Roopak Sinha · Parthasarathi Roop
Samik Basu

# Correct-by-Construction Approaches for SoC Design

Springer

# Correct-by-Construction Approaches for SoC Design

Roopak Sinha • Parthasarathi Roop • Samik Basu

# Correct-by-Construction Approaches for SoC Design

Springer

Roopak Sinha
Electrical and Computer Engineering
The University of Auckland
Auckland, New Zealand

Parthasarathi Roop
Electrical and Computer Engineering
The University of Auckland
Auckland, New Zealand

Samik Basu
Department of Computer Science
Iowa State University
Ames, IA, USA

ISBN 978-1-4614-7863-8 ISBN 978-1-4614-7864-5 (eBook)
DOI 10.1007/978-1-4614-7864-5
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2013944687



Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

*Roopak: To Anaya and Dhrov.*

*Partha: To Bapa, Maa and Guruji.*

*Samik: To my parents.*

# Foreword

It is always a good idea to begin a foreword for a computer science book with Moore's law! Indeed, this self-fulfilling prediction has led to a miniaturization such that 2012s processors have up to five billions transistors, such as the Intel's Xeon Phi. However, this miniaturization comes with a proportional complexity price, no engineer being able to conceive a processor with that many transistors. It was therefore natural to use a divide-and-conquer approach, which resulted first in the advent of multi-core processors and system-on-a-chip (SoC) and finally multi-processor-SoC (MPSoC). This is where the field of processor design (hardware) meets the field of component-based design (software). In both fields, the issue is quite simple: how can an engineer design a huge and complex system (be it an MPSoC or a large software system) from simpler and already validated components, possibly third-party? This looks very appealing, but it raises several crucial challenges, which are precisely what this book is about: the design, verification, and validation of systems composed of many components, such that global properties can be preserved and such that mismatches of any kind between the components can be avoided (mismatches either with respect to the control signals, to the clocks, or to the data exchanged between the components). The methods and tools presented in this book will prove essential for designing the next generation of MPSoCs, ever more powerful, ever more embedded, and ever more reliable.

Grenoble, France — Dr. Alain Girault

# Preface

A rapid surge in the consumer electronics revolution of the past decade may be attributed to the advancement in the automated design techniques for systems-on-chips (SoCs). SoCs are excellent metaphors for well-known computing terminologies such as *abstraction* and *reuse*, which have been in vogue for many decades both in industry and in academia. The mobile phone stands out as the significant benefactor of the abstraction and reuse approaches in computing. One key ingredient for the success of the mobile phone has been not only the advancement of RF technology but also the rapid evolution of the phone as a multi-functional, user friendly device that can perform wide-ranging tasks from movie making, transport planning to being a portable medical device. Would this have been possible without rapid advancement in SoC design techniques that incorporated state-of-the-art multicore processors, low-power design and many pre-verified components and buses?

The impetus for this monograph has been twofold. Firstly, Partha came across the reuse methodology manual (RMM) as a graduate student. RMM-like design methodology, and its variants practiced in industry, has been very successful in dealing with many challenges of SoC design. However, RMM and similar design approaches state that the system-level verification effort is still a major challenge as it requires significantly higher effort in comparison to other phases of SoC design. This motivated Partha and Roopak in early 2004 to examine this issue. Roopak worked on this problem from 2004 to 2008 and proposed the use of rigorous techniques such as model checking to aid the system-level verification process. They worked jointly with Samik who helped shape this research by leveraging many interesting ideas from software engineering. This joint effort led to many publications in embedded system conferences such as design automation and test in Europe (DATE) and VLSI Design. This research and associated publications form the main foundations of this book.

A second impetus of this monograph is motivated by the rapid adoption of SoCs in safety-critical applications in robotics, automotive and medical devices. Here, a major concern is the overall functional safety of the product, i.e., the device has to be designed to consider all possible risks and safety functions that have been integrated

to mitigate these risks. We noticed that the research we undertook since 2004 may be ideal not only for reducing the system-level verification effort but also for helping safety analysis and associated certification.

While our work is already published and hence available to expert researchers, we felt that a monograph on this topic of system-level verification of SoCs may have wider interest. Firstly, we wanted to make these research results available to researchers, designers and students alike. Hence, we embarked on a pedagogic presentation of formal methods that is widely accessible. We also simplified the process of requirement elicitation of SoCs through natural language-based boiler plates. This aspect of our work was never published before and is a significant addition to facilitate the adoption of formal methods in industry. Secondly, we developed this monograph with practitioners in mind. ARM is one of the most widely adopted platforms for SoCs (http://www.arm.com/community/soc/index.php). Hence, we used ARM and AMBA-based examples throughout this text to motivate our approach.

Last but not the least, our work has been accomplished not in isolation but due to research efforts of many researchers, who have also published on related topics and have proposed many concepts that we have either extended or reused. As it is often stated, "we stood on the shoulder of other researchers", to develop the proposed methodology. We have carefully examined and presented the latest status of this research on the use of formal methods in SoC design and system-level verification (see Chap. 7 for a state-of-the-art review of related research). This may be helpful for starting graduate students, SoC designers and researchers to access such related research quickly.

This being the first version of the book, we envisage that there are some errors and omissions. We welcome any feedback in this regard. We are also developing supplementary material such as exercises and lecture notes for use of this text in advanced undergraduate or postgraduate classes.

## *Organization and Reading Guide*

This book is not a traditional book on formal methods or formal verification. In these books, a set of formal techniques are described, in an algorithmic manner. Our book, in contrast, is developed for engineers so that they may use a design methodology that is based on sound mathematical principles. Hence, it is more a book on system design rather than system analysis, though the latter is built into the design process.

The book is organized into seven chapters. Each chapter deals with some key concepts related to the correct-by-construction design approach for SoCs. Figure 1 provides an overview of the chapters and the main ideas conveyed in them. Each chapter has a set of key concepts (presented in cyan) and the main idea (presented in red).

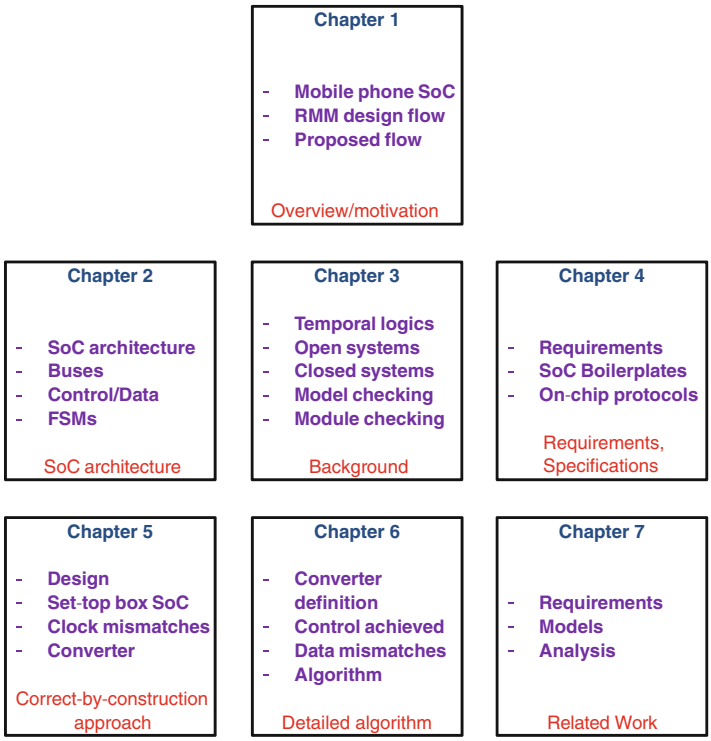



**Fig. 1** Key concepts of the different chapters

Chapter 1 provides the overview of SoC design and explains one particular approach based on the reuse methodology manual (RMM) [KB02]. We then motivate the need for the proposed approach, especially to tackle the problem of system-level verification. The key concepts presented in this chapter include the "inside view" of a mobile-phone SoC, the RMM design flow for SoCs and a proposed design flow for the correct-by-construction approach. This chapter may be viewed as the "overview/motivation" chapter.

Chapter 2 provides details of SoC internals, including key concepts such as the SoC architecture based on ARM, internals of buses used, the need for capturing the control-flow and data-flow in on-chip communication protocols of SoC components (also called IPs). This chapter may be viewed as the "SoC architecture" chapter.

Chapter 3 equips the reader with the necessary background material needed for later chapters. Key concepts covered in this chapter include the distinction between closed and open systems, temporal logic (to capture specifications), model checking (to verify a closed system) and module checking (to verify open systems). This chapter may be viewed as the "background" chapter.

Chapter 4 provides the concept of SoC boiler plates. These provide an approach by which engineers can capture correctness criteria of an SoC at the system-level using "structured English" requirements. We then show how these can be automatically mapped to temporal logic formula. This chapter also introduces a type of finite state machine called synchronous Kripke structures (SKS). These are used for the formal representation of on-chip communication protocols of the IPs. This chapter may be viewed as the "requirements/specification" chapter.

Chapter 5 provides a SoC design approach, where the on-chip protocols are described as SKS and requirements are captured as boiler plates. It then develops an approach called oversampling to bridge the clock mismatches between IPs. Finally, it uses an approach based on "converter synthesis" to propose the design methodology. The concepts in this chapter are illustrated using a set-top box example. This chapter may be viewed as the "correct-by-construction design methodology" chapter.

Chapter 6 provides further details of the converter synthesis algorithm. Key concepts covered in this chapter include data-buffers and data-related properties, converter definition and control and the converter generation algorithm. We provide classifications of the inputs and outputs of a converter. Then the converter is formalized and its control actions are described using an example. The details of this algorithm with an appropriate illustration appear in Appendix A. This chapter may be viewed as "converter synthesis" chapter.

Chapter 7 summarizes related work. We discuss the system-level verification literature and SoC design literature. Key concepts covered in this chapter include literature related to requirements, modelling and analysis.

Chapters 1–3 and 7 are self-contained and may be read in any order. Chapters 4–6, on the other hand, will make more sense if read in sequence. Here are some possible/suggested reading orders:

- $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4 \Rightarrow 5 \Rightarrow 6 \Rightarrow 7$. This is a very conventional flow and we have organized the chapters so that a reader can read the book top-down using this flow. This can also form the basis for any SoC design course, where these chapters may be covered over one semester, in this sequence.
- $1 \Rightarrow 7 \Rightarrow 3 \Rightarrow 2 \Rightarrow 4 \Rightarrow 5 \Rightarrow 6$. This order allows the reader to get good understanding of related work and the formal background (Chap. 3), before exploring the rest of the technical details. This is good for readers, who want a good grasp of both the background and the formal methods before progressing further.
- $1 \Rightarrow 2 \Rightarrow 7 \Rightarrow 4 \Rightarrow 5 \Rightarrow 6$. This order is suitable for more astute readers, who would like to skip the introductory material related to model/module checking and temporal logic. The contents of Chap. 7 can be suitably intertwined with the other chapters. For example, when reading Chap. 4, related work on modelling may be useful. Similarly, when reading Chaps. 5 and 6, related work on analysis presented in Chap. 7 may be useful.

The technical details of the converter generation algorithm are presented in Chap. 6 and Appendix A. An algorithm for convertibility verification and associated converter synthesis is presented in Appendix A. This forms the main basis of the proposed design methodology. However, Chap. 5 is fairly self-contained and provides the key idea of SoC design at a high level. Hence, it may be feasible that some readers may skip the technical details of Chap. 6 and Appendix A.

Auckland, New Zealand — Roopak Sinha
Auckland, New Zealand — Parthasarathi Roop
Ames, IA, USA — Samik Basu

# Acknowledgements

We would like to acknowledge the following individuals, who have been either our collaborators, co-authors or people with whom we have discussed regarding the proposed research.

- Dr. Alain Girault, INRIA, Grenoble
- Dr. Gregor Goessler, INRIA, Grenoble
- Dr. S. Ramesh, General Motors R&D, USA
- Prof. Arcot Sowmya, UNSW, Australia
- Prof. Zoran Salcic, University of Auckland, New Zealand
- Dr. Karin Avnit, UNSW, Australia

# Contents

# Acronyms

- SoC: System-on-a-chip. Plural: SoCs (systems-on-chip)
- IP: Intellectual property (block)
- FSM: Finite state machine
- SKS: Synchronous Kripke structures
- CTL: Computation tree logic
- AMBA: Advanced microcontroller bus architecture

# Chapter 1
# System-on-a-Chip Design

A rapid surge in the consumer electronics revolution of the past decade has been fuelled by a design methodology that started in the mid 1990s called *core based design* [GZ97]. During this period, hardware design methods based on logic synthesis techniques [GDWL92] were exacerbating the design productivity gap i.e., the productivity of human designers lagged behind the projections made by Moore's law. It was predicted that for an average engineer, completing a design for a chip in the year 2001 would require around 500 years [KB02]. Hence, there was a lot of impetus in the mid 1990s to develop alternative design techniques that were inspired by the concept of design reuse that was well known for software. Reusable software components, also called "abstract interfaces", were introduced as early as 1977 [Par77]. Subsequently, Goldberg introduced the concept of object-oriented programming as an alternative paradigm to structured programming. Ever since, technologies such as CORBA, .NET, and more recently web-services encapsulate a piece of software as a reusable *component* and provide a range of methods for component reuse and composition [OSB11].

The hardware design community, around the early nineties, started exploring ways to bridge the design productivity gap. The concept of core based design was developed to reuse pre-designed and pre-verified cores such as microprocessors, interface controllers, network controllers, ports, buses and timers. The main objective was to create a design using existing cores rather than creating a full-custom design. Cores could be kept at different levels of abstractions such as:

1. A *hard core* is a pre-designed block with minimal scope for modification. It is fully designed with placement and routing completed (physical design steps following high level synthesis).
2. A *soft core* is a synthesizable HDL description.
3. A *firm core* is at an intermediate level between hard and soft cores. It can be in register transfer level (RTL) or netlist form.

As the appetite for consumer electronic devices such as mobile phones increased, time to market pressures also increased significantly. Hence, a design paradigm for reusable cores that meet minimum quality standards [GLK+99] was needed.

R. Sinha et al., *Correct-by-Construction Approaches for SoC Design*,
DOI 10.1007/978-1-4614-7864-5_1, © Springer Science+Business Media New York 2014

An alliance called virtual socket interface alliance (VSIA) was created in 1996 and they developed a range of silicon intellectual property (SIP) standards to ensure the design of IP-cores that meet a desired quality standard.

These efforts combined with Keting and Bricaud's *Reuse methodology manual* [KB02] and platform-based design approaches [SVCBS04] have been widely adopted by industry. These led to the design of many systems-on-chips (SoCs) such as those used in mobile phones, MP3 players, set-top-boxes and many other consumer electronic devices. Techniques adopted during the late 1990s and early 2000s for SoC design helped in bridging the design productivity gap significantly and this led to a consumer electronics revolution unseen before. This success is exemplified by companies such as Apple, Samsung, Nokia, TI and Cisco just to name a few. It is worth noting that Apple emerged as one of the top companies in the world (with an estimated net worth of over $500 Billion) by 2010.

## 1.1 A Generic SoC Architecture

A system-on-a-chip, or SoC, is an embedded system all of whose components are integrated onto a single chip [Due04]. The components of an SoC are called intellectual property blocks, or IPs. IPs of an SoC include one or more processors (also called cores or macros) and several peripheral devices. These IPs are chosen depending on the desired application of the SoC. IPs may be implemented in hardware (such as a processor core, RAM, or USB devices) or in software (device drivers and algorithms running on a processor).

IPs communicate with each other through one or more well-defined communication channels. Typically, an SoC contains a common *SoC bus* that interconnects all its IPs [ARM99, GL00]. The bus has a well-defined communication protocol, described by several *bus policies*. An example of a bus policy is that an IP may initiate communication with another IP only if no other IPs are using the bus. IPs that can initiate communication with another IP over the bus are called *masters* and the IPs that respond to such communication requests are called *slaves* [Gut99]. At any given instance, the master using the bus (to communicate with another IP) is called the *active master*. If there are no active masters, the bus is said to be idle.

Since mid 2000s multiprocessor systems were becoming more popular in the desktop segment to provide better power-performance trade-off compared to single processor based designs. This also led to the recent widespread adoption of SoCs with multiple processor cores called multi-processor SoCs, or MPSoCs [CLN+02], which are very common in recent smart phones.

Figure 1.1 shows the layout of an SoC for a typical mobile phone. The SoC is based on the ARM Cortex-A15 multi-core processor and the ARM advanced microcontroller bus architecture (AMBA) [ARM99]. The SoC contains two processing

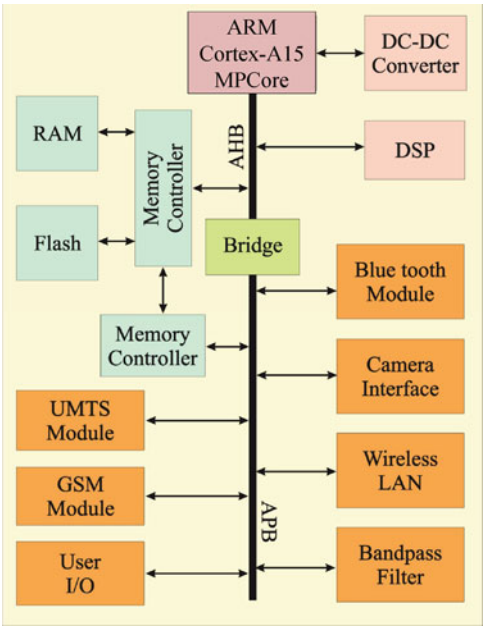



**Fig. 1.1** An SoC for a typical mobile phone

cores: an ARM Cortex-A15 general purpose multi-processor and a digital signal processor (DSP). The two cores are connected to the memory controller, that allows them to interact with the system memory and external flash memory, using the AMBA AHB bus (High performance bus). The peripheral IPs of the system, including the universal mobile telecommunications system (UMTS) and global system for mobile communications (GSM) modules are connected to the AMBA APB (peripheral bus). The two buses are connected via a *bridge*, which allows transfer of information between devices connected to the two buses.

An SoC can be designed to achieve complex functionality by reusing pre-implemented IPs. IP reuse considerably speeds up the process of building a single-chip application-specific computer system and is one of the main reasons for the ever-increasing popularity of SoCs [SCC00]. The mobile phone SoC shown in Fig. 1.1 can be built by reusing existing modules such as the ARM processor core, memory controllers and peripherals such as the GSM and UMTS modules. The software drivers for the modules are often available as pre-compiled libraries to the system programmer. The system can be extended by adding another IP, for example a digital TV receiver module, that extends the functionality of the system design.

## 1.2 Current Design Flow

A very well known design flow of SoCs is based on a structured methodology called the reuse methodology manual (RMM) [KB02]. This methodology in now adopted by industry and the most recent version of this book is in its third edition. The RMM approach develops an overall design flow that is combined with a range of guidelines which can be used for the SoC design process and associated best design practices. The overall design process of RMM is depicted in Fig. 1.2.

SoC design begins with the overall system specification, which is mainly expressed in natural language (step 1 in Fig. 1.2). This specification includes both the functional aspects (such as the desired functionality of the SoC and the associated design functions) along with non-functional constraints such as size, features, timing constraints and price. From this, a design team develops a behavioural model (step 2) in languages such as SystemC. This behavioural model is validated and based on the results of validation, the behavioural model may be further refined (step 3). Sometimes behavioural models in multiple languages are developed and tested. Following this, the system architects perform manual hardware-software partitioning (step 4). A library of IPs (both hardware and

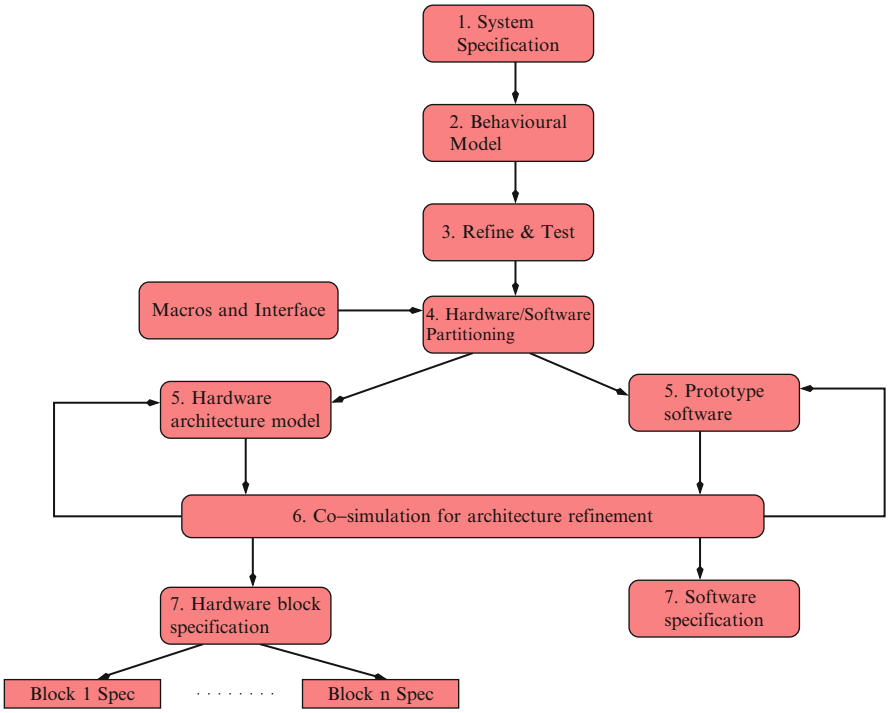



**Fig. 1.2** RMM system design flow (adapted from [KB02])

software) may be used for performance estimation purposes. Once the partitioning is done, the communication protocols between the hardware and software IPs need to be defined.

The next steps involve the development of the hardware architecture and the prototype software (step 5). Like the mobile phone SoC shown in Fig. 1.1, most SoCs use many IPs that are connected over multiple buses. SystemC TLM-based simulations come in handy here. Such simulations are needed to ascertain the bus structure and bandwidth needed. The simulations also include application code to make it realistic. On the software side, all device drivers for the various peripherals are made available for programming the cores. Then hardware-software co-simulation (step 6) is employed to validate the overall design. At the conclusion of this step the specification of the hardware and software IPs need to be completed (step 7). From this, the actual implementation of a given IP block is performed. RMM details the development of each core, from specification to logic synthesis, in detail. It also suggests a structured way using which detailed documentation could be developed for each core. This could be an existing pre-designed IP or an IP for which the design is yet to be performed.

## 1.3 Proposed Design Flow

The RMM design-flow discusses how stating with a high-level specification, the specification of the IPs and their interaction could be derived. An orthogonal question is the issue of IP integration through system-level verification. IP integration and system-level verification poses considerable challenges. To quote RMM, "verifying functionality and timing at the system-level is probably the most difficult and important aspect of SoC design. ... For many teams, verification takes 50–80% of the overall design effort". We start by discussing some major issues with IP integration. Again quoting RMM,

- "The low-level interfaces do not work; for example, a handshake signal is inverted.
- There was a misunderstanding in the functionality of the block.
- There are functional bugs in the design."

Some of these problems are related to the lack of rigour in the specification of functionality and the interfaces. Additional problems occur when IPs from diverse vendors need to be integrated. RMM identifies several problems such as the lack of proper documentation of the interface, and mismatches between the IP interface and the bus. Other problems such as mismatching naming conventions, data-widths and IP timing need to be considered. RMM goes on to provide several guidelines to deal with these problems. RMM also identifies several system-level verification issues and strategies for addressing these problems. These mostly rely on strategies for managing the complexity of testing such a large-scale system. Formal verification is sometimes employed to perform equivalence checking of the RTL and the derived netlist.

### 1.3.1 Motivation for the Book

While traditional SoCs are primarily used in consumer electronics applications that are not safety-critical, there has been an increasing demand for using SoCs in safety-critical application domains. Examples include avionics, automotive, military and medical devices. This is where our approach, as expounded in this monograph, comes in. Instead of using a set of guidelines to overcome system-level design and verification issues, we propose a correct-by-construction approach. Here, correct-by-construction[1] refers to the use of rigorous techniques in the SoC design process particularly targeting system-level IP integration and system-level validation. Our idea of correct-by-construction is similar to [HC02]. Our approach is also amenable to SoCs that need functional safety assessment [IEC]. While the reliability assessment part (which is primarily applied to the hardware part) for such SoCs has received some attention [CJ09], the use of mathematical techniques for enhanced safety of the overall system is yet to be considered. To this end, we propose the following:

1. We propose an approach by which the interface of every IP, i.e. the on-chip communication protocol of the IP, can be described in a formal manner using a special type of finite state machine called synchronous Kripke structure (SKS). This will eliminate any misunderstandings in the way each IP communicates with its environment. We propose that this formal specification be part of the documentation of every IP core. This is presented, in detail, in Chap. 4.
2. To correctly specify the interaction of IPs, we propose the need for system-level correctness criteria. Such criteria will ensure the correct sequencing of control and data exchanges while enforcing the lack of deadlocks. We seek to use the notion of boiler-plates [JD11], which provide a structured way of gathering such requirements. This step of obtaining and specifying systematic requirements is lacking in current design flow for SoCs. This is also presented in detail in Chap. 4.
3. The IP composition and validation task is achieved using a *model/module checking*-based algorithm, presented in Chap. 3, that checks whether the interface descriptions of a set of IPs match. Inputs to the algorithm are the SKSs of the IP's to be composed and a set of correctness requirements in the form of boiler plates. Mismatches such as incompatible control signals, incorrect sequencing of control operations, data-width mismatches and clock mismatches are considered. If some protocols are incompatible, appropriate *converters* are generated to facilitate correct composition which meets all high-level requirements. Converters can be either generated in one step or iteratively. When such composition is not feasible, an error trace is generated indicating the reasons for the failure to achieve composition. A system-design methodology that is correct-by-construction is presented in Chap. 5. The actual algorithm for IP composition is described in Chap. 6.

[1] This is unlike the approach taken by a single design team to apply the full life cycle to the development of a complete SoC as was done for the SUN SPARC [KB02].

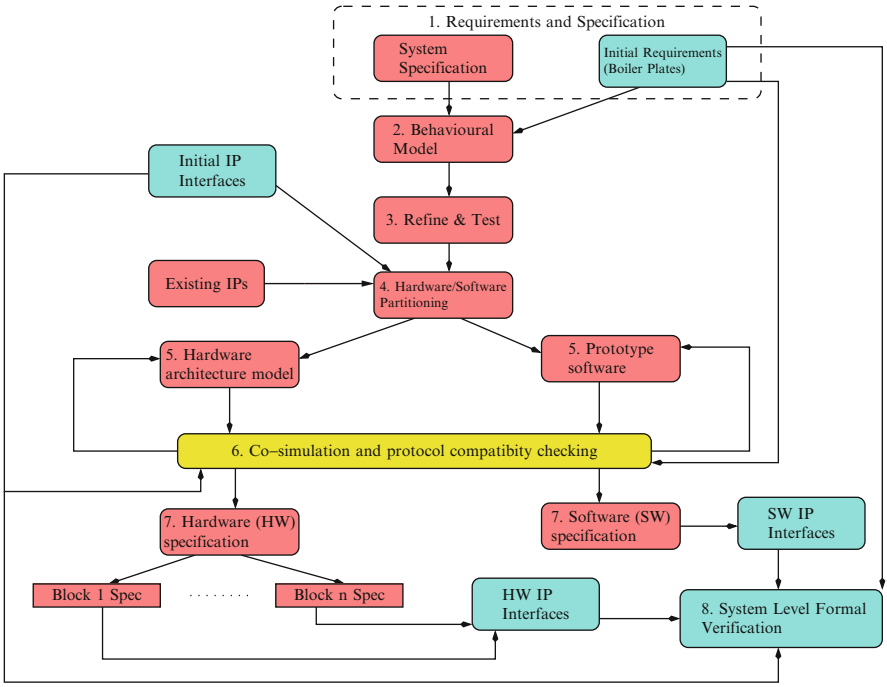



**Fig. 1.3** Design flow for correct-by-construction SoC design

The proposed design flow is shown in Fig. 1.3. This design flow can fit into existing design flow such as RMM and could be also used along side alternative correct-by-construction design flow say using platform-based approaches [SVCBS04].

In Fig. 1.3, we have revised the RMM flow to illustrate how the proposed methodology can be used to augment an existing design flow. Revised steps and inputs are indicated by cyan and yellow rectangles in contrast to the original pink rectangles of original RMM. All revised steps that are cyan in colour will be covered in this text, while one step that is coloured yellow is beyond the scope of this text. Revisions to the RMM flow presented in Fig. 1.2 include a new starting step called requirements and specification (Chap. 4). During this step, boiler plate based correctness requirements are gathered prior to the creation of the overall specification. We have revised the inputs to step 4 by separating the actual IP to be reused from the description of its interface (also termed as on-chip protocols of IPs). This interface needs to be specified in a formal way for every IP to be reused using SKS notation. We have revised step 6 of RMM so that in addition to the co-simulation based validation, we have added a compatibility checking step for the IPs that need to interact. This step, though a revision, is coloured yellow to indicate that we won't present this step in detail in this text. The proposed revision, however, will achieve better refinement and iteration between steps 5 and 6 of the

RMM flow. Once all IPs are selected and the specifications of each of the cores is available, we have added an explicit need for defining the interface for each newly created IP (both software and hardware). Finally, we have augmented the RMM flow with an 8th and final step, where correct by construction design can be performed using model/module checking based IP composition and validation step. This step takes as input boiler plates as correctness requirements. These are automatically converted to a set of temporal logic properties in CTL [CGP00]. This step also takes into account the SKS specification of every IP and the buses. Composition is then performed either iteratively (one IP at a time) or in a single step depending on the constraints to be satisfied. During composition, either a failure happens and an error trace is generated to highlight the reasons for such failure. Alternatively, a set of *converters* are generated to facilitate the seamless composition between the IPs. The composition algorithm can automatically resolve *control mismatches, data-width mismatches, and clock mismatches*. This algorithm for compatibility checking is further illustrated in the next section.

