Remote transmission request, Identifier extension bit, Reversed bit, and Data length code), Data filed, CRC, CRC delimiter, ACK, ACK delimiter, and End-of-frame(EOF). In our CAN bus model, the bit time units required for each communication step (field) in CAN bus protocol are specified with do-timing construct as prerequisite, and we analyze the satisfiability of timing constraints based on this prerequisite.

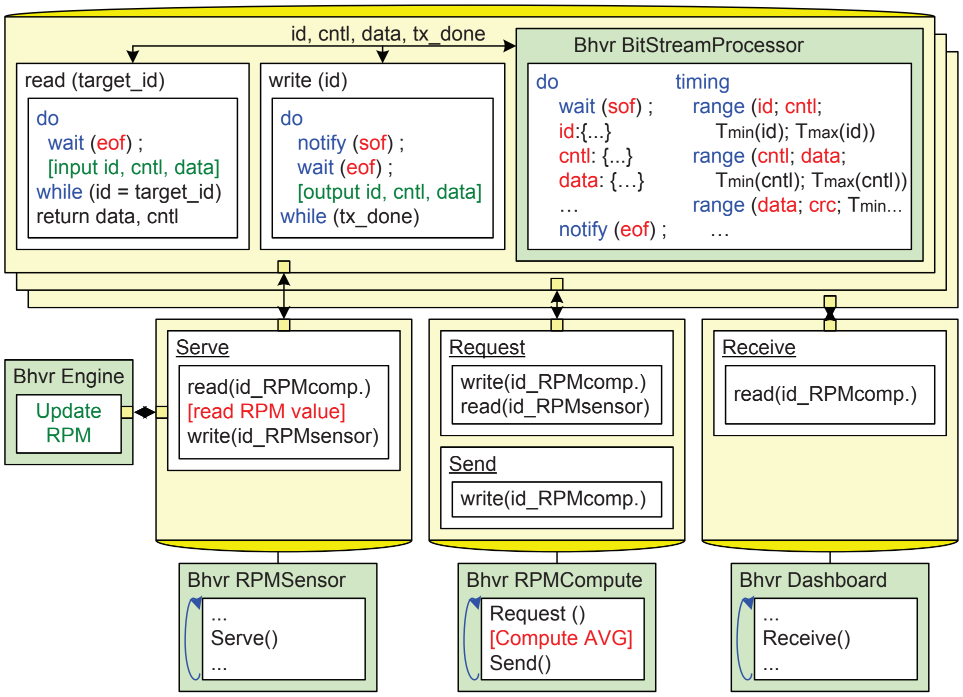


Figure 3.14: TLM of automotive example using CAN bus

3.4.3 Experimental Results

The statistics of TLM timing analysis for both bus protocols are listed in Table 4.2. In the experiments, we verify the satisfiability of liveness and timing constraints. Experiment 1 and Experiment 6 are the SMT2-LIB model generated for the Producer-Consumer and the 3-ECU example with augmented assertions specified by the designer to reflect the real use case. In these two examples, we simply check if there is any conflict caused by circular waiting in the model, and then we augment the model with one assertion asking if the execution of the model can finish in Experiment 2 and Experiment 7. Experiment 3 and Experiment 8 is for checking the minimal execution time of the mode. In this two experiment we augment the models with assertions asking if the execution can finish within a time violating the protocol specification. As for Exp.4, Exp.5, and Exp.9-11, we test the satisfiability of given timing constraints under certain conditions. For example, Experiment 11 in the table shows a scenario where we allow the model to utilize the bus up to 60% maximum, that is, on average over 5 slots only 3 may be used. In these experiment, we model conditons specified in the *Condition* field of the table by giving assertions to control boolean function symbols representing the condition of the while loop statement in the TLM.

Except for the condition and the results, statistics results such as number of assertions, lines of code (LOC) and the run time of the analysis for those assertions are also listed in the table. According to the measured time, the satisfiability searching for these two models and the constraints we added is reasonably fast, and as is often expected, unsatisfiable solutions are faster obtained than satisfiable ones. Note that for the test case 6 the solver gave no answer after two hours of calculation. According to our observation, models with less constraints usually take more time to analysis. To reduce the search time, we add an assumption that the entire transaction finishes in finite time (Experiment 2 and Experiment 7). All experiments have been performed on a host PC with a 4-core CPU (Intel$^{(R)}$ Core$^{(TM)}$2 Quad) at 3.0 GHz.

3.5 Summary

In this chapter, we have proposed an improved approach to verify liveness and timing constraints by extracting timing relations from a TLM design model and using a SMT solver to verify the satisfiability of the corresponding assertions. Compared with the method proposed in the previous chapter, the improved approach can verify the timing information specified in computation as well as in communication. Also, we introduce a hierarchical method to cope with the complexity growth of the model. We demonstrated our approach with two standard bus protocols AMBA AHB and

Table 3.2: Static SMT Analysis of TLM examples using AMBA AHB and CAN bus protocols

Experi– ment	*Constraint*	*Condition*	*#of assertions*	*LOC*	*Time*	*Result*
Liveness and timing analysis for AHB TLM						
1	None	No Circular Waiting	240 (6 aug.)	19389	332s	SAT
2	T_{end}(Prod) $< T_{never}$	No Circular Waiting	241 (7 aug.)	19392	313s	SAT
3	T_{end}(Prod) $<$ 6 cycles	Min. execution time	242 (8 aug.)	19395	4s	UNSAT
4	T_{end}(Prod) $\leq$ 10 cycles	Polling succeeds in the first 2 attemps	243 (9 aug.)	19401	333s	SAT
5	T_{end}(Prod) $\geq$ 10 cycles	Polling succeeds in the first 2 attemps	243 (9 aug.)	19401	114s	UNSAT
Liveness and timing analysis for CAN TLM						
6	None	No Circular Waiting	382 (3 aug.)	24963 24963	$>$ 2hr	UNKNOWN
7	T_{end}(DashDisp) $< T_{never}$	No Circular Waiting	383 (4 aug.)	24972	189s	SAT
8	T_{end}(RPMcmp) $\leq$ 200 units	Min. execution time	384 (5 aug.)	24975	50s	UNSAT
9	T_{end}(RPMcmp) $\leq$ 500 units	Write always fails in the 1st attempt	387 (8 aug.)	24984	284s	SAT
10	T_{end}(RPMcmp) $\geq$ 300 units	Write succeeds in the 1st attempt	387 (8 aug.)	24984	5s	UNSAT
11	T_{end}(RPMcmp) $\leq$ 500 units	Bus utilization $\leq$ 60%	385 (6 aug.)	25054	135s	SAT

CAN bus. Our approach utilizes the designer's augmented assertion reflecting the properties of interest. In future work, we plan to improve the interaction between the designer and the SMT assertion generator.

Chapter 4

May-Happen-in-Parallel Analysis of ESL Models

4.1 Introduction

For concurrent and parallel languages, the May-Happen-in-Parallel (MHP) problem asks for two given statements whether or not there is a possibility where these two statements are executed at the same time. MHP analysis is useful as a basis for static model analysis and debugging, such as resource allocation and contention or race condition detection. In this chapter[57] [58], we propose an approach to abstract an UPPAAL model [60] from a system in SpecC system level description language (SLDL) and analyze the MHP statements by verifying the model with the UPPAAL verifier. In contrast to other techniques, such as [65], our approach can not only check the MHP property of two statements, but also of any number of statements and other properties.

4.1.1 MHP Analysis using UPPAAL Model Checker

Figure 4.1 illustrates the analysis flow and the tool chain we use in this work to analyze MHP statements.

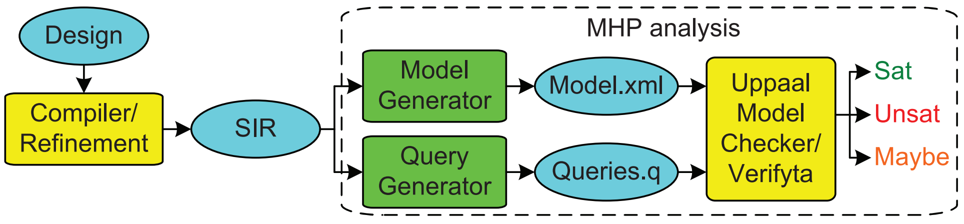


Figure 4.1: MHP analysis flow with UPPAAL model checker

Before the analysis, the system design is compiled into a System Internal Representation (SIR) data structure [14]. The internal representation is then read by the Model Generator and Query Generator modules of the MHP analysis tool. The yellow blocks in the illustration are the existing tools to generate the SIR model and verify it, and the green blocks represent the tool we created to connect the design flow and the analysis. The Model Generator abstracts an automata model from the SIR structure, and the Query generator generates the queries for MHP analysis, asking if any two given scheduling points could possibly happen at the same time. According to the model and queries, the UPPAAL model checker will give one of the following answers: *satisfied*, *not satisfied*, or *maybe*. Each satisfied query represents a pair of MHP scheduling points, while an unsatisfied query means the two scheduling points will not happen at the same time. The *maybe* answer is given when the upper approximation option in the state space representation of UPPAAL verifier is enabled. In our experiments, we will use an example to demonstrate these three cases.

This chapter is organized as follows. Section 4.1.2 reviews the related work of MHP analysis as well as system modeling with automata. The motivation of this work is discussed in Section 4.2. Model and Query Generator are described in Section 4.3 and Section 4.4. In Section 4.5 the model optimization to shorten analysis time is introduced. Section 4.6 shows the experimental results for multimedia applications (two JPEG encoders, and MP3 decoder). In addition to MHP analysis, we extend this work to support timing delay and power consumption analysis, and the concept is introduced in Section 4.7. Finally, summary and future work are discussed in Section 4.8.

4.1.2 Related Work

MHP analysis and race condition detection in concurrent and parallel language has been broadly studied. [69] proposed an approach to find MHP statements in a untimed concurrent program with trace flow graph (TFG). [66, 67, 68] detect races and analyze non-deterministic anomalies in timed concurrent models, but in those methods simulation is required. In [65], an approach using static segment aware detection to identify MHP segment pairs is proposed. Like [65], our approach focuses on the timed model and does not require simulation. Compared to [65], our approach reports more precise results at the price of longer analysis time. Another advantage of our approach is that instead of just giving the MHP analysis results, our method can report the trace of transitions showing how statements can be executed in parallel. In addition, our approach can also identify the MHP sets of any number of statements and verify other properties like liveness (deadlock detection) and timing guarantees.

As for the system modeling, [62] and [63] propose approaches to model the behavior of SLDL designs with automata in PROMELA, and in [**?**] the design is modeled as a network of timed automata. The model is then analyzed with SPIN and UPPAAL model checker, respectively. Our approach [57] [58] also models the application with a network of automata and analyzes it with UPPAAL model checker. Compared with other works, our approach supports the modeling of richer design compositions and channel communication. More important, instead of only supporting the behavior of regular discrete event simulation (DES), the scheduler process in our model also supports *parallel* DES (PDES) [64], which is essential to our MHP analysis.

4.2 May-Happen-in-Parallel Analysis

In this work, the definition of May-Happen-in-Parallel (MHP) analysis is defined as *"for two given statements in the design, is it possible that these two statements are executed at the same time?"*. In Chapter 1 we have used an example illustrated in Figure 1.6 to show how a MHP case can cause a wrong execution result. Here we describe why this issue cannot or may not be detected with simulation-based validation and requires other analysis techniques to identify the MHP statements in the design.

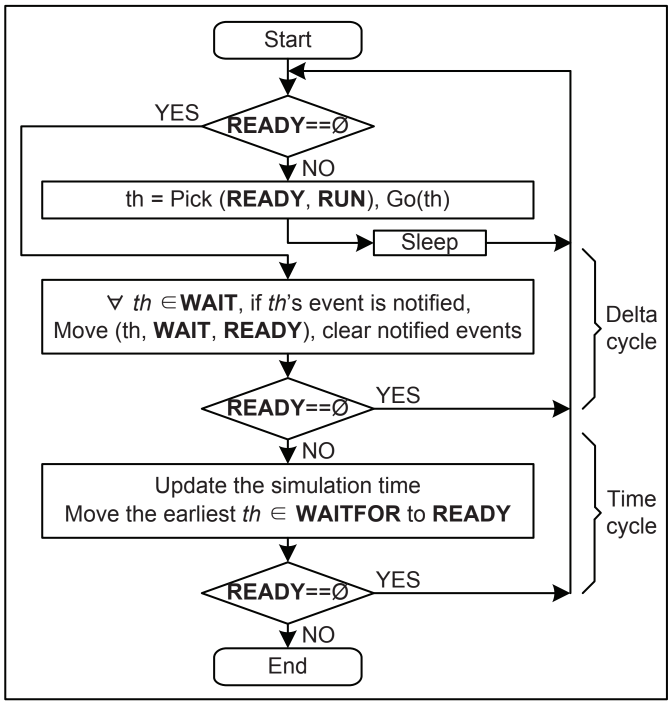

Start
YES
READY==Ø
NO
th = Pick (READY, RUN), Go(th)
Sleep
∀ th ∈WAIT, if th's event is notified,
Move (th, WAIT, READY), clear notified events
Delta cycle
READY==Ø
YES
NO
Update the simulation time
Move the earliest th ∈ WAITFOR to READY
Time cycle
READY==Ø
YES
NO
End

Figure 4.2: Discrete event simulation (DES) algorithm (source[36])

Figure 4.2 and Figure 4.3 show two discrete event simulation algorithms. The (regular) discrete event simulation algorithm [36] is illustrated in Figure 4.2 is used in the SpecC scheduler. In SpecC, concurrent threads are created for explicit parallelism description, and their executions are coordinated by the scheduler. The threads ready to be executed are stored in the READY queue, and one thread is picked among the threads in READY queue for execution at a time. When the READY queue is empty, the scheduler will fill the queue again by waking up the threads who have received the event they are waiting for. These thread are moved from WAIT queue to READY queue, and a new delta cycle begins. If the queue is still empty after event delivery, the scheduler advances the time and moves the threads with the earliest waiting time from WAITFOR queue to READY queue. If there is no more thread waiting in the WAITFOR queue, the simulation is done. In this algorithm, threads in the READY queue are picked and run one-by-one in non-deterministic order to model the concurrent behavior, i.e., even with the explicit parallelism there is only one thread executed at a time. In this case, the race condition caused by parallel access to shared variables can not be triggered and detected with simulation.