### *1.3.2 Benefits of the Proposed Approach*

The proposed approach has several benefits compared to existing approaches. These are outlined below:

- *Systematic approach to elicit requirements*: As IPs are connected to buses, there is a need to specify at a high-level abstraction the requirements for correct interaction. Such specification is either ignored at the start of design or captured informally. By introducing the well known idea of boiler plates, we propose a very systematic approach for eliciting and documenting correct system-level requirements.
- *A structured and unambiguous approach to modelling of interfaces*: RMM reports that many times there could be a misunderstanding regarding the functionality, timing or interface issues related to an IP block. We argue that it is possible to separate the functionality from the interface and timing issues using the proposed approach of SKS-based formal modelling. SKSs allow the precise modelling of IP interfaces including the IP clock.
- *A rigorous approach for system level IP integration and verification*: Using boiler plates and SKSs as inputs, we have developed a model checking based IP composition algorithm that is unique as well as scalable. We demonstrate that the proposed approach can be used for the IP integration and system-level verification tasks of real SoCs based on ARM.
- *Early detection of errors*: Detecting errors early is essential to the success of any commercial project. One of the major benefits of the proposed approach is that it can be used to detect errors early in the design cycle. By separating IP interface

from functionality and by including formal requirements, we have revised step 6 of RMM so that in addition to co-simulation, compatibility checking can be performed much earlier in the design cycle.
- *Better quality assurance for functional safety*: Functional safety assessment requires the use of rigorous quality assurance process, often involving formal methods. These are essential for dealing with the systematic errors found in both hardware and software. While the approach in [CJ09] deals with the issue of random errors in SoCs, the proposed approach targets systematic errors.

## 1.4 Organization of the Rest of the Book

The rest of this text is organized as follows:

- Chapter 2 presents a typical SoC architecture using the Advanced Microcontroller Bus Architecture, or AMBA developed for the ARM family of microprocessors introduced in the mobile phone SoC example in Fig. 1.1. It provides an in-depth discussion of the different buses used, their internal structures and how these can be used as the main vehicle for interconnection of several IPs. Following this, it motivates the usage of formal modelling in the context of examples presented in this chapter.
- Chapter 3 provides the main theoretical basics that underpin the rest of the book. In particular, it provides an in-depth treatment of two types of formal verification, namely *model checking* and *module checking*. Model checking can be used to verify SoCs, when the system may be considered *closed*, i.e., it has no interaction with the external environment. Module checking, on the other hand, is used for verifying SoCs that are *open* to environment interaction.
- Chapter 4 builds on the foundations of the previous two chapters to propose formal modelling of the on-chip protocols of IPs using Synchronous Kripke structures (SKSs). These are subsequently used in later chapters as IP interfaces (see Fig. 1.3). This chapter also provides a systematic approach for requirement elicitation using SoC boiler plates (which are developed, for the first time, in this book).
- Chapter 5, presents a correct-by-construction design methodology using the concepts introduced in the previous chapters. We use SKS for modelling on-chip protocols and boiler plates to capture the requirements for correct composition. Users are also required to provide the relationships between the clocks of the IPs and it is assumed that all clocks are derived from the fastest master clock (as is typical in SoCs). Clock mismatches are first removed using the approach of *oversampling* [Hal94]. It then uses the idea of converter synthesis (presented in Chap. 6) to design the SoC. This chapter uses an AMBA-based example of a set-top box to motivate the proposed design methodology.
- IPs may have three distinct types of mismatches: control mismatches, data-width mismatches and clock mismatches. Chapter 6 uses the formal models of IPs and

the requirements specified as boiler plates (that are converted to temporal logic CTL, which is introduced in Chap. 3), to propose a system-level verification methodology for IPs (step 8 in Fig. 1.3). Appendix A provides an algorithm for convertibility verification and converter synthesis. Technical details of the algorithm, its complexity and soundness are presented here. The algorithm is also illustrated using an example.

- The final chapter, Chap. 7, discusses related work and future directions for this book.

All the above chapters use many illustrative examples. Some chapters of this book are self contained while others are related to previous chapters. For example, Chaps. 2, 3 and 7 are fairly self-contained and could be read as individual chapters. However, the book as a whole is written such that there is strictly linear progression of ideas and associated flow.

## 1.5 Conclusions

This chapter provides a bird's-eye view of the current practice in SoC design. It then motivates the correct-by-construction approach particularly to tackle the system-level verification challenges in the current approaches to SoC design. The proposed flow is incremental and can be used in conjunction with existing design approaches such as platform-based design. We illustrate this incremental strategy by suggesting a revised RMM flow that can be augmented with the proposed system-level formal verification.

The next chapter deals with the architecture of typical SoCs that are designed using the ARM family of processors. We elaborate on the internals of the buses and how IPs communicate using these buses. We also provide some initial ideas on describing on-chip protocols of ARM-based SoCs.

# Chapter 2
# The AMBA SOC Platform

SoCs contain numerous IPs that provide varying functionalities. The interconnection of IPs is non-trivial because different SoCs may contain the same set of IPs but have different behaviour. In this chapter, we focus on how IPs are connected in a typical SoC, and how the information about their interconnection can be used to lead us to a formal model for IPs and SoCs. Figure 2.1 presents the mobile phone SoC example introduced previously in Chap. 1. This SoC contains processors, memories, and several peripherals. These IPs have varying functionalities, speeds, and data-exchange behaviours, and hence their interconnection must be comprehensive enough to be compatible to their individual behaviours, and at the same time provide a standard method to connect such diverse IPs.

Typically, IPs are connected using a *SoC bus*. A SoC bus is the interconnect fabric that provides a standard interface, or a *platform*, that can be used to connect different IPs in essentially the same way using *platform-based design* [SV02]. A *platform* is a combination of generic hardware (buses, processors, memory etc.) and software (microcontroller code) that can be customized by incremental changes in software and hardware to allow for a faster turnaround for differentiated products. Popular SoC buses available today include Altera Avalon [XZ03], IBM CoreConnect [HD02], IDT IPBus [IDT04], Open Core Protocol [Web00], and Wishbone [Pet99]. In this chapter we study the Advanced Microcontroller Bus Architecture, or AMBA [ARM99] in detail, since it is the de facto standard for creating SoCs. Examples of AMBA's use in industry include many chip makers, such as NVidia, Qualcomm, Actel, etc [HLP+10, AZ10].

AMBA, provided by ARM, describes a generic set of buses for use in SoCs, application-specific integrated circuits (ASICs), and traditional micro-controllers. The AMBA standard was first released in 1996, and is an open standard, which allows it to be used by all SoC manufacturers that use ARM-based processors in their SoCs. AMBA contains two types of buses: high performance *system* buses to connect core IPs, and a flexible *peripheral* bus to connect a multitude of peripherals and components. In Fig. 2.1, the AMBA advanced high performance bus (AHB) (a system bus) is used to connect a processor, a DSP, and high-performance memory controllers, and the AMBA advanced peripheral bus (APB) is used to connect

R. Sinha et al., *Correct-by-Construction Approaches for SoC Design*,
DOI 10.1007/978-1-4614-7864-5_2, 

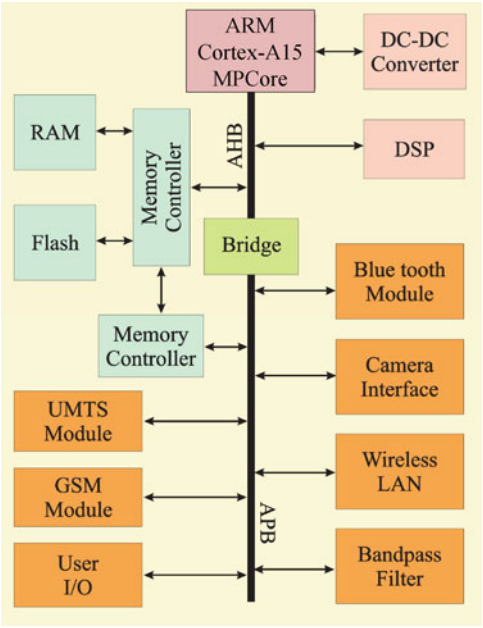



**Fig. 2.1** An AMBA-based mobile phone SoC example

various peripheral IPs. AMBA allows us the flexibility of upgrading this SoC by replacing some IPs with newer versions (such as the Bluetooth module) or by the addition of more IPs that add functionality to the existing design (such as an FM receiver). Note that Fig. 2.1 also contains a *Bridge*, which connects the AHB and APB buses. Bridges are standard bus-to-bus interfaces that allow IPs connected to different buses to communicate with each other in a standardized way.

This chapter first describes the AMBA standard in Sect. 2.1. This allows us to gain insights into how typical IPs are interconnected to create a SoC. As discussed previously in Chap. 1, our goal is to formally model IPs and SoCs in order to precisely capture individual IP behaviours as well as their interactions. Therefore, in Sect. 2.2, we look at how we can move towards a formal model for IPs and their interactions in SoCs based on AMBA.

## 2.1 The AMBA Standard

### *2.1.1 Terminology*

AMBA uses specific terminology for SoC components. Since this terminology will be used throughout this book, we introduce it here with an illustration of a simplified SoC based on the AMBA advanced system bus (ASB) shown in Fig. 2.2.

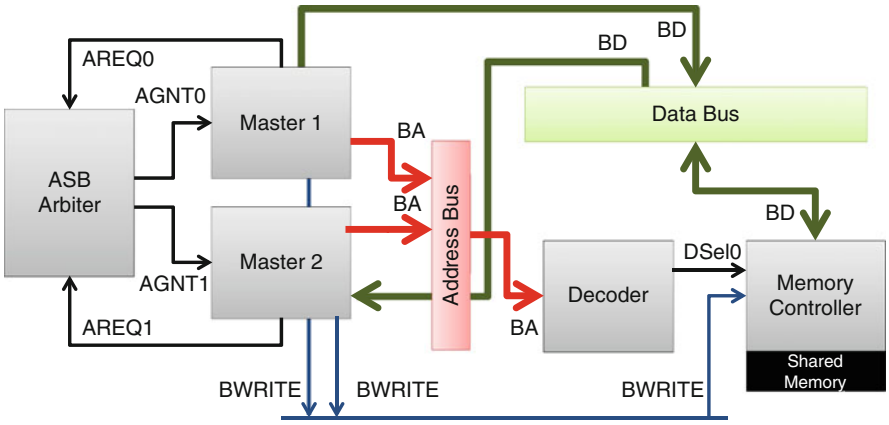



**Fig. 2.2** A simple ASB system

1. *Master*: A master device, usually connected to a high-throughput system bus, is an IP which is capable of initiating communications on the bus. Figure 2.2 contains two such masters: *Master 1* and *Master 2*.
2. *Arbiter*: As an SoC may contain multiple masters, an appropriate arbitration algorithm [Fly97] is needed. In the ASB system as shown in Fig. 2.2, a central arbiter is available, which connects to all masters in the system directly, and can provide bus access to them using an appropriate scheduling scheme. The central arbitration scheme can be round-robin, prioritized, etc.
3. *Slave*: A slave device responds to communication requests from a master. While peripheral devices, memory controllers, and network controllers are categorized as slaves, even processing elements such as DSPs may act as slaves. In Fig. 2.2, the SoC contains a single memory controller slave.
4. *Decoder*: Masters use the address bus of the interconnect to invoke slaves. A decoder decodes the address (or a part of it) into relevant control signals to enable slaves. For example, in Fig. 2.2, the decoder reads the address issued by the masters, and translates it into a slave select signal `DSel0` for the memory controller slave.
5. *Bridge*: AMBA provides the Advanced Peripheral Bus which is used to connect multiple peripheral IPs to a SoC. As shown in Fig. 2.1, a mobile phone SoC can contain a number of such peripherals which provide key functionalities to the SoC. Peripheral buses are slower and than system buses like ASB, and provide different communication protocols to the IPs connected to them. A bridge allows a master on the system bus to communicate with a peripheral device even though they do no share the same communication protocol.

In addition to the above IP-types, an SoC contains the following mechanisms for inter-IP communications:

- *Control signals*: AMBA buses provide dedicated point-to-point lines for IPs to communicate meaningful control information using control signals. Examples include signals for arbitration (such as $AREQ0$, $AGNT0$ in Fig. 2.2), write/read signals (such as $BWRITE$ in Fig. 2.2), slave select signals (such as $DSel0$ in Fig. 2.2), etc.
- *Data bus*: A data bus is used by IPs to write and read data. It is generally of a fixed width, and in AMBA data buses are 32-bits. Each master and slave in an SoC can read or write data through the data bus, as shown in Fig. 2.2. However, only one device is allowed to use the data bus at any one time in order to maintain data consistency.
- *Address bus*: An address bus allows IPs to specify physical addresses. AMBA address buses are 32-bits typically. Each IP in the system may connect to the address bus, but only one IP is allowed to read/write at any stage. In addition to specifying physical addresses, the address bus is read by the bus decoder to generate slave-select signals (like $DSel0$ in Fig. 2.2), which allows masters to invoke specific slaves.

The communications between IPs in an AMBA SoC happen as a series of *transactions*. A transaction is invoked by a master, and consists of the following steps: securing bus access, initiating read/write operation, and completing read/write operation. Figure 2.3 presents one possible transaction that can happen in the simple SoC shown in Fig. 2.2. It is shown as a message sequence chart [ITR96], and consists of the following steps. First, master $M1$ (and then master $M2$) requests bus access from the bus arbiter by asserting $AREQ0$ ($AREQ1$) control signal. The arbiter allows $M1$ access to the bus by asserting the $AGNT0$ control signal. Master $M2$ has to wait for $M1$ to relinquish the bus access after completing the current transaction. $M1$ then places a 32-bit address on the bus (also available to read by the slave), and a 32-bit data packet on the data bus. It also asserts the $BWRITE$ control signal, which means that it is initiating a write transaction. The decoder reads the address bus and then generates the slave select control signal $DSel0$ to invoke the slave. The slave then reads the data on the data bus and completes the write transaction.

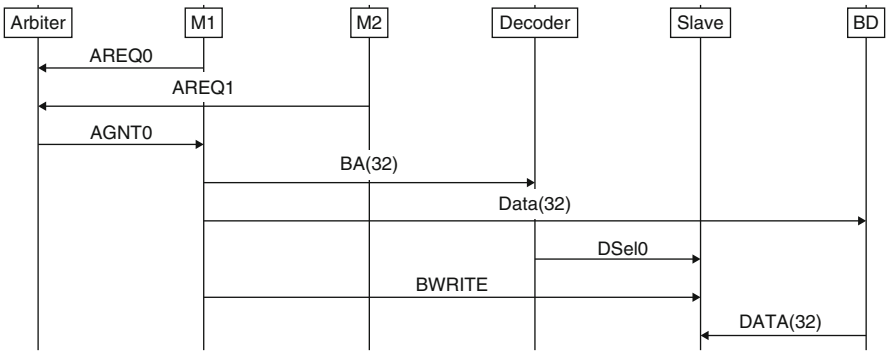



**Fig. 2.3** Example of a transaction in the ASB system shown in Fig. 2.2

AMBA based buses provide a variety of transactions. A transaction may be read (a master reading from a slave) or write (a slave reading data written by the master). Similarly, a transaction may be a *single transfer*, where a single data packet is exchanged (read/written), or a *burst transfer*, which allows multiple packets to be exchanged in the same transaction. Burst transfers are useful in high band-width data operations such as cache-line fills. In addition, AMBA provides mechanisms for error reporting (e.g. when a slave fails to correctly read or write). Each AMBA bus has different capabilities in terms of the transactions it can support, and the communication protocol may also differ from other AMBA buses.

### *2.1.2 AMBA Buses*

AMBA, first introduced in 1996, describes a number of buses and interfaces for on-chip communications. The first version of AMBA contained only two buses—a system bus and a peripheral bus. The current version, AMBA 4, contains a wide collection of high performance buses and interfaces. In this section, we look at some key AMBA buses, based mostly on the AMBA 3 standard [ARM].

#### Advanced High Performance Bus

The AHB bus was first proposed in AMBA 2, and was upgraded later in AMBA 3. A simple AHB system consists of one or more masters connected to slave devices (such as memory controllers), as shown in Fig. 2.4. This arrangement is known as a *layer*. A layer contains a unique decoder, which resolves master requests into slave activation signals. Additionally, if there are multiple masters, an arbiter is included,

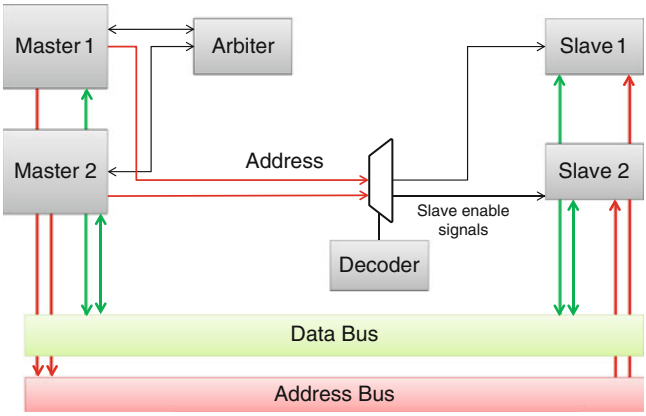



**Fig. 2.4** Single-layer AHB

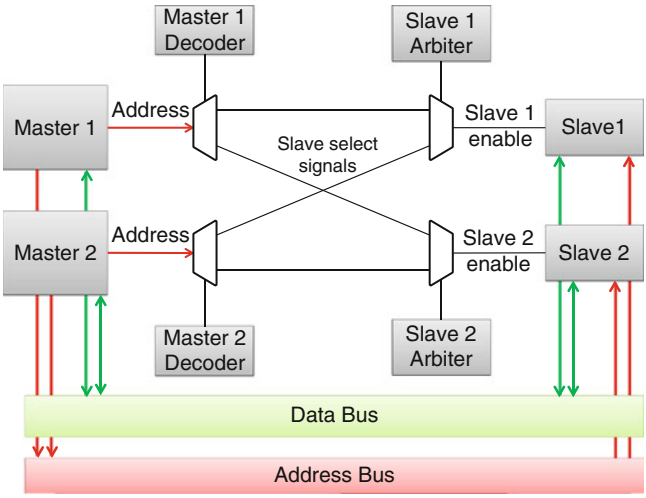



**Fig. 2.5** Multi-layer AHB

which decides which master gets to access the bus at any time. AHB masters can perform burst transfers where multiple data elements are read/written from/to a slave in a single transaction. A number of variants of AHB exist, as discussed below.

A multi-later AHB, as shown in Fig. 2.5 contains one layer (decoder and slave select signals) for each connected master in the AHB. Each slave has a slave arbiter, which decides which layer (master) is given access in case two masters request access to the slave at the same time. A multi-layer arrangement therefore allows more than one master to be actively talking to a (unique) slave in the SoC at the same time. The arbitration scheme for every slave-arbiter can be different, for example, round robin, or fixed priority. Of course, as the numbers of masters and slaves increase, the interconnect matrix becomes complex. This problem can be minimized by making some slaves *local* to a layer (and hence removing the need for separate arbiters for them), allowing multiple slaves to be accessed as a single device (reducing the number of arbiters), using multi-port slaves which do not need arbiters, and/or allowing multiple masters on a single layer (reducing the number of decoders).

The AHB-lite protocol is a high-bandwidth system that supports a single master connected to multiple slaves (shown in Fig. 2.6). This protocol does not need an arbiter due to the presence of a single master, and is therefore simpler to synthesize. A multi-multiplexer is used to ensure that only one slave writes to or reads from the data bus at any time. Multiple AHB-lite layers can be connected together using the multilayer interconnect, allowing for the synthesis of complex SoCs.

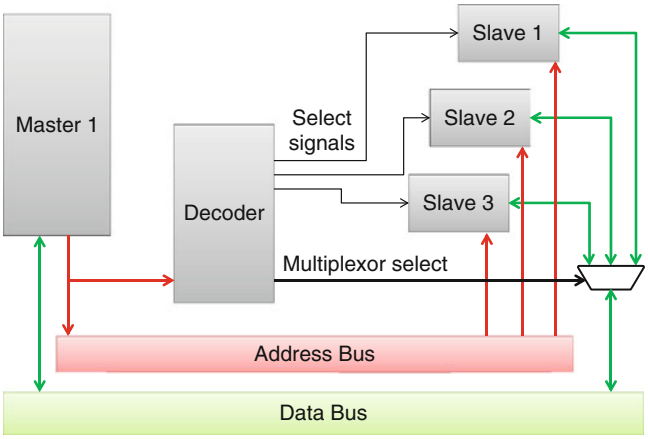



**Fig. 2.6** AHB-lite

### Advanced System Bus

Introduced in AMBA version 2.0, the ASB is a high performance synchronous bus. Multiple bus masters connect to an arbiter, which grants access to a master using the implemented arbitration scheme. ASB offers a pipelined protocol, where arbitration, address transfers and data transfers can occur simultaneously.

Figure 2.2 shows a simple ASB-based SoC. It contains two masters. Master 1 writes data to a shared memory via a memory controller slave, while master 2 reads data from this shared memory. We assume that the shared memory is simply a circular buffer, which reads or write data without needing an address. Only one master may access the slave at any time. This arbitration is achieved through the ASB arbiter, which reads requests from the two masters through the `AREQ0` and `AREQ1` signals. The arbiter then grants access to either master 1 or master 2 by asserting the signal `AGNT0` or `AGNT1`, respectively. The master granted access is called the *current bus master*, or simple the bus master.

If master 1 is chosen as the bus master, it drives a write cycle as follows. It asserts the control signal `BWRITE` which indicates a write operation. It then places an address on the 32-bit address bus. The decoder reads this address and generates the control signal `DSel0`, which activates the memory controller slave. At the same time, master 1 also writes 32-bit data `BD` onto the data bus. The slave then reads the data on the data bus into the shared memory. Note that in ASB systems, slaves can read some lines of the address bus to determine where to store the data being saved onto memory. However, as stated earlier, we assume that in our simple system, the shared memory is simply a circular buffer, and the new data is saved onto the next available location.

If master 2 is chosen as the bus master, it drives a *read* cycle. It first de-asserts the control signal `BWRITE` to indicate a read operation. It then places an address on the 32-bit address bus. The decoder reads this address and generates the control signal

DSel0, which activates the memory controller slave. When the slave is activated (and BWRITE is de-asserted), it places 32-bits of data from the shared memory onto the data bus. Master 2 can then read this data and complete the transaction.

#### Advanced Peripheral Bus

The APB bus is a low-cost non-pipelined bus used to connect low-bandwidth peripherals of an SoC. The APB is connected to a system-bus (such as AHB and AXI) using a bridge. The bridge allows masters on the AHB to address the APB system as a single slave. However, a bridge does not ensure that the interactions between masters and slaves are *correct*, i.e., as intended by the designer. APB transfers (read and write) take a minimum of 2 clock cycles with further wait states added for slower peripherals. Each transfer involves the exchange of 1 unit of data between the invoking master and the responding slave. Error responses for both reads and writes are allowed to provide extra functionality.

#### Other Buses

The advanced trace bus (ATB) provides functionality for on-chip debugging and trace analysis for AMBA-based SoCs. Each IP in the SoC that has trace capabilities is connected to the ATB. In the context of ATB, master interfaces write trace data on the bus, while slave interfaces receive the trace data from the ATB. The advanced eXtensible interface (AXI) is a high-speed, burst-only interface introduced first in AMBA 3 specifications. The AXI interface is backward compatible to AHB and APB, and hence can be used with legacy components. This generic interface is extensible to be used in a range of settings ranging from basic single-address bus single-data bus system to multilayer systems with multiple address and data buses.

## 2.2 Formal Modelling of AMBA SoCs

As discussed in the previous chapter, our aim is to use mathematically sound techniques to analyse an SoC for the satisfaction of functional specifications. In this chapter, we illustrate how to formally model AMBA systems. A formal model allows the use of mathematical reasoning to analyse a system. A formal model must be *abstract* so that it is easier to analyze and does not carry redundant details, but at the same time, it must capture adequate details about the system to be meaningful. In our context, the following details must be captured by the formal model:

1. *Control flow*: The model must be able to capture the sending and receiving of control signals of the IPs of a SoC.

2. *Data flow*: The model must be able to capture the reading and writing of data by the IPs of a SoC.
3. *Timing*: The model must be able to capture the synchronous nature of AMBA systems, where execution proceeds with respect to clocks. Note that different peripherals may use different clocks, but that these clocks are generally obtained from the same clock source.
4. *Compositionality*: The model must be modular and compositional, such that individual IPs can be modelled, and the SoC can be rendered as a composition of its IPs.

We now expand on each of the above criteria, and illustrate how these can be modelled.

### 2.2.1 Control Flow

In AMBA systems, control flow consists of the exchange of control signals between the bus components, masters and slaves. The control signals used in every AMBA bus are well defined by the AMBA specifications. For example, in the AMBA ASB (advanced system bus), a master may request bus access by sustaining the `AREQ` signal, and the bus arbiter responds using the `AGNT` signal when access is granted (as shown in Fig. 2.2).

The behaviour of a single IP can be captured using a state machine which has a finite set of states $S$ with a unique initial state $s_0$. States have transitions between them, which describe possible control flow between them. For example, consider master 1 of the AMBA ASB example shown in Fig. 2.2. This master writes data to a shared memory via the memory controller slave. We can capture the control flow in the execution of master 1 as follows. Firstly, we can identify all control states that this IP can be in at any time. These states are:

1. *Init*: The initial state of the IP upon start up.
2. *Request*: The master has requested for bus access and is waiting for bus access to be granted by the arbiter.
3. *Active*: Access has been granted, and the write operation has been started.
4. *Complete*: The write operation has been completed, and the master can relinquish bus access.

The control flow within master 1 can be described using transitions between the above states, as shown in Fig. 2.7a. Starting from the *Init* state, the IP enters the *Request* state, and waits there until the arbiter allows bus access. It then enters the *Active* state to start the transfer. A transition to *Complete* state is taken when the transfer is complete. The IP then takes a transition back to *Init* to repeat the above control flow infinitely often.

Note that, as shown in Fig. 2.7, states and transitions can have *labels*. State labels may refer to state names (as in the above figure), or to the values of the variables of

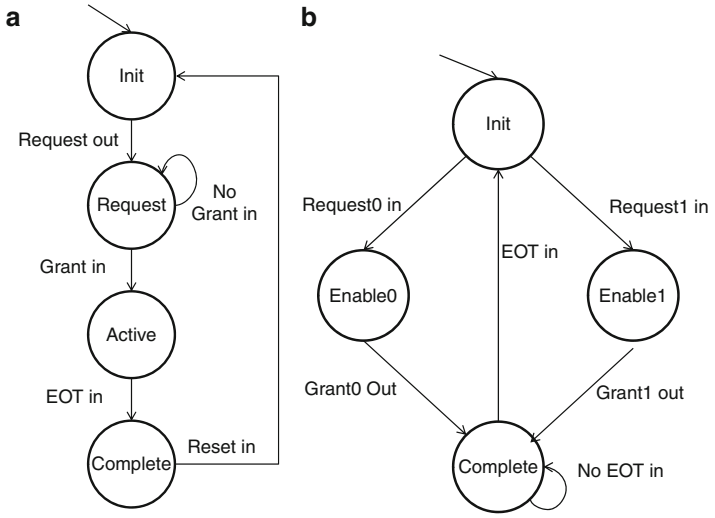



**Fig. 2.7** Modelling control flow using finite state machines. (**a**) Modelling control flow of a master IP using finite state machine. (**b**) Control flow model of the bus arbiter

an IP when it is in a specific state. Similarly, transition labels may refer to the inputs and outputs generated by the IP when a transition is taken. For example, when a transition is taken from state *Init* to state *Request*, the transition label `Request Out` indicates the emission of a request signal by the IP (to the arbiter).

The level of detail captured in each state and transition of the above model can be increased or decreased by using a denser or sparser set of propositions. Consider Fig. 2.7b, which provides a control flow model of the bus arbiter shown in Fig. 2.2. This model reads access requests from two masters, and grants access on a first-come first-served basis. When access is granted to a master, the arbiter waits in the *Complete* state until it receives the control signal `EOT` signalling an end-of-transaction. A more detailed model of this arbiter can carry actual signal names of the AMBA ASB, such as $AREQ0$ and $AGNT0$ (as shown in Fig. 2.2) instead of the more abstract labels $Request1\,in$ and $Grant0\,out$.

### 2.2.2 Data Flow

A typical single-layer AMBA system uses two main data flow channels: the address bus and the data bus. The address bus is used by the current bus master to invoke a slave. The data bus is used to read/write the actual data produced by the slave or master during the transaction. A formal model must capture all relevant data flow information and must leave out any redundant information.

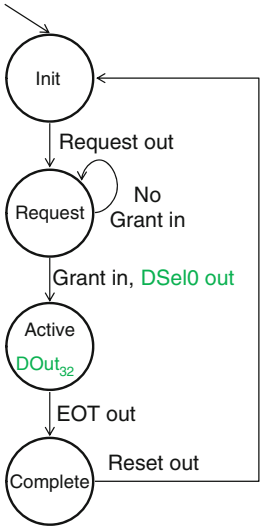



**Fig. 2.8** Modelling data flow using finite state machine

A naive way to model the above data flow is to precisely enumerate all possible addresses and data that buses can contain. Given that each of these buses is usually 32-bits in size, there are $2^{32}$ possible unique values. Capturing all these in the formal model can result in state explosion. On the other hand, we can ignore data flow all together, or in other words, we can remove all data elements from the formal model. However, such abstraction will limit the effectiveness of the model significantly, and it may become useless in most cases.

We can also take a third, more balanced approach. For the address bus, we can abstract all data out and replace them with pre-interpreted extra control signals. For example, the master may refer to a slave $S$ using the range of addresses $Ox00000000$ to $Ox000000FF$. We can simply replace the emission of any address in this range by a control signal $Invoke\_S$, which can act as an additional output of the state machine representing the master IP, and will be included as an additional control input of the model of the slave $S$. The extra control signals can be captured simply as inputs and outputs of the state machine. Figure 2.8 extends Fig. 2.7, by capturing the data flow in master 1 (shown in Fig. 2.2). Note that the label of the transition from state *Request* to state *Active* contains the `DSel0 out`, which abstracts the actual address (on the address bus) issued by the master to invoke Slave0 (via the decoder) to a control signal.