To make use of the computation power provided by multi-core processor, our lab developed a *Parallel Discrete Event Simulation* (PDES) [64] algorithm in which multiple threads can be activated at the same time and threads can be really executed in parallel. The scheduler keeps picking threads in the READY queue for execution as long as there is available core. This algorithm is illustrated in Figure 4.3.

With multi-core processors and PDES algorithm, now it is possible to detect the race condition with simulation. However, triggering a potential race condition and detect it is still not guaranteed with parallel simulation. First, there is still a chance that the shared variable is accessed by multiple threads in correct order, and the simulation outcome just happens to be correct. Besides, race condition can be input data dependent, i.e., it may only be triggered by certain input vectors. Since the result

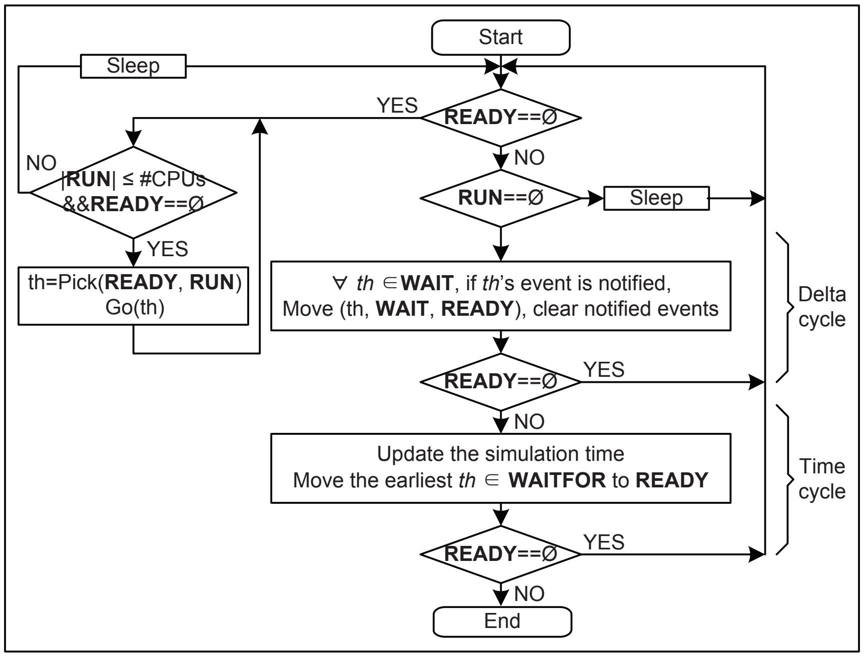

Start
Sleep
YES
READY==Ø
NO
NO
\|RUN\| ≤ #CPUs &&READY==Ø
RUN==Ø
Sleep
YES
th=Pick(READY, RUN) Go(th)
∀ th ∈WAIT, if th's event is notified, Move (th, WAIT, READY), clear notified events
Delta cycle
READY==Ø
YES
NO
Update the simulation time Move the earliest th ∈ WAITFOR to READY
Time cycle
READY==Ø
YES
NO
End

Figure 4.3: Parallel discrete event simulation (PDES) algorithm source[64])

of simulation-based verification is input-vector dependent, we could encounter a case that for certain input vector the result is correct even race condition happens but for another set of input the result is wrong.
Finally, detecting the occurrence of race condition is not good enough. When race condition is observed, the designer needs to identify where it happens in the model. It is difficult to find out the statements causing the race condition simply with the simulation result, especially the result can vary from time to time when race condition happens.

The reasons described above inspire us to develop an approach which is capable of detecting MHP statements in the design and is also input vector independent. Also, the proposed approach should be able to assist designers in debugging when MHP statements are detected, i.e., report the condition in which statements can be executed in parallel. As the analysis flow illustrated in Figure 4.1, our concept here contains two steps: The first step is to abstract a system level design into an UPPAAL automata model in which the state transition simulates the behavior of the PDES algorithm. The second step is that for all possible combinations of two given statements in the design, we generate a query asking whether or not these two statements can possibly be executed in parallel. The model and the set of queries are analyzed with UPPAAL model checker, and an answer will be given for each query. If a query is satisfiable, which means the corresponding statements in the query can possibly be executed in parallel, a trace recording the transition to state satisfying this query can be given by the UPPAAL model checker. With this trace, the designer can easily recreate the scenario in which these two statements happen concurrently.

4.3 SLDL Design to UPPAAL Model

As we described in the previous section, the proposed method in this paper includes two parts: model generator and query generator. In this section, we first briefly introduce the basic concept of mapping a system design into an UPPAAL system model. After the introduction, we describe how we convert the details of behaviors and scheduler for parallel discrete event simulation algorithm into UPPAAL automaton templates.

4.3.1 UPPAAL Automaton and System Model

Before the description of our approach, we first briefly introduce the basic concept of an UPPAAL system model. An UPPAAL model consists of a network of concurrent processes which are created by instantiating the pre-defined timed automaton templates, and these concurrent processes can communicate and synchronize with each other through parameters and channels defined. The system can be seen as a set of

automata running concurrently, i.e., when there are multiple transitions enabled in the instance processes, these enabled transitions can take place in non-deterministic order. An UPPAAL system model is usually composed of three parts:
1) definition of data structures, functions and global variables declaration,
2) definition of automaton templates, and
3) system definition.
The first part is quite similar to programming language like C. In an UPPAAL model the designers can define global variables and function to be accessed and called by all instance processes. Except for the basic variable types supported by UPPAAL modeling language, such as integer and Boolean variable, designers can also define their own complex data structures using `struct` construct.
As for the second and third parts, we use a simple model in Figure 4.4 to illustrate the basic components in an UPPAAL model. In this example, the model is composed of two processes `Inst1` and `Inst2` (illustrated as green blocks in the figure) communicating through channel [sync] and integer [a].

The templates of the automata has to be defined first and then they can be instantiated in the system definition to create the processes and build the model. To build a model in Figure 4.4, automaton template `TA1` and `TA2` have to be defined first. In the definition of a template, states in the automaton, transitions between states, the conditions to enable a transition and expression to be evaluated on the transition are clearly specified. In UPPAAL model they are named as *location*, *transition*, and *label* respectively.
Take template TA1 as the example. Four locations X1∼ X4, transitions X1→X2, X1→X3, X2→X4, and X3→X4 are defined. Labels are shown as blocks on transitions in the illustration, and they are attached to transitions to specify the expressions and conditions in which transitions are enabled.
UPPAAL model checker supports three types of label for different purposes. The

first type are *update* labels (b=1, b=3, or a=b+1 in black in this example) define the expression to be evaluated during the transition. The second type are *guard* labels represent the condition when transitions are enabled. When process Inst2 is at location Y2, integer *a* defines which transition is enabled in this process. Note that when *a* equals 3, process Inst2 stays at location Y2.

The third type are *synchronisation* labels which define the event synchronization between transitions in multiple processes. The synchronisation labels with exclamation mark are event producers and the labels with question marks are consumers. In this example, whenever transition X2→X4 or X3→X4 happens, the transition from Y1→Y2 happens at the same time if process Inst2 is at location Y2.

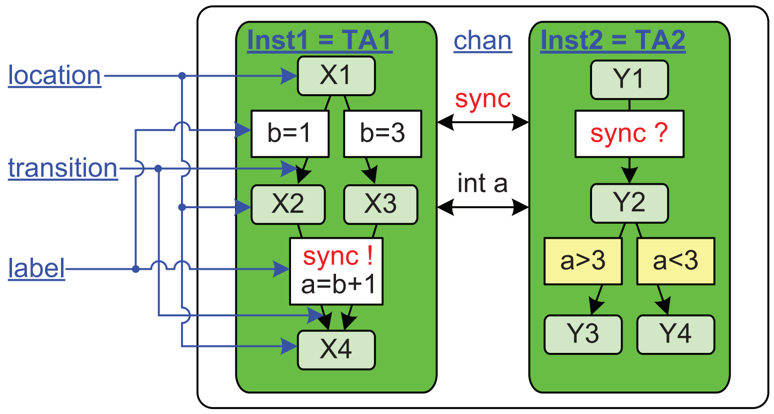


Figure 4.4: Example of an UPPAAL system model

The final step to build an UPPAAL model is to instantiate predefined templates and create a network with concurrent processes in the system definition. The instance processes created in the system definition can communicate with parameter and channel. In this example, channel [sync] and integer [a] are defined in the system definition and used to connect processes Inst1 and Inst2.

4.3.2 PDES Model in UPPAAL

Figure 4.5 shows our structure of the UPPAAL model for a system model. A system model is usually composed of multiple computation blocks(modules, behaviors) with communication (port, channel, event synchronization) between those blocks. Using SpecC SLDL, the computation block is described as `behavior`, and our approach distinguishes two types of behaviors: *Leaf* and *Hierarchical* behavior. A leaf behavior is purely composed local variables, local methods, and a main method which implements the computation and the communication, while a hierarchical behavior is purely composed of child behavior instances and a main method specifying the composition of the instances.

A system model is constructed with a topmost behavior `Main` and the sub-instances of hierarchical and leaf behaviors.

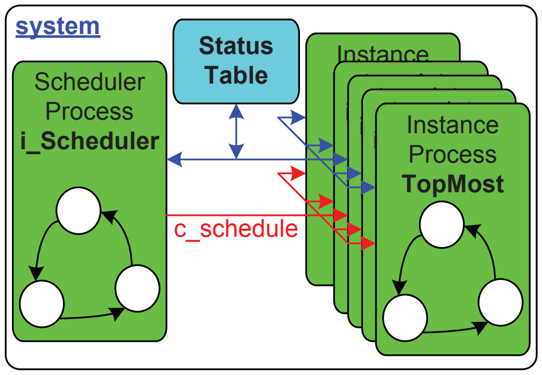


Figure 4.5: SLDL Design to UPPAAL automata conversion

In our approach, we first abstract an automaton template from each behavior. These templates are then instantiated to build a process network modeling the system. Each behavior instance in the design is one-to-one mapped to a process through template instantiation.

Except for the behavior instance processes, the system also contains a *scheduler* process. Like in the discrete event simulation we have a central scheduler to coordinate the execution of concurrent threads, in our approach we also have a scheduler process to control the transitions in the instance processes. All instance processes are connected to the scheduler process through a structure *status_tree* and a channel *c_schedule*. The status tree is a tree structure designed to keep the status information for all behavior instances. The structure of the status tree reflects the hierarchy of the design: roof of the tree is the status node for the topmost behavior Main, and the nodes beneath are the status nodes for its child instances.

To better understand our approach, we provide an introductory SLDL example from [65] in Figure 4.6 to demonstrate the structure of our UPPAAL model as well as the [status_tree].

As shown in Figure 4.6, four processes are created through template instantiation in the system definition for the scheduler, topmost behavior Main, and its child instance A and B, respectively. All instance processes are connected to the scheduler process through [c_schedule] and [status_tree]. Note that there is also a channel *c_call* between the parent process and its child processes, as for some compositions the activation of child processes are coordinated by the parent rather than the scheduler. The detail of channel c_schedule and c_call will be described in the following sections.