For data transfers, we can restrict our model such that it only shows the *number of bits* written to/read from the data bus by a slave/master during a transaction. An IP may partially use the data bus. For example, an 8-bit serial UART IP may only read/write 8 bits from/to the 32-bit data bus. While this approach does not enumerate all possible data exchanged between IPs (and at the same time reduces state explosion), it can be useful in ensuring that all data generated by one IP is read by other IP(s), or that no data-corruption due to overflows (when existing data on

the bus is overwritten) or underflows (when an IP tries to read non-existent data). As shown in Fig. 2.8, the state label $DOut_{32}$ indicates that the IP outputs 32 bits of data to the data bus every time state *Active* is reached.

### 2.2.3 Timing

AMBA has synchronous buses, which enforce connected IPs to execute with respect to a common clock signal. Different buses in a system might use different clocks. For example, the APB will usually have a slower bus than the ASB. Usually, these different clocks are *rational*, or are obtained by dividing the system clock by different integer values.

We can capture timing aspects by simply ensuring that the control and data flow of the system is divided into discrete steps (or states) such that each step represents the state of the IP at a different clock edge, or tick. For example, we can assume that the state machine describing the control-data flow of master 1 (Fig. 2.2) as illustrated in Fig. 2.8 executes synchronously. In other words, only one transition is taken at every tick of the clock.

### 2.2.4 Compositionality

We require that the formal model be capable of not only describing individual IPs, but also partial and complete SoCs. For this, we require a *composition operator* to combine individual models into a single model. Given that our model is based on finite state machines that execute synchronously, we can use existing operators such as the synchronous parallel [BCE+03] operator.

As discussed above, the state machine shown in Fig. 2.8 executes synchronously with the common bus clock of the SoC shown in Fig. 2.2. Moreover, all other IPs connected to the same bus (master 2, arbiter, the memory controller slave) also execute with respect to the same clock, and therefore will *synchronize* their execution with master 1 at each clock tick. In other words, we can describe the SoC as a *composition* of its connected IP protocols. More details about compositionality and how we compose IP protocols will appear later in the book.

## 2.3 Conclusions

This chapter provides a detailed look at ARM-based SoC architectures. We explain the architecture of a typical mobile phone SoC. We discuss the AMBA family of SoC buses, and show how different interconnects can be used to implement different parts of an SoC. As AMBA is an open standard that can be used royalty-free, it is currently the de facto standard for SoC design.

Once the SoC architecture is introduced, we need to discuss how the on-chip communication protocols of the IPs are described. We introduce the concept of a finite state machine (FSM)-based representation of the on-chip communication protocols of SoCs. This model captures the control flow, data flow, and timing characteristics of these protocols. A formal representation of these FSMs is presented in Chap. 4 as synchronous Kripke structures (SKS). SKS allow us to formally capture key details of IPs and SoCs, which can be used to analyze them statically using techniques such as model checking/module checking. The next chapter provides such background material i.e., formal modelling and analysis using model/module checking.

# Chapter 3
# Automatic Verification Using Model and Module Checking

The process of checking whether a system (software or hardware) conforms to or violates a pre-specified set of desired properties (often referred to as the requirements) is called *verification*. Verification can involve a wide variety of techniques ranging from design review, code inspection to (semi-)automated analysis of system, which may include testing and automatic test case generation [SR07]. Verification also covers rigorous techniques involving some form of *static analysis* such as model checking [CGP00] and abstract interpretation [CC10]. These are used to ensure the functional correctness of a system so as to guarantee 100 % coverage for the property being checked. Hence, these techniques are usually classified as *formal verification* techniques, where the objective of verification is to prove the absence of certain errors. Here, the term *formal* refers to the use of a range of mathematical techniques as the underlying basis of the analysis. This is in contrast to informal techniques such as testing that may find the presence of certain errors but cannot prove the absence of these errors. This chapter provides the formal notation and the underlying algorithms needed in order to understand later chapters on SoC design that are founded on the principles introduced here. However, this chapter is fairly self-contained and could be used as an introduction to formal verification. An excellent introduction to formal verification techniques for hardware may be obtained through the survey [KG99] and that of software through the survey [DKW08].

Formal verification of complex systems must consider two distinct types of systems, namely *open systems* and *closed systems*. Open systems, such as SoCs, used in game consoles and mobile phones, continuously interact with the adjoining environment. Here, the system must react to stimulus from the environment and the environment is uncontrollable. Closed systems, such as payroll processing and inventory control, read inputs and process these algorithmically to produce some outputs. We will specifically focus on the verification of open systems, since SoCs are inherently open. However, for completeness, we start with the verification of closed systems, called *model checking* [CGP00]. We then present module checking [KV96] for verifying open systems in Sect. 3.2

R. Sinha et al., *Correct-by-Construction Approaches for SoC Design*,
DOI 10.1007/978-1-4614-7864-5_3, © Springer Science+Business Media New York 2014

## 3.1 Model Checking

For formal analysis to proceed, model checking requires the following:

1. Mathematical model of the design.
2. Language to express properties mathematically.
3. A method of proof.

Finite state machines (FSMs introduced in Chap. 2) are very well known models of both hardware and software. We will use a variant of FSMs, called Kripke Structures, to represent a digital system formally.

### *3.1.1 Basic Model: Kripke Structure*

**Definition 3.1 (Kripke structure).** A Kripke structure (KS) is defined as a tuple $\langle AP, S, s_0, T, L \rangle$ where $AP$ is the set of atomic propositions, $S$ is the set of states with $s_0$ being an unique start state, $T \subseteq S \times S$ is the set of transition/edge relations between two states and $L : S \to 2^{AP}$ is the labeling function mapping each state to a set of atomic propositions $AP$ that holds at that state. We denote a transition $(s_1, s_2) \in T$ as $s_1 \longrightarrow s_2$.

Dynamics of systems are described using KS models, where the states in a given KS denotes the different configurations of the system, transitions denote the evolution of the system from one configuration to another, and the labeling function captures the properties that are satisfied at each configuration. Additionally, meta information such as start configurations, final configurations and actions that result in evolutions may also be included. Conditions that enable evolution are represented by appropriately augmenting the KS model (by adding transition labels, by marking specific states at start and/or final states, etc.).

In the proceeding, we focus on the Definition 3.1 for setting up and explaining background knowledge necessary for the rest of the chapter. Eventually, we present required meta information and the corresponding KS augmentation for modeling the SoCs (refer to Sect. 3.2.1.1).

**Definition 3.2 (Path).** A path in a KS $M$ starting from a state $s$ is an infinite sequence of states $\pi = s_o s_1 s_2 \ldots$. such that $(s_i, s_{i+1}) \in T$ holds for all $i \geq 0$.

We will use $\text{PATH}(s)$ to denote the set of paths starting from state $s$ in KS.

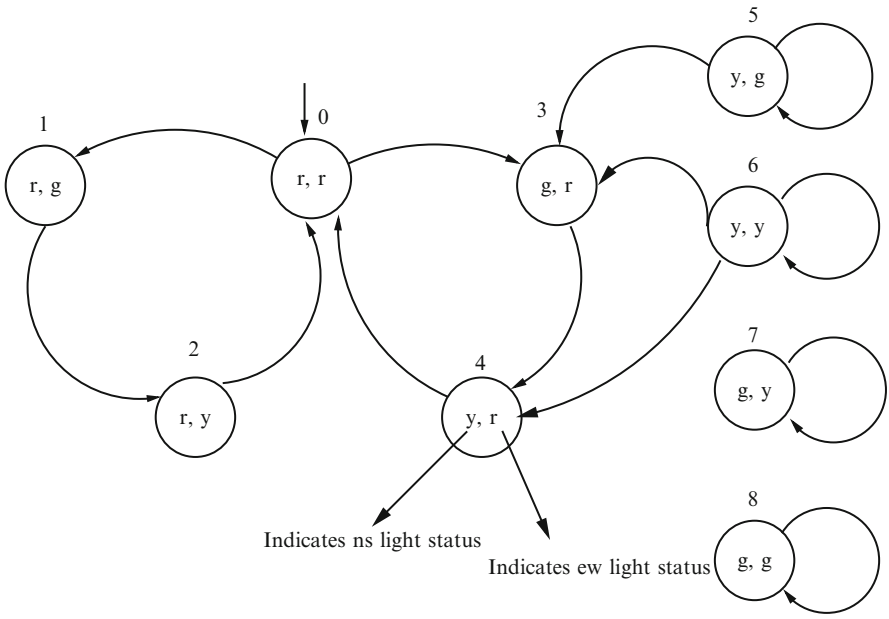



**Fig. 3.1** A KS model of a simple traffic light controller [KG99]

### *3.1.2 Example: Model of a Traffic Light Controller*

We use a pedagogic example of a traffic light controller presented in [KG99] to motivate formal modelling. Consider the controller for a traffic intersection for a crossing between a north-south (ns) road and an east-west (ew) road. Let $\mathscr{C} = \{r, y, g\}$. Then, the state space of the controller is given by $\mathscr{S} = \mathscr{C} \times \mathscr{C}$ and hence the controller has a maximum of nine states. This controller may be represented formally using a KS as shown in Fig. 3.1. Here the states $S = \{0, 1, 2, 3, .., 8\}$, with the initial state being the state 0. $AP = \{ns = r, ns = g, ns = y, ew = r, ew = g, ew = y\}$. The transition relation $T = \{(0,1), (0,3), (1,2), \ldots\}$ and the labeling function returns the value of the state label for any given state i.e., $L(0) = \{ns = r, ew = r\}$ and $L(8) = \{ns = g, ew = g\}$. It may be noted that states on the right of Fig. 3.1 are not desirable to be reached in a correct and properly initialized controller.

### *3.1.3 Specification Using Temporal Logic*

Temporal logic may be used to describe the evolution of system over time. Such properties are traditionally classified as either *safety properties* or *liveness properties*. A safety property asserts that *nothing bad ever happens*. For example,

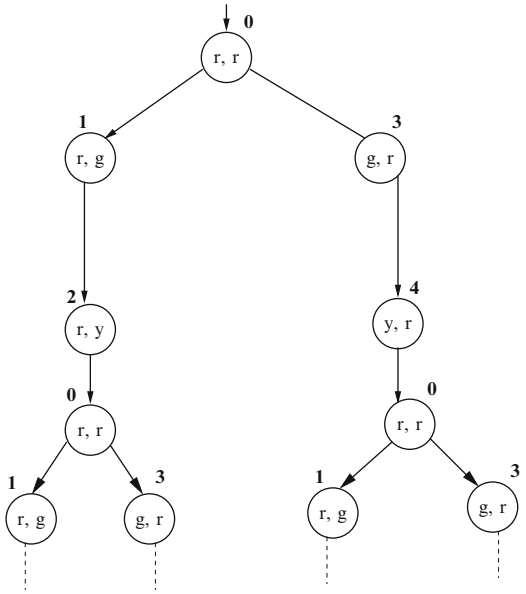



**Fig. 3.2** An example computation tree for the traffic light controller

in a traffic light controller, we may demand that no reachable state in the controller must have green lights in both directions. Safety properties ensure that the system is never allowed to enter a bad state. However, for overall system correctness, safety properties are not enough. For example, we could design a "safe" traffic light controller that has a single state in which it turns the north-south light green and the east-west light red. However, such a controller is unusable as it has no mechanism for ensuring progress in the system. Liveness properties are required to ensure that a system makes progress. Hence, they demand that *something good eventually happens*. In the traffic light controller example, a liveness property might be that every direction eventually sees the green light. Temporal logics are used to describe such safety and liveness properties and may be classified as follows:

1. Linear temporal logic: Time progresses in linear steps corresponding to real or natural numbers. Example: LTL.
2. Branching time logic: Progression of time could have branching in the future, while the past is linearly ordered. Example: CTL, ACTL and CTL$^*$.

In this manuscript we will concentrate only on Computation Tree Logic or CTL. CTL properties are expressed over computation trees. A computation tree is an infinite tree starting from a given state of a Kripke model. The tree is obtained by unwinding the model. For example, a computation tree starting at the state 0 of the traffic light model is shown in Fig. 3.2.

CTL uses both temporal operators and path quantifiers to construct complex properties starting with atomic propositions. The temporal operators of CTL are:

1. $X$, the next time operator: this hold over a given path if the property holds in the second state of the path.
2. $F$, in the future operator: this holds over a path if the property holds in the current state or any state in the future i.e., eventually.
3. $G$, the globally operator: this holds over a path if the property holds over every state in the path.
4. $U$, the until operator: the property $(\varphi_1 U \varphi_2)$ holds over a path if there is a state on the path where $\varphi_2$ holds and that $\varphi_1$ must hold along every preceding state (if such a state is available).

In addition to the temporal operators, CTL also uses the following path quantifiers:

1. $E$ denoting the existential path quantifier.
2. And $A$ denoting the universal path quantifier.

We will start by first presenting the temporal operators that are defined over paths of a computation tree. Any CTL formula is essentially a *state formula* (which hold over individual states), which may consist of sub-formulae that hold over entire paths also known as *path formulae*. Any state formula is also a valid CTL formula. Figure 3.3 presents visually how CTL formulae may be built by composing simpler formulae iteratively to construct more complex formulae. A valid CTL formula is obtained when the accepting state (i.e., state 1) is reached. The process starts in state 0 when any atomic proposition is selected and this automatically leads to a transition to state 1. Any time a temporal operator $(X, F, G, U)$ prefixes a formula in state 1, a transition is made to state 2, since this creates a path formula. Only when a path formula (in state 2) is prefixed with a path quantifier, a transition is again made to state 1, since this leads to the creation of a state formula.

Lets illustrate this process as follows. Simplest state formulae are atomic propositions from the set $AP$ defined in the model. For example, $(ew = g)$ in the example in Fig. 3.1 is a state formula. When you prefix any state formula with a path quantifier it becomes a path formula. For example, the formula $\mathtt{F}(ew = g)$ is a path formula. A path formula can be converted to a state formula by quantifying this using a path quantifier. For example, if we prefix the previous path formula using the existential path quantifier, it becomes a state formula $\mathtt{EF}(ew = g)$. This CTL formula expresses the fact that there exists a path in the model such that eventually the east-west light turns green. Continuing in this manner, we can obtain the CTL formula $\mathtt{AG}(\mathtt{EF}(ew = g) \vee \mathtt{EF}(ns = g))$. In contrast, the formula $\mathtt{AG}(\mathtt{EF}(ew = g) \vee \mathtt{EFX}(ns = g))$ is not a valid CTL formula (left as an exercise for the reader based in Fig. 3.3).

Similarly, the formula $\mathtt{AFEX}(ew = g)$ can be built iteratively starting with $(ew = g)$, which is a state formula and when prefixed with $X$ becomes a path formula $X(ew = g)$. When this is prefixed again by a path quantifier, it becomes a state formula $\mathtt{EX}(ew = g)$, which states that there exists a path such that in the next state the east-west light turns green. Continuing in this manner, the final formula $\mathtt{AFEX}(ew = g)$ is built, which specifies that eventually along all paths a state is

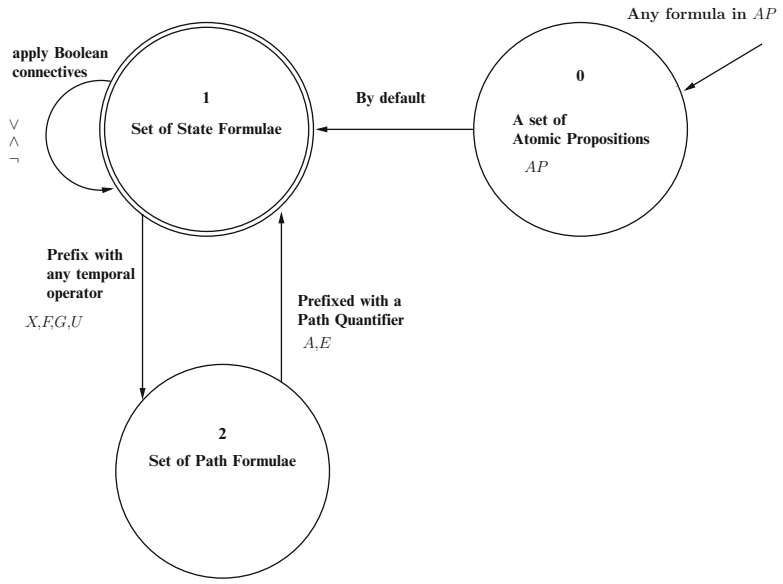



**Fig. 3.3** How to build CTL formulae

reachable such that in the next state the east-west light turns green. We refer the readers to [CGP00] for detailed discussion of CTL$^*$ and its branching fragment CTL and linear fragment LTL.

Formally, CTL formulas is the set of all *state formulas* generated by the following definition.

**Definition 3.3 (State formulae).**

- If $p \in AP$ then $p$ is a state formula.
- If $\varphi_1$ and $\varphi_2$ are state formulae, then $\neg\varphi_1, (\varphi_1 \vee \varphi_2), (\varphi_1 \wedge \varphi_2)$ are also state formulae.
- If $\varphi$ is a *path formula* then $\mathtt{E}\varphi$ and $\mathtt{A}\varphi$ are also state formulas, where path formula are defined as in Definition 3.4.

**Definition 3.4 (Path formulae).** If $\varphi_1$ and $\varphi_2$ are *state* formulae then $\mathtt{X}\varphi_1$, $\mathtt{F}\varphi_1$, $\mathtt{G}\varphi_1$, $(\varphi_1 \ \mathtt{U} \ \varphi_2)$ are path formulae.

We will use $\varphi$ and its subscripted versions to denote state formulas, i.e, CTL properties and $\psi$ and its subscripted versions to denote path formulas.

**CTL Semantics.** The meaning of the CTL properties are given in terms of the configurations (i.e., states in a KS model) which satisfy the (state) properties. The standard notation $[\![\varphi]\!]_M$ denotes the semantics of $\varphi$ in terms of sets of states in KS model that satisfy the state formula $\varphi$. Similarly, we will use $[\![\psi]\!]_M$ to denote the semantics of $\psi$ in terms of sets of paths in KS model that satisfy $\psi$. Formally,

**Semantics for CTL state property:** $\varphi \rightarrow \texttt{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \texttt{E}\psi \mid \texttt{A}\psi$

$$[\![\texttt{true}]\!]_M = S$$
$$[\![p]\!]_M = \{s \mid p \in L(s)\}$$
$$[\![\neg\varphi]\!]_M = S - [\![\varphi]\!]_M$$
$$[\![\varphi_1 \vee \varphi_2]\!]_M = [\![\varphi_1]\!]_M \cup [\![\varphi_2]\!]_M$$
$$[\![\texttt{E}\psi]\!]_M = \{s \mid \exists \pi \in \textsc{Path}(s) : \pi \in [\![\psi]\!]_M\}$$
$$[\![\texttt{A}\psi]\!]_M = \{s \mid \forall \pi \in \textsc{Path}(s) : \pi \in [\![\psi]\!]_M\}$$

*where*

**Semantics of path property:** $\psi \rightarrow \texttt{X}\varphi \mid \texttt{F}\varphi \mid \texttt{G}\varphi \mid \varphi\ \texttt{U}\ \varphi$

$$[\![\texttt{X}\varphi]\!]_M = \{\pi \mid \pi[1] \in [\![\varphi]\!]_M\}$$
$$[\![\texttt{F}\varphi]\!]_M = \{\pi \mid \exists i \geq 0 : \pi[i] \in [\![\varphi]\!]_M\}$$
$$[\![\texttt{G}\varphi]\!]_M = \{\pi \mid \forall i \geq 0 : \pi[i] \in [\![\varphi]\!]_M\}$$
$$[\![\varphi\ \texttt{U}\ \varphi_2]\!]_M = \{s \mid \exists j \geq 0 : \pi[j] \in [\![\varphi_2]\!]_M \wedge \forall i < j : \pi[i] \in [\![\varphi_1]\!]_M\}$$

The propositional constant `true` holds in any state in the model $M$. The proposition $p$ is satisfied in all KS states, which are labeled by a set of propositions that includes $p$. The negation of a CTL property is satisfied in states which do not satisfy the property. The disjunctive CTL property is satisfied in all states that satisfy either of the disjuncts. The CTL property $\texttt{E}\psi$ is satisfied by any state from where there exists one path which, in turn, satisfies $\psi$. Similarly, the CTL property $\texttt{A}\psi$ is satisfied by any state from where all paths satisfy $\psi$. The path property $\texttt{X}\varphi$ is satisfied by any path where the second state in the path satisfied $\varphi$. The path property $\texttt{F}\varphi$ is satisfied by any path which has some state in the path that satisfies $\varphi$. On the other hand, a path satisfies $\texttt{G}\varphi$ if every state in the path satisfies $\varphi$. Finally, a path satisfies $\varphi_1\ \texttt{U}\ \varphi_2$ when there exists a state in that path that satisfies $\varphi_2$ and in all state before that $\varphi_1$ is satisfied.

For instance, the CTL property $\texttt{EX}\varphi$ is satisfied by the set of states which has *at least* one next state (denoted by $\pi[1]$) which, in turn, belongs to the semantics of $\varphi$.

When a state $s$ in the model $M$ belongs to the semantics of CTL property $\varphi$, it is denoted by $M, s \models \varphi$. If the start state (denoted by $s_0$) of $M$ belongs to the semantics of the CTL property $\varphi$, then it is denoted by $M \models \varphi$.

We will write $[\![\varphi]\!]$ ($[\![\psi]\!]$) instead of $[\![\varphi]\!]_M$ (respectively, $[\![\psi]\!]_M$) when the $M$ is immediate from the context.

**Temporal Equivalences and Adequate Set.** As is evident from CTL syntax, every path quantifier (E and A) must be matched with a state qualifier (X, F, G and U) and vice versa. Therefore, $\mathtt{EG}\varphi$ is a valid CTL formula, while $\mathtt{XEG}\varphi$ or $\mathtt{EGF}\varphi$ or $\mathtt{GF}\varphi$ are not valid CTL formulas.

Several equivalences can be established based on the semantics of the CTL formulas. These instances are based on the equivalences of state and path formulas.

First, consider state formulas. For instance,

$$[\![\neg \mathtt{A}\psi]\!] = S - \{s \mid \forall \pi \in \textsc{Path}(s) : \pi \in [\![\psi]\!]\} = \{s \mid \exists \pi \in \textsc{Path}(s) : \pi \in [\![\neg\psi]\!]\} = [\![\mathtt{E}\neg\psi]\!]$$

Therefore, $\neg \mathtt{A}\psi = \mathtt{E}\neg\psi$. Similarly, the following equivalences hold for the path formulas.

$$\begin{aligned}
\mathtt{X}\varphi &= \neg\mathtt{X}\neg\varphi \\
\mathtt{F}\varphi &= \neg\mathtt{G}\neg\varphi \\
\mathtt{true}\ \mathtt{U}\ \varphi &= \mathtt{F}\varphi \\
\neg(\varphi_1\ \mathtt{U}\ \varphi_2) &= (\neg\varphi_2\ \mathtt{U}\ (\neg\varphi_1 \wedge \neg\varphi_2)) \vee \mathtt{G}\neg\varphi_2
\end{aligned}$$

While the first three equivalences of path formulas follow directly from the boolean manipulation of negation and universal/existential quantifiers, the last equivalence is somewhat involved. There are two possible ways the left-hand side path formula can be satisfied in a path.

- In the path, $\varphi_1$ is not satisfied in some state before a state satisfying $\varphi_2$.
- In the path, $\varphi_2$ is not satisfied in any state.

Based on the above equivalences, there are five key CTL formula equivalences:

$$\begin{aligned}
\mathtt{AX}\varphi &= \neg\mathtt{EX}\neg\varphi \\
\mathtt{AG}\varphi &= \neg\mathtt{EF}\neg\varphi \\
\mathtt{AF}\varphi &= \neg\mathtt{EG}\neg\varphi \\
\mathtt{A}(\varphi_1\ \mathtt{U}\ \varphi_2) &= \neg\mathtt{E}(\neg\varphi_2\ \mathtt{U}\ (\neg\varphi_1 \wedge \neg\varphi_2)) \wedge \neg\mathtt{EG}\neg\varphi_2 \\
\mathtt{EF}\varphi &= \mathtt{E}(\mathtt{true}\ \mathtt{U}\ \varphi)
\end{aligned}$$

Note that, $\neg\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)$ cannot be expressed as AU-CTL formula. This is because

$$\neg\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2) = \mathtt{A}\neg(\varphi_1\ \mathtt{U}\ \varphi_2) = \mathtt{A}(\ (\neg\varphi_2\ \mathtt{U}\ (\neg\varphi_1 \wedge \neg\varphi_2)) \vee \mathtt{G}\neg\varphi_2\ )$$

As universal quantification (in this case A—universal path quantifier) does not distribute over disjunction, it is not possible to express the right-hand side as a valid CTL formula (recall that, valid CTL formula requires pairing of path quantifiers with state quantifier).

The CTL formula equivalences provides an important insight: all properties expressible using CTL can be represented using a subset of CTL formula constructs. Such a subset referred to as the *adequate set*. There are many adequate sets. For instance, $\{\texttt{true}, AP, \neg, \vee, \texttt{EX}, \texttt{EU}, \texttt{EG}\}$ and $\{\texttt{false}, AP, \neg, \vee, \texttt{EX}, \texttt{EU}, \texttt{AF}\}$ are two different adequate sets.[1] Any adequate set must contain EU. The importance of adequate set lies in the fact that it is sufficient for any method for verifying the satisfiability of CTL formula to consider elements in some adequate set. In the following section, we will consider a model checking algorithm for CTL which considers the adequate set $\{\texttt{true}, AP, \neg, \vee, \texttt{EX}, \texttt{EU}, \texttt{EG}\}$.

### 3.1.4 Explicit Sate Model Checking

The simplest form of model checking may be performed using simple state-labeling using *reachability analysis*. This process is known as *explicit state* model checking since, in the worst case, the complete state space of the model needs to be explored in order to verify a property.

**Definition 3.5 (Model checking).** Given a *KS* instance $M$ and a CTL property $\varphi$, determine a $S' \subseteq S$ such that $S' = \{s \mid M, s \models \varphi\}$. We say that $M \models \varphi$ iff $s_0 \in S'$. In this case, we say that the model satisfies the property.

The algorithm starts with the initial state labeling present in the KS model $M$ due to the state labeling function $L(s)$. The algorithm proceeds in a series of iterations. In the $i$th stage of the algorithm, sub-formulas with $(i-1)$ nested CTL operators are processed. When a given sub-formula is processed, it is added to the labeling of state $s$ if it holds in $s$. Given the adequate set of CTL operators presented above, the algorithm needs to consider six different types of formulae, namely $p \in AP$, $\varphi_1 \vee \varphi_2$, $\neg\varphi$, $\texttt{EX}\varphi$, $\texttt{EG}\varphi$ and $\texttt{E}(\varphi_1\ \texttt{U}\ \varphi_2)$. The labeling procedure works as follows:

- $[\![p]\!]$ For any atomic proposition $p$, the labeling function of the model ($L$) can be used to determine a state is labelled with $p$ or not.
- $[\![\varphi_1 \vee \varphi_2]\!]$ For formulas of the form $\varphi_1 \vee \varphi_2$, we label all states that are labelled by either $\varphi_1$ or $\varphi_2$.
- $[\![\neg\varphi]\!]$ For formulas of the form $\neg\varphi$, we label all states that are not labelled by $\varphi$.
- $[\![\texttt{EX}\varphi]\!]$ For formulas of the form $\texttt{EX}\varphi$, we label all states that have some successor labelled by $\varphi$.
- $[\![\texttt{E}(\varphi_1\ \texttt{U}\ \varphi_2)]\!]$ For formulas of the form $\varphi = \texttt{E}(\varphi_1\ \texttt{U}\ \varphi_2)$, we start by labeling all states that are labelled by $\varphi_2$ and label them with $\varphi$. Following this,

[1] $AP$ denotes set of propositions.

we perform backward reachability to determine all predecessors of $\varphi$-states that are labelled with $\varphi_1$. These are then also labelled with $\varphi$.

$[\![\texttt{EG}\varphi]\!]$ For formulas of the form $\texttt{EG}\varphi$, firstly all states that don't satisfy $\varphi$ are identified and removed along with any associated transitions. Then the algorithm identifies all non-trivial strongly connected components (SCCs) [CGP00] in this revised model. All states in these SCCs are labelled with $\texttt{EG}\varphi$. Finally, any state that can reach these SCCs are also labeled with $\texttt{EG}\varphi$.

We start by illustrating the labeling of $\texttt{EG}\varphi$, which is the most involved step from the above. Note that this property requires that a path (by Definition 3.2 a path is infinite unless explicitly stated otherwise) must exist in the model where every state is labelled by $\varphi$.

This algorithm is based on the identification of SCCs in the graph of the model $M$. Any KS $M$ has an associated graph consisting of states and transitions. An SCC is a largest subset of states in the KS $M$ such that each state in the subset can reach each other via at least one transition. This is also referred to as the non-trivial SCC as it involves at least one transition.

Figure 3.4 demonstrates the verification of property $\texttt{EG}p$ on a sample KS. The algorithm starts by deleting all states and transitions that are not labelled by $p$ (step 1). After this step, state 6 and associated transitions are deleted. Next we identify that the only non-trivial SCC consists of the states 1, 2, 4 and 5. All these states are labelled by the property $\texttt{EG}p$ (step 2). Finally, we identify any state in the reduced KS (after step 1) that can reach a state labelled by $\texttt{EG}p$. We label these states (i.e., state 0) also by $\texttt{EG}p$.

We now illustrate the explicit state model checking process using the traffic light controller example from the previous section. We need to verify the property that *we can always reach a state such that the east west light is green in one of its successors*. This may be expressed in CTL as $\varphi = \texttt{AF}(\texttt{EX}[ew = g])$.

We may express $\varphi = \texttt{AF}(\texttt{EX}[ew = g])$ as: $\neg\texttt{EG}[\neg\texttt{EX}(ew = g)]$. The nesting of CTL operators in the above formula are:

- i=1: $(ew = g)$
- i=2: $\texttt{EX}(ew = g)$
- i=3: $\neg\texttt{EX}(ew = g)$
- i=4: $\texttt{EG}(\neg\texttt{EX}(ew = g))$
- i=5: $\neg\texttt{EG}(\neg\texttt{EX}(ew = g))$

Due to the above nesting, the verification of this property over the model will require five iterations. The first iteration labels all states where the proposition $(ew = g)$ holds. Figure 3.5 illustrates this step and shows that state 1, 8 and 5 satisfy this formula, since they are already labelled with $(ew = g)$. The next formula to be examined is $\texttt{EX}(ew = g)$. This iteration is illustrated in Fig. 3.6. During this iteration, any state from which a successor labelled by $(ew = g)$ can be reached is determined. This is then labelled by $\texttt{EX}(ew = g)$ Hence, we label state 0 (since it has state 1 as its successor), state 5 and state 8 (as 5 and 8 have self loops and they are already labelled by $(ew = g)$).

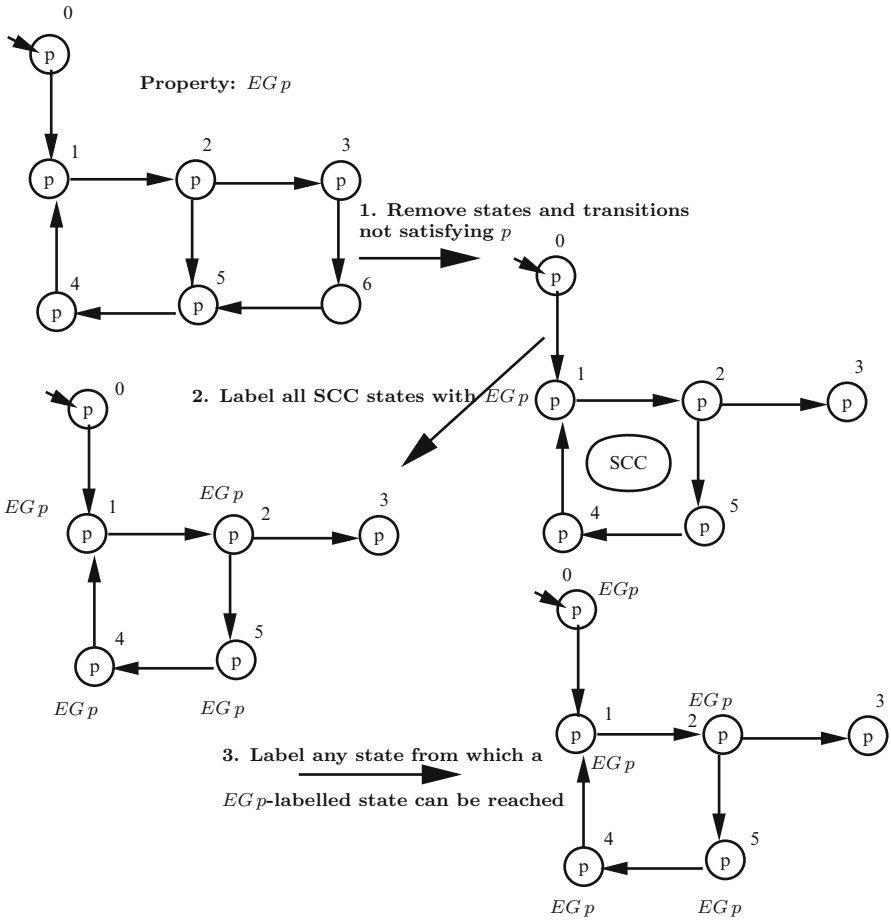



**Fig. 3.4** Example illustrating the verification of the property $\mathtt{EG}p$

In the third iteration, the algorithm identifies all states that don't satisfy the property $\mathtt{EX}(ew = g)$. Hence, all states in the model except state 0, 5 and 8 are labelled by the formula $\neg(\mathtt{EX}(ew = g)$. This is illustrated in Fig. 3.7. In the fourth iteration, the algorithm performs the labeling of the formula $EG(\neg\mathtt{EX}(ew = g))$. First all states that don't satisfy $\neg\mathtt{EX}(ew = g)$ are identified and removed. Hence, the states 0, 8 and 5 and all transitions associated with them are removed. Subsequently, all non-trivial SCCs in the reduced graph are identified. The only non-trivial SCCs are due to the self-loops on states 6, 7. Hence, these two states are labelled with the formula $\mathtt{EG}(\neg\mathtt{EX}(ew = g))$. This is illustrated in Fig. 3.8.

The fifth and the final iteration considers the formula $\neg\mathtt{EG}(\neg\mathtt{EX}(ew = g))$. As only states 6 and 7 satisfy $\mathtt{EG}(\neg\mathtt{EX}(ew = g))$, all the remaining states, except these satisfy the desired CTL property. This is illustrated in Fig. 3.9.

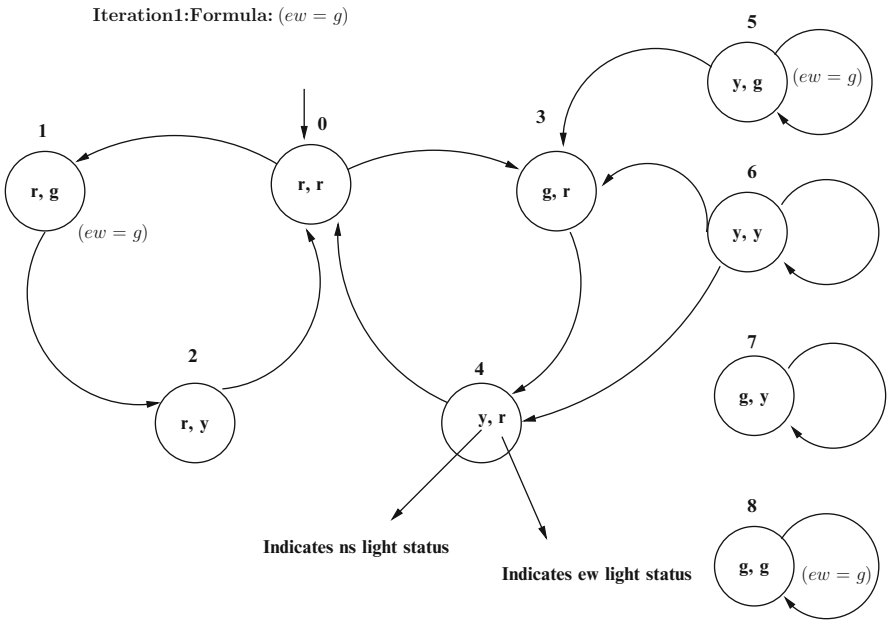



**Fig. 3.5** First iteration of the verification of the property $\neg\mathtt{EG}[\neg\mathtt{EX}(ew = g)]$

The complexity with respect to the model is linear to the number of transitions and states in the model. This is because of the complexity of finding SCCs and performing backward depth-first traversal of graph. The complexity with respect to the CTL property is linear to the number of subformulas in the property. This is because labeling any state with a property requires that labeling of states with all its subformulas are already performed. Therefore, the overall complexity of model checking is of the order of product of the formula size and the sum of transitions and states i.e., $O(|M| \times |\varphi)$.

Note that, the number of states and transitions in a KS can grow exponentially with respect to the number of propositions that are present in the KS definition. This is referred to as the state-space explosion. The explosion is particularly severe for compositional systems, where each participant in the composition contributes to the set of propositions. Consequently, this algorithm will not scale while verifying large systems. Several techniques for dealing with this state space explosion problem have been discussed in [CGP00,DKW08]. These include symbolic techniques using BDD and SAT, abstraction based techniques such as refinement, and compositional verification techniques. A detailed discussion of these is beyond the scope of this text.

- i=1: $(ew = g)$
- i=2: $\mathtt{EX}(ew = g)$

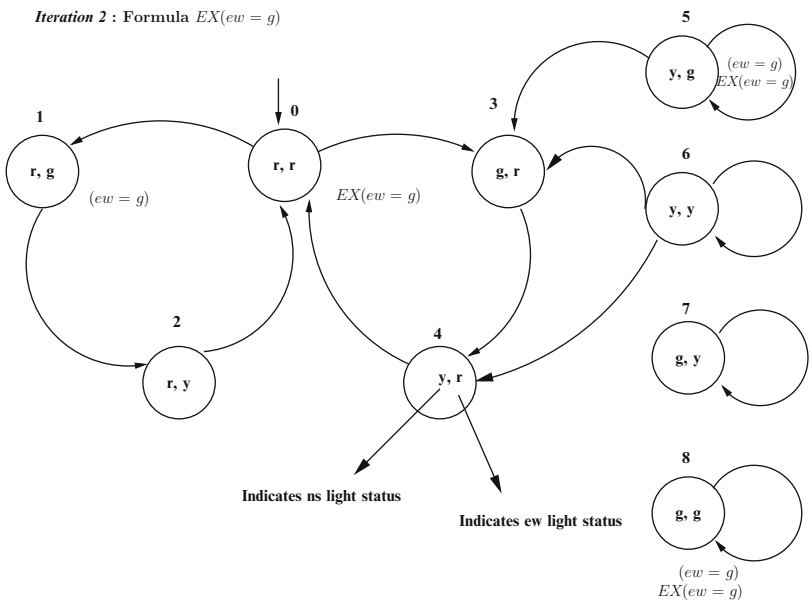



**Fig. 3.6** Second iteration of the verification of the property $\neg\mathtt{EG}[\neg\mathtt{EX}(ew = g)]$

## 3.2 Module Checking

In the previous section, we considered the verification of closed systems. Closed systems don't interact with the environment in order to evolve. For example, the traffic light controller presented in Fig. 3.1 is an instance of older generation systems that are known as *un-actuated intersections*. Here, the controller is pre-programmed so that the timing of the state transitions is not changeable based on traffic patterns. Hence, these systems don't work very well as traffic demand varies.

To overcome the above problems, actuated intersections and sophisticated controllers such as SCATS [CWCS11, oNSW] were introduced. These controllers gather additional control information using sensors such as underground inductive loops. These loops can detect the presence of cars and can also be used to compute control parameters such as the *degree of saturation* (which indicates how effectively the green time was used). Use of additional sensors improve the efficiency of traffic control. However, this also introduces new challenges for the verification of the system. Consider a revised traffic light controller, as shown in Fig. 3.10. Here, the transition from state 0 to 1 happens only when a car is detected along the east-west direction. Similarly, a transition from the state 0 to 3 is made when a car

- i=3: $\neg\mathtt{EX}(ew = g)$
- i=4: $\mathtt{EG}(\neg\mathtt{EX}(ew = g))$
- i=5: $\neg\mathtt{EG}(\neg\mathtt{EX}(ew = g))$

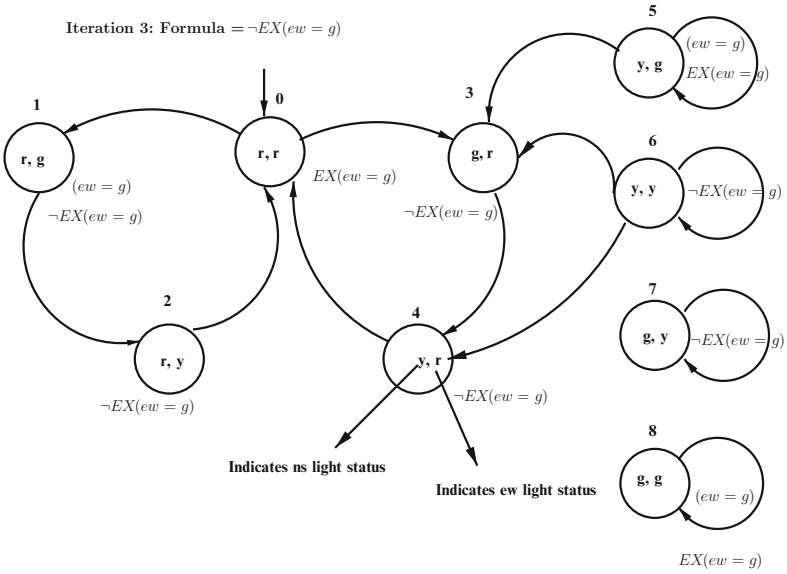



**Fig. 3.7** Third iteration of the verification of the property $\neg\mathtt{EG}[\neg\mathtt{EX}(ew = g)]$

is detected in the north-south direction. When both these conditions are true, the controller makes a non-deterministic choice. Consider that we intend to verify the property in this controller that *starting from the initial state, the east-west light will eventually turn green*. This property will be satisfied in the model in Fig. 3.1 since this model is closed. Here, every transition requires no external events and hence will be eventually taken. However, in the open system as illustrated in Fig. 3.10, this property will fail. This is because the transition to state 3 is dependent on an input from the environment, which may never be provided. Hence, the verification of open systems needs to consider the inherent non-determinism of the environment during the verification process. As SoCs are inherently open, we need to consider this aspect carefully. In the following, we consider any open system as a *module*, in contrast to models of closed systems as defined in Definition 3.1 (this terminology is borrowed from the original formulation regarding the verification of open systems in [KVW01]).

*Given a module, does it satisfy a temporal property when used in any environment?*

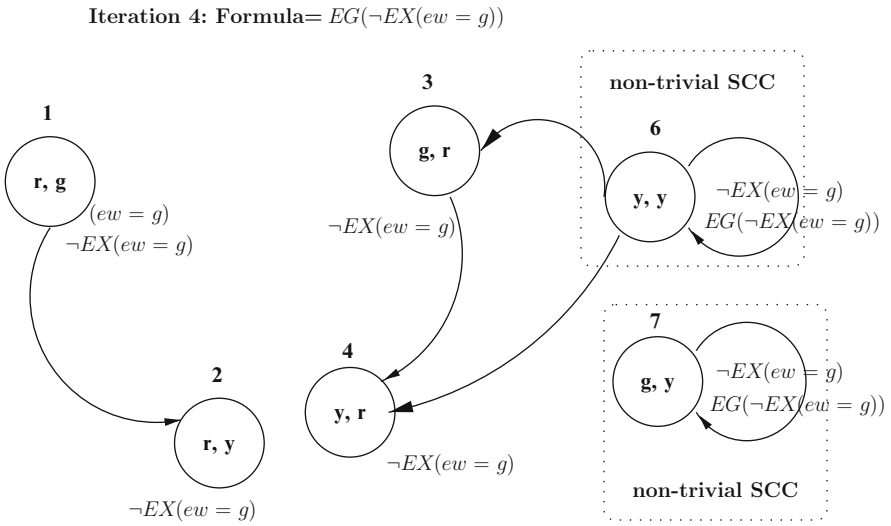



**Fig. 3.8** Fourth iteration of the verification of the property $\neg\texttt{EG}[\neg\texttt{EX}(ew = g)]$

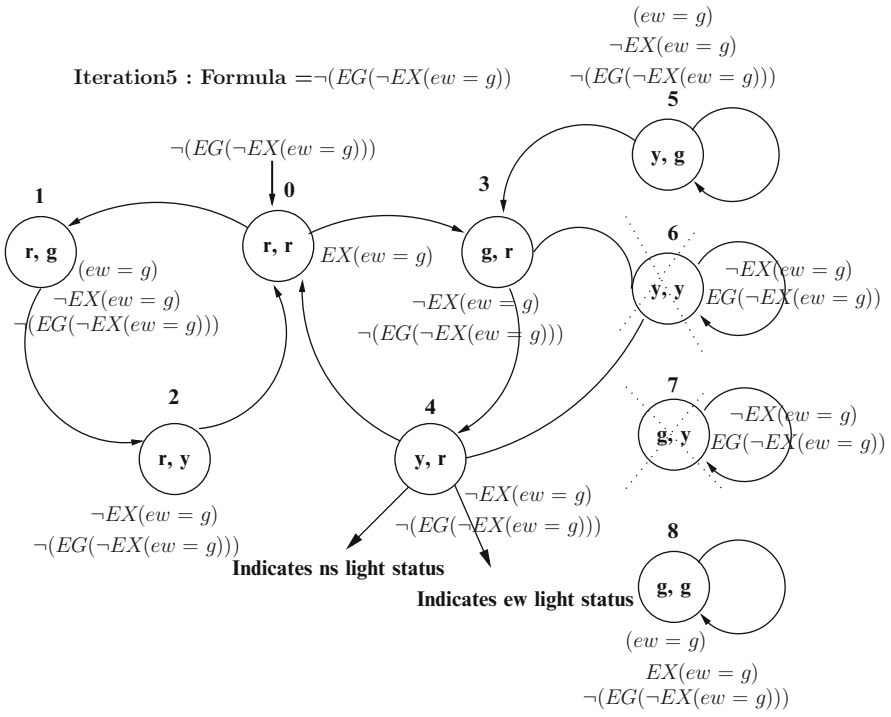



**Fig. 3.9** Fifth iteration of the verification of the property $\neg\texttt{EG}[\neg\texttt{EX}(ew = g)]$

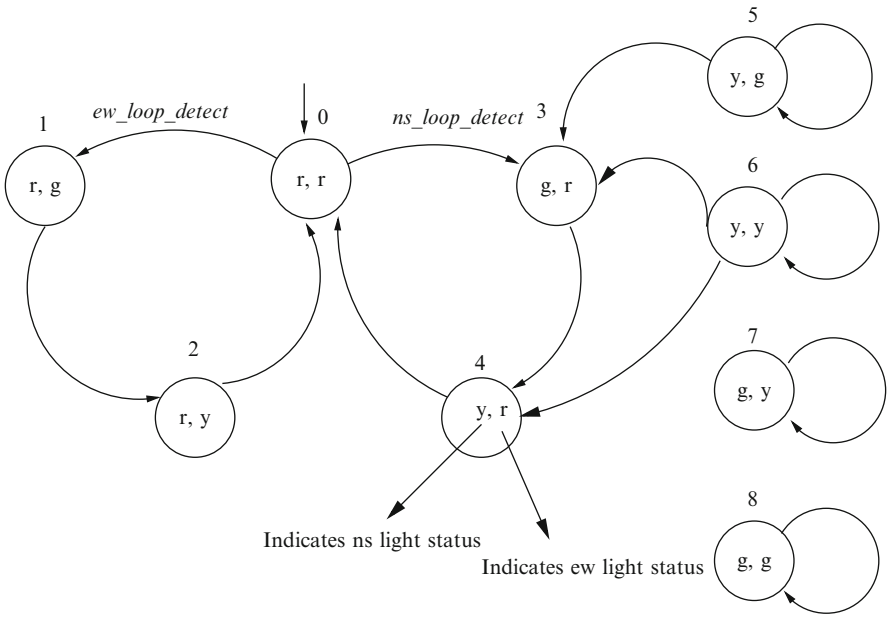



**Fig. 3.10** A traffic light controller with loop detectors

Consider that a module $\mathscr{M}$ is specified, and the ways in which it can interact and/or effect rest of the system is also given (interfaces). Then question is whether for a given temporal property $\varphi$, does $\forall \mathscr{E} : \mathscr{M} \times \mathscr{E} \models \varphi$ hold. We will use the following notational convenience to denote the this verification obligation: $\mathscr{M} \models_o \varphi$.

In the above, $\mathscr{E}$ denotes the (unknown/unspecified) behavior of the rest of the system, which includes the module $\mathscr{M}$. It is often referred to as the *environment* of the module $\mathscr{M}$. The environment can never disable or interfere with the events that are internal to $\mathscr{M}$; it can only affect $\mathscr{M}$ behavior which depend on the rest of the system (possibly obtaining input). It is immediate that often enumerating all possible instantiations of $\mathscr{E}$ and deploying model checking is not an effective or viable solution. Instead, specialized technique need to be developed which can address the above verification objective by just relying on the information at hand—specification of the module and its interfaces with the rest of the system. Such a technique is referred to as *module checking* [KV96], introduced by Kuperman and Vardi in 1996.

The complexity for CTL module checking is `EXPTIME`-complete (unlike polynomial complexity of CTL model checking). The reason is attributed to the presence of temporal operator `E` in the logic of CTL, which allows to express properties of states which contains *at least* one path with certain path property. The presence of temporal properties `EG`, `EF` and `EU` requires that all possible nondeterministic choices in the model must be considered [KVW01], which in turn, results in increased computational complexity. For universally quantified properties

containing only AX, AG, AF and AU, the complexity of module checking coincides with that of model checking [KV97].

While the complexity of CTL module checking sounds like bad news, a practical implementation of module checking with better average time complexity is possible by the following observation. The negation of the verification objective, i.e., a module does not satisfy a CTL property ($\mathscr{M} \not\models_o \varphi$), can be directly addressed by checking for existence of an environment $\mathscr{E}$ for which $\mathscr{M} \times \mathscr{E} \not\models \varphi$. The environment $\mathscr{E}$ can be viewed as the *witness* under which the $\mathscr{M}$ does not satisfy the property under consideration. In [BRS07], we have proposed a tableau-based technique for which such an environment can be identified. In the following sections, we will discuss the tableau-based technique, which will form the basis of module-based SoC design.

## 3.2.1 Tableau-Based Local Module Checking

The tableau-based local module checking is based on the following concepts:

1. Kripke structure representation of the module
2. Kripke structure representation of a generic environment of the module
3. Composition between the module and its environment

In the following subsections, we discuss each of the above concepts.

### 3.2.1.1 Modules as Kripke Structure

Kripke structure, augmented appropriately to capture the interfaces, is used to model the behavior of $\mathscr{M}$.

**Definition 3.6 (Modules as Kripke structures).** A module Kripke structure (MKS) is a tuple $\langle AP, S, s_0, \Sigma, R, L \rangle$ where:

1. $AP$ is a set of atomic propositions.
2. $S = S_I \uplus S_E$ is a finite set of states where $S_I$ is the set of internal system states and $S_E$ is the set of environment-controlled states of the module.
3. $s_0 \in S$ is a unique start state.
4. $\Sigma = \Sigma_I \uplus \Sigma_E$ is a set of events where $\Sigma_I$ is the set of all internal events and $\Sigma_E$ is the set of all external events.
5. $R \subseteq S \times B(\Sigma) \times S$ is a total transition relation where $B(\Sigma)$ is the set of all Boolean formulas over the elements of $\Sigma$. Further, for any transition $(s, b, s') \in R$, if $s \in S_I$, then $b \in B(\Sigma_I)$. Similarly, if $s \in S_E$, then $b \in B(\Sigma_E)$.

(continued)

**Box 3.8 (continued)**

6. $L : S \rightarrow 2^{AP}$ is the state labeling function mapping each state to a set of propositions.

In the above definition, the module specification partitions the set of states into internal and system. When the module is in an internal state, it does not interact with its environment; on the other hand, when the module is in a system state, it can interact with its environment. Furthermore, the module evolves from one state to another when a guard over the events at the source state are satisfied. These events are either provided by the environment (when the source state is environment state) or generated internally. The events and the environment-controlled states correspond to interfaces between the module $\mathscr{M}$ and its environment.

Consider the Kripke structure representation of AMBA ASB (advanced system bus) arbiter.[2] The arbiter is meant to ensure that only one of the two masters in the system is allowed to access the bus at any given instance. The active master is called the *bus master* and is allowed to access all SoC resources (slaves and bus data) while the other master waits. The shaded states are internal states; others are environment-controlled. The propositions $M = 1$ and $M = 2$ denote whether master 1 or 2 has been granted access. The propositions $R1$ and $R2$ denote whether a request from the master 1 or 2 has been received and the request has not been serviced by the arbiter. The events $req1$ and $req2$ are external events that are supplied by the environment; $req1$ is supplied by the master 1, $req2$ is supplied by master 2. The events $switch$ and $noswitch$ are internal events, which enables the switching of access different masters and disables the switching of access, respectively.
A property of interest for this module is

*The arbiter eventually gives bus access to master 1.*

The property can be expressed in the logic of CTL as $\texttt{AGEF}(M = 1)$, which states that every state of the system (arbiter with the masters) eventually ends in a state where $M = 1$. Note that, this behavior is satisfied by the system, only when the external events of the arbiter $req1$ are provided by its environment (note that, internal events such as $switch$ or $noswitch$ are under the control of the arbiter itself). As explained above, module checking amounts to identifying whether the arbiter module in any environment satisfies the property.

[2]This example is an adapted version of the NuSMV implementation of a two-master arbiter presented and verified in [RMK03] in Fig. 3.11, earlier shown in Fig. 2.2. The bus arbiter is simplified by assuming that the masters never perform burst transfers (multiple transfers in the same transaction). Further, the example is restricted to a 2-master arbiter for the sake of concise illustration of the concepts provided in the chapter.

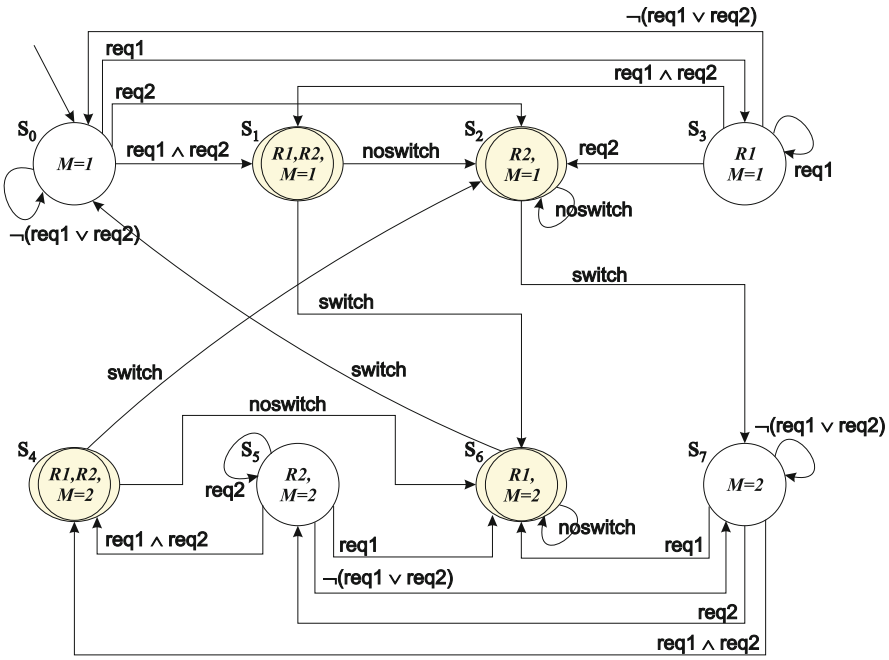



**Fig. 3.11** The AMBA bus arbiter

### 3.2.1.2 Environment as Kripke Structure

Recall that, the environment of a module represents how the rest of the system interacts with the module; it cannot disable any internal events of the module and can disable environment-controlled events. The environment can be also specified using the augmented Kripke structure used to model the module (see Definition 3.6). Formally,

**Definition 3.7.** An environment $\mathscr{E} = \langle AP_{\mathscr{E}}, S_{\mathscr{E}}, s_{0\mathscr{E}}, \Sigma_{\mathscr{E}}, R_{\mathscr{E}}, L_{\mathscr{E}} \rangle$ is such that

1. $AP_{\mathscr{E}} = \emptyset$.
2. $S_{\mathscr{E}}$ is a finite set of states.
3. $s_{0\mathscr{E}}$ is the initial state.
4. $\Sigma_{\mathscr{E}} \subseteq \Sigma_{\mathscr{M}}$.
5. $R_{\mathscr{E}} \subseteq S_{\mathscr{E}} \times B(\Sigma_{\mathscr{E}}) \times S_{\mathscr{E}}$.
6. $L(e) = \emptyset$ for any state $e \in S_{\mathscr{E}}$.

The states of an environment do not contain any labels $AP_{\mathscr{E}} = \emptyset$ and therefore, the labeling function denotes that no proposition is satisfied at any state (note, however, propositional constant `true` holds in all states).

#### 3.2.1.3 Composition of a Module and Its Environment

Given a module $\mathscr{M}$ and an environment $\mathscr{E}$ specified as Kripke structure, the system behavior from the perspective of $\mathscr{M}$ can be described as a *composition*. The composition follows specific rules by which $\mathscr{M}$ and $\mathscr{E}$ interact. Formally,

**Definition 3.8 (Parallel composition).** Given a module $\mathscr{M} = \langle AP_{\mathscr{M}}, S_{\mathscr{M}}, s_{0\mathscr{M}}, R_{\mathscr{M}}, \Sigma_{\mathscr{M}}, L_{\mathscr{M}} \rangle$, its environment $\mathscr{E} = \langle AP_{\mathscr{E}}, S_{\mathscr{E}}, s_{0\mathscr{E}}, \Sigma_{\mathscr{E}}, R_{\mathscr{E}}, L_{\mathscr{E}} \rangle$, their parallel composition resulting in $\mathscr{E} \times \mathscr{M} \equiv \mathscr{P} = \langle AP_{\mathscr{P}}, S_{\mathscr{P}}, s_{0\mathscr{P}}, \Sigma_{\mathscr{P}}, R_{\mathscr{P}}, L_{\mathscr{P}} \rangle$ is defined as follows:

1. $AP_{\mathscr{P}} = AP_{\mathscr{M}}$.
2. $S_{\mathscr{P}} \subseteq \{(e,s) : e \in S_{\mathscr{E}} \wedge s \in S_{\mathscr{M}} \wedge \mathscr{N}(e,s)\}$, where $\mathscr{N} \subseteq S_{\mathscr{E}} \times S_{\mathscr{M}}$ is an *environment refinement relation* such that $(e,s) \in \mathscr{N}$ when the following holds
   a. $e$ must have at least one outgoing transition $e \xrightarrow{\cdots} e'$ for some $e' \in S_{\mathscr{E}}$.
   b. If $s$ is environment-controlled then for every $e \xrightarrow{b} e'$, there exists $s \xrightarrow{b} s'$ and $(e',s') \in \mathscr{N}$.
   c. If $s$ is internal state then for every $s \xrightarrow{b} s'$ there exists $e \xrightarrow{b} e'$ and $(e',s'_e) \in \mathscr{N}$.
3. $s_{0\mathscr{P}} = (s_{0\mathscr{M}}, s_{0\mathscr{E}})$.
4. $\Sigma_{\mathscr{P}} = \Sigma_{\mathscr{E}}$.
5. $R_{\mathscr{P}} \subseteq S_{\mathscr{P}} \times \Sigma_{\mathscr{P}} \times S_{\mathscr{P}}$ is a total transition relation where for each state $s_{\mathscr{P}} \in S_{\mathscr{P}}$ such that $s_{\mathscr{P}} = (e,s)$,
$$\left[ e \xrightarrow{b} e' \ \wedge \ s \xrightarrow{b} s' \right] \Rightarrow (e,s) \xrightarrow{b} (e',s')$$
6. $L_{\mathscr{P}}(s_{\mathscr{M}}, s_{\mathscr{E}}) = L_{\mathscr{M}}(s_{\mathscr{M}})$.