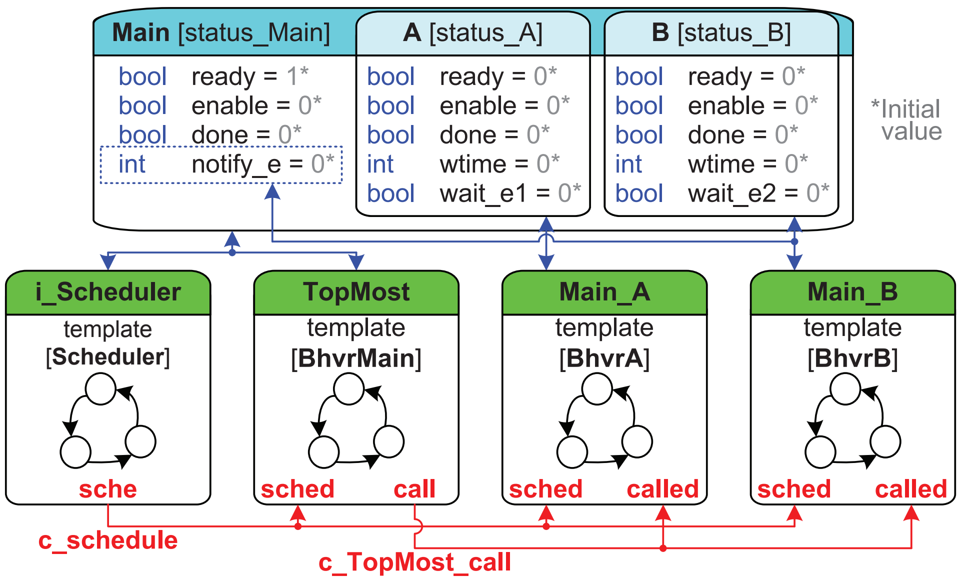
In Figure 4.6 we also show the detail of [status_tree]. The status flags *ready*, *enable*, and *done*, are kept in the node to represent the status of the corresponding instance, and certain additional flags such as *wtime* or *notify_X* are added to the status tree to store the information for event synchronization and time advancement. For example, flag `notify_e` is added the status node of Main for `event e` in the behavior. The detail of the additional flag will be described in the following sections. Based on the information in [status_tree], the scheduler activates instance processes in the proper order and ensures the transitions are compliant with the parallel discrete event execution semantics.

```
int array[10] = {0, 1, 2,..., 9};
int x = 0, y = 0, z = 0, w = 0 ;
behavior BhvrA (event e1)
{
  void main(){
    int i = 0;
    for (i=0; i<9; i++){
      y = x + 27 ;
      waitfor 1;
      w++ ;
      wait e1 ;
      x = array[i]*42; }}
} ;

behavior BhvrB (event e2)
{
  void main(){
    int i = 0;
    for (i=0; i<9; i++){
      y = y*42 + z ;
      waitfor 2;
      array[i] = array[i]*4+x++;
      notify e2 ;
      wait e2 ;
      z ++ ; }
  }
} ;

behavior Main ()
{
  event e ;
  BhvrA A(e) ;
  BhvrB B(e) ;
  int main() {
    par
    {
      A.main() ;
      B.main() ;
    }
  }
} ;
```

(A) SLDL source code for a simple design example



(B) UPPAAL model for the simple design example

Figure 4.6: SLDL source code for an introductory design example

While the hierarchy of instances has been flattened in the system definition of the UPPAAL model, [status_tree] still maintains the hierarchy of the design. The reference of each node is passed to the corresponding instance process as parameter so that the process can access its flags and its children's.

Except for the reference of the node, the reference of a flag can also be passed to a process as needed. Take flag [notify_e] as an example. The reference of this flag is passed to process Main_B for statement "notify e2"in BhvrB as process Main_B needs to set this flag when it reaches the location of the notify statement.

In the following sections, we first introduce the UPPAAL automaton template for hierarchical and leaf behavior as well as the status flag updating in the transition. Then, we describe the template of scheduler automaton in detail, and explain how scheduler process interacts with the instance process according to the status flags to simulate the behavior of Parallel Discrete Event Simulation.

4.3.3 Automaton Template for Hierarchical Behaviors

To build an UPPAAL model simulating a SLDL design, each behavior in the SLDL design is abstracted to an automaton template. In the system description of the UPPAAL model, each behavior instance in the design is one to one mapped to a process in the system by instantiating the defined automaton template.

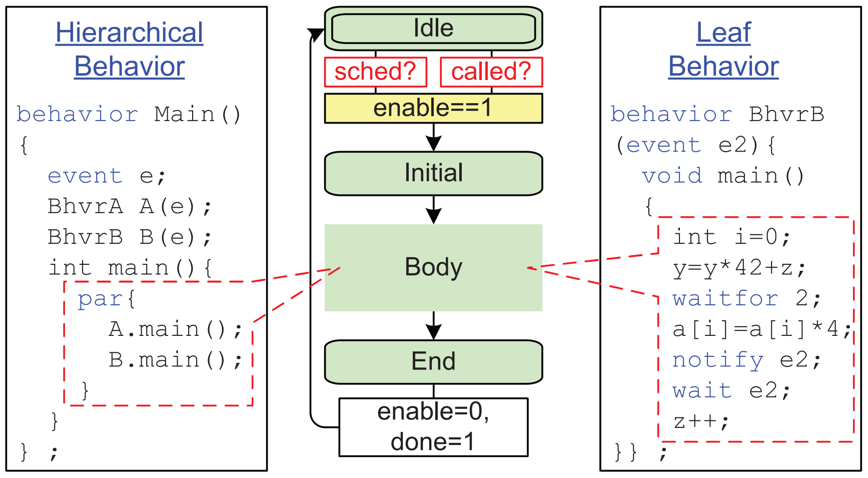


Figure 4.7: Automaton template for both hierarchical and leaf behavior

Figure 4.7 shows the common automaton template for both hierarchical behavior and leaf behavior in our approach. The central part of the illustration shows the basic structure for all types of behaviors. In each behavior template there are at least three locations: [Idle], [Initial], and [End]. All behavior processes start at [Idle] which represents the status where the corresonding behavior instance are waiting for activation, and wait for transition [Idle]→[Initial] to be activated. Location [Initial] marks the moment when a behavior process is activated by the scheduler or by its parent process. Transition [Idle]→[Initial] is only activated when the enable flag in the status node for the corresponding instance is set and the synchronization is triggered through channel [c_schedule] or [c_call]. As described before, channel [c_schedule] is used to synchronize scheduler and behavior instance processes, while channel [c_call] implements the synchronization between parent process and child processes.

The statements in the main method of the behavior are converted to corresponding locations and transitions, and then inserted into the automaton template between location [Initial] and End]. Location [End] in the model represents the state where the execution of the process is finished. After the execution of the behavior instance is finished, the process reaches [End] and then goes back to [Idle]. On the transition [End]→[Idle], the done flag of this instance is asserted and enable flag is reset. Figure 4.8 also shows the locations and transitions for the four types of composition defined in SpecC SLDL.

In the following subsections, we describe the corresponding locations and transitions generated for four types of composition in hierarchical behavior, and the automaton template is illustrated in Figure 4.8.

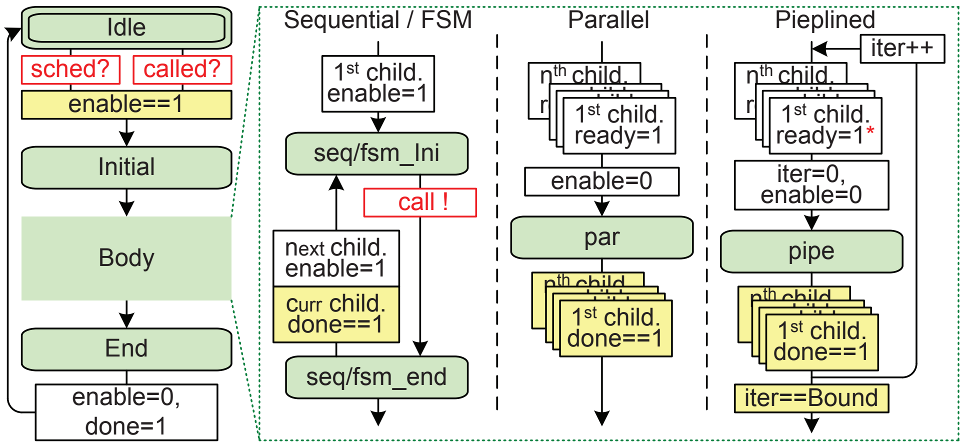


Figure 4.8: Representation of hierarchical behaviors in UPPAAL

Sequential Composition

Based on the execution semantics of sequential composition, the children instances are executed in the order in which they are instantiated in the parent behavior and their execution does overlap with each other. This semantic is modeled in the transition between location [seq_ini] and [seq_end]. When the parent process reaches location [seq_ini], the enable flag of the first child instance is set by the parent process. The parent process then activates the transition [Idle]→[Initial] in child by triggering the synchronization over channel [c_call] with synchronize label in transition [seq_ini]→[seq_end]. After the activated child process reaches [End], the done flag of the child process is set and transition [seq_end]→[seq_ini] is enabled. The parent process then activates the next child process in the same manner. After the execution of all children finishes, transition [seq_end]→[End] is enabled, and the parent process sets its done flag, resets the enable flag, and goes back to [Idle].

FSM Composition

Our approach also supports the modeling of finite-state-machine composition `fsm` in which the child instances are executed conditionally. The template for the FSM composition is very similar to the sequential composition. For both compositions only one child instance is activated at a time and the next is activated after the current one is finished. However, the child automata is activated in the order specified in the FSM transition statements (in one-to-one fashion). Each transition statement in the FSM composition is one-to-one mapped to an update label which enabling the next child instance on the transition [fsm_end]→[fsm_ini].

Parallel Composition

The instances in a behavior with `par` composition are executed in parallel manner. To model the parallel execution semantics, all child processes are activated at the same time and then executed concurrently. As shown in Figure 4.8, in transition [Initial]→[par] the update label sets the ready flags of all child instances and clears enable flag of the parent instance. The scheduler process detects the assertion of the ready flags and activates child processes by setting the enable flags of all child instances and triggering the synchronization over channel [c_schedule]. Note that in order to synchronize with multiple processes, [c_schedule] must be a channel of *broadcast* type. Before the execution of all child instances are finished, the parent process waits at location [par] until the done flags of all child instances are set.

Pipelined Composition

Another composition in which child instances are executed in parallel manner is pipeline composition. Similar to the parallel composition, the child instances of active stages in a `pipe` composition are activated at the same time, too.
There are two major differences between pipelined and parallel composition. The first difference is that instead of being executed once in the parallel composition, the child instances are executed iteratively in pipelined composition. The number of iteration can be specified just like the condition expression of a for-loop statement. However, considering the pipeline filling and flushing stage, not all child instances are activated in all iterations. Therefore, the way to set the ready flags for the child instances in par and pipe composition are different. For a pipeline composition with n instances and m iterations, the ready and done flag of i-th instance at s-th iteration, $i \in \{1, 2, ..., n\}, s \in \{1, 2, ..., m\}$, are set in the iterative transition [pipe]→[pipe] as

follows:

$$\begin{array}{ll} \texttt{if } \texttt{i} \leq \texttt{s} \leq \texttt{m} + \texttt{i} - 1, & \texttt{Inst(i).ready} = 1, \texttt{Inst(i).done} = 0 \\ \texttt{else} & \texttt{Inst(i).ready} = 0, \texttt{Inst(i).done} = 1 \end{array}$$

4.3.4 Automaton Template for Leaf Behaviors

As for the abstraction of leaf behaviors, instead of generating location and transition for every statements, only certain statements of interest are taken into consideration in the model generation. Here we categorize the statements of interest into three types:

1) control-flow statement,

2) waitfor and wait-notify synchronization, and

3) channel communication

Statements other than these three types are abstracted away (ignored) since they have no influence on the transition in the automaton. Note that if there is no waitfor, wait-notify statement or channel communication in the sub-statements, the control-flow statement is abstracted away, too.

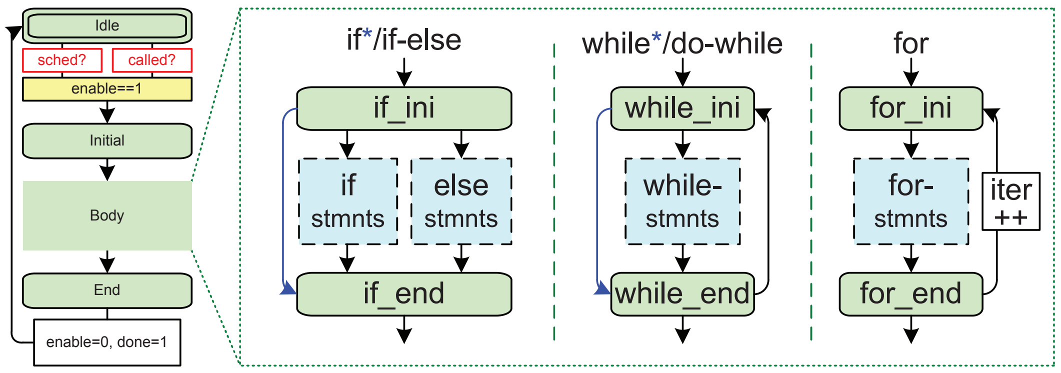


Figure 4.9: Control flow statements reflected in leaf automata

Control-Flow Statement

Figure 4.9 shows the corresponding locations and transitions generated for if/if-else, while/do-while, and for loop. We generate a pair of locations [ini] and [end] for these three types of statements to encapsulate their sub-statements. For if/if-else statements, we create transitions from location [ini] into the sub-statements for both cases, and two paths merge at location [end].

For the do-while/while loop statement, transition [end]→[ini] is inserted to execute the sub-statement for non-deterministic times, and a transition bypassing the sub-statement is provided for the while statement in case the condition is false at the first iteration. The for-loop statement is similar to the while-loop with guard and update labels in the transition to count the iteration. The difference is that guard and update labels are created to count the iteration based on the condition expression if the iteration number is clearly specified.

Event Synchronization and Time Advancement

In SpecC language, the event synchronization and time advancement are implemented with *wait-notify* and *waitfor* statements. The locations and transitions for waitfor, wait, and notify statements are illustrated in Figure 4.10.

Two locations [ini] and [end] are created for each wait and waitfor statement. Locations [wait_ini] and [waitfor_ini] represent the states in which the instance is waiting for event delivery and time advancement, while locations [wait_end] and [waitfor_end] represent the states where the instance is waked up. As for *notify* statements, one location [notify] is created for each event notification statement in the design. According to the execution semantics, it takes at least one delta cycle or simulation clock advancement to wake up an automaton from suspension caused by wait or waitfor.

Since the scheduler process is the only module which keeps track of the event delivery and time advancement, the suspended automata are re-activated by the scheduler. When a process reaches location [ini] for a wait or waitfor statement, the process suspends itself by clearing its enable flag and waits at [ini] until it is woken up by the scheduler.

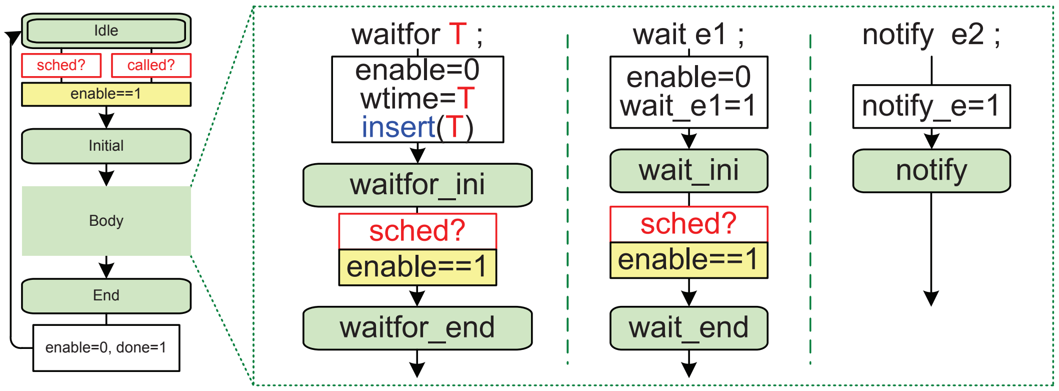


Figure 4.10: `waitfor` statement and `wait-notify` synchronization

A *waitfor* statement with argument N suspends the current instance from execution for N time units. In our model, a global sorted queue is used to store the waiting time of the suspended instance, and flag [wtime] is also added to the status node of each leaf instance to identify if an instance is suspended by a waitfor. When a process reaches location [ini] of statement `waitfor T`, it suspends itself and set the wtime variable to `T`. A predefined function `insert` is called to insert `T` into the queue for time advance in the scheduler. The waiting time in the queue will be read out in order in the scheduler process to decide what is the next time units to be advanced. This suspended process is reactivated by the scheduler after the simulation time is advanced by `T` units.

A wait statement suspends the current thread from execution and waits for a statement notifying the same event is executed. In our model, a `wait` flag is added to the status node for each wait statement in the instance. Take the introductory design in Figure 4.6 as the example. Flag [wait_e1] and [wait_e2] are added to the status node of instance A and B for statement `wait e1` and `wait e2`. When a process reaches location [ini] of a wait statement, it suspends itself and sets the corresponding wait flag. When another process reaches the location of a notify statement delivering the event, the suspended process is reactivated by the scheduler.

A notify statement wakes up all suspended threads waiting for the notification of a certain event. In our model, a *notify* flag is added to the status node for each event. The reference of the flag is passed to all instances notifying the event so that those processes can assert the flag when they reaches the notify location. For example, in Figure 4.6 flag *notify_e* is added to the status node of Main, and the reference of this flag is passed to the process of instance B for the statement `notify e2`at line 22.

Channel Communication

Channel communication is essential in system level modeling, and SpecC SLDL supports various standard channels, such as semaphore, mutex, handshake, double-handshake, and queue. In SLDLs, the channel communication between blocks is implemented by making function calls to the method defined in the channel instances to transfer data from sender to receiver. Our approach supports the modeling of the three mostly used channels, which are handshake, double-handshake, and queue. Here we show the standard double handshake channel illustrated in Figure 4.11. For other two channel types, the corresponding locations and transitions will be generated according to their detailed implementation respectively. In this example, channel instance C is connected to instances S and R so that these two instances can call functions send() and receive() defined in the channel to communicate.

Instead of create a process for the channel instance like for behavior instance, we inline the communication method into the sender and receiver process. The right part of

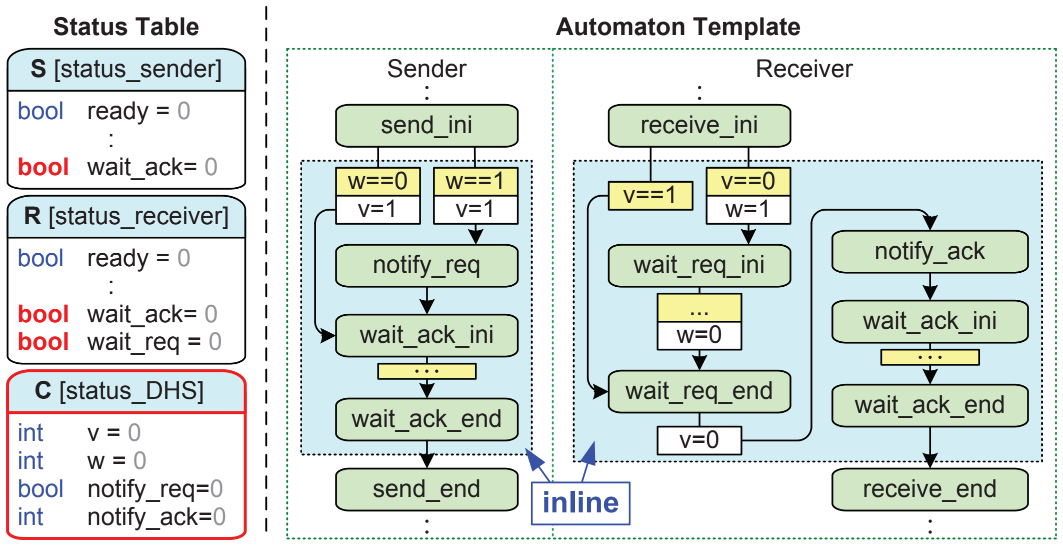


Figure 4.11: Communication using standard double handshake channel

Figure 4.11 illustrates the inlining. Except for locations [ini] and [end] inserted for the channel function call, the locations and transitions modeling the detail of the communication method are also inlined between [ini] and [end]. The text and block in red in the left part shows the additional node and flags for using a double handshake channel. Flags *wait_ack* and *wait_req* are added to the status node of the sender S and receiver R, and a status node C is inserted in the tree for the channel instance C. The wait and notify locations here synchronize the sender and receiver. The guard and update labels in the transitions make sure no matter which function is called first, the send and receive function finish at the same time.

4.3.5 Scheduler Automaton

In this section we show the scheduler process modeling the discrete event simulation. Note that our scheduler automaton supports the modeling for both regular DES and parallel DES. The difference is that the regular DES mode allows one active process at a time, while the parallel DES mode allows many activated processes. Since in this paper the scheduler needs to run in PDES mode for MHP analysis, the following description of scheduler automaton is for PDES mode.

Figure 4.12 illustrates the template of the scheduler automaton. The composition of the scheduler automaton can be roughly divided into three parts:

1) instance activation,

2) event delivery, and

3) time advancement.

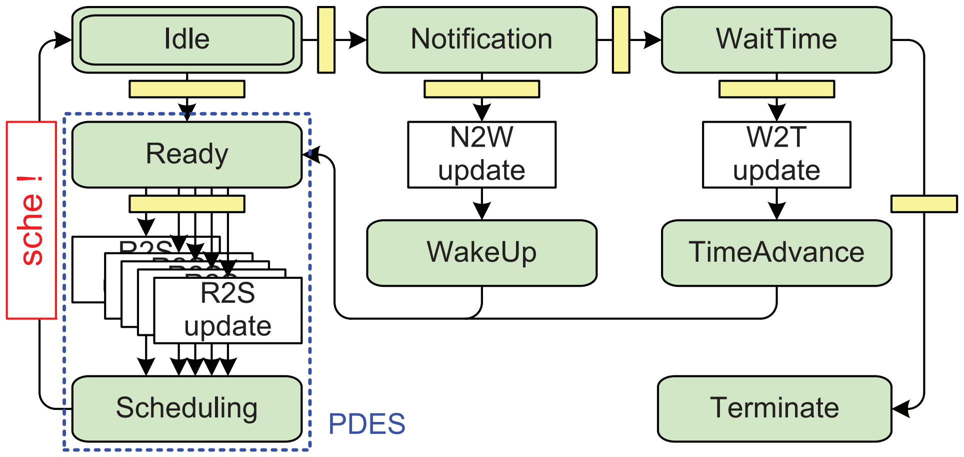


Figure 4.12: Scheduler automaton with delta and time advance cycles

The instance activation contains the loop from [Idle] to [Ready] and [Scheduling], and then back to [Idle]. Transition [Idle]→[Ready] is enabled when there is any asserted ready flag in the status tree. The enable flags of all instances with asserted ready flags are set in transition [Ready]→[Scheduling], and the all instances with asserted enable flag are activated by the synchronisation label in transition [Scheduling]→[Idle].

The event delivery includes the path from [Idle] to [Ready] via [Notification] and [WakeUp]. This part simulates the delta cycle increment in the DES. Transition [Idle]→[Notification] is enabled when all instances are suspended. The guard in transition [Notification]→[WakeUp] checks if there is any asserted notify flag, and the update label sets the ready flags of the suspended instances waiting for the same event. For example, the labels below are annotated to the transition to wake up instance A and B in Figure 4.6 from suspension.

[guard] `Main.notify_e` == `1`

[update] `Main.A.ready` = (`Main.A.wait_e1` == `1`)? `1` : `0`,

`Main.B.ready` = (`Main.B.wait_e2` == `1`)? `1` : `0`

The time advancement is the path from [Idle] to [Ready] via [WaitTime] and [TimeAdvance]. This part simulates the simulation time advancement in the DES. Transition [Notification]→[WaitTime] is enabled when there is no asserted notify flag in the status tree. The guard in transition [WaitTime]→[TimeAdvance] reads the minimal waiting time *min_clk* from the sorted queue and advance the time by *min_clk*. If min_clk is 0, i.e., there is no instance waiting for time advancement, the transition to [Terminate] is enabled and the scheduler process can end. If the minimal time value is greater than 0, the update label in transition [WaitTime]→[TimeAdvance] set the ready flag of the suspended instance if its wtime flag matches min_clk. The following labels are annotated to transition [WaitTime]→[TimeAdvance] to wake up instance A and B from suspension in Figure 4.6.

[guard] `min_clk` > `0`

[update] `Main.A.ready` = (`Main.A.wtime` == `min_clk`)? `1` : `0`,

`Main.B.ready` = (`Main.B.wtime` == `min_clk`)? `1` : `0`

min_clk will then be subtracted from all wtime flags greater than 0 in the status tree as well as from all waiting times in the sorted queue.

4.3.6 UPPAAL System Description for a PDES Model

After automaton templates for behaviors and central scheduler are defined, the last step is to instantiate the defined templates in the system description to build the model. As we described before, each instance in the design is one-to-one mapped to a instance process in the system description. Our approach flattens the hierarchy of the system, and we rely on the synchronization channels between scheduler and instance processes as well as the channels between parent and child instances processes to coordinate the transitions in these concurrent processes and simulate the execution semantics defined in SpecC language.

Here we use the introductory example in Figure 4.6 to demonstrate the generated system description. An UPPAAL system model illustrated in Figure 4.13 is generated for the introductory example. For this example, templates for behavior Main, BhvrA, and BhvrB are instantiated to build the system model. As we described above, a Scheduler process is created to coordinate state transactions in processes for behavior instances and a structure [status_tree] is created according to the hierarchy of the model to store the status of processes. Note that the labels on the transition and communication between processes are not shown in the figure for simplicity.

4.4 Queries for May-Happen-in-Parallel Analysis

In this section we introduce our idea of analyzing a MHP pair of statements by asking the model checker whether a corresponding query is satisfiable, as well as how the query generator creates a set of queries for MHP analysis.