In the above, note that the atomic propositions describing states in the composition are same as those describing the states in the module. Recall that (see Definition 3.7), there are no propositions that hold in any state of the environment Kripke structure.

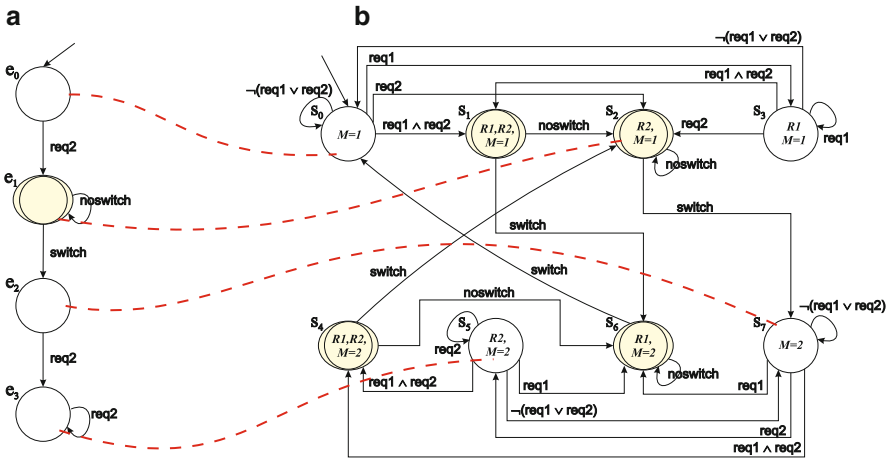



**Fig. 3.12** A sample environment $\mathscr{E}$ for the ASB arbiter module

The second item describes the states in the composition as a tuple containing states from the module $\mathscr{M}$ and its environment $\mathscr{E}$, which ensures that $\mathscr{E}$ is *compatible* with $\mathscr{M}$. The tuple is required to satisfy a refinement relation $\mathscr{N}$. This relation ensures that the environment state is not a deadlocked state (item 2a), the environment state does not contain labeled transitions that are not present in the state of the module (item 2b), and the environment state replicates all transitions from the corresponding state of the module if the latter is an internal state (item 2c).

The start state is described in terms of the start states of $\mathscr{M}$ and $\mathscr{E}$ (item 3). The events of the composition are same as the events in $\mathscr{E}$ (item 4) and the transition relation is based on the matching transitions between $\mathscr{M}$ and $\mathscr{E}$ (item 5). Finally, the labeling function coincides with the labeling function of $\mathscr{M}$.

Note that, based on the refinement relation and the transition relation, in the composition the transitions from the internal states of the module are never disabled (items 2a, c) and every transitions from the environment states of the module may not be enabled (items 2a, b). Consider the environment in the Fig. 3.12. The red dotted lines represent the refinement relation; e.g., $(e_0, s_0) \in \mathscr{N}$. The state $e_0$ has one outgoing transition on the event $req2$ (interacts with the ASB arbiter module via $req2$) and disables all other transitions in the module ($req1 \wedge req2$, $\neg(req1 \vee req2)$, and $req1$). Also note that, $(e_1, s_2) \in \mathscr{N}$ and $e_1$ does not disable any of the transitions from the internal state $s_2$. Based on the composition definition, the behavior of the composition obtained from the module and its environment is illustrated in Fig. 3.13.

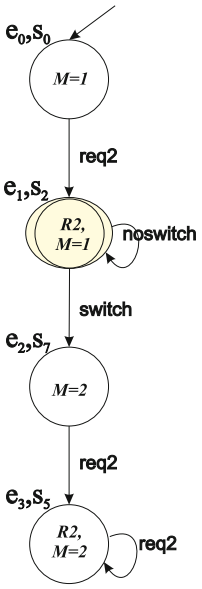



**Fig. 3.13** The behavior of the arbiter under the environment $\mathscr{E}$ presented in Fig. 3.12

#### 3.2.1.4 Tableau for Module Checking and Environment Generation

Recall that (Sect. 3.2) module checking amounts to finding a witnessing environment $\mathscr{E}$ such that $\mathscr{M} \times \mathscr{E} \models \neg\varphi$, which holds only when $\mathscr{M} \not\models_o \varphi$. In this section, we discuss a proof system for establishing whether a module satisfies a property and generate a witnessing environment if it does not.

> The constraint imposed on this technique is that the CTL property is expressible in negative normal form, one where negations are only applied on propositions.

The proof system is tableau-based, which is constructed using a set of tableau-rules of the following form:

$$\frac{\textit{Conclusion}}{\textit{Premise}}$$

In the above, the conclusion is the proof objective (or intermediate proof objective) and the premise describes the requirements that imply the objective.

Given the objective to verify $\mathscr{M} \not\models_o \varphi$, i.e, $\mathscr{M} \times \mathscr{E} \not\models \varphi$, the proof objective is equivalently written as $(e_0, s_0) \models \neg\varphi$, where $e_0$ and $s_0$ are start states of $\mathscr{M}$ and the witnessing environment $\mathscr{E}$, respectively. Starting with this proof objective as the *conclusion*, tableau-rules are applied to obtain the *premise*. In the subsequent steps, the newly obtained premise becomes the new proof objective on which tableau-rules

are again applied to obtain another premise. The process is repeated until a proof objective is obtained

1. For which no tableau-rule can be applied leading to failure to prove the original proof obligation. This denotes that the applied tableau-rules cannot prove the proof obligation.
2. For which there is a tableau-rule with no premise denoting that the proof obligation has been successfully released. The applied tableau-rules are successful in proving the obligation.

There are two sets of tableau-rules in the proof system depending on whether the state of the module appearing in the proof obligation is an internal state or an environment-controlled state. The difference is primarily because the premise of the tableau-rule captures (imposes) the property of the environment. When the proof obligation contains an internal state, the corresponding state of the environment must enable all the transitions from that state; while when the proof obligation contains module state that is environment-controlled, the corresponding environment state will allow a non-trivial subset of transitions from the environment-controlled state.

**Tableau-Rules for Internal State.** Figure 3.14 presents the tableau-rules, which are applied when the state of the module in the conclusion is an internal state. Consider first the $prop_{I1}$-rule, which states that $(e, s_s)$ satisfies $p$ only when $p$ holds in the state $s_s$; no other proof obligations are imposed on the states. Note that, environment states are not labeled with any proposition; the labeling of the states in the composition of the environment and the module solely depends on the labeling of the states in the module. The negation of the proposition is considered in $prop_{I2}$-rule. The $\wedge$-rule states a conjunctive property is satisfied only when each conjunct is satisfied. The $\vee$-rules correspond to disjunctive properties. The $unr_{I,eu}$-rule states that $\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)$ is satisfied only when the equivalent expansion is satisfied. The expansion is based on the fact that

$$\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2) \equiv \varphi_2 \vee (\varphi_1 \wedge \mathtt{EXE}(\varphi_1\ \mathtt{U}\ \varphi_2))$$

Similar, equivalences are used for the rules $unr_{I,au}$, $unr_{I,eg}$ and $unr_{I,ag}$. These rules correspond to *unrolling* of the properties.

The $unr_{I,ex}$-rule states that there exists some *matching* (over $b$) transition from $e$ to a destination state $e_b$, such that $(e_b, s_b)$ satisfies $\varphi$. The environment state is also required to have transitions matching the other transitions from $s_I$. The $unr_{I,ax}$, on the other hand, requires that each destination composed of $e_b$ and $s_b$ satisfies $\varphi$.

The above rules directly follows from the semantics of CTL as described in Sect. 3.1.3.

Finally, the *reorg*-rule is used re-organize the expression from set-based notation to logical notation; the set of properties in the rhs of $\models$-notation is considered to be in conjunctive form.

$$reorg\ \frac{(e,s_s)\ \models\ \{\varphi_1,\ldots,\varphi_n\}}{(e,s_s)\ \models\ \varphi_1\ \ldots\ (e,s_s)\ \models\ \varphi_n}$$

$$prop_{I1}\ \frac{(e,s_s)\ \models\ p}{.}\ p\in L(s_s) \qquad prop_{I2}\ \frac{(e,s_s)\ \models\ \neg p}{.}\ p\notin L(s_s) \qquad \wedge\ \frac{(e,s_s)\ \models\ \varphi_1\wedge\varphi_2}{(e,s_s)\ \models\ \varphi_1 \quad (e,s_s)\ \models\ \varphi_2}$$

$$\vee_1\ \frac{(e,s_s)\ \models\ \varphi_1\vee\varphi_2}{(e,s_s)\ \models\ \varphi_1} \qquad \vee_2\ \frac{(e,s_s)\ \models\ \varphi_1\vee\varphi_2}{(e,s_s)\ \models\ \varphi_2}$$

$$unr_{I,eu}\ \frac{(e,s_s)\ \models\ \mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)}{(e,s_s)\ \models\ \varphi_2\vee(\varphi_1\wedge\mathtt{EXE}(\varphi_1\ \mathtt{U}\ \varphi_2))} \qquad unr_{I,au}\ \frac{(e,s_s)\ \models\ \mathtt{A}(\varphi\ \mathtt{U}\ \varphi_2)}{(e,s_s)\ \models\ \varphi_2\vee(\varphi_1\wedge\mathtt{AXA}(\varphi_1\ \mathtt{U}\ \varphi_2))}$$

$$unr_{I,eg}\ \frac{(e,s_s)\ \models\ EG\varphi}{(e,s_s)\ \models\ \varphi\wedge EXEG\varphi} \qquad unr_{I,ag}\ \frac{(e,s_s)\ \models\ \mathtt{AG}\varphi}{(e,s_s)\ \models\ \varphi\wedge\mathtt{AXAG}\varphi}$$

$$unr_{I,ex}\ \frac{(e,s_s)\ \models\ \mathtt{EX}\varphi}{\exists b\in B: e\xrightarrow{b}_{\mathscr{E}} e_b\ \wedge\ (e_b,s_b)\ \models\ \varphi,\ \forall b'\in B/b: e\xrightarrow{b'} e_{b'}\ \wedge\ (e_{b'},s_{b'})\models\mathtt{true}}\ B=\{b\mid s\xrightarrow{b}_{\mathscr{M}} s_b\}$$

$$unr_{I,ax}\ \frac{(e,s_s)\ \models\ \mathtt{AX}\varphi}{\forall b\in B: e\xrightarrow{b}_{\mathscr{E}} e_b\ \wedge\ (e_b,s_b)\models\varphi}\ B=\{b\mid s\xrightarrow{b}_{\mathscr{M}} s_b\}$$

**Fig. 3.14** Tableau rules for module checking system states

**Tableau-Rules for Environment-Controlled State.** Figure 3.15 presents that tableau-rules, which are applied when the state of the module in the conclusion is environmentally controlled. The *reorg*-rule re-organizes the property in the form of set; the conclusion holds when there is no property to be satisfied—set is empty (*emp*-rule). The $prop_{E1}$ and $prop_{E2}$ rules correspond to the presence of a proposition and a negation of a proposition in the set $C$. If one of the properties in the set is a conjunctive property, then the set is updated to contain the conjuncts ($\wedge$-rule). Similarly, $\vee$-rules correspond to disjunctive properties.

The $unr_{E,eu}$-rule states that if one of the properties in the set is $\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2)$ then in the premise the set is updated to include the expanded version of the property $\varphi_2 \vee (\varphi_1 \wedge \mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2))$. Similar expansions are present in rules $unr_{E,au}$, $unr_{E,eg}$ and $unr_{E,ag}$.

The final rule $unr_E$ is applied when none of the other rules can be applied, i.e., when the set of properties in the conclusion contains properties over AX and EX. This rule (similar to $unr_{I,ex}$ and $unr_{I,ax}$ in Fig. 3.14) demands a specific set of next states for the environment state $e$ as part of the premise of the rule. Recall that, the state $e$ can disable some transitions from the environment-controlled state $s_e$. Let $S$ be some set of tuples, each of which captures the enabled event and the destination state from $s_e$. The set of existential next-state properties $C_{ex}$ can be partitioned into $|S|$ partitions. The premise demands that for all properties in each partition, there exists some next state $(e_b, s_b)$ reachable via an enabled event $b$ that satisfies those properties as well as the universal next-state properties $C_{ax}$.

### 3.2.1.5 Finitizing Proofs

The proof system is constructed by application of tableau rules starting with a conclusion that need to be proved. A path in the proof system consists of one or more applications of tableau rules. A proof path can end successfully where the last application is $prop_{I1}$-, $prop_{I2}$- or *emp*-rule. On the other hand, a proof path can end unsuccessfully when the last application of tableau rule is unsuccessful either because the side condition is not satisfied or the premise is not immediately valid. Note however, that the above tableau-based proof system can be of infinite depth as the properties of the form EU, AU, EG, AG are unfolded without any base case. The finitization of the proof system relies on the fact that there are finite number of states in the module under consideration, and that the semantics of EU, AU and the semantics of EG, AG have least fixed point and greatest fixed point characterizations.

The fixed point representation of CTL formulas AU and EU are

$$\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2) \equiv Z =_\mu \varphi_2 \vee (\varphi_1 \wedge \mathtt{EX}Z) \quad \mathtt{A}(\varphi_1\ \mathtt{U}\ \varphi_2) \equiv Z =_\mu \varphi_2 \vee (\varphi_1 \wedge \mathtt{AX}Z)$$

We overload the logical operators to also denote the corresponding set operators ($\wedge$ and $\cap$, $\vee$ and $\cup$). Also, the formulas denote their semantics (e.g., $\varphi$ and set of states that satisfy $\varphi$ are identically represented).

$$reorg\ \frac{(e,s_e)\ \models\ \varphi}{(e,s_e)\ \models\ \{\varphi\}}$$

$$emp\ \frac{(e,s_e)\ \models\ \{\}}{.} \qquad prop_{E1}\ \frac{(e,s_e)\ \models\ \{p,C\}}{(e,s_e)\ \models\ C}\ p\in L(s_e) \qquad prop_{E2}\ \frac{(e,s_e)\ \models\ \{\neg p,C\}}{(e,s_e)\ \models\ C}\ p\notin L(s_e)$$

$$\wedge\ \frac{(e,s_e)\ \models\ \{\varphi_1\wedge\varphi_2,C\}}{(e,s_e)\ \models\ \{\varphi_1,\varphi_2,C\}} \quad \vee_1\ \frac{(e,s_e)\ \models\ \{\varphi_1\vee\varphi_2,C\}}{(e,s_e)\ \models\ \{\varphi_1,C\}} \quad \vee_1\ \frac{(e,s_e)\ \models\ \{\varphi_1\vee\varphi_2,C\}}{(e,s_e)\ \models\ \{\varphi_2,C\}}$$

$$unr_{E,eu}\ \frac{(e,s_e)\ \models\ \{\mathtt{E}(\varphi_1\ \mathtt{U}\ \varphi_2),C\}}{(e,s_e)\ \models\ \{(\varphi_2\vee(\varphi_1\wedge\mathtt{EXE}(\varphi_1\ \mathtt{U}\ \varphi_2))),C\}} \qquad unr_{E,au}\ \frac{(e,s_e)\ \models\ \{\mathtt{A}(\varphi_1\ \mathtt{U}\ \varphi_2),C\}}{(e,s_e)\ \models\ \{(\varphi_2\vee(\varphi_1\wedge\mathtt{AXA}(\varphi_1\ \mathtt{U}\ \varphi_2))),C\}}$$

$$unr_{E,eg}\ \frac{(e,s_e)\ \models\ \{\mathtt{EG}\varphi,C\}}{(e,s_e)\ \models\ \{\varphi\wedge\mathtt{EXEG}\varphi,C\}} \qquad unr_{E,ag}\ \frac{(e,s_e)\ \models\ \{\mathtt{AG}\varphi,C\}}{(e,s_e)\ \models\ \{\varphi\wedge\mathtt{AXAG}\varphi,C\}}$$

$$unr_E\ \frac{(e,s_e)\ \models\ C}{\exists S\subseteq NS:\exists\Pi=\{C_1,C_2,\ldots,C_{|S|}\}:\bigcup_{i\leq|S|}C_i=C_{ex}\ \wedge\ \forall C_i\in\Pi:\exists e_b:e\xrightarrow{b}_{\mathscr{E}}e_b\ \wedge\ (b,s_b)\in S\ \wedge\ (e_b,s_b)\models C_i\wedge C_{ax}}\begin{cases}C_{ax}=\{\varphi_k\mid \mathtt{AX}\varphi_k\in C\}\\ C_{ex}=\{\varphi_k\mid \mathtt{EX}\varphi_k\in C\}\\ NS=\{(b,s_b)\mid s\xrightarrow{b}_{\mathscr{M}}s_b\}\end{cases}$$

**Fig. 3.15** Tableau rules for module checking environment states

In the above equations, $\mu$ represents the sign of the equation and is used to denote the least fixed point and $Z$ is recursive variable whose valuation (set of model states) is the least fixed point computation of its definition. Similarly, the CTL formulas EG and AG can be represented by greatest fixed point recursive equations:

$$\texttt{EG}\varphi \equiv Z =_{\nu} \varphi \wedge \texttt{EX}Z \quad \texttt{AG}\varphi \equiv Z =_{\nu} \varphi \wedge \texttt{AX}Z$$

The fixed point computation proceeds by iteratively computing the approximations of $Z$ over the lattice of set of states in the model. A solution is reached only when two consecutive approximations are identical.

**Least Fixed Points.** The first approximation (say $Z_0$) of the least fixed point variable is the empty set (bottom of the lattice). The second approximation corresponding to EU is computed using the least fixed point equation as follows:

$$Z_1 = \varphi_2 \vee (\varphi_1 \wedge \texttt{EX}Z_0)$$

The value of $Z_1$ captures the set of states that satisfy $\varphi_2$. Note that the second disjunct $\varphi_1 \wedge \texttt{EX}Z_0$ is empty set (false) because $\texttt{EX}Z_0$ is empty set (false). Subsequently, $Z_2$ captures the set of states that satisfy $\varphi_2$ or that satisfy $\varphi_1$ and have at least one next state satisfying $\varphi_2$. Proceeding further, the successive computation of $Z_i$'s terminates, when $Z_k = Z_{k+1}$ for some $k$. It can be shown that the value of $Z_k$ is the semantics of $\texttt{E}(\varphi_1 \texttt{ U } \varphi_2)$.

Intuitively, in order to prove the satisfiability of a property such as $\texttt{E}(\varphi_1 \texttt{ U } \varphi_2)$, which has a least fixed point characterization, one needs to show that $\varphi_2$ is satisfied in finite number of transitions. The same argument is valid for $\texttt{A}(\varphi_1 \texttt{ U } \varphi_2)$ (recall that, unlike EU, AU universally quantifies path).

If the tableau-rules in Fig. 3.14 are applied on a pair of module state and least fixed point property more than once, the second application is replaced with `false`. If the tableau-rules in Fig. 3.15 are applied on a pair of module state and a set of properties $C$ more than once and the set $C$ contains a least fixed point property, then the second application is replaced with `false`. This is because application of tableau-rules does not generate a finite path in the proof system for proving the satisfiability of the least fixed point property.

**Greatest Fixed Points.** The first approximation ($Z_0$) of the greatest fixed point variable is the set of all states (top of the lattice). The second approximation corresponding to EG is computed using the greatest fixed point equation as follows:

$$Z_1 = \varphi \wedge \texttt{EX}Z_0$$

The value of $Z_1$ captures the set of states that satisfy $\varphi$ and has some next state. Subsequently, $Z_2$ captures the set of states that satisfy $\varphi$ and that have at least one next state that satisfy $\varphi$. Proceeding further, the successive computation of $Z_i$'s terminates, when $Z_k = Z_{k+1}$ for some $k$. Essentially, $Z_k$ captures the set of states which belong to some infinite path where each state in the path satisfies $\varphi$. Note

that, as the number of states in a model is finite, an infinite path can be only obtained via a loop in the model. It can be shown that the value of $Z_k$ is the semantics of $\texttt{EG}\varphi$.

Intuitively, in order to prove the satisfiability of a property such as $\texttt{EG}\varphi$, which has a greatest fixed point characterization, one needs to prove that $\varphi$ is satisfied in all states of an infinite path. The same argument is valid for $\texttt{AG}\varphi$ (the difference is in terms of universal quantification of paths).

If the tableau-rules in Fig. 3.14 are applied on a pair of module state and greatest fixed point property more than once, the second application is replaced with `true`. This replacement denotes that an infinite path is obtained via a loop. Similarly, if the tableau-rule in Fig. 3.15 are applied on a pair of module state and set of properties $C$ more than once and $C$ contains only greatest fixed point properties, then the second application is replaced with `true`.

The above strategy based on fixed point characterization of CTL properties finitizes the tableau.

#### 3.2.1.6 A Simple Example

Consider the example module in Fig. 3.16a. Assume that all states are environment controlled and the proposition $p$ is true at states $s_1$, $s_4$, $s_5$ and $s_6$. The obligation is to prove $(e_0, s_0)$ satisfy the set of properties: $C = \{\texttt{AXEX}p, \texttt{EXEX}\neg p, \texttt{EX}p\}$. Fig. 3.16b presents the possible choices made by the application of the rule $unr_E$ as described in Fig. 3.15. The $\surd$ denotes the partition which satisfies the premise of the tableau rule.

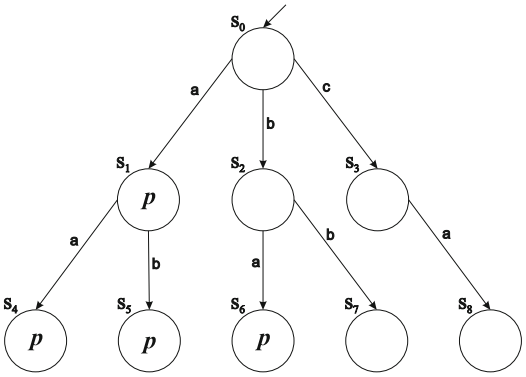
#### 3.2.1.7 Complexity

The complexity of the generating the proof system based on the tableau-rules is driven by the $unr_E$-rule in Fig. 3.14. In this rule, all possible subsets of set of next states are considered ($k = 2^{|S|}$, where $|S|$ is the set of states in the module $\mathscr{M}$). Then, for each subset, all possible partitions of the set of next-state obligations ($C_{ex}$) are considered. Note that, the size of $C_{ex}$ is of the order of $n = |\varphi|$ ($\varphi$ is the property under consideration). Note that, given $n$ elements, the number of ways it can be partitioned into $k$ groups where the $i$th group contains $m_i$ elements is $P(m_1, m_2, \ldots, m_k) = \left(\dfrac{n!}{m_1!m_2!\ldots m_k!}\right)$. Therefore, the number of possible choices considered in $unr_E$ is of the order of $O(n! \times k!)$.

#### 3.2.1.8 Soundness and Completeness

**Theorem 3.1.** *Given a module $\mathscr{M}$ with start state $s_0$ and CTL property $\varphi$ in negative normal form, a successful tableau is obtained and an environment with start state $e_0$ is generated for the $(e_0, s_0) \models \neg\varphi$ if and only if $s_0 \not\models_o \varphi$.*

**a**




**b**

ToProve: $(e_0, s_0) \models C$, $C = \{\texttt{AXEX}p, \texttt{EX}p, \texttt{EXEX}\neg p\}$

$C_{ax} = \{\texttt{EX}p\}, C_{ex} = \{p, \texttt{EX}\neg p\}, NS = \{s_1, s_2, s_3\}$

| $S$ | $\Pi$ | Premise | Result |
|---|---|---|---|
| $\{s_1\}$ | $\{p, \texttt{EX}\neg p\}$ | $(e_1, s_1) \models \{\texttt{EX}p, \texttt{EX}\neg p, p\}$ | Does not satisfy $\texttt{EX}p$ |
| $\{s_2\}$ | $\{p, \texttt{EX}\neg p\}$ | $(e_1, s_1) \models \{\texttt{EX}p, \texttt{EX}\neg p, p\}$ | Does not satisfy $p$ |
| $\{s_1, s_2\}$ | $\{\{p\}, \{\texttt{EX}\neg p\}\}$ | $(e_1, s_1) \models \{\texttt{EX}p, p\}$<br>$(e_2, s_2) \models \{\texttt{EX}p, \texttt{EX}\neg p\}$ | $\surd$ |
| | $\{\{\texttt{EX}\neg p\}, \{p\}\}$ | $(e_1, s_1) \models \{\texttt{EX}p, \texttt{EX}\neg p\}$ | Does not satisfy $\texttt{EX}p$ |
| | | $(e_2, s_2) \models \{\texttt{EX}p, p\}$ | Does not satisfy $p$ |
| ⋮ | ⋮ | ⋮ | |

**Fig. 3.16** Illustration of $unr_E$ tableau rule (**a**) An example module. (**b**) Possible choices during application of rule $unr_E$

*Proof.* First, note that the generated environment is compatible with the module, i.e., it does not disable any transitions from the internal states and it is capable of enabling subset of transitions from the environment controlled states. This is ensured by

1. The tableau rules for the internal states in Fig. 3.14, where the environment moves for every transition from the module state.
2. The tableau rules for the environment controlled states in Fig. 3.15, where the environment selects some subset of transitions from the module state ($unr_E$-rule).

Second, note that the tableau rules follow the semantics of the CTL properties, which ensures the soundness of each rule.

Finally, there is a rule for each syntactic form of the CTL properties, which ensures the completely of the tableau-based proof system. □

## 3.3 Conclusion

In this chapter, we have discussed the formalisms for representing and reasoning about system behaviors expressed as finite state Kripke structures. The central theme is the usage of model and module checking techniques using temporal properties in the logic of computation tree temporal logic (CTL). This chapter forms the basis of the techniques for automatically analyzing and constructing SoCs (Systems on Chip) with correctness guarantees. Here we summarize the motivation behind the usage of formalisms and model/module checking discussed in this chapter for SoC design.

SoCs are complex embedded systems built using pre-verified components (called intellectual property blocks or IPs) chosen from available IP libraries. The various IPs of an SoC may be interconnected via a central system bus on a single chip. Several problems need to be addressed before such interconnection can be realized in practice; collectively the problems are classified as *protocol mismatches* referring to inherent communication incompatibilities in IPs. Protocol mismatches include *control mismatches*, which disallows correct exchange of control signals between IPs; *data-width mismatches*, which restricts exchange of data; and *interface mismatches*, which disallows connecting two or more IPs. In short, mismatches must be resolved before SoCs with desired system-level behavior can be built from existing IPs. A possible solution to address protocol mismatches relies on manually modifying all the IPs participating in an SoC. However, this process of manual modification is usually ineffective because firstly, it requires significant time and effort to modify complex IPs, and secondly, if requirements change later in the design cycle, further repetitions of manual modification might be required. An alternate approach is to develop a *converter* that acts as an intermediary between IPs using mismatched protocols. The advantages of this approach are that (a) converters can be generated automatically and (b) changes to the requirements can be incorporated in the design by simply updating the converter. This technique is referred to as the protocol conversion technique. Protocol conversion involves two aspects: verifying the existence of a converter that can address protocol mismatches (covertibility verification) and generating one, if it exists (converter synthesis). In the following chapters, we discuss how module checking [KV96] (see Sect. 3.2) is adapted for convertibility verification and converter synthesis in protocol conversion.

Next chapter will deal with the process of systematically obtaining the requirements for SoCs at the system-level. This chapter will also provide a variant of Kripke structures, called synchronous Kripke structures (SKS), to represent the on-chip communication protocols of IPs.

# Chapter 4
# Models for SoCs and Specifications

Formal methods have the potential to automate many processes of the SoC design cycle and reduce time-to-market. However, the larger scale adoption of formal methods is hampered due to two primary reasons. The first reason is the well-known *state space explosion* phenomenon. State explosion is experienced when formal models, obtained from translating systems, have too many states. Consequently, this affects analysis time. Exponential growth of states may occur due to the inclusion of data variables in a model. A 32-bit integer can result in a $2^{32}$ times increase in state space if all its values are explicitly (and naïvely) encoded in the model. Also, a system composed of multiple sub-systems may result in a model that faces combinatorial state explosion. For instance, a model containing two 5-state components can have 25 states. Thankfully, many techniques such as symbolic representation [JEK+90], compositional reasoning [CGP00], abstract interpretation [Cou96] etc., have given birth to many optimised formal models and formal methods that can be used to analyse very large models. The second reason for the limited adoption of formal methods are human barriers to their use [Par10]. In many cases, SoC designers may not have the necessary expertise to write formal properties, or to use formal methods effectively. Designers require tool support to use formal methods in more intuitive ways.

Let us explore the above issues in a SoC context. Figure 4.1 presents a SoC containing two masters and a slave connected using the AMBA system bus (ASB). It is desired that the master *Producer* writes data to the memory slave that the master *Consumer* reads. The masters gain access to the bus (and the memory slave) through the bus arbiter. Each IP in this SoC executes using its own clock. In order to verify the correctness of this system with respect to intended desirable functionality, we can use formal methods such as model checking (Chap. 3). In order to use model checking, we require that the whole system is described formally using a suitable model and that desired functionality is specified using temporal logic, such as CTL (see Chap. 3).