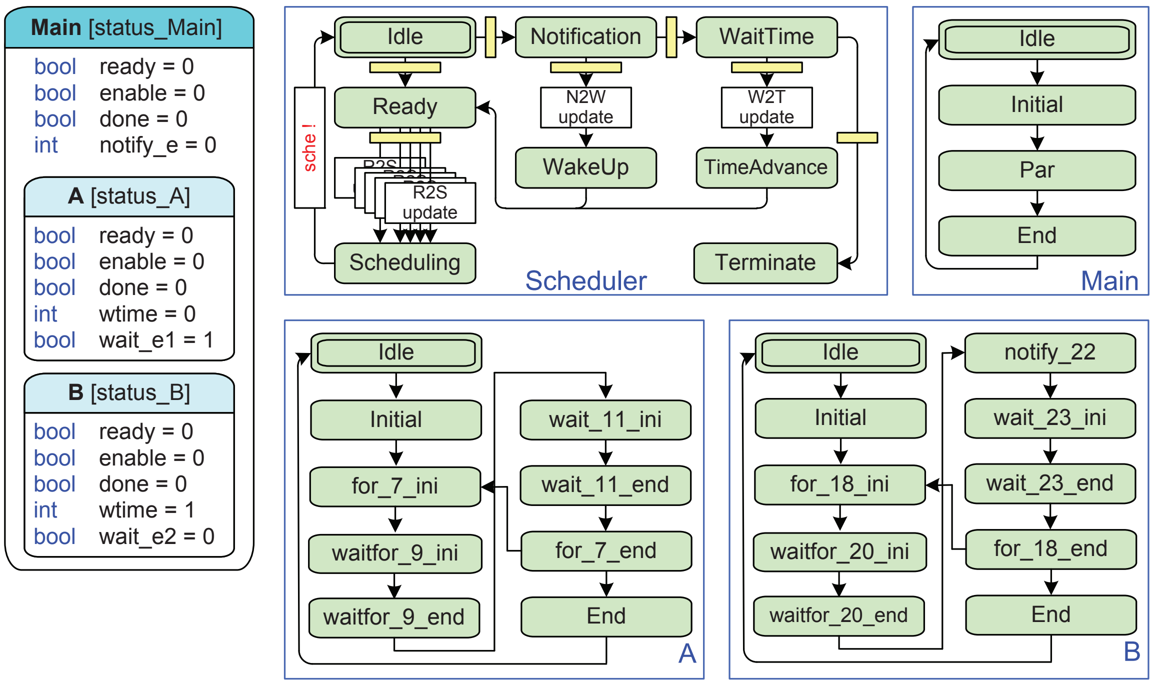


Figure 4.13: UPPAAL system description for the introductory example

4.4.1 Query in UPPAAL Model Checker

In the UPPAAL verifier, a query is described in the UPPAAL requirement specification language which supports five types of properties, namely

Possibly (`E<>`)

Invariantly (`A[]`)

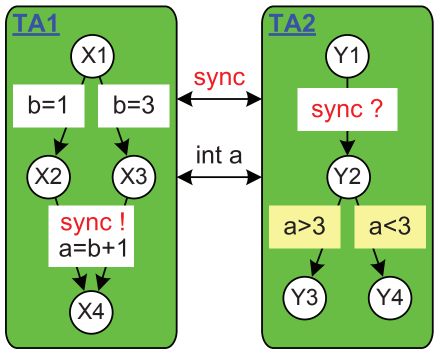
Potentially always (`E[]`)

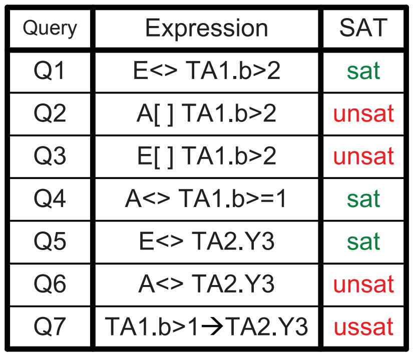
Eventually (`A<>`) and

Lead to (`-->`).

Here we use an example shown in Figure 4.14 to demonstrate how the query can be asked and their satisfiability.

The *Possibly* property `E<>p` tests if there is a reachable state where property `p` is satisfied. Query Q1 in Figure 4.14 asks if variable `b` in process `TA1` can be greater than 2 in this model. The result is satisfiable since transition `X1` to `X3` in `TA1` can set





Query	Expression	SAT
Q1	E<> TA1.b>2	sat
Q2	A[ ] TA1.b>2	unsat
Q3	E[ ] TA1.b>2	unsat
Q4	A<> TA1.b>=1	sat
Q5	E<> TA2.Y3	sat
Q6	A<> TA2.Y3	unsat
Q7	TA1.b>1→TA2.Y3	ussat

Figure 4.14: Queries for the model and their satisfiability

`b` to 2.

The *Invariantly* property `A[]p` tests if every reachable state satisfies property `p`. Query Q2 asks if `b` in `TA1` is always greater than 2, and the result is unsatisfiable since `b` is 1 in X2.

The *Potentially always* property `E[]p` tests if there is a sequence of transitions in which all states satisfy `p`. Query Q3 asks if there is a sequence of transitions where `b` in `TA1` is greater than 2 in all states. This properties is unsatisfiable because `b` is zero in location X1 (assuming `b` is not initialized). However, the result is satisfiable if the initial value of `b` is greater than 2. In this case, any sequence of transitions containing transition X1$\rightarrow$X3 satisfies this properties.

The *Eventually* property `A<>p` tests if all possible sequences of transitions eventually reach a state satisfy `p`. Query Q4 asks if `b` in `TA1` is eventually greater than or equal to 1. The queries is satisfiable since for all possible sequences of transition `b` will eventually be either 1 or 3.

Finally, the *leadto* property `p --> q` can be expressed as the property `A[] (p imply A<> q)`. Note that in the example query Q7 is unsatisfiable because TA1.b>1 only guarantee TA2 cannot be at location Y4, but not guarantee TA2 will always be at location Y3.

In the UPPAAL requirement specification language, it is possible to test whether or not a certain process is at a given location with the query of the form [`process.location`]. Also, the user can use the expression as below to verify if certain processes are at certain locations at the same time. For example, query of the form [`process1.location1 and process1.location1`] can test whether a process1 is at location1 and process2 is at location2 at the same time.

4.4.2 Queries for MHP Analysis

With the basic concept of the UPPAAL requirement specification language, the next step is to decide what the state properties and temporal properties should be for the MHP analysis. For two given statements Stmnt1 and Stmnt2 in instances Inst1 and Inst2 respectively, the following query will be created for MHP analysis:

```
E <> Proc_Inst1.Loc_Stmnt1 and Proc_Inst2.Loc_Stmnt2
```

In our approach, since the query is "whether or not two statements can possibly happen in parallel", we use the *Possibly* property `E<>p` in the expression to test if there is a reachable state where property `p` is satisfied. As for the state property, since in the model generator we have mapped instances into processes, statements into locations and execution of statements into transitions, *Proc_Inst1.Loc_Stmnt1* and *Proc_Inst2.Loc_Stmnt2* are used to represent Stmnt1 in instance Inst1 and Stmnt2 in instance Inst2 respectively. Finally, an `and` is used to state "happen in parallel" in the query.

The most intuitive approach to generate a set of queries for MHP analysis for any two given statements is to generate a query for each possible combination of statement pair. This approach certainly will do the work, but the number of queries may be tedious even for a simple model. To reduce the number of queries, we apply three approaches to generate a compact set of queries for MHP analysis.

First, instead of generating query for all possible statement pairs, we only identify the suitable scheduling points that may happen in parallel. To analyze the May-Happen-in-Parallel properties without exhaustively generate query for all possible combinations of statement pairs in the design, we need to identify suitable points that can represent the simulation times and delta cycles for a group of statements, and here we use the concept of scheduling point.

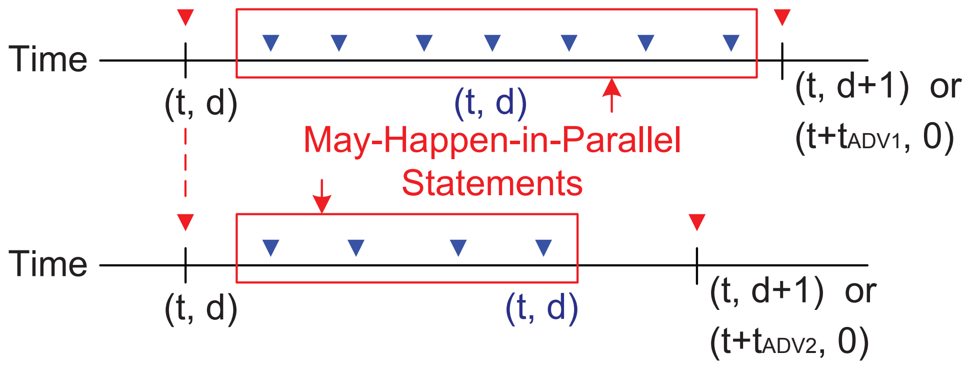
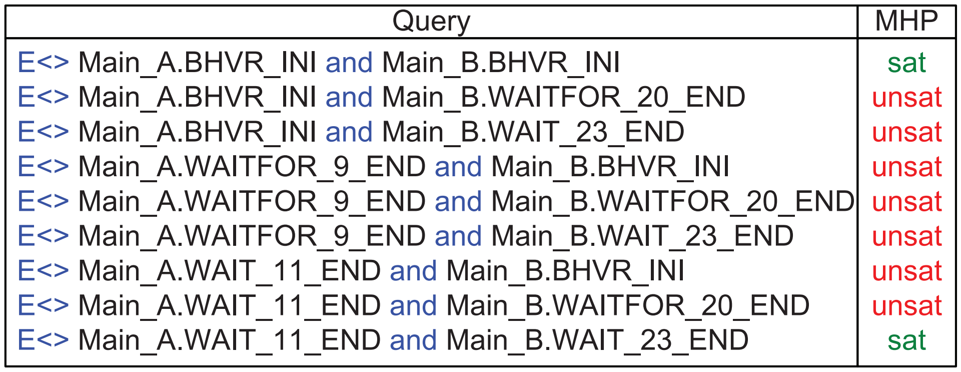


Figure 4.15: Scheduling points

Figure 4.15 illustrates the concept of scheduling points and how they can be used in the analysis to reduce the number of queries. This illustration shows two timelines of two concurrent processes respectively, and each point pointed by an arrow in the figure represents the timing stamp for a statement in the design. Statements in the model are further divided into two categories: statements which can affect the simulation time and delta cycle(pointed by red arrows) and statements that cannot(pointed by blue arrows). Points pointed by red arrows are called scheduling points in our approach.

The scheduling points mark the moments when instances are activated or woken by the scheduler. According to the semantics, the statements between two scheduling points share the same simulation clock and delta cycle. Take Figure 4.15 as the example. Between scheduling points, there can be any number of statements which does not affect the simulation time and delta cycle, i.e., statements other than `waitfor` and `wait`. These statements (pointed by blue arrows) share the same simulation time and delta cycle with the preceding scheduling point.

The concept here is that if we prove two scheduling points in two concurrent processes share the same simulation and delta cycle (`(t,d)` in this example), then according to the execution semantics these two groups of statements are supposed to be executed concurrent. In this case, any two statements from these two groups and one from each separately are MHP pair of statements. Therefore, by checking the MHP pairs of these scheduling points, we actually check the MHP pairs of all statements in the design. The scheduling points in our model include location [Initial] of all instances, and location [end] of wait and waitfor statements.



Query	MHP
E<> Main_A.BHVR_INI and Main_B.BHVR_INI	sat
E<> Main_A.BHVR_INI and Main_B.WAITFOR_20_END	unsat
E<> Main_A.BHVR_INI and Main_B.WAIT_23_END	unsat
E<> Main_A.WAITFOR_9_END and Main_B.BHVR_INI	unsat
E<> Main_A.WAITFOR_9_END and Main_B.WAITFOR_20_END	unsat
E<> Main_A.WAITFOR_9_END and Main_B.WAIT_23_END	unsat
E<> Main_A.WAIT_11_END and Main_B.BHVR_INI	unsat
E<> Main_A.WAIT_11_END and Main_B.WAITFOR_20_END	unsat
E<> Main_A.WAIT_11_END and Main_B.WAIT_23_END	sat

Figure 4.16: Queries for Figure 4.6 example for MHP analysis

The second approach is that we only generate queries for leaf instances instead of all instance. The reason is the computation and communication statements only exist in leaf instances. Since at this point the main purpose of our MHP analysis is to analyze the concurrent computation, we only need the queries for leaf instances. Note that in order to simplify the analysis, we take the end locations of the channel function calls as scheduling points instead of generating queries for the wait statements inlined for the communication method.

The final approach is to use static analysis to rule out statement pairs which are executed sequentially for sure and generate queries for pairs that may happen in parallel. The approach includes two steps. The first step is to generate all possible MHP pairs of instances. In this step all combinations of any two leaf instances are generated, except combinations where two instances share the same parent with sequential or FSM composition in their hierarchy, since with these composition the execution of the child instances cannot overlap.
The second step is to generate queries for all combinations of the scheduling points in each MHP pair of instances. Take the introductory design as the example. The red arrows in Figure 4.6(A) mark the scheduling points in both leaf instances. Since instance A and B do not have the same parent with sequential or FSM composition, instance A and B are a MHP pair. Figure 4.16 shows the queries generated by our tool as well as their satisfiability. According to the result, the statement set at lines 6∼8 and statement set at lines 17∼19 are MHP statements. Also statements at line 12 or 8 and statement at line 24 or 19 are MHP statements as well.