The choice of formal model to describe the SoC shown in Fig. 4.1 is an important one. We would like to choose the most compact model to minimize verification time. At the same time, the model must contain sufficient information

R. Sinha et al., *Correct-by-Construction Approaches for SoC Design*,
DOI 10.1007/978-1-4614-7864-5_4, 

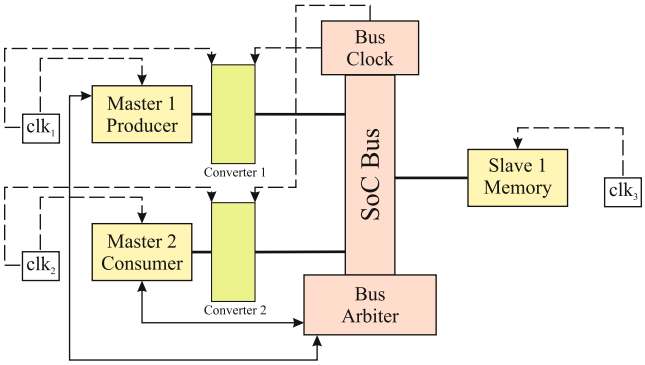



**Fig. 4.1** A producer-consumer SoC based on AMBA ASB

for meaningful verification. To counter state explosion, we must focus on encoding data and handling combinatorial factors. As discussed previously in Chap. 2, we can model control, data and timing aspects of SoCs in a compact manner using finite state machines. The informal model presented in Chap. 2 is formalised, based on our experience with SoC modelling and verification, in this chapter as *synchronous Kripke structures* [SRBS09]. Synchronous Kripke structures are precise and compact models that encode control, data, and timing characteristics of IPs and SoCs precisely. This allows verifying commonly-used properties efficiently. Synchronous Kripke structures are defined and explained in Sect. 4.1. Synchronous Kripke structures are SoC-specific models that are extensions of the more generic Kripke structure models presented in Chap. 3.

We resolve human barriers to formal methods by providing an intuitive framework to write properties. A more *natural* way to write properties, as compared to temporal logic, is the use of natural language. Consider the following sentence

> "The arbiter always eventually grants access to master 1."

Such a property is easier to write and understand than the CTL specification $Arbiter \models \texttt{AGAF}(GNT1Out)$. Moreover, this same property can be modified just slightly to create the following natural language specification

> "The arbiter always eventually grants access to *master 2*."

The two properties shown above use the same *pattern*. The text "The Arbiter always eventually grants access to" is repeated in both cases, and the only difference is the name of the IP (master 1 or master 2). We call such recurring patterns

*boilerplates*. Used historically in programming, a boilerplate is known as an oft-repeated pattern in programs. Similarly, in the SoC context, we capture the repeating patterns in properties using boilerplates, such as

> "The Arbiter always eventually grants access to ⟨*system*⟩."

This boilerplate is an incomplete sentence with the *place holder* ⟨*system*⟩. The place holder can be replaced by the name of any master in any SoC to create a customised property. We can get designers to complete such natural language boilerplates using an appropriate software tool. We can then extract the CTL properties corresponding to the completed boilerplates automatically, without user involvement. *SoC Boilerplates* allow mitigating the human barriers to specifying formal properties, and are described in Sect. 4.2.

## 4.1 IP Modelling Using Synchronous Kripke Structures

### *4.1.1 Synchronous Kripke Structures*

SoC behaviour can be captured by modelling *IP protocols* and their interactions. IP protocols provide details about IP interfaces, or how IPs interact in a SoC, and abstract out any non-relevant internal details. As discussed previously in Sect. 2.2, a formal model for IP protocols (addressed as IPs henceforth) must be able to model *control flow*, *data flow*, and *timing* characteristics precisely. In this section, we formalise a finite-state machine based model, called a *synchronous Kripke structure*. We first present this model intuitively before defining it.

Figure 4.1 contains a memory slave IP, which reads and writes data in the SoC using a memory controller IP. In this case, the protocol of the memory slave IP is simply the protocol of the memory controller. The memory controller IP has two sub-protocols: read and write. Figure 4.2 presents the reader protocol of the memory controller shown in Fig. 4.1. This protocol captures the operation of reading data into the memory. The protocol is described as a finite state machine, which has an initial state $u_0$. Control flow within the protocol is modelled as transitions between states,

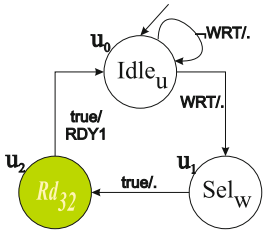



**Fig. 4.2** The synchronous Kripke structure for the slave reader IP

with signal exchanges described as transition labels. For example, a transition from state $u_0$ to state $u_1$ happens when the input control signal `WRT` is present. State labels, such as $Idle_u$ of state $u_0$, are atomic propositions that can be used for verification. State labels describe the status of the model in different states, such as the values of control and data variables. Data flow is modelled by special data labels. For example, the label $Rd_{32}$ of state $u_2$ indicates that when the model enters this state, the protocol reads 32-bits of data from the data bus. Timing is captured by assuming that transitions between states synchronize with respect to a common clock.

We now present a formal definition for synchronous Kripke structures (SKS), and then describe their features in more detail.

**Definition 4.1 (SKS).** A Synchronous Kripke structure SKS is a *finite state machine* represented as a tuple $\langle AP, S, s_0, I, O, R, L, clk\rangle$ where:

- $AP = AP_{control} \uplus AP_{data}$ is a set of propositions partitioned into the set of control labels $AP_{control}$ the set of data labels $AP_{data}$.
- $S$ is a finite set of states.
- $s_0 \in S$ is the initial state.
- $I$ is a finite, non-empty set of inputs.
- $O$ is a finite, non-empty set of outputs.
- $R \subseteq S \times \{\mathtt{t}\} \times B(I) \times 2^O \times S$ is the transition relation where $B(I)$ represents the set of all boolean formulas over $I$. The event $\mathtt{t}$ represents ticking of the clock $clk$.
- $L : S \to 2^{AP}$ is the state labelling function.

A SKS has a finite set of states $S$ with a unique start state $s_0$. Each state $s$ is labelled by a subset of the atomic propositions in $AP$ using the state labelling function $L$. $AP$ is partitioned into two sets: the set of control propositions $AP_{control}$, and propositions indicating data flow in $AP_{data}$. All transitions in a Kripke structure trigger with respect to the *ticks* of the clock $clk$ and a boolean formula over the set of inputs. This formula is evaluated after sampling inputs at the ticks of $clk$. A SKS transition $(s, \mathtt{t}, b, o, s') \in R$ is described using the short-hand $s \xrightarrow{b/o} s'$. This transition triggers when the formula $b \in B(I)$ evaluates to true (depending on sampled inputs), and when the clock $clk$ ticks. A set of output signals $o$ is then emitted and the SKS moves from state $s$ to state $s'$. To ensure *deterministic* execution, we restrict the model such that each unique subset of inputs (of $I$) enables the trigger condition (a boolean formula over inputs) of precisely one outgoing transition of each state.

The slave reader IP model shown in Fig. 4.2 is the SKS $P_U = \langle AP_U, S_U, u_0, I_U, O_U, R_U, L_U, clk_3\rangle$, where:

- $AP_U = \{Idle_U, Sel_w, Rd_{32}\}$ with $AP_{Ucontrol} = \{Idle_U, Sel_w\}$ and $AP_{Udata} = \{Rd_{32}\}$.
- $S_U = \{u_0, u_1, u_3\}$ with $u_0$ being the initial state.

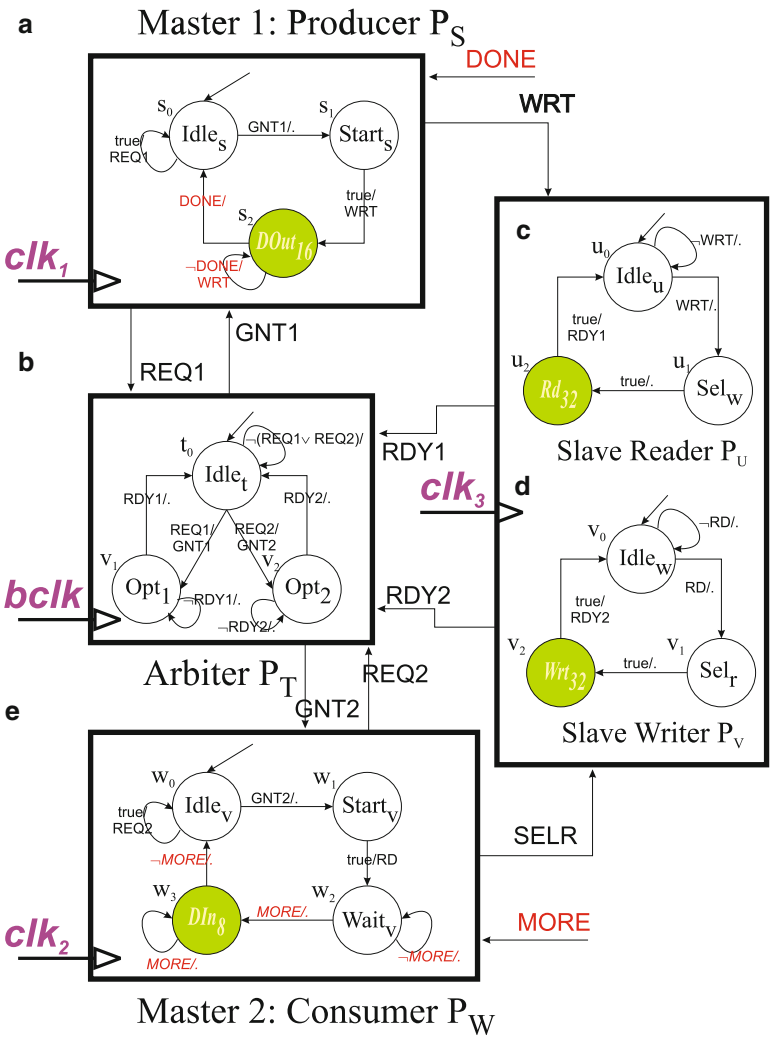



**Fig. 4.3** SKS models of the IPs of the SoC shown in Fig. 4.1

- $I_U = \{\texttt{WRT}\}$ and $O_U = \{\texttt{RDY1}\}$.
- $R_U = \{u_0 \xrightarrow{\neg\texttt{WRT}/.} u_0, u_0 \xrightarrow{\texttt{WRT}/.} u_1, u_1 \xrightarrow{true/.} u_2, u_2 \xrightarrow{true/\texttt{RDY1}} u_0\}$. We use the symbol "." to indicate no output (or $\emptyset$). Each transition triggers with respect to the ticks of the clock $clk_3$. This model is deterministic because precisely one transition is enabled from any state at any clock tick.
- $L_U(u_0) = \{Idle_u\}$, $L_U(u_1) = \{Sel_w\}$, and $L_U(u_2) = \{Rd_{32}\}$.

Figure 4.3 presents the SKS models for all IPs of the SoC system presented in Fig. 6.1. The bus arbiter $P_T$ allows the producer master protocol $P_S$ and the consumer master protocol $P_W$ to carry out transactions. The arbiter and masters use signals

REQ1, REQ1, GNT1 and GNT2 to communicate. Every transaction, the producer master $P_S$ writes data to the data bus, in multiples of 16-bit units (indicated by the label $DOut_{16}$ of state $s_2$). The consumer master $P_W$ can read data in 8-bit units every transaction (indicated by the label $DIn_8$ of state $w_3$). The two protocols exchange data via the memory slave. The memory slave first reads the data generated by the producer master and then makes it available for the consumer master. The protocol of the memory slave IP is divided into two parts: a reader protocol $P_U$, and a writer protocol $P_V$. The reader protocol, as described earlier, reads 32-bit data from the data bus into the memory, and the slave writer protocol writes 32-bit data to the data bus. A master can choose to either write data to (via the slaver reader protocol) or read data from (via the slave writer protocol) the memory slave using the signals WRT or SELR respectively. We now describe how control, data and timing are captured in SKS by using the models shown in Fig. 4.3.

#### 4.1.1.1 Control Flow

IPs use lines of the SoC control bus (such as read/write enable signals) to exchange control information. Different buses may have different signalling methods. For example, a read/write signal may be reversed on two different buses. However, capturing the precise signalling method is not essential during IP modelling. We can *abstract* these details out, and can see control flow between IPs as the flow of output and input signals between them. These output and input signals can be captured with descriptive names to make the model easier to understand. SKS models therefore simply contain such input and output control signals. E.g., the output control signal WRT, signifying a write operation on the data bus, is emitted by the producer master SKS $P_S$ and read as an input signal by the slave reader $P_U$ (Fig. 4.3).

Another way control flow happens in an SoC is through the SoC decoder, which selects and enables a slave by converting (some bits of) the current address bus value into a slave select signal. In order to capture this slave-selection by a master in its formal model, we can simply abstract the model such that it issues slave select signals directly. For example, the consumer master $P_W$ emits the signal SELR to select the slave writer $P_V$ (Fig. 4.3). The decoder logic is abstracted out, without the loss of information.

In addition to the use of the control bus, IPs may also exchange control information by using the data and/or address bus. For example, the data sent by a master to a slave might contain control information that the slave may use to generate an appropriate response. Automatically capturing such *implicit* and *application-dependent* control information is very difficult, if not impossible. However, SKS enable designers to manually embed such information by using additional input and output signals or additional state-labels.

#### 4.1.1.2 Data Flow

IPs use the SoC data and address buses to exchange data. In some cases, IPs may also share data over shared memories accessed using additional connections outside of the SoC bus. A naïve way to capture data flow in an SoC model is to encode all possible data values that are exchanged between IPs. However, this leads to state explosion for even very small models. For the SoC shown in Fig. 4.1, the AMBA ASB data bus is 32-bits. Therefore they are $2^{32}$ distinct data values that may be captured, resulting in an enormously large state space for a simple SoC model. Capturing all possible data values is wasteful in many cases, making this method unattractive for most cases. On the other hand, abstracting out all data bus (and address bus) values might result in a very compact but simplified and useless model. Therefore, we need to find a way to encode all meaningful data bus values (as well as address bus values) in a compact manner. We can explore possible solutions by focussing on the type of information data or address values contain.

1. *Control information*: As noted previously in Sect. 4.1.1.1, data or address bus values can contain implicit control information. This information can be encoded by simply adding control signals or state labels to the formal model.
2. *Data value*: Capturing the actual value of the data or address bus at any instance is usually unnecessary to verify most key properties. Hence, we can abstract out values in the formal model to avoid state-explosion. For example, in Fig. 4.3, the producer master $P_S$ writes 16-bit data to the data bus. We capture this data write operation simply as the label $DOut_{16}$ of state $s_2$ in $P_S$.

   While this abstraction works most of the time, in some cases, IPs may generate key values that must be included in the model. SKS allow designers to include such key values in the models by using additional control signals or state labels.
3. *Data-width*: This refers to the size, in number of bits, of the value of a data packet. While SoC data bus sizes are usually fixed (e.g., 32-bits for AMBA buses), an IP may be able to read/write data at a lower width. Also, two IPs communicating via a separate shared memory might read or write from it at different data-widths. For example, the producer and consumer masters shown in Fig. 4.1 have data-widths of 16 and 8 bits respectively. SKS capture data-widths associated with specific read and write operations by using data labels. For example, the data label $DOut_{16}$ of state $s_2$ in the producer master model $P_S$ corresponds to the generation of 16 bit data by the IP. Each data label, such as $DOut_{16}$, captures the *channel*, the *type* (read/write) of the operation, and the *width* (in number of bits) of the data packet. The label $DOut_{16}$ corresponds to channel $D$ (the data bus), operation *Out* (write), and data-width 16 (bits).

#### 4.1.1.3 Timing

Multiple IPs of a SoC typically synchronize their execution by using the SoC bus clock. Some IPs like processors typically execute at faster frequencies than peripheral devices. An on-chip system usually contains a single clock crystal, which

generates a *master* clock signal. Different clock frequencies for different IPs are obtained by *dividing* the master clock. SKS allow explicit modelling of the driving clock of an IP. For example, the IPs of the producer-consumer SoC shown in Fig. 4.3 all execute on different clocks which are contained in the respective SKS models explicitly.

### 4.1.2 Composition of Synchronous Kripke Structures

SKS capture the control, data and timing (clock-driven) characteristics of IPs. The composition of SKS, as defined below, captures the control, data and timing behaviour of multiple interacting IPs. In order to keep the presentation simple and concise, we choose to focus only on the control and data aspects of the composition, and assume that all protocols in the composition execute using a common clock. This restriction allows us to define SKS composition using the more-familiar synchronous parallel composition. We deal with multi-clock issues later in Chap. 5.

**Definition 4.2 (Parallel Composition).** Given two SKS $P_1 = \langle AP_1, S_1, s_{0_1}, I_1, O_1, R_1, L_1, clk \rangle$ and $P_2 = \langle AP_2, S_2, s_{0_2}, I_2, O_2, R_2, L_2, clk \rangle$, such that $I_1 \cap I_2 = \emptyset$, their parallel composition is the SKS $P_1||P_2 = \langle AP_{||}, S_{||}, s_{0_{||}}, I_{||}, O_{||}, R_{||}, L_{||}, \text{clk} \rangle$ where:

- $AP_{||} = AP_1 \cup AP_2$.
- $S_{||} = S_1 \times S_2$.
- $s_{0_{||}} = (s_{0_1}, s_{0_2})$.
- $I_{||} = I_1 \cup I_2$.
- $O_{||} = O_1 \cup O_2$.
- $L_{||}((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$.
- $R_{||} \subseteq S_{||} \times \{\mathtt{t}\} \times B(I_{||}) \times 2^{O_{||}} \times S_{||}$ is the transition relation such that:

$$\frac{(s_1 \xrightarrow{b_1/o_1} s_1') \ \wedge \ (s_2 \xrightarrow{b_2/o_2} s_2')}{(s_1, s_2) \xrightarrow{b_1 \wedge b_2 / o_1 \cup o_2} (s_1', s_2')}$$

The parallel composition of two protocols contains all states and transitions that can be reached by making simultaneous transitions from each protocol. Figure 4.4 presents the parallel composition of slave reader $P_U$ and slave writer $P_V$ shown in Fig. 4.3. Each state in the composition corresponds to a state in each protocol. E.g., the initial state $(u_0, v_0)$ of the parallel composition shown in Fig. 4.4 corresponds to the initial states of the two protocols. The state labels of each composed state is the

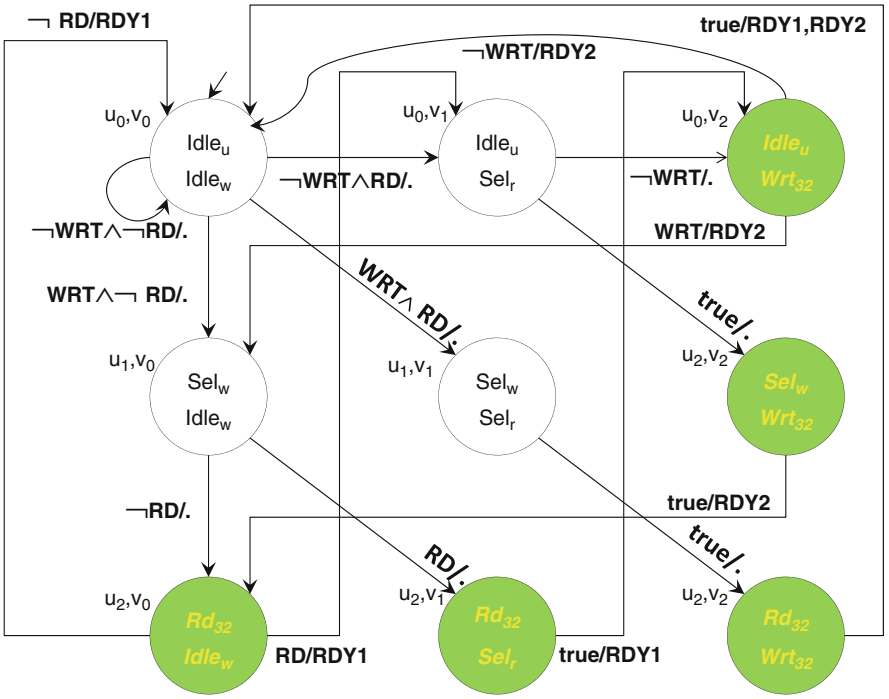



**Fig. 4.4** Parallel composition of the slave reader and writer protocols $P_U$ and $P_V$

union of the labellings of its constituent states $s_1$ and $s_2$. For example, the labelling of state $(u_2, v_2)$ in Fig. 4.4 is $\{Rd_{32}, Wrt_{32}\}$ because $\{Rd_{32}\}$ and $\{Wrt_{32}\}$ are the labellings of states $u_2$ and $v_2$ respectively.

The composition can read and write all input and output signals of the two protocols. Each transition in a state $(s_1, s_2)$ combines a unique transition in each state $s_1$ and $s_2$. For example, the state $(u_0, v_0)$ shown in Fig. 4.4 has four transitions. Each of these transitions corresponds to a unique combination of transitions in states $u_0$ and $v_0$. For example, the transition $(u_0, v_0) \xrightarrow{\neg \mathtt{WRT} \wedge \neg \mathtt{RD}/.} (u_0, v_0)$ combines the transitions $u_0 \xrightarrow{\neg \mathtt{WRT}/.} u_0$ and $v_0 \xrightarrow{\neg \mathtt{RD}/.} v_0$ in $P_U$ and $P_W$ respectively.

SKS models and their parallel composition precisely and succinctly describe the behaviour of SoCs in terms of their control, data and timing characteristics. The next section introduces SoC boilerplates, which allow users to write formal properties about SoC behaviour using natural language.

## 4.2 SoC Boilerplates

Boilerplates allow users to use natural language to specify desired formal properties. Boilerplates are semi-complete natural language sentences that can be completed to create context-specific properties. The completed sentences can then be automatically translated into CTL formulas. The following is a boilerplate:

$$\text{The } \langle system \rangle \text{ must always eventually reach } \langle state\,label \rangle \qquad \text{(B1)}$$

In boilerplate (B1), $\langle system \rangle$ and $\langle state\,label \rangle$, are called *place holders*. Place holders are replaced by system-specific information to complete boilerplates. We utilize the following place holders in our context:

- *System place holders*: A $\langle system \rangle$ place holder can be replaced by the name of the entire system being developed, or one of its sub-systems or components. For instance, the system place holder in boilerplate (B1) above can be replaced with *Arbiter* in Fig. 4.3.
- *State label place holders*: A $\langle state\,label \rangle$ place holder corresponds to the control and/or data labels of a SKS models. For example, boilerplate (B1) contains a system label, which can be replaced by the IP name *Arbiter*, and a state label place holder which can contain any state label contained in the SKS description of *Arbiter* (shown in Fig. 4.3). The use of a system-label automatically limits the state-labels that can be used in the following state-label field. A completed requirement based on boilerplate (B1), and the SoC example shown in Fig. 4.3 could be:

$$\text{The } \mathit{Arbiter} \text{ must always eventually reach } \mathit{Idle}_t \qquad \text{(b1)}$$

In this chapter, we capitalise boilerplate (such as (B1)), and write requirement names using lower case (such as (b1)).

- *Signal place-holders*: A $\langle signal \rangle$ place holder is replaced by signal names, contained in the sets $I$ and $O$ of a SKS model. We can have even more precise signal place-holders such as $\langle input\,signal \rangle$ and $\langle output\,signal \rangle$ to clearly distinguish between inputs and outputs. Consider the following boilerplate:

$$\text{The } \langle system \rangle \text{ must always eventually emit } \langle output\,signal \rangle \qquad \text{(B2)}$$

Boilerplate (B2) can then be completed to the following requirement:

The *Arbiter* must always eventually emit `GNT2` (b2)

- *Boilerplate place holders*: A $\langle boilerplate \rangle$ place holder allows the *nesting* of boilerplates. For example, consider the boilerplate:

The $\langle system \rangle$ satisfies $\langle boilerplate1 \rangle$ and $\langle boilerplate2 \rangle$ (B3)

Boilerplate (B3) is a conjunctive boilerplate which can combine two previously-created requirements (the same IP) into a single requirement. For example, we can combine the requirements (b1) and (b2) using boilerplate (B3) and create the following requirement:

The *Arbiter* satisfies (b1) and (b2) (b3)

- *Combined place holders*: A $\langle type1/type2 \rangle$ combined place holder can be completed using entities of both $\langle type1 \rangle$ and $\langle type2 \rangle$. For example, as we will be described later in Sect. 4.2.1, we can insert a boilerplate-based requirement into another boilerplate to create a *nested* requirement. We use $\langle boilerplate/statelabel \rangle$ place holders to allow such nesting, where boilerplate-based requirements or state labels are used to write completed requirements.

Each boilerplate corresponds to a property written using a specific pattern. In our case, a boilerplate represents a specific CTL property. A completed boilerplate, or a requirement (such as (b1)), is automatically translated into a CTL property, without user involvement. A simple software tool with a graphical user interface can allow users to select, and complete boilerplates interactively. This tool can automatically collect all relevant system information, and can dynamically display and constrain the values a place holder can be replaced by.

### 4.2.1 Building a Meaningful Set of Boilerplates

CTL provides limitless possibilities to create boilerplates. For example, a boilerplate could be written for the formula AGAFEGEF⟨*state label*⟩, and for even more complex formulas. However, providing users with a large set of boilerplates is not ideal. It is difficult and confusing to choose a boilerplate for a specific property from a large collection. On the other hand, providing too few boilerplates constrains the expressiveness of the requirement framework. In this section, we focus on ways to create a balanced set of boilerplates.

#### 4.2.1.1 Providing a Generic Set of Extendible Boilerplates

Exploiting the structure of CTL, we can provide a set of generic boilerplates that can be extended to write any CTL property. More complex CTL properties are essentially created from the nesting of simpler sub-properties (refer to Definition 3.3). Hence, it is sufficient to provide extendible boilerplates for basic CTL operators like AG, EG, AX, EX, AF, EF, AU, EU, ∧, ∨, and ¬. Consider for example the following boilerplate:

The ⟨*system*⟩ always reaches ⟨*state label* 1⟩
infinitely often and from there, it can possibly reach ⟨*state label* 2⟩ (B4)

A requirement based on this boilerplate for the SoC shown in Fig. 4.3 could be:

The *Arbiter* always reaches $\text{Idle}_t$
infinitely often and from there, it can possibly reach $\text{Opt}_1$ (b4)

Requirement (b4) may be expressed as the CTL property $\texttt{AGAF}(Idle_t \wedge \texttt{EF}(Opt_1))$. This seemingly complex property can be built using the following *generic* boilerplates (we omit the ⟨*system*⟩ place holder for the sake of readability):

There is a path to reach ⟨*state label*/*boilerplate*⟩ (EF)

Both ⟨*state label*/*boilerplate*1⟩ and ⟨*state label*/*boilerplate*2⟩ hold (AND)

Always eventually ⟨*state label*/*boilerplate*⟩ is reached (AF)

Globally ⟨*state label*/*boilerplate*⟩ holds (AG)

The boilerplates shown above can be used to write requirement b4, as well as many other requirements (such as $\texttt{AGEF}Idle_t$) that use these boilerplates in different sequences. Providing users with extendible boilerplates that correspond to all CTL operators (as shown in Table 4.1) allows them to write any CTL property.

#### 4.2.1.2 Constraining Boilerplates to Commonly-Used Specifications

Since we are focussing on SoC design, we can look at designing boilerplates to capture commonly-used properties in SoC verification. More specifically, we look at the following types of properties:

- *Single IP properties*: When verifying single IPs, designers may want to focus on properties based on specific patterns within IPs. Consider the following boilerplates

The ⟨*system*⟩ initializes with ⟨*state label*⟩ (B5)

The ⟨*system*⟩ always eventually reaches ⟨*state label*⟩ (B6)

  Boilerplate (B5) refers to an initialization condition and boilerplate (B6) is a liveness property which requires a state label to be visited infinitely often during the execution of an IP. Such properties are commonly encountered during single IP verification.
- *Sequencing of control states within single or multiple IPs*: In many cases, we require that an IP must follow a sequence of execution. Such a sequence can be represented as a sequence of states in the IP's SKS description. Similarly, we may require the IPs of a SoC to execute in specific sequences. As an example, consider the following boilerplate:

The **System** reaches $\langle state\,label1 \rangle$ and from there, it always eventually $reaches \langle state\,label2 \rangle$. This behaviour is repeated infinitely often. (B7)

The keyword **System** in boilerplate (B7) refers to the complete SoC, which is the composition of all constituent IPs. Boilerplate (B7) requires that the system always visits a state labelled with $\langle state\,label1 \rangle$ before visiting another state labelled with $\langle state\,label2 \rangle$. A requirement based on boilerplate (B7) could be:

The **System** reaches $Rd_{32}$ and from there, it always eventually reaches $Wrt_{32}$. This behaviour is repeated infinitely often. (b7)

Requirement (b7) demands the slave reader protocol $P_U$ to first read 32-bits of data (into the memory) before the slave writer protocol $P_W$ writes data out from the memory (see Fig. 4.3). Converted to CTL, requirement (B7) is the property $\texttt{AG}(Rd_{32} \Rightarrow \texttt{AF}Wrt_{32})$. We can create more boilerplates that require specific sequencing of control states between more than two IPs in a similar fashion.