In the end, for any two leaf instance processes which are potentially activated by the scheduler at the same time, the query generator create a set of queries for all combinations of scheduling points in these two processes, and let UPPAAL model checker verify the satisfiability of these queries.

4.5 Model Optimization

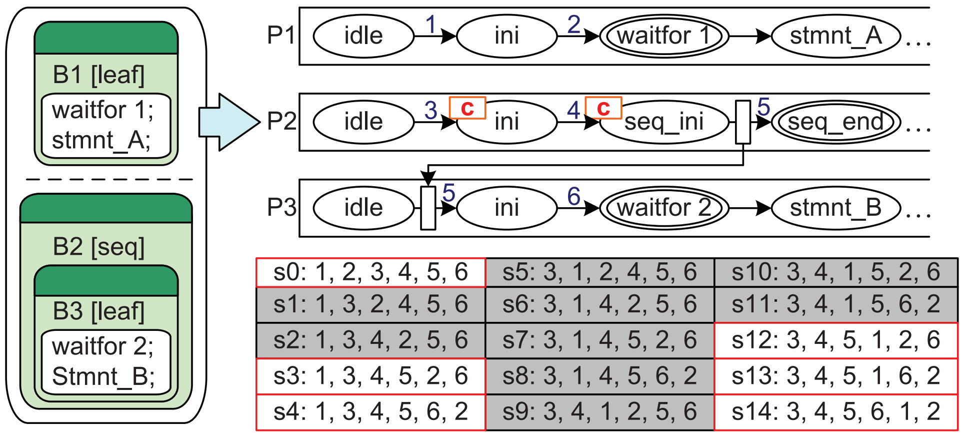
In this paper, we apply two main methods from different aspects to shorten the run time of MHP analysis. The first one is to generate a compact set of queries to shorten the analysis time, which has been described in the previous section. Another method is to trim the redundant search space in the model so that the solver can still give the identical result with less search time.

In order to analyze the MHP statements in the design, our model supports PDES and activates as many instances as possible in parallel. The downside of this method is that for most of the steps in the trace there are multiple enabled transitions and therefore the search space is much larger than regular DES. We assume the UPPAAL model checker will try all possible interleaving between enabled transitions in processes to prove the satisfiability of the given query – especially when the query itself turns out to be unsatisfiable.
Based on this assumption, we developed an approach to trim the search space in the model by removing redundant interleaving, i.e., interleaving leading to the same states and does not affect the analysis result. In the optimized model, a great portion of redundant interleaving is removed and interleaving modeling the concurrency in the system design are still kept. The experimental results match our assumption as the analysis time of the optimized model is shortened and the result is identical to the original model.

The concept of this method is to give certain transitions in processes higher priority than others and use the priority to prevent redundant interleaving from happening. The priority of a transition in the UPPAAL model can be determined by setting the priority of the location involving the transitions. For example, for two enabled transitions X1→X2 and Y1→Y2, transition X1→X2 will always happen before transition

Y1→Y2 if X1 has higher priority than Y1.

In UPPAAL model, different priority can be given to a location by specifying the type of the location. There are three types of locations supported in UPPAAL: *committed*, *urgent*, and regular. Among these three types of locations, locations of committed type have the highest priority. If any automaton is in a committed location, the next transition must depart from one of the committed locations. Transitions departing from locations of urgent and regular type can only take place when there is no automaton in a committed location, i.e., we can block part of the interleaving and then reduce the search space.



s0: 1, 2, 3, 4, 5, 6	s5: 3, 1, 2, 4, 5, 6	s10: 3, 4, 1, 5, 2, 6
s1: 1, 3, 2, 4, 5, 6	s6: 3, 1, 4, 2, 5, 6	s11: 3, 4, 1, 5, 6, 2
s2: 1, 3, 4, 2, 5, 6	s7: 3, 1, 4, 5, 2, 6	s12: 3, 4, 5, 1, 2, 6
s3: 1, 3, 4, 5, 2, 6	s8: 3, 1, 4, 5, 6, 2	s13: 3, 4, 5, 1, 6, 2
s4: 1, 3, 4, 5, 6, 2	s9: 3, 4, 1, 2, 5, 6	s14: 3, 4, 5, 6, 1, 2

Figure 4.17: Optimization with location prioritization

Let's use a UPPAAL model of a system with three instances in Figure 4.17 to illustrate how the prioritized locations can trim the search space. In this example, instance B1 and B2 are executed concurrently and B3 is a child instance of B2. Processes P1, P2, and P3 are created for these three instances in the UPPAAL model. If now the query is for MHP analysis as we defined above , the solver shall try all possible transitions listed in Figure 4.17 to determine the satisfiability since this query is not satisfiable (in fact, stmnt_A and stmnt_B are executed at different simulation time). Figure 4.17 lists all 15 possible sequences of transitions before P1 wakes up and moves into location for stmnt_A. Here, we give P2.ini and P2.seq_ini the *committed* priority to reduce the number of possible sequences. After the location prioritization, transition 4 and 5 must occur right after transition 3, and thus the number of possible sequences is reduced from 15 to 6 (s0, s3, s4, s12, s13, s14) while the interleaving properties between the transitions in the bodies of leaf instances (transition 2 and 6) are still intact.

In previous section, we mentioned that our approach will only generate queries for statements of interest in leaf instances because time advancement, event synchronization, and channel communication can only happen in the leaf behavior. Based on this rule, we trim the search space in our model by removing transition interleaving between processes for hierarchical instances and leaf instances but still keeping the possible interleaving between leaf instances to model the concurrent composition.

In the optimized model, we assign the following locations with the *committed* priority: [Ready] and [Scheduling] in the scheduler automaton, [Initial] and [End] of processes for hierarchical instances, and [seq_ini] and [fsm_ini]. Note that the prioritization trims the search space significantly without violating the execution semantics. The remaining possible sequences still keep the concurrency between transitions in the bodies of leaf instances (for example, transition 2 and 6 in Figure 4.17).
Our experimental results show that our assumption is valid. Table 4.1 in Section 4.6 shows one example demonstrating the analysis run time for the model before optimization and after. The analysis results for both models are identical, but the difference in runtime and memory requirements is tremendous.

4.6 Experiments and Results

We now show experimental results of running MHP analysis on the introductory example and three in-house models of embedded applications, a grey-scale JPEG encoder[70], color JPEG encoder, and MP3 decoder[71]. The generation of the models and queries is very quick. The analysis, however, takes time to verify the satisfiability of the queries.

Table 4.1: Run time and memory requirement for optimized model

Optimization (JPEG)	# of queries	# of mhp pairs	total runtime	memory req.
Before	143	51	1h:44m:41s	310MB
After	143	51	42s	26MB

Table 4.2: MHP Analysis of SLDL Design Using UPPAAL Model Checker

Application	lines of codes	# of queries	# of mhp pairs	total runtime
Intro	39	9	2	<1s
Intro-M	39	9	-	∞
Intro-M*	39	9	2*	3s*
Mono-JPEG	1.5k	143	51	42s
Color-JPEG	2.5k	210	25	16m:18s
MP3 Decoder	7k	141	24	21h:32m:36s

Table 4.1 shows the comparison before and after the search space optimization described in Section 4.5. Here we use our JPEG encoder model as the example and list the runtime and memory requirement before the optimization and after. Note that the memory requirements listed here are obtained by running the verification on an unsatisfiable query since for an unsatisfiable query the solver needs to explore all search space to disprove the satisfiability. In this table, we can see that for the same model and query, the optimized model takes significantly less computation resource to obtain the identical result.

Table 4.2 first shows the result for the introductory example of Figure 4.6. The verification takes less than one second and reports two out of nine MHP pairs of scheduling points are true. Compared to [65] in which four out of nine are reported true, our approach is more precise. Intro-M is a modified introductory example where the loop is replaced by a while loop. For a design with unbounded loop transitions, the verification tool keeps expanding the search space and tries to find a trace to satisfy the query. For satisfiable properties like the first and last query in Figure 4.16 the verification is still quick, but for unsatisfiable properties the verification will not terminate. To deal with this situation, our tool identifies the while and do-while loop, and insert a guard label as the upperbound in the transition created for the loop statement. In our model, an upperbound for a while or do-while loop is added to the model as an argument so that the designer can easily specify the upperbound in the system description. Note that the model checker assigns zero to all uninitialized integer variables and arguments, and the labels we generate for loop statements identifies the case where "upperbound=0" and allows the loop running unboundedly.

As we mentioned in the previous paragraph, the search space can keep expanding if there is an unbounded loop in the model. Instead of giving loops an upperbound, an alternative approach is to use the under approximation option provided by the verifier. According to the information provided by UP4ALL Inc., the under approximations use bit-state hashing to represent the state space. Users can also choose the size of hash table to adjust the degree of approximation, i.e., the size of the table results in the size of the state space being searched. The result of this option is an approximate answer. The results listed in Intro-M* row are obtained with this approximation option. Instead of keeping searching until running out of memory, the verifier replies `"MAY NOT be satisfied"` for unsatisfiable queries.

The fourth and fifth experiment are MHP analysis of greyscale and color JPEG encoder. We can see that the number of MHP queries for color JPEG is more than greyscale JPEG because there are more leaf instances and scheduling points in the color JPEG encoder. The true MHP pairs in the greyscale encoder, on the other hand, are more than the MHP pairs in the color encoder. The reason is that in the color encoder the communication between modules are implemented with double handshake channels, while the communication in greyscale encoder are implemented with queues. Given the medium size design, the analysis run time is acceptable.

The MP3 decoder is a large example with three times as many instances as the greyscale encoder (34 and 12 respectively). The left and right channel are decoded in parallel and each channel contains multiple instances with children below them. The search space is much larger than the JPEG encoder and it takes much more time to verify the satisfiability. Giving the complexity of formal verification, the run time of less than a day is still reasonable.

4.7 Extension to Timing and Power Consumption Analysis

In this chapter we propose an approach which converts system level designs into UPPAAL automaton models and make use of UPPAAL model checker to prove or disprove the satisfiability of queries asking whether two given statements can possibly be executed in parallel. The experimental results shows this approach works well and matches our expectation. Except for MHP analysis, we also found that this approach can be used for verification of other properties of interest. In this section, we present our concept of applying this apporach to timing delay and power constraint analysis.

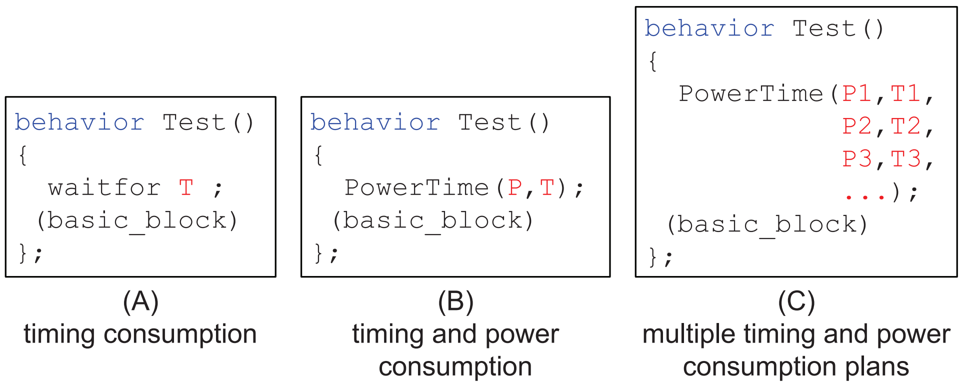
4.7.1 Timing Delay Analysis

Except for the functionality of the design, the timing delay of the design can also be captured in SLDL. In our model, we capture the time spent by a module to complete the computation using `waitfor T` statement. As the illustration in Figure 4.18(A), `waitfor T` statement is inserted at the beginning of the main method of a leaf instance as the annotation for timing delay, and the value of `T` represents the time required to finish the computation. In simulation-based verification, the discrete event simulator identifies time advancement statements, and activate or wakeup the idling or suspended behavior instances at the correct simulation time. After the simulation, the timing delay can be easily obtained by returning the simulation time advancement in the discrete event simulator.