- *Sequencing of signals:* Since IP interaction over a SoC bus involves the exchange of signals, we can also write boilerplates that involve signal emissions and receptions. For example, we can have a boilerplate:

In the **System**, emission of $\langle signal1 \rangle$ is always eventually followed by the emission of $\langle signal2 \rangle$. This behaviour is repeated infinitely often. (B8)

Consider the following requirement, based on boilerplate (B8), and constructed for the SoC shown in Fig. 4.3:

In the **System**, emission of `REQ1` is always eventually followed by the emission of `GNT1`. This behaviour is repeated infinitely often. (b8)

Requirement (b8) requires the Arbiter to always eventually grant bus access to master $P_S$ when it requests bus access using the signal `REQ1`. Such requirements on the sequencing of control signals are frequently used in SoC verification.
However, while boilerplate (B8) is very similar to boilerplate (B7), requirement (b8) cannot be translated into a CTL property in a straightforward manner. CTL properties are based on IP state-labels. Signals in SKS are transition labels, and we cannot directly write CTL properties to reason about signals. We can however capture transitions (and therefore signals) indirectly by encoding them as CTL formulas. For example, consider the transition $s_0 \xrightarrow{true/\texttt{REQ1}} s_0$ in the producer master protocol $P_S$ shown in Fig. 4.3. State $s_0$ is labelled by $Idle_s$. We can therefore capture this transition as the CTL formula $Idle_s \wedge \texttt{AX}Idle_s$. Since signal `REQ1` is only emitted along this specific transition in the SoC, the formula $Idle_s \wedge \texttt{AX}Idle_s$ also captures the emission of `REQ1`. Similarly, the transition $t_0 \xrightarrow{\texttt{REQ1/GNT1}} t_1$, corresponding to the emission of `GNT1`, can be captured as the formula $Idle_t \wedge \texttt{AX}Opt_1$. We can now convert requirement (b8) to the following CTL formula:

$$\texttt{AG}[(Idle_s \wedge \texttt{AX}Idle_s) \Rightarrow \texttt{AF}(Idle_t \wedge \texttt{AX}Opt_1)]$$

As shown above, any transition $s \xrightarrow{\texttt{signal1/signal2}} s'$ can be written as the formula $L(s) \wedge \texttt{AX}L(s')$. Such a formula implicitly captures the emission of `signal1` or reception of `signal2`. In general, it is possible that a signal is emitted or received along multiple transitions in a system. For example, a signal `signal` may be emitted along $n$-transitions $s_i \xrightarrow{b_i/\texttt{signal}} b_i'$ $(i \in [1,n])$. In this case, `signal` emission can be captured using the following CTL formula:

$$[(L(s_1) \wedge L(s_1')) \vee (L(s_2) \wedge L(s_2')) \vee \ldots] \qquad \text{(Sig2CTL)}$$

Note that there are cases when we cannot capture the emission (or reception) of `signal` in CTL. Consider the case when we have a model containing two transitions $s_1 \xrightarrow{\texttt{true/signal}} s_2$ and $s_3 \xrightarrow{\texttt{true/.}} s_2$ where both states have the same labels $(L(s_1) = L(s_3))$. In this case, the formula $L(s_1) \wedge \texttt{AX}L(s_2)$ captures both transitions because the labels of $s_1$ are identical to the labels of $s_3$. However, since only one transition involves the emission of `signal`, we fail to create a precise formula for its emission. We face the same issue when $s_1 = s_3$.

**Table 4.1** SoC Boiler-plates

| No. | Boilerplate | CTL equivalent |
|---|---|---|
| | *Generic CTL Boilerplates* | |
| 1 | The ⟨*system*⟩ satisfies ⟨*statelabel*/*boilerplate*⟩ | ⟨*system*⟩ $\models$ ⟨*statelabel*/*boilerplate*⟩ |
| 2 | The ⟨*system*⟩ does not satisfy *statelabel*/*boilerplate* | ⟨*system*⟩ $\models \neg$⟨*statelabel*/*boilerplate*⟩ |
| 3 | The ⟨*system*⟩ satisfies *statelabel*/*boilerplate* always in the next state | ⟨*system*⟩ $\models$ AX⟨*statelabel*/*boilerplate*⟩ |
| 4 | The ⟨*system*⟩ possibly satisfies *statelabel*/*boilerplate* in the next state | ⟨*system*⟩ $\models$ EX⟨*statelabel*/*boilerplate*⟩ |
| 5 | The ⟨*system*⟩ eventually reaches *statelabel*/*boilerplate* | ⟨*system*⟩ $\models$ AF⟨*statelabel*/*boilerplate*⟩ |
| 6 | The ⟨*system*⟩ can possible eventually reach *statelabel*/*boilerplate* | ⟨*system*⟩ $\models$ EF⟨*statelabel*/*boilerplate*⟩ |
| 7 | The ⟨*system*⟩ satisfies *statelabel*/*boilerplate* in all states | ⟨*system*⟩ $\models$ AG⟨*statelabel*/*boilerplate*⟩ |
| 8 | The ⟨*system*⟩ can satisfy *statelabel*/*boilerplate* globally | ⟨*system*⟩ $\models$ EG⟨*statelabel*/*boilerplate*⟩ |
| 9 | In ⟨*system*⟩, along all possible executions, *statelabel*1/*boilerplate*1 is satisfied until a state is reached where *statelabel*2/*boilerplate*2 is satisfied | ⟨*system*⟩ AU(⟨*statelabel*1/*boilerplate*1⟩ ⟨*statelabel*2/*boilerplate*2⟩ |
| 10 | In ⟨*system*⟩, there exists at least one execution where *statelabel*1/*boilerplate*1 is satisfied until a state is reached where *statelabel*2/*boilerplate*2 is satisfied | ⟨*system*⟩ EU(⟨*statelabel*1/*boilerplate*1⟩ ⟨*statelabel*2/*boilerplate*2⟩ |
| | *Single IP properties* | |
| 11 | The ⟨*system*⟩ initializes with ⟨*statelabel*⟩ | ⟨*system*⟩ $\models$ ⟨*statelabel*⟩ |
| 12 | The ⟨*system*⟩ always reaches ⟨*statelabel*2⟩ after reaching ⟨*statelabel*1⟩ | ⟨*system*⟩ $\models$ ⟨*statelabel*1⟩ $\Rightarrow$ AF⟨*statelabel*2⟩ |
| | The ⟨*system*⟩ can possibly reach ⟨*statelabel*2⟩ after reaching ⟨*statelabel*1⟩ | ⟨*system*⟩ $\models$ ⟨*statelabel*1⟩ $\Rightarrow$ EF⟨*statelabel*2⟩ |
| 13 | The ⟨*system*⟩ always eventually reaches ⟨*statelabel*⟩ | ⟨*system*⟩ $\models$ AGAF⟨*statelabel*⟩ |
| | The ⟨*system*⟩ can possibly reach ⟨*statelabel*⟩ infinitely often | ⟨*system*⟩ $\models$ EGEF⟨*statelabel*⟩ |
| | The ⟨*system*⟩ can always possibly reach ⟨*statelabel*⟩ | ⟨*system*⟩ $\models$ AGEF⟨*statelabel*⟩ |
| 14 | The ⟨*system*⟩ always eventually emits/reads ⟨*signal*⟩ | ⟨*system*⟩ $\models$ AG[*Sig2CTL*(⟨*signal*⟩)] |
| | *Multiple IP properties* | |
| 15 | ⟨*system*1⟩ reaches ⟨*statelabel*1⟩ and ⟨*system*2⟩ reaches ⟨*statelabel*2⟩ simultaneously, and infinitely often | ⟨*system*1$\|\|$*system*2⟩ $\models$ AGAF(⟨*statelabel*1⟩ $\wedge$ ⟨*statelabel*2⟩) |
| 16 | ⟨*system*1⟩ remains in a state labelled by ⟨*statelabel*1⟩ until ⟨*system*2⟩ reaches ⟨*statelabel*2⟩ | ⟨*system*1$\|\|$*system*2⟩ $\models$ A(⟨*statelabel*1⟩ U ⟨*statelabel*2⟩) |

(continued)

**Table 4.1** (continued)

| No. | Boilerplate | CTL equivalent |
|---|---|---|
| 17 | In **System**, emission of $\langle signal1 \rangle$ is always eventually followed by the emission of $\langle signal2 \rangle$ | **System** $\models$ AG[$Sig2CTL(\langle signal1 \rangle) \Rightarrow$ AF($Sig2CTL(\langle signal2 \rangle)$)] |
| 18 | In **System**, emission of $\langle signal \rangle$ is always eventually followed by the reaching of $\langle statelabel/boilerplate \rangle$ | **System** $\models$ AG[$Sig2CTL(\langle signal1 \rangle) \Rightarrow$ AF($\langle statelabel/boilerplate \rangle$)] |

The encoding of signals as CTL properties, and the exclusion of signals that cannot be precisely captured in CTL, can be added as simple extensions to a boilerplate tool. This allows the user to use boilerplates for signals without having to worry about their encoding into formal logic.

We can use the above categories to collect boilerplates into smaller and easier to use sets. For example, a user wanting to verify a single IP would not need to look at all possible sequencing boilerplates.

### *4.2.2 SoC Boiler-Plates*

Based on the discussion above, we propose SoC boilerplates shown in Table 4.1. SoC Boilerplates are arranged in 3 categories. The first category contains generic boilerplates corresponding to every CTL operator. These boilerplates can be used to build any CTL formula, as discussed in Sect. 4.2.1.1. Categories 2 and 3 contain single and multiple IP properties that are frequently encountered during SoC verification. Both categories contain properties involving the sequencing of control states and/or signals, as discussed in Sect. 4.2.1.2. Since signals are converted into CTL formulas (if possible), boilerplates with signals (such as boilerplates 14,17 and 18) are converted to CTL by the application of Eq. Sig2CTL. (Eq. Sig2CTL converts a signal into a CTL property). For multiple IP properties, users can explicitly choose IPs to be included in verification (boilerplates 15, 16) or use the keyword **System** to verify over the entire SoC (boilerplates 17, 18). Note that categories 2 and 4 in Table 4.1 are not exhaustive, and additional boilerplates can be added in a straightforward manner.

## 4.3 Conclusions

This chapter presented two important frameworks. The synchronous Kripke structures (SKS) framework allows precise formal modelling of IPs, and can be used to compose two or more IPs into a SoC formal. SKS are compact state-machine based models that encode the most relevant control, data, and timing details of IPs. We also presented SoC boilerplates framework as an alternative to CTL. Boilerplates allow users to write natural language properties that can be automatically translated

into temporal logic (CTL in our case). A user may write any CTL property about a system using generic CTL boilerplates, or may use single or multiple-IP boilerplates to write commonly-used properties for SoC verification.

Next chapter will build on this foundation to develop a correct-by-construction approach to SoC design. We will use the example of a set-top box to motivate the proposed approach.

# Chapter 5
# SoC Design Methodology

SoCs are built using pre-verified on-chip protocols chosen from available IP libraries. There are two key problems with integrating selected protocols into a SoC:

1. Addressing protocol *mismatches* [PdAHSV02]: On-chip protocols may differ in the way they exchange control and data information. Moreover, if they execute using different clocks, they may have additional timing-related differences between them. These *control*, *data* and *clock mismatches* may prevent meaningful communication.
2. Enforcing system-level behavioural constraints: Even if on-chip protocols can be composed, there is no guarantee that the resulting system will satisfy desired system-level specifications.

As stated in Chap. 1, system-level verification, which ensures that protocols interact correctly, can take up to 80% of the total design time. Any manual modification of on-chip protocols is an expensive process which may add further delays. Hence, to address the two issues stated above, we look at the automatic generation of glue logic, called a *converter*, to enable composing mismatched on-chip protocols in a manner such that the desired system-level specifications are satisfied. A converter acts as an interface between on-chip protocols and guides their interactions. In this chapter, we provide a methodology to automatically generate converters to address mismatches and enforce system-level specifications.

An automatic converter generation framework requires formal descriptions of on-chip protocols and system-level behavioural constraints. For this purpose, we can use synchronous Kripke structures to model on-chip protocols in a precise manner, and SoC boilerplates (Chap. 4) to capture natural language requirements as formal CTL formulas. We can then use formal methods such as module and model checking presented in Chap. 3 to verify if on-chip protocols satisfy system-level specifications, However, we cannot automatically enforce the satisfaction of system-level specifications using verification algorithms. This objective is achieved via a converter, which is generated by the methodology presented in this chapter.

R. Sinha et al., *Correct-by-Construction Approaches for SoC Design*,
DOI 10.1007/978-1-4614-7864-5_5, © Springer Science+Business Media New York 2014

This methodology uses a converter generation algorithm based on model and module checking. The details of this algorithm appear in Chap. 6. In this chapter, we focus on presenting a bird's eye view of the converter generation process.

## 5.1 Protocol Mismatches

Figure 5.1 presents a SoC for a set-top receiver box, adapted from a similar example presented in [SRBS09]. The set-top box contains four on-chip protocols, and Fig. 5.1 shows their SKS models. The infra-red (IR) receiver reads key-codes generated by a user using a remote control, and writes them to the IR buffer. The control unit verifies these key codes, and enables the video decoder which decodes satellite data into an output AV stream. Finally, the PAL/NTSC encoder encodes the AV stream into the user's TV format (PAL or NTSC).

The individual protocols, which execute on different clocks, operate as follows. The IR receiver on-chip protocol $P_S$ writes 8-bit characters (represented by the label $COut_8$ of state $s_1$) onto an IR buffer. The control unit $P_T$ awaits an input `ready` and then reads two characters or 16 bits from the IR buffer (represented by the label $KeyIn_{16}$ of state $t_1$). It emits `start` if the key is valid, which is read by the

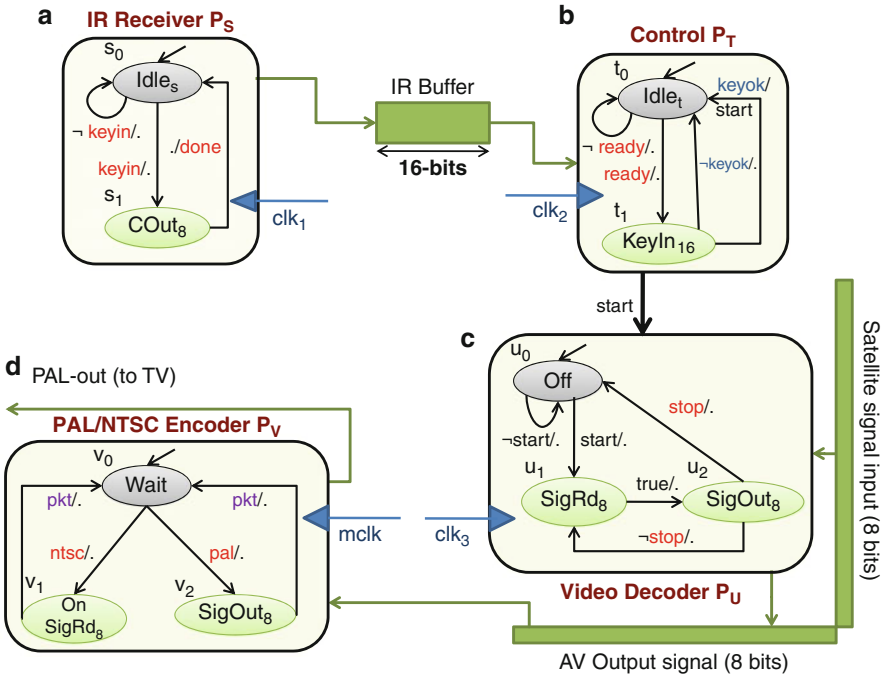



**Fig. 5.1** Protocols of the set-top receiver box SoC

video decoder $P_U$. $P_U$ then alternates between reading 8-bit data from the satellite signal input stream (in state $u_1$) and writing it to the AV output signal stream (in state $u_2$) until it receives the `stop` signal. The PAL/NTSC encoder $P_V$ can convert an AV packet stream to a PAL or NTSC stream depending on the corresponding input received before each packet.

The on-chip protocols shown in Fig. 5.1 have a number of mismatches:

1. *Control mismatches:* The on-chip protocols have unconnected control signals such as `keyin`, `done`, `ready`, and *stop*. Also, some input signals such as *pkt* are not emitted by any protocol in the system. Also, the protocols are free to execute without any explicit need to synchronize shared control signals (like `start`). These anomalies can lead to improper communication between the on-chip protocols.
2. *Data-width mismatches:* While the IR receiver writes single 8-bit characters to the IR buffer, the control unit reads two characters or a 16-bit data from the IR buffer each time, resulting in a *data-width* mismatch. When the protocols execute in an unrestricted manner, it is possible to have overflows and underflows in this buffer.
3. *Clock mismatches:* The on-chip protocols operate on different clocks, resulting in timing issues such as an inability to synchronize the protocols and the loss of shared control signals.

The above mismatches may prevent the on-chip protocols of the SoC to interact in the desired manner. Hence, while addressing mismatches is important, a SoC built from multiple components must also satisfy system-level requirements. For the set-top receiver example in Fig. 5.1, the SoC must satisfy the following user-specified requirements, described as SoC boilerplates:

1. Control constraints are `CTL` properties over state labels in the protocols (discussed in detail in Sect. 4.2). For the SoC shown in Fig. 5.1, we capture the following control properties:
   (a) The system must visit the state $t_1$ (in $P_T$) infinitely often in order to check codes entered by the user. This informal property can be captured as the boilerplate requirement "The SoC must always eventually reach $KeyIn_{16}$", obtained from the completing the boilerplate "The $\langle system \rangle$ must always eventually reach $\langle state\,label \rangle$". The completed requirement corresponds to the CTL formula $\mathtt{AGAF}KeyIn_{16}$.
   (b) Once decoding starts, it must continue until the user provided key-codes are checked again by the control unit. We capture this property using the requirement "Whenever SoC reaches $SigRd_8$, it must then not reach $Off$ until reaching $KeyIn_{16}$.", obtained from completing the boilerplate "Whenever $\langle system \rangle$ reaches $\langle state\,label1 \rangle$, it must then not visit $\langle state\,label2 \rangle$ until reaching $\langle state\,label3 \rangle$." When translated to CTL, we get the formula $\mathtt{AG}(SigRd_8 \Rightarrow \mathtt{A}(\neg Off\ \mathtt{U}\ KeyIn_{16}))$.

2. *Data constraints*, discussed later in Sect. 6.2, restrict the minimum and maximum number of bits a data buffer in the system can contain at any time. For example, for the SoC shown in Fig. 5.1, we may require the IR buffer to contain between 0 and 16 bits (its capacity) at any time to avoid overflows and underflows. We capture this as the requirement "In the SoC, the number of bits in $IRBuffer$ is always between 0 and 16", obtained from completing the boilerplate "In $\langle system \rangle$, the number of bits in $\langle buffername \rangle$ is always between $\langle integer1 \rangle$ and $\langle integer2 \rangle$".

   In order to transform boilerplates for data constraints into CTL, we introduce *data counters* that track the number of bits contained in a data buffer at any time. For the set-top box example, we introduce a *counter* `I` for the IR Buffer. The use of counters is explained in detail in Chap. 6.

   A data counter is associated with a data *channel* (such as a buffer or data bus), and is incremented/decremented whenever data is added to/removed from the channel. For example, we introduce a *counter* $\mathtt{I}_{IR}$ for the IR Buffer, and then reduce the above requirement to $AG(0 \leq \mathtt{I}_{IR} \leq 16)$. In addition to data constraints, we can have *control-data* constraints that combine control and data aspects within a single requirement. Details about data constraints appear later in Chap. 6, Sect. 6.2.

In order to address possible mismatches, and ensure that the SoC satisfies all system-level requirements, an automatic converter generation algorithm must do the following:

1. Ensure that on-chip protocols with different clocks can be composed by addressing clock mismatches.
2. Generate a converter to ensure that:
   (a) Control signals are always exchanged between on-chip protocols in the desired manner.
   (b) All data buffers remain within capacity.
   (c) All system-level requirements are satisfied.

In the next section, we look at how we can compose multi-clock on-chip protocols, and then provide a methodology to generate converters in the next section.

## 5.2 Composition of Multi-clock IPs

The composition operator presented in Chap. 4, Definition 4.2, can only be used when on-chip protocols execute using a common clock. However, as illustrated in Fig. 5.1, a SoC may contain protocols that execute on different clocks. This section describes how we can compose on-chip protocols to create multi-clock SoCs.

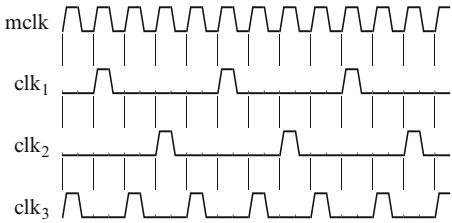



**Fig. 5.2** Timing diagram for the clocks of the set-top box SoC

### 5.2.1 Clocks

A clock can be viewed as an input stream where each rising edge signifies a *tick*. The time difference between two consecutive ticks of a clock is called the time period $T$ of the clock. The frequency $F$ of a given clock is computed as $1/T$ and represents the number of clock ticks that happen per unit of time (seconds). The scope of this chapter is limited to clocks where the time between any two ticks is a constant. Given two clocks $clk1$ and $clk2$, their *clock ratio* is the ratio $F_1 : F_2$ of their frequencies. The clock ratio is used to compare the relative speeds of two or more clocks. Consider, for example, the clocks *mclk*, $clk_1$, $clk_2$ and $clk_3$ in Fig. 5.2. The ratio between *mclk* and $clk_1$ (and also $clk_2$) is 4:1 and the ratio between *mclk* and $clk_3$ is 2:1. This means that *mclk* is four times faster than $clk_1$ and $clk_2$, and two times faster than $clk_3$.

### 5.2.2 Clock Automata

Clocks *synchronize* whenever their ticks align. From Fig. 5.2, it can be seen that while the clocks $clk_1$, $clk_2$ and $clk_3$ always synchronize with the clock *mclk* individually (each of their ticks are aligned with some ticks of *mclk*), they never synchronize with each other. A SoC may contain several IPs that may execute using different clocks. Typically, a system-on-chip has a *base* or *master* clock which is the fastest clock in the system [CP03]. All other clocks in the SoC are *derived* from the master clock. Each tick of a derived clock synchronizes with some tick of the master clock. For example, in Fig. 5.2, clocks $clk_1$, $clk_2$ and $clk_3$ are derived from the master clock *mclk*. A derived clock always has a frequency less than or equal to the frequency of the master clock. Two clocks are *identical* if they are both derived from each other. The relationship between multiple clocks of a SoC can be described using a *clock automaton*, defined below.

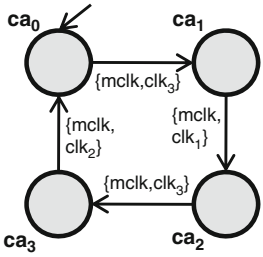



**Fig. 5.3** Clock automaton *CA* for the set-top box SoC

**Definition 5.1 (Clock Automaton).** A Clock automaton (*CA*) is a *finite state machine* represented as a tuple $\langle S, ca_0, CLK, R, mclk \rangle$ where:

- $S$ is a finite set of states.
- $ca_0 \in S$ is the initial state.
- $CLK$ is a finite non-empty set of output *clock signals*.
- $R : S \rightarrow 2^{CLK} \times S$ is the transition function.
- *mclk* is the driving clock of *CA*.

A clock automaton has a finite set of states with a unique initial state. Each state in the clock automaton, by definition, has exactly one outgoing transition. During each transition, the clock automaton emits a subset of clocks ticks (rising edges). Transitions in the clock automaton are triggered using the master clock *mclk*. The transitions of the form $R(ca) = (CLK_o, ca')$, where $CLK_o \subseteq CLK$, are represented using the shorthand $ca \xrightarrow{CLK_o} ca'$. As the clock automaton transitions are driven by the clock *mclk*, it serves as the master clock and all other clocks are derived from it.

Figure 5.3 shows the clock automaton *CA* for the set-top box SoC. It captures clock relationships of the SoC as described in the timing diagram in Fig. 5.2. *CA* contains four states and four transitions which form a cycle. The cycle represents an infinitely repeating pattern of four ticks of the master clock *mclk* (the driving clock of *CA*). During the first and third transitions of this repeating pattern, *CA* emits the clock signal $clk_3$, capturing the exact relationship between $clk_3$ and *mclk* shown in Fig. 5.2. The same holds true for clocks $clk_1$ (emitted during the second transition) and $clk_2$ (emitted during the fourth transition).

### 5.2.3 SKS Oversampling

An on-chip protocol, described as a SKS, executes only when its clock ticks. At a clock tick, the IP can sample environment inputs, take a transition, emit outputs and move to a new state. It must then wait for the next clock tick to execute further. Two

IPs, executing on different clocks, may communicate with each other (exchange control signals) only when their corresponding clocks synchronize. When their ticks do not align, they cannot execute together and may fail to exchange control signals. For example, since the control unit and video decoder protocols of the set-top box (Fig. 5.1) execute using clocks $clk_2$ and $clk_3$ that never synchronize, the shared control signal *start* emitted by the control unit will never be read by the video decoder.

The composition of two SKS, as defined in Definition 4.2, can be computed only when the two SKS execute using the same clock. In order to compose on-chip protocols with different clocks, we use the concept of *oversampling* [CT92]. Given a SKS $P$ that executes using a specific clock $clk$, and a clock automaton $CA$ that emits $clk$ and executes on the master clock $mclk$, we can transform $P$ to an oversampled SKS that executes using $mclk$. If we oversample all on-chip protocols of a SoC, the oversampled protocols will all execute using $mclk$, and can be composed using Definition 4.2. SKS oversampling is defined as follows:

**Definition 5.2 (SKS Oversampling).** Given a SKS $P = \langle AP, S, s_0, I, O, R, L, clk\rangle$ and a clock automaton $CA = \langle S^{CA}, ca_0, CLK^{CA}, R^{CA}, mclk\rangle$, such that $clk \in CLK^{CA}$, the oversampled SKS is $P_{CA} = \langle AP, S_{CA}, s_{CA0}, I, O, R_{CA}, L_{CA}, mclk\rangle$ where

1. $S_{CA} \subseteq S \times S^{CA}$ is the set of all reachable states.
2. $s_{CA0} = (s_0, ca_0)$ is the initial state with $L(s_0, ca_{A0}) = L(s_0)$.
3. $R_{CA} \subseteq S_{CA} \times \{\mathtt{t}\} \times B(I) \times 2^O \times S_{CA}$ is the transition relation where the event $\mathtt{t}$ represents ticking of the clock $mclk$.

   Given $ca \xrightarrow{\{CLK_o\}} ca'$, each state $(s, ca) \in S_{CA}$ has the following transitions:

   a. If $clk \in CLK_o$, then for every transition $s \xrightarrow{b/o} s'$, $(s, ca)$ has a transition $(s, ca) \xrightarrow{b/o} (s', ca')$. Also, $L(s', ca') = L(s')$.

   b. If $clk \notin CLK_o$, then $(s, ca)$ has only one transition $(s, ca) \xrightarrow{true/\emptyset} (s, ca')$. Also, $L(s, ca') = L(s) \cap AP_{control}$.

Given a SKS $P$ executing on clock $clk$ and a clock automaton $CA$ with the master clock $mclk$, the oversampled SKS $P_{CA}$ describes the behaviour of $P$ with respect to the master clock $mclk$. Each state of $P_{CA}$ is a pair $(s, ca)$ where $s$ and $ca$ are states in $P$ and $CA$ (point 1 in Definition 5.2). The initial state is $(s_0, ca_0)$, and corresponds to the initial states of $P$ and $CA$. This initial state has the same state labels as the initial state $s_0$ of $P$ (point 2 of Definition 5.2). The oversampled protocol also has the same inputs and outputs as $P$. We formalise the transitions of any state $(s, ca)$ in $P_{CA}$ as follows (point 3). If the lone transition $ca \xrightarrow{\{CLK_o\}} ca'$ out of $ca$ emits the clock signal $clk$ (point 3a), then the state $(s, ca)$ can take any of the enabled transitions of $s$ (because the current tick of $mclk$ synchronizes with a tick of $clk1$). In this case,

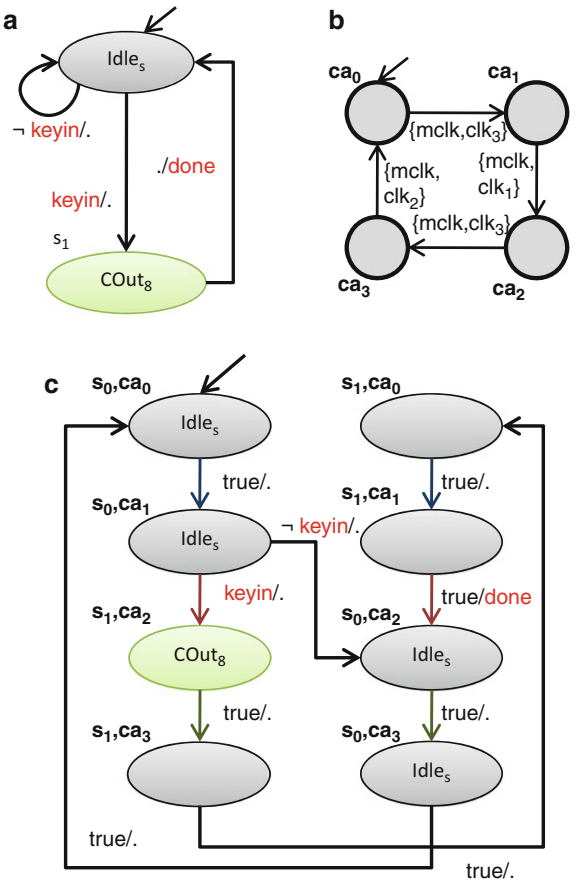



**Fig. 5.4** Oversampling of the IR receiver SKS. (**a**) IR receiver SKS with driving clock $clk_1$. (**b**) Clock automaton $CA$. (**c**) Oversampled IR receiver SKS

$(s, ca)$ will have transitions to states of the form $(s', ca')$ where $s'$ are successors of $s$ in $P$. Each such successor state $(s', ca')$ in this case has the same state labels as state $s'$ in $P$. On the other hand, if the lone transition $ca \xrightarrow{\{CLK_o\}} ca'$ out of $ca_0$ does not emit $clk$ (point 3a in Definition 5.2), then state $(s, ca)$ cannot sample any inputs of $P$, and hence no outgoing transitions of $s$ can be allowed. In this case, $(s, ca)$ has a single transition to $(s, ca')$. Moreover, state $(s, ca')$ retains only the *control* labels of $s$, and all *data* labels of $s$ are removed. Recall, from Definition 4.1, that a data label represents a data operation in a SKS. Since the transitions in the oversampled protocol created using point 3b do not correspond to an actual transition in the original protocol, we must remove any data labels to capture the fact that no data operations are carried out during such transitions.