Similar to the advantages and disadvantages we described before, simulation-based timing analysis is also input vector dependent. The simulation cycle time obtained during the simulation is influenced by the input vector. Unless we try all possible input vector to cover all conditional execution cases in the model, the captured cycle time cannot guarantee the simulation will finish within the timing constraints for all possible input vectors. Here we can use our UPPAAL automaton model to address this issue since the scheduler process in our UPPAAL system model simulates the behavior of the parallel discrete event simulator. In the scheduler process we also have a local integer *sim_clk* to keep track of the time advancement and make sure the scheduler process activate or wakeup behavior processes from suspension at correct time. To ask whether for all possible conditions the execution of the system model can finish within a time `T`, we can simply add a query as below:

```
E <> i_Scheduler.Terminate and i_Scheduler.sim_clk > T
```

Here `i.Scheduler` is the instance name of the scheduler process. If the query is satisfiable, the solver proves that there is at a possible condition in which the execution of the system model can be greater than timing constraint `T`; otherwise, the model is proved to meet the timing constraint since the solver exhaustively searches the state space but cannot find a state satisfying this query.



```
behavior Test()
{
  waitfor T ;
 (basic_block)
};
```

(A)
timing consumption

```
behavior Test()
{
  PowerTime(P,T);
 (basic_block)
};
```

(B)
timing and power
consumption

```
behavior Test()
{
  PowerTime(P1,T1,
            P2,T2,
            P3,T3,
            ...);
 (basic_block)
};
```

(C)
multiple timing and power
consumption plans

Figure 4.18: Power and timing annotation in the model

4.7.2 Timing Delay and Power Consumption Analysis

The concept of using `waitfor` to model the timing delay can also be applied to power consumption analysis. In [72], an approach based on virtual power meter is proposed. In this approach, instead of just inserting `waitfor` statements to represents the timing delay, the power consumption for each basic block is estimated with a computation profiler and then annotated to the model with a function call to a power API `PowerTime_Meter(P,T)` for each basic block, where P and T represents power and time required to complete the computation of the basic block respectively. The annotated ESL model is illustrated in Figure 4.18(B). In [72], the power API is used to keep track of the power and time spent in execution of the model, and after the simulation the trace can be visualized so that designers can have better understanding how power is dissipated over time. Here we take power APIs as statements providing information about power consumption and time advancement of the model, and generate corresponding locations, transitions, and label for it.

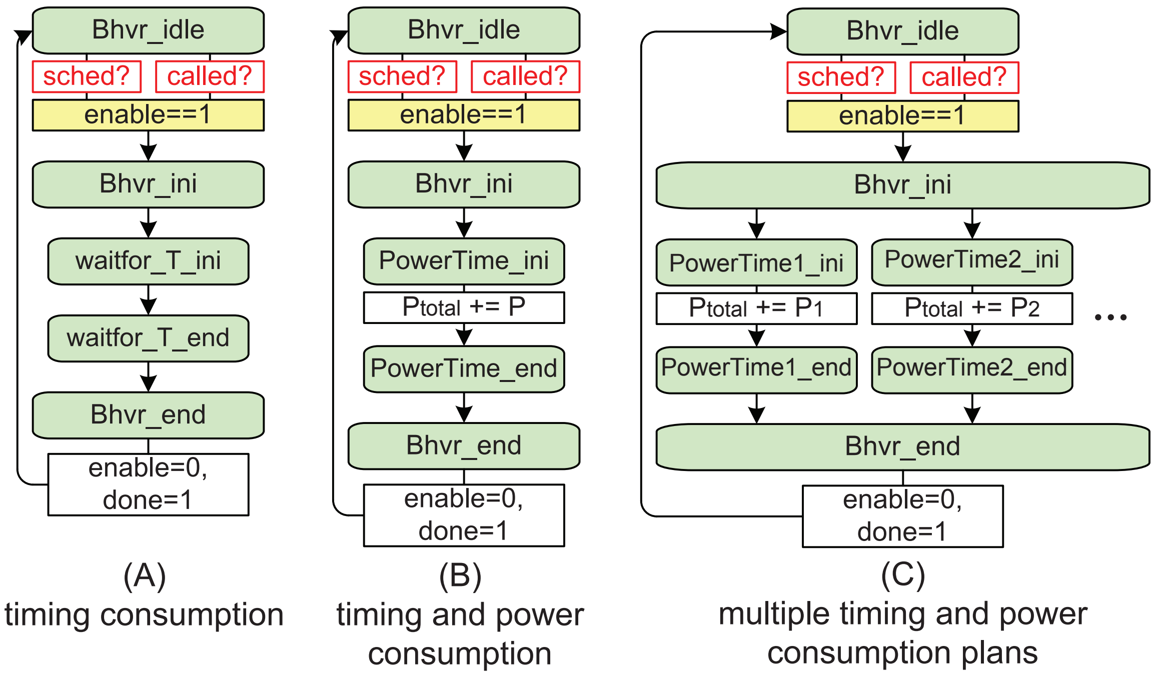


Figure 4.19: UPPAAL model for power and timing annotation

Figure 4.19(A) and (B) shows the generated automaton templates for Figure 4.18(A) and (B) respectively. As the illustration in Figure 4.19(B), the locations generated for power APIs are almost identical to the locations for `waitfor` statements. The difference here is the additional update label on the transition from the initial location for the statement to the end. We insert a global variable $\mathtt{P}_{total}$ to keep the accumulated power consumption, and it will be updated by the update label in the transition. With this modification, the total power consumption is taken into consideration when we verify the system design with the corresponding UPPAAL model. To ask whether for all possible conditions the execution of the system model can finish within a time $\mathtt{T}_{con}$ and power consumption $\mathtt{P}_{con}$, we can simply create a query as below:

$$\mathtt{E} <> \mathtt{i\_Scheduler.Terminate}\ \mathtt{and}\ (\mathtt{i\_Scheduler.sim\_clk} > \mathtt{T}_{\mathtt{con}}\ \mathtt{or}\ \mathtt{P}_{\mathtt{total}} > \mathtt{P}_{\mathtt{con}})$$

If the query is satisfiable, it is possible that the execution time or the power consumption can be greater than the specified constraints.

4.7.3 UPPAAL Modeling for Dynamic Voltage and Frequency Scaling

The research to reduce power consumption of a design has been broadly studied, and one technology in this realm is Dynamic Voltage and Frequency Scaling (DVFS). In DVFS, the speed of the circuit can be modified through adjusting the voltage and frequency. Designers can use this technology to balance the timing delay and power consumption and find the best combination which uses minimal power to complete the job on time. In a circuit supporting DVFS technology, a few set of power and time consuming plans are provided. To model this technology in our system level model, the power API is modified to take multiple sets of power and time as arguments. The modified ESL model is illustrated in Figure 4.18(C). In this illustration, the power API takes three sets of delay and power consumption {`P1,T1`}, {`P2,T2`}, and {`P2,T2`} to represent three consuming plans respectively. Conceptually the greater the power consumption, the faster the computation speed, i.e., T1 > T2 > T3 if P1 < P2 < P3. Here we also modify our UPPAAL model generator to support the DVFS technology with finite set of power/time consuming plans.

Figure 4.19(C) illustrate the modified automaton template. Instead of inserting a pair of locations for each power APIs, multiple arcs representing different options are created to model the specified power and time consuming plans. With the modified model, we can use the model checker to find out the feasible plan to meet the time and power constraints. To ask whether it is possible that the execution of the system model can finish within a time $\mathtt{T}_{con}$ and power consumption $\mathtt{P}_{con}$, we can simply create a query as below:

$$\texttt{E <> i\_Scheduler.Terminate and (i\_Scheduler.sim\_clk} < \texttt{T}_{\texttt{con}} \texttt{ and P}_{\texttt{total}} < \texttt{P}_{\texttt{con}})$$

If the query is satisfiable, it means there is a feasible plan to meet constraints $\texttt{T}_{con}$ and $\texttt{P}_{con}$. As we described before, the model checker can generate a trace for a satisfiable case. With this trace, designers can replay state transitions and find out at what time the behavior process should take which path to satisfy the query, i.e., find out when the processing element should use more power to complete the computation fast and when to reduce the speed to save power.

4.8 Summary

In this chapter, we have described a new approach to identify the MHP statements in a system. Our approach includes the abstraction of the automata network from a design and the generation of queries for MHP analysis. The satisfiability of MHP queries is verified using UPPAAL model checker, and we demonstrated our method with an introductory and three models of embedded application. Compared to state-of-the-art other work [65], our results are more precise, but take longer to compute. We also propose a concept of using UPPAAL model checker to verify the satisfiability of timing and power consumption constraints. This concept is further evolved into an approach using model checker to find out the feasible DVFS plan to satisfy the giving constraints.

In terms of future work, first we plan to shorten the analysis time by introducing more static analysis and generating a more compact set of queries as well as further exploit the priorities of locations to trim the search space. As we demonstrate in the experimental results, the state space reduction greatly improves the run time as well as resource requirement, and we think there are still lots of room to improve the reduction technique. Second, we will keep explore the possibility of using this model to verify other properties of interest. Here we have proposed a concept of verifying the total consumption of time and power, and we believe this model can be used to verify more properties in both time and power domain.

Chapter 5

Conclusion

5.1 Contributions

In this chapter, we summarize our contributions to formal analysis of ESL models. There are three specific contributions, namely
1) formal deadlock detection,
2) formal timing analysis, and
3) May-Happen-in-Parallel analysis.

5.1.1 Formal Deadlock Detection using SMT

In chapter 2 [35], we have proposed an approach to formally detect potential deadlocks using Satisfiability Modulo Theories solver. Our method first abstracts the system-level model into a time interval model composed of assertions in SMT-LIB2 language, and then use the SMT solver to prove or disprove the existence of conflicts in the model. This approach mostly deals with the design at specification level, and it detects the deadlock caused by the improper execution order and communication assignment. If the model is unsatisfiable, i.e., deadlock is detected, based on the indices reported by the solver which indicate assertions leading to the conflict, our tool generates a brief report stating the structural hierarchy and channel communication causing the deadlock. This work enables the detection of deadlock caused by the improper combination of execution order and communication and therefore improves the stability of embedded systems. Above of that, our approach can also give designers good hints of which parts of design causing the deadlock by reporting the indices of the assertions causing the timing conflict and generating a error report.

5.1.2 Formal Timing Analysis using SMT

In Chapter 3 [48], we further extended the approach in Chapter 2 so that we can support the analysis of properties of interest in time domain. The improved approach can not only identify the composition in hierarchical behaviors and pre-defined channels, but also support the modeling of statements for conditional execution, time advancement and event synchronization in leaf behaviors. Furthermore, this approach also supports the modeling of customized channels. After the time interval model is extracted from the system-level design, designers can augment the model with assertions representing real-world use cases and the properties of interest and then use SMT solver to verify the satisfiability of the augmented assertions.

We use this approach to analyze the liveness and timing constraint properties of two transaction level models in which processing elements communicate with each other over AMBA AHB protocol and CAN bus protocol. The experimental results shows that this work is capable of analyzing the satisfiability of timing constraints under various assumptions within reasonable analysis time. This work also shares the same advantage of the previous approach, which is capable of reporting useful information back to designers for both satisfiable and unsatisfiable cases. By integrating this work into the design loop, we can identify an application will finish on time or not, which is a critical concern in real-time system.

5.1.3 May-Happen-in-Parallel Analysis using UPPAAL Model Checking

In Chapter 4[57] [58], we have proposed a method to identify May-Happen-in-Parallel statements in the system model. May-Happen-in-Parallel analysis can answer the questions for two given statements in the design, whether or not they can be executed in parallel. It is important because processing elements running in parallel in the system-level design can lead to race conditions caused by parallel accesses to shared variables. To identify this issue, we need to identify the statements which can be executed in parallel. Our approach here is to use our model generator to convert the system-level design into an UPPAAL system model which simulates the behavior or parallel discrete event simulation on the target design. Also, queries asking whether or not the given two statement can possibly be executed in parallel are generated automatically, and our query generator creates a set of queries for all possible combinations of any two statements. The model and queries are then analyzed by UPPAAL model checker to identify the satisfiability of each query.

Except for the model and query generator, we also proposed an approach to optimize the model and trim the state space. With the optimization, we found that the run time and memory requirements are reduced significantly. This work enables precise detection of MHP statements. Compared with a state-of-the-art MHP analysis approach proposed in [65], this work shows the experimental result is more precise. Also, since a query is satisfiable only when the corresponding MHP property is true, the UPPAAL model checker can generate a trace to show how the query is satisfiable, i.e., show under what conditions the two given statements can happen in parallel. Another benefit of this approach is that the automata model generated in the work can also be used to analyze other properties, such as timing delay and power consumption. With this approach, we can identify the potential race condition precisely, and therefore improve the stability of the system.

5.2 Future Work

5.2.1 SMT Modeling for System-level Design

The approach of using SMT modeling for deadlock detection and timing constraint analysis certainly can do the work, but we also encountered some inconvenience with this approach and that could be the possible improvement direction in the future. First, some test cases we run the analysis did not finish in two hours (and probably will take more since we terminated the analysis), and the situation will be even worse if the design is more complex. For this reason, scalability of the analysis using SMT solver is definitely a direction worth the consideration.

In Section 3.3.4, we have proposed an idea of hierarchical timing analysis which provides a certain degree of scalability to this approach, but this approach is limited to the timing information specified with do-timing constructs by the designer. We think further improvement such as to decompose the analysis to multiple levels and make use of intermediate results provided by the solver in analysis of higher hierarchical levels is possible.

Another direction for future development is to improve the user interface. One difficulty we faced in this approach is that even though the Z3 SMT solver can report the indices of assertions causing the conflict when potential deadlock is detected, it is still hard to identify where the problem is when the structure of the system and the communication between modules are complicated. A future improvement to address this issue would be visualizing the structure of the system and displaying the conflicting timing relations in the structure illustration. We believe it will greatly reduce the effort designers need to make to identify the issue in the system.

5.2.2 UPPAAL Modeling for System-level Design

As for the system modeling using UPPAAL automaton, the future work can be discussed from two aspects. The first aspect is the possible application with this UPPAAL model for system-level design. In Chapter 4, we have proposed an application which identifies MHP statements in the system with our generated UPPAAL model for the system, and we believe it can be use to analyze the constraints in power domain, such as the concept we brought out in Section 4.7 which uses this model for verifying the timing and power constraints as well as finding feasible plans with dynamic voltage and frequency scaling. Except for the analysis of the properties in power and time domain, we also think it can be used for analysis of other properties of interest.

For example another possible application we come up with is to use this model to find out the maximum parallelism in the model. Since the model itself simulates the behavior of the execution in a system-level model, we can ask many kind of questions to it as long as the question can be described as a query in requirement specification language.

Another aspect of future work here is the optimization of the model and further reduce the run time and resource requirement. With more static analysis to the system level design and to the query we want the solver to prove/disprove, the UPPAAL model we proposed in this dissertation can be further optimized to reduce the state space.

Bibliography

[1] D. Gajski and R. Kuhn "*Guest Editors Introduction: New VLSITools,*" IEEE Computer, December, 1983.

[2] W. Chen, X .Han, C. Chang, G. Liu, and R. Döomer, "*Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models,*" IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.33, no.12, pp. 1859-1872, December, 2014

[3] L. Cai, A. Gerstlauer, S. Abdi, J. Peng, D. Shin, H. Yu, R. Dömer, D. Gajski "*System-On-Chip Environment Manual*", Center for Embedded Computer Systems, Technical Report 03-45, December 2003

[4] "*International Semiconductor Industry Association. International Technology Roadmap for Semiconductors (ITRS)*", http://www.itrs.net, 2004

[5] M. Fujita, I. Ghosh, and M. Prasad, "*Verification Techniques for System-Level Design*". Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007

[6] J. Staunstrup, and W. Wolf "*Hardware/Software Co-Design: Principles and Practice*", Springer-Verlag US, 1997

[7] G. D. Micheli, and R. K. Gupta "*Hardware/Software Co-Design*", IEEE MICRO, vol. 85, pp 349-365, 1997

[8] A. Sangiovanni-Vincentelli, H. Zeng, M. Di Natale, and P. Marwedel, "*Embedded Systems Development: From Functional Models to Implementations*", Springer New York, 2013

[9] Grant Martin, Brian Bailey, and Andrew Piziali, "*ESL Design and Verification: A Prescription for Electronic System Level Methodology*". Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 2007.

[10] A. Gerstlauer, R. Dömer, J. Peng, D. Gajski, "*System Design: A Practical Guide with SpecC*", Kluwer Academic Publishers, 2001.

[11] D. Gajski, S. Abdi, A. Gerstlauer, G. Schirner "*Embedded System Design: Modeling, Synthesis, and Verification*", Springer Publishing Company, Incorporated, 2009

[12] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, "SpecC: Specification Language and Methodology" Kluwer Academic Publisher, Boston, March, 2000

[13] W. K. Lam. "*Hardware Design Verification: Simulation and Formal Method-Based Approaches"* Prentice Hall PTR, Upper Saddle River, NJ, USA. 2005.

[14] I. Viskic, R. Dömer, "*A Flexible, Syntax Independent Representation (SIR) for System Level Design Models*", Proceedings of EuroMicro Conference on Digital System Design, Dubrovnik, Croatia, August 2006.

[15] "*SystemC 2.3.1 (Includes TLM)*" http://www.accellera.org/downloads/standards/systemc

[16] D. C. Black, and J. Donovan, B. Bunton SystemC: From the Ground Up, Second Edition Springer Verlag Gmbh, 2014

[17] E. M. Clarke and J. M. Wing "*Formal methods: state of the art and future directions*", ACM Comput. Surv. 28, 4 (December 1996)

[18] J. B. Almeida "*Rigorous Software Development*" Springer-Verlag London, 2011

[19] C. Kern and M. R. Greenstreet "*Formal Verification in Hardware Desgin: A Survey*", CM Trans. Des. Autom. Electron. Syst. 4, 2 April 1999, 123-193.

[20] V. D'Silva, D. Kroening, and G. Weissenbacher, "*A Survey of Automated Techniques for Formal Software Verification*" IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.7, pp. 1165–1178

[21] A. J. Hu "*Formal Hardware Verification with BDDs: An Introduction*", IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM), pp. 677-682, 1997

[22] A. Koelbl, Y. Lu, and A. Mathur, "*Formal equivalence checking between system-level models and RTL*" IEEE/ACM International Conference on Computer-Aided Design, 2005. ICCAD-2005

[23] J. P. Marques-Silva , K. A. Sakallah "*Boolean satisfiability in electronic design automation*" Design Automation Conference, 2000. pp675-680

[24] E. M. Clarke, O. Grumberg and D. Peled "*Model Checking*", MIT Press, Cambridge, MA, USA. 2000

[25] R. Jhala and R. Majumdar "*Software model checking*" ACM Comput. Surv. 41, 4, Article 21, October 2009

[26] S. Edelkamp , S. Leue , A. Lluch-Lafuente "*Directed explicit-state model checking in the validation of communication protocols*", International Journal on Software Tools for Technology (STTT), 2004. volume 5, pages 247-267

[27] "*Synopsys Formality*", `http://www.synopsys.com/Tools/Verification/FormalEquivalence/Pages/default.aspx`

[28] "*Synopsys Formality*", `http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/HECTOR.aspx`

[29] "*Cadence Encounter Conformal Equivalence Checker*", `http://www.cadence.com/products/ld/equivalence_checker/pages/default.aspx`

[30] "*The Yices SMT Solver*" `http://yices.csl.sri.com/`

[31] "*CVC4 the SMT solver*", `http://cvc4.cs.nyu.edu/web/`

[32] Conchon, S., Contejean, E., Kanig, J "*Alt ERGO SMT Solver*" `http://alt-ergo.lri.fr/`

[33] M. Garey and D. Johnson. "*Computers and Intractability: A Guide to the Theory of NP-Completeness*", W. H. Freeman, 1979

[34] Luca Aceto, "*Is your model checker on time? On the complexity of model checking for timed modal logics*" In Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science (MFCS '99), Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki (Eds.). Springer-Verlag, London, UK, UK, 125-136

[35] C. Chang, R. Dömer, "*Formal Deadlock Analysis of SpecC Models Using Satisfiability Modulo Theories*", Proceedings of the International Embedded Systems Symposium, Springer, Paderborn, Germany, June 2013

[36] R. Döomer, A. Gerstlauer, and D. Gajski, "*SpecC Language Reference Manual Version 2.0*" SpecC Technology Open Consortium, Japan, December 2002.

[37] M. Fujita, H. Nakamura. "*The Standard SpecC Language*" Proceedings of the International Symposium on System Synthesis, Montreal, October 2001.

[38] A. Habibi, H. Moinudeen, and S. Tahar. "*Generating Finite State Machines from SystemC*", In *Design, Automation and Test in Europe*, pages 76-81, 2006.

[39] A. Habibi and S. Tahar. "*An Approach for the Verification of SystemC Designs Using AsmL*", In *Automated Technology for Verification and Analysis*, pages 69-83, 2005.

[40] P. Herber, J. Fellmuth, and S. Glesner "*Model Checking SystemC Designs Using Timed Automata*", In *Int. Conf. on HW/SW Codesign and System Synthesis.* ACM, press, 2008.

[41] P. Herber, M. Pockrandt, and S. Glesner "*Transforming SystemC Transaction Level Models into UPPAAL timed automata*", In *2011 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 161-170, 2011.

[42] D. Karlsson, P. Eles, and Z. Peng. "*Formal verification of SystemC Designs using a Petri-Net based Representation*", In *DATE*, pages 1228-1233, 2006.

[43] Chun-Nan Chou, Yen-Sheng Ho, Chiao Hsiehand Chung-Yang Huan "*Formal Deadlock Checking on High-Level SystemC Designs*" In *2010 2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 794-799, 2010.

[44] Chun-Nan Chou, Yen-Sheng Ho, Chiao Hsiehand, Chung-Yang Huan "*Symbolic Model Checking on SystemC Design*" In *Design Automation Conference (DAC) '12 Proceedings of the 49th Annual Design Automation Conference*, pages 327-333, 2012.

[45] C. Barrett, A. Stump, C. Tinelli "*The SMT-LIB Standard Version 2.0*", March 30, 2010

[46] "*Z3 theorem prover*" `http://z3.codeplex.com/`

[47] David R. Cok. "*The SMT-LIB v2 Language and Tools: A Tutorial*" `http://www.grammatech.com/resources/smt/SMTLIBTutorial.pdf`

[48] C. Chang and R. Dömer, "*Communication Protocol Analysis of Transaction-Level Models using Satisfiability Modulo Theories*", Accepted for publication at the Asia and South Pacific Design Automation Conference 2015, Tokyo, Japan, January 2015

[49] L. De Moura and N. Bjørner, "*Z3: An efficient smt solver*", in TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag.

[50] C. P. Gomes, H. Kautz, A. Sabharwal, B. Selman, "*Satisfiability solvers*" Handbook of Knowledge Representation 3 (2008): 89-134

[51] L. De Moura and N. Bjørner, "*atisfiability modulo theories: Introduction and applications*", in Commun. ACM, 54(9):69–77, Sept. 2011.

[52] T. Sakunkonchak, S. Komatsu, and M. Fujita, "*Synchronization verification in system-level design with ILP solvers*", Proceedings of Formal Methods and Models for Co-Design, pp.121,130, 11-14 July 2005

[53] Advanced RISC Machines Ltd., "*AMBA Specification (Rev. 2.0)*", ARM IHI 0011A

[54] Robert Bosch GmbH. "*CAN Specification, 2.0 edition*", 1991. http://www.can.bosch.com/

[55] G. Schirner and R. Dömer, "*Quantitative analysis of the speed/accuracy trade-off in transaction level modeling*", *ACM Transactions Embedded Computing Systems*, 8:4:1–4:29, Jan. 2009.

[56] G. Schirner and R. Dömer, "*Abstract Communication Modeling: A Case Study*

Using the CAN Automotive Bus", *Proceedings of the International Embedded Systems Symposium*, Springer, Manaus, Brazil, August 2005.

[57] C. Chang and R. Dömer, "*Abstracting ESL Designs to UPPAAL System Models*", Center for Embedded and Cyber-Physical Systems, Technical Report 14-13, November 2014

[58] C. Chang and R. Dömer, "*May-Happen-in-Parallel Analysis of ESL Models using UPPAAL Model Checking*", Accepted for publication at the Design, Automation and Test in Europe Conference 2015, Grenoble, France, March 2015.

[59] G. Behrmann, A. David, K. G. Larsen "*A Tutorial on UPPAAL 4.0*" http://www.uppaal.org/

[60] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P.Pettersson, W. Wi, and M. Hendrikes, "*UPPAAL 4.0*", in Proc. QEST, 2006, pp. 125-126.

[61] P. Herber, and S. Glesner, "*A HW/SW co-Verification framework for SystemC*", in ACM Trans. Embed Comput, Syst. 12, 1s Article 61, 2013

[62] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, "*A SystemC/TLM Semantics in PROMELA and its Possible Application*", in Proc. SPIN, LNCS, 2007, pp 204-222

[63] A.Cimatti. I. Narasamdya, and M. Roveri, "*Software Model Checking SystemC*", in IEEE Computer-Aid-Design of Integrated Circuits and Systems, Vol. 32, May, 2013

[64] R. Dömer, W. Chen and X. Han, "*Parallel Discrete Event Simulation of Transaction Level Models*", Proceedings of the Asia and South Pacific Design Automation Conference, 2012, Sydney, Australia, Feb 2012

[65] W. Chen, X. Han, and R. Dömer, "*May-Happen-in-Parallel Analysis based on Segment Graphs for Safe ESL Models*", in DATE '14 European Design and Automation Association.

[66] C. Schumacher, J. Weinstock, R. Leupers, and G. Ascheid, "*Scandal: SystemC Analysis for Nondeterminism Anomalies*", in Forum on Specification and Design Languages, 2012

[67] A. Sen, V. Ogale, and M. S. Abadir, "*Predictive Runtime Verification of Multiprocessor SoCs in SystemC*", Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE

[68] N. BLANC, E. Zurich, and D. Kroening, "*Race Analysis for SystemC using Model Checking*", in ACM Trans. Des. Autom. Electron. Syst. 15, 3, Article 21

[69] G. Naumovich and G.S. Avrunin, "*A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel*", in ACM SIGSOFT on Software Engineering Notes,vol.23,pp.24-34,1998

[70] K. P. Kim, R. Dömer, "Design Exploration using Multiple ARM Instruction Set Simulators - A Case Study on a JPEG Encoder", Center for Embedded Computer Systems, Technical Report 09-08, May 2009.

[71] P. Chandraiah, H. Schirner, N. Srinivas, R. Dömer, "*System-On Chip Modeling and Design: A Case Study on MP3 Decoder*", Center for Embedded Computer Systems, Technical Report 04-17, June 2004.

[72] Y. Samei and R. Dömer, "*MAVO: An Automated Framework for ESL Design: Monitor, Analyze, Visualize and Optimize*", Center for Embedded and Cyber-Physical Systems, Technical Report 14-12, November 2014.