Figure 5.4 shows how the IR receiver protocol (Fig. 5.1) can be oversampled with respect to the driving clock *mclk* of the clock automaton shown in Fig. 5.3. Each state in $P_{CA}$ corresponds to a unique state in $P$ and a unique state in $CA$. For example, the initial state of the oversampled IR receiver is $(s_0, ca_0)$ where $s_0$ is the initial state of the IR receiver SKS, and $ca_0$ is the initial state of the clock automaton. The oversampled SKS has the same inputs and outputs as the original SKS.

The transitions of a state $(s, ca)$ in $P_{CA}$ depend on the outgoing transition of the clock automaton state $ca$. If the outgoing transition from $ca$ emits the clock signal $clk$ (the driving clock of $P$), $(s, ca)$ preserves all transitions in $s$. On the other hand, if the outgoing transition from $ca$ does not emit $clk$, $(s, ca)$ has a single transition to state $(s, ca')$ where $ca'$ is the successor of $ca$. For example, consider state $(s_1, ca_1)$ in the oversample IR receiver SKS. The clock automaton state $ca_1$ has a transition to state $ca_2$ where the clock $clk_1$ (which drives the IR receiver) is emitted. Therefore, state $(s_1, ca_1)$ has a transition to $(s_0, ca_2)$, enabling state $s_1$ to take a transition to $s_0$. Similarly, state $(s_0, ca_0)$ has a transition to $(s_0, ca_1)$ which shows that while the clock automaton moves to a new state, the IR receiver does not progress since its driving signal $clk_1$ is not emitted by the clock automaton. We call such transitions delay transitions.

Each $(s, ca)$ in the oversampled SKS has the same state labels as state $s$ in the original SKS. However, if $(s, ca)$ is reached via a delay transition, all its data labels are removed. This is done because a delay transition models the lack of progress in the oversampled SKS. Since data-labels model explicit data operations in the SKS, they are removed from all states that are reached via delay transitions. For example, both states $(s_1, ca_2)$ and $(s_1, ca_3)$ in Fig. 5.4 correspond to state $s_1$ in the IR receiver protocol. While $(s_1, ca_2)$ contains the data label $COut_8$ (a label of $s_1$), $(s_1, ca_3)$ does not, because state $(s_1, ca_3)$ is reached after a delay transition.

## 5.3 Design Methodology Using Protocol Conversion

This section provides an overview of the design methodology where we use converter generation to build SoCs. We discuss here an overview of the process to automatically build SoCs from mismatching on-chip protocols, and the technical details of the proposed converter generation algorithm is presented in Chap. 6.

Figure 5.5 provides an overview of the converter generation process. It consists of the following steps:

1. *Inputs*: The converter generation algorithm takes in the following inputs:
   - *SKS descriptions of on-chip protocols:* As discussed in Sect. 4.1, SKS can formally model the control, data and timing (clock-driven) behaviour of on-chip protocols. SKS allow us to formally model the IPs of a system and use these models for system-level verification (see Fig. 1.3). In addition to the SKS models, we require the user to provide the relationships between the clocks of the various protocols. This information is used to resolve clock mismatches using oversampling, as presented in Sect. 5.2. For the set-top box SoC shown in Fig. 5.1, we require the user to provide SKS descriptions of all four protocols, as well as the clock relationships which result in the creation of the clock automaton shown in Fig. 5.3.
   - *Control and data requirements as SoC boilerplates:* SoC boilerplates (introduced in Sect. 4.2) allow users to build system-level requirements efficiently. These requirements can be used for the system specification, compatibility

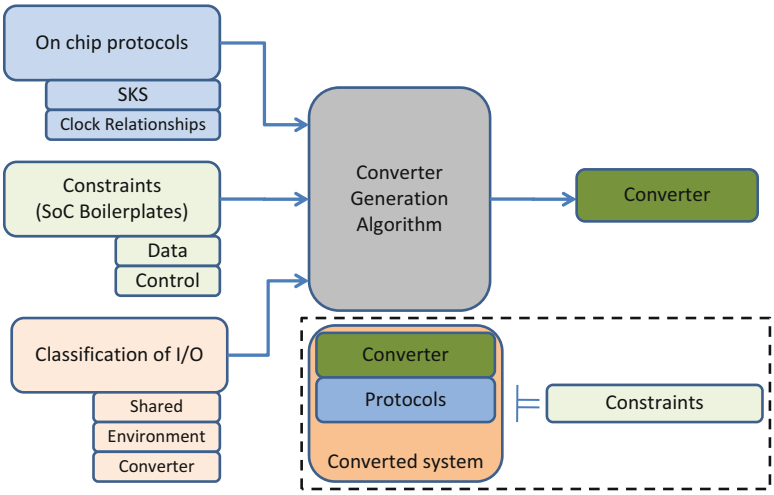



**Fig. 5.5** Overview of converter generation

checking, as well as the system-level verification of the SoC design cycle, as shown in Fig. 1.3. We can automatically extract CTL properties from such requirements. Control requirements describe sequences of control-signals exchange between the protocols. Data requirements on the other hand allow tracking of the amount of data present in data channels (such as the IR buffer in Fig. 5.1). Details about the use of data requirements appear later in Sect. 6.2. For the set-top box SoC example, the user may provide the system-level constraints discussed in Sect. 5.1.

- *Classification of input and output control signals*: We require the user to classify the input and output signals of the protocols. This helps constrain how the converter (to be generated) deals with different types of signals. We require each signal to be marked *shared* (emitted and read within the protocols of the SoC), *environment*-generated (generated in the environment of the SoC), or *converter-generated* (to be generated by the converter whenever needed). Figure 5.6 illustrates classification of control signals. It shows how the converter for the set-top box deals with different types of control signals. We classify inputs `keyin` and `keyok` read by the IR receiver and the control unit respectively as *uncontrollable* inputs because they are emitted by the environment of the SoC (a user and a remote server respectively). Such signals must be relayed by the converter immediately to the protocols. The control unit and video decoder protocols share the signal `start`. Hence, we classify `start` as a *shared* signal. The converter can read a shared signal, *buffer* it for some time, and then *forward* it to a reading protocol at an opportune time. Finally, we classify unconnected protocol signals such as `done`, `ready`, `stop`, `pal` and `pkt` as *converter-generated* signals, allowing the converter

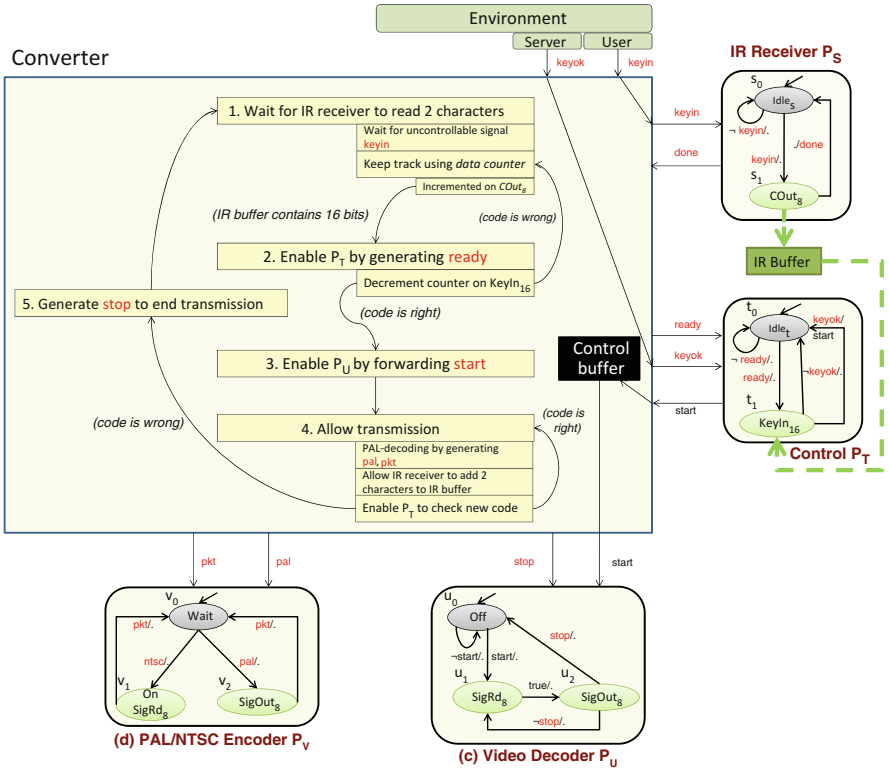



**Fig. 5.6** Illustration of protocol conversion for the set-top SoC case study

to read or write them at any time without restriction. The technical details of signal classification appear later in Sect. 6.3.1.

2. *Converter generation:* A converter is generated using the following steps.

- *Oversampling:* All input SKS are oversampled to describe their behaviour with respect to a common clock. For the set-top box example, all SKS shown in Fig. 5.1 are oversampled (using the clock automaton in Fig. 5.3) so that their behaviours are described with respect to the common SoC master clock *mclk*. This step reduces the problem of creating protocol converters for multi-clock protocols to the problem of creating protocol converters for synchronous protocols.
- *Synchronous composition:* The synchronous composition (Definition 4.2) of the oversampled SKS protocols is computed.
- *Converter generation algorithm:* (Details appear in Chap. 6 and Appendix A) In Sect. 6.4, we propose a converter generation algorithm based on module checking. This algorithm reads as inputs a composition of protocols, all system-level CTL constraints, and the classification of control signals. It

then generates a converter, if one exists, which is guaranteed to control the protocols to satisfy system-level constraints.

Figure 5.6 shows how a converter, automatically generated using our algorithm, can control the four IPs of the set-top box SoC example. Figure 5.6 provides an abstract view of the converter (converters are formalised later in Sect. 6.3). The converter shown in Fig. 5.6 operates as follows. It first allows the IR receiver to read two characters from the user and add them to the IR buffer. This step involves waiting for the uncontrollable input `keyin` (provided by the user) twice, which results in the writing of 16-bits of data onto the IR buffer (two visits to $COut_8$ data-label in the IR receiver). The converter tracks the amount of data present in the IR buffer by using a counter (intuitively described in Sect. 5.1, and formalized later in Sect. 6.2). Each time IR receiver adds 8-bit data to the buffer by visiting the label $COut_8$, this counter is incremented by 8. Next, in step 2, the converter enables the control unit by generating the signal `ready`. The control unit reads the two-character key in the IR buffer (which decrements the counter on IR buffer by 16) and sends them to the remote server (part of the SoC's environment). The server responds with the signal `keyok` if authentication succeeds. In case authentication fails, the converter returns to step 1. Otherwise, the converter *buffers* the shared signal `start` emitted by the control unit. Then, in step 3, the converter forwards the buffered signal `start` to the video decoder to enable decoding. In step 4, the converter provides the generated signals `pkt` and `pal` to the encoder such that the transmission continues. At the same time, it tracks the amount of data put in the IR buffer by the user (using the counter on IR buffer). When a new key is present, the control unit is enabled to authenticate the new key. The converter moves to step 5 when a new key fails to authenticate. In that case, the converter generates the signal `stop` to disable the decoder, and then moves back to step 1.

3. *Composition of converter and protocols:* A converter serves as an interface between the protocols their control signals (uncontrollable, shared, and generated). The composition of a converter and protocol is called their *lock-step* composition (defined later in Sect. 6.3) because the converter enforces state-by-state control over the protocols. The lock-step composition is a SKS, and it is guaranteed to satisfy all system-level constraints provided by the user as an input to the converter generation algorithm.

   For the set-top box example, the converted system shown in Fig. 5.6 is the lock-step composition of the converter and the four protocols, and is guaranteed to satisfy the system-level constraints provided in Sect. 5.1.

## 5.4 Conclusions

This chapter presented our design methodology to build SoCs from on-chip protocols with possible protocol mismatches between them. We presented control, clock and data-width mismatches that can affect the interaction of on-chip protocols. We also discussed how clock mismatches between on-chip protocols using different clocks can be resolved together using oversampling. Finally, we presented the steps involved in generating converters to build SoCs from mismatching protocols.

# Chapter 6
# Automatic Protocol Conversion

In Chap. 4, we discussed the characteristics of SoCs and their behavioral models in terms of Synchronous Kripke structures. We have also presented SoC boilerplates to capture temporal logic formulas to succinctly and precisely capture the desired behaviors of SoCs. In Chap. 5, we discussed the problem of *protocol mismatches*, where independently-developed IPs are unable to "correctly" communicate with each other to realize the desired system. We showed how clock mismatches can be resolved using oversampling, and presented a methodology to resolve control and data-width mismatches using an automatic protocol conversion algorithm. In this chapter, we present the details of the protocol conversion methodology, primary focusing on the technical intricacies of dealing with data-width mismatches and the communication between the IPs and the converter to be generated to resolve the IP protocol mismatches. We also provide an outline of how converter generation can be achieved via the module checking technique described in Chap. 3.

## 6.1 Illustrative Example

Consider a SoC containing a master and a slave connected via the AMBA system bus Fig. 6.1,[1] the objective is to connect a consumer master to a slave memory block. The SoC bus requires the master to first request bus access[2] through its arbiter and once access is granted, it can attempt to read data from the slave. We consider that the master and slave have some control and data mismatches, which hinder in their integration into the AHB system.

**An Abstract Model.** In order to illustrate such mismatches, we consider the abstracted behaviors of the SoC as presented in Fig. 6.2. Assume that the data-bus

[1] A similar example with two masters and one slave was presented in Fig. 4.1 in the last chapter.

[2] For the time being, readers can ignore the presence of "Converter" between the Master and the SoC Bus.

R. Sinha et al., *Correct-by-Construction Approaches for SoC Design*,
DOI 10.1007/978-1-4614-7864-5_6, © Springer Science+Business Media New York 2014

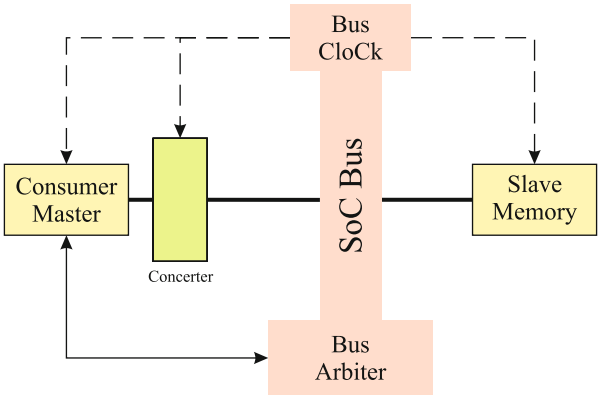



**Fig. 6.1** Convertibility verification overview

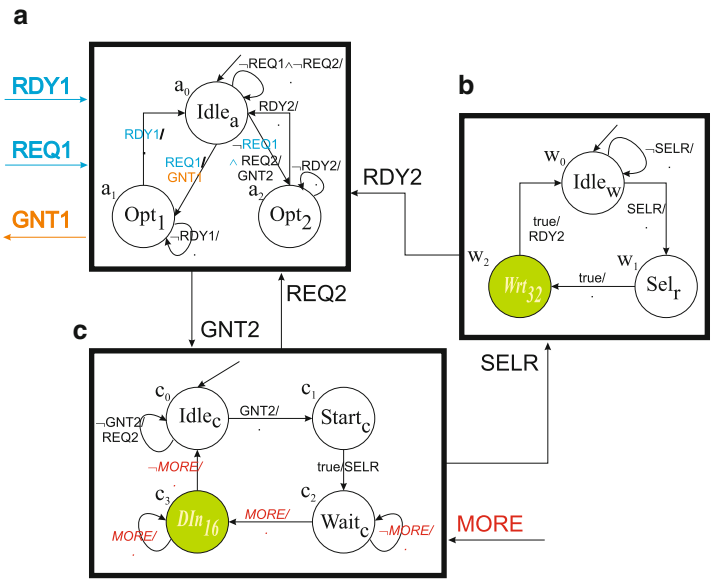



**Fig. 6.2** Representation of the on-chip protocols of the IPs in Fig. 6.1. (**a**) Arbiter. (**b**) Slave writer. (**c**) Consumer master

size of the AMBA AHB is 32 bits and all IPs use the bus clock to execute. Recall that, in our representation scheme (Definition 4.1 in Sect. 4.1), a transition is enabled and triggered only when the input signals (represented by conjunction of propositions) are present or absent (represented by negation). The transitions when executed can result in a set of outputs (or no output as represented by "·").

Figure 6.2a shows the protocol $P_A$ of a bus arbiter that can arbitrate bus access between two masters (even though the SoC currently contains only one master).

In its initial state $a_0$, the arbiter awaits bus request signals `REQ1` or `REQ2` from the masters, and grants access to the first requester by issuing the corresponding grant signal `GNT1` or `GNT2` respectively. In case both request signals are present at the same time, it gives access to master 1 by default. Master 2 is given access only when `REQ2` is present and `REQ1` is absent. Once access has been granted, the arbiter waits for the completion of a transfer by awaiting the `RDY1` or `RDY2` signals (depending on the active master), and then moves back to its initial state.

The slave memory block's writer protocol $P_W$ is shown in Fig. 6.2b. In its initial state $w_0$, the protocol awaits the signal `SELR` (signifying a read request by the current master), and moves to state $w_1$. From $w_1$, in the next tick of the bus clock, the protocol enters state $w_2$. This transition results in the slave writing a 32-bit data packet to the SoC data bus (represented by the label $Wrt_{32}$ of the state $w_2$). After data is written, the protocol resets back to $w_0$.

The protocol $P_C$ for the consumer master is shown in Fig. 6.2c. From its initial state $c_0$, it keeps requesting bus access by emitting the `REQ2` signal. When the grant signal `GNT2` is received, it moves to state $c_1$ where it emits the control signal `SELR` (requesting a read from the slave) and moves to state $c_2$. It then awaits the control signal `MORE` to move to state $c_3$ (labeled by $DIn_{16}$), where it reads a 16-bit data packet from the SoC's data bus. If the signal `MORE` is still available, the protocol can read multiple times from the data bus by making a transition back to $c_3$. When `MORE` is de-asserted, the protocol resets back to $c_0$.[3]

**The Protocol Mismatches.** The various parts of the AMBA AHB system shown in Fig. 6.2 have the following inconsistencies or mismatches that may result in improper communication between them:

1. *Data-width mismatches*: The consumer master has a data-width (word size) of 16-bits which differs from the word size of 32-bits for both the AMBA AHB bus and the slave memory. This difference in word sizes may result in improper or lossy data transfers. For instance,
   (a) **Overflows**: During one write operation, the slave protocol places 32-bits data to be read by the master onto the data-bus of the AMBA AHB (of size 32-bits). However, it is possible that the slave protocol attempts to write more data before the previous data has been completely read by the master. In this case, some data loss may occur due to the overflow as the data-bus can only contain 32-bits.
   (b) **Underflows**: Similarly, if the master attempts to read data before the slave protocol has placed any data onto the data bus, an underflow may happen.
2. *Control mismatches*: The exchange of some control signals between the various IPs of the system may result in deadlocks, data errors and/or the violation of bus

[3]The use of `MORE` captures a typical *burst transfer* operation in SoCs where a master may allow multiple data transfers during the same bus transaction.

policies when an IP produces some signals that are not consumed any other IP or when an IP expects a signal that is not produced by any other IPs. For instance,

(a) **Lack of synchronization**: It is possible that the consumer master is reading data from the data-bus whereas the arbiter has reset back to its initial state (to signal the end of the transaction). This is possible because the reset signal `RDY2` might be issued by the slave and read by the arbiter while the master is still reading data.
(b) **Missing control signals**: The control signal `MORE`, required by the consumer master to successfully read data is not provided by any other IPs in the system. Without this signal, the master would deadlock at state $c_2$.

3. *Clock mismatches*: There can be clock mismatches; however as discussed in Chap. 5, one can deal with clock mismatches by using oversampling (Definition 5.2). Oversampling each on-chip protocol creates a fully synchronous system. Henceforth, in this chapter we only consider the control and the data mismatches.

**Objective.** The objective of correct SoC design is to resolve the protocol (data and control) mismatches and additionally ensure the satisfaction of desired functionalities. We will show that the above objective can be encoded as CTL properties over states of IPs and SoC, and applying model/module checking based technique to generate a converter (if necessary and possible) which when composed with the given IPs satisfies the CTL properties (thereby resolving protocol mismatches and ensuring the satisfaction of desired functionalities).

## 6.2 Modeling Data as Labels on States

The first step in addressing data-width mismatches is to appropriately represent the size of the data being exchanged by IPs in an SoC and the *channel* associated with this data exchange. A channel here represents the buffer that is used for storing and retrieving the data being exchanged. Consider, for example, the 32-bit AMBA ASB data bus of the SoC in Fig. 6.1. Recall that SKS is used to represent the behavioural model of IPs and the communication model is concerned with the capturing the fact that some data (signal) is being exchanged between the on-chip protocols, rather than the contents of the data. Therefore, data operations can be captured by the data labels of the states in the SKS. Each data label must provide the following information:

1. *Data source*: The data label must indicate the *data channel* to (from) which the data is written (read). Examples of data channels include the data bus and shared memories.
2. *Read or write*: Each label must represent whether the data is being consumed or produced.

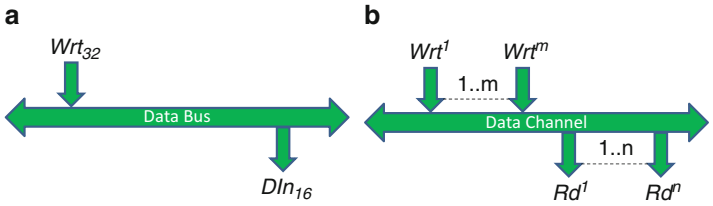



**Fig. 6.3** Data operations for a data channel. (**a**) Data operations for the case study. (**b**) Data operations in general

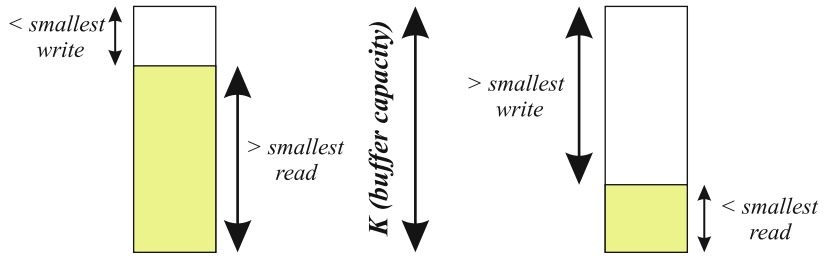
3. *Size of data*: Each label must indicate the amount of data, in bits, being transferred.

The above information corresponding to the data labels can be captured as a *mapping* of each data label to a source, type, and size. For the SoC shown in Fig. 6.2, the data label $Wrt_{32}$ of the slave writer state $w_2$ is mapped to a write operation of 32 bits on the data bus. Similarly, the label $DIn_{16}$ of the consumer master state $c_3$ is mapped to a read operation of 16 bits from the data bus of the SoC.

In a more generic setting, SoC can contain multiple data channels and multiple read and write operations (captured by data labels in the SKS) for each data channel. For each data channel, we can assume that there are $n$ data labels corresponding to read operations and $m$ data labels corresponding to write operations in the SKS. For example, for the on-chip protocols presented in Fig. 6.2, the consumer master and the slave writer communicate over a common medium—the SoC data bus. The data labels $DIn_{16}$ and $Wrt_{32}$ in the SoC correspond to read and write operations from/to the data bus. Hence, in this case, $n = 1$ and $m = 1$, as shown in Fig. 6.3a. Figure 6.3b illustrates the more general case with $m$ write and $n$ read operations for a data channel. Each data operation (read or write) has its own *size*, or the number of bits transferred.

### 6.2.1 Data Constraints

Given the data labels, there are two types of constraints that are imposed for correct behavior. The first type can be viewed as "static" constraint, which is based on purely the size of the data being exchanged between communicating IPs. In other words, static data constraint does not rely on the dynamic behavior of the IPs. This constraint imposes a lower bound on the size of the data channel. The other type of data constraint is dynamic and is based on the behaviour of IPs. This constraint is specified using temporal logic and can be imposed on the communication behavior by appropriately constructing converters (as mentioned before). In the following, we will describe both these types of constraints.




Case 1: Not enough buffer space for a write: allow at least the smallest read.

Case 2: Not enough data on buffer for a read: allow at least the smallest write.

**Fig. 6.4** Computation of buffer bounds: Eq. 6.2.2

**Lower Bound on Medium Size.** The lower bound on the size of the data channel via which data is being exchanged naturally depends on the size of the data being written to it and consumed from it. It is immediate that the lower bound is equal to the maximum size of the data that can be written to or read from a channel. That is, if $k$ is the size of the data channel, then

$$k \geq \max \begin{pmatrix} \texttt{Size}(Wrt^1), \texttt{Size}(Wrt^2), \ldots, \texttt{Size}(Wrt^n), \\ \texttt{Size}(Rd^1), \texttt{Size}(Rd^2), \ldots, \texttt{Size}(Rd^m) \end{pmatrix} \tag{6.2.1}$$

where $Wrt^i$ (respectively, $Rd^i$) is the data label corresponding to the $i$-th type of write (respectively, read) operation and $\texttt{Size}(\cdot)$ represent the size of the data read and write. For instance, going back to Fig. 6.2, $\texttt{Size}(DIn_{16}) = 16$ and $\texttt{Size}(Wrt_{32}) = 32$, i.e., the size of the SoC data bus, $k \geq 32$.

In addition to the above lower bound, it is also necessary to impose another condition on the data channel size. This condition requires that

1. If the channel cannot accept any write operation, then it must have enough data to allow a read operation (shown as case 1 in Fig. 6.4).
2. Similarly, if the channel contains data that is smaller than what any IP can read, then the channel must have enough space to allow a write operation (shown as case 2 in Fig. 6.4).

The above conditions can be expressed as a lower bound on the size of the medium $k$. We require that $k$ must be large enough to allow the minimum sized read in the system when it does not have enough data for even the minimum sized write, and vice versa. The bound on $k$ can be stated as follows.

$$k \geq \begin{bmatrix} \min\{\texttt{Size}(Wrt^1), \texttt{Size}(Wrt^2), \ldots, \texttt{Size}(Wrt^n)\} \\ + \\ \min\{\texttt{Size}(Rd^1), \texttt{Size}(Rd^2), \ldots, \texttt{Size}(Rd^m)\} \end{bmatrix} - \texttt{GCD} \tag{6.2.2}$$

where GCD is the greatest common divisor of the read and write data sizes. The *minimum* value of read/write operations and the GCD are used to capture the

tightest lower bound for $k$. The minimum value provides the information regarding the smallest size for the read operation and the write operation. The GCD is subtracted to capture the fact that we are interested in the smallest number of read (or write) operations necessary to allow for at least one write (or read, respectively).

For our running example, $k \geq [32+16]-16$, i.e., $k \geq 32$. While in this example, the necessary lower bounds as per the above two requirements are identical, there are cases where they may be different. Consider an example where

$$\texttt{Size}(Wrt^1) = 10 \text{ and } \texttt{Size}(Rd^1) = 4 \text{ and } \texttt{Size}(Rd^2) = 6$$

Then as per Eq. 6.2.1, $k \geq 10$, and as per Eq. 6.2.2, $k \geq 12$, which requires that $k$ must be greater than equal to 12. Consider that 10 bits of data is written to the channel, the maximum number of reads (of size 4 bits) that are possible is 2 which results in 2 bits of data still present in the channel. At this point, no read operation is possible, however it is necessary to allow a write operation and that is why, the minimum size of the channel is set to 12.

Note that the lower bounds discussed above do not ensure deadlocks due to data-width mismatches between the IP-protocols can be avoided. This is because the protocols are not taken into consideration for identifying these bounds. Our objective, in this section, is to show the subtleties of the impact of data-width irrespective of the behavior exhibited by the IPs following different protocols. If the size of the channel is less than the lower bound, then there is a possibility of a deadlock. On the other hand, if the size of all the writes are the same and the size of all the reads are the same, then one can ensure that there will be no deadlock as long as the channel size is greater than the lower bound.

**Dynamic or Temporal Data Constraints.** So far, we have discussed the minimum size of the data channel which is likely to help avoid deadlocks in the communication between IPs. Dynamic data constraints, on the other hand, ensure the correct communication between IPs in terms of data-width. These constraints are impose temporal restrictions on the behavior of the IPs in terms of the data being written to and read from the data channel. Knowing the capacity $k$ of the channel and ensuring that it satisfies the lower bound constraint as mentioned above, one can also keep track of the amount of data (yet to be consumed) present in the channel. A data counter, say `cntr`, can be maintained to capture the amount of data being written to and read from the channel. Whenever an IP moves to a configuration/state with a data label $Wrt^i$, the counter is incremented by $\texttt{Size}(Wrt^i)$; on the other hand, whenever an IP moves to a configuration/state with a data lable $Rd^j$, the counter is decremented by $\texttt{Size}(Rd^j)$. Finally, the temporal property

$$\texttt{AG}(0 \leq \texttt{cntr} \leq k) \tag{6.2.3}$$

if satisfied by the behavior of the IPs, ensures that there is never any data overflow or underflow. Note that, in general such a property may not be satisfied by the IPs

(due to protocol mismatch) and as a result, it is the job of a coverter to restrict the communication between the IPs in such a way that the above property is satisfied.

### 6.2.2 Control Constraints

As noted before, protocol mismatch due to lack of synchronization and missing signals are referred to as the control mismatch. Avoiding such mismatch will require that the IPs' communication exhibit certain desired properties over control or are forced to exhibit the same due to the presence of a converter (responsible for mitigating protocol mismatches). We represent such properties in the logic of CTL.

Consider the illustrative example discussed in this chapter. One can require that the consumer master can always eventually read data from the system data bus. Such a property can be easily expressed using boiler plates described in the previous chapter. The CTL formula representing such a property will be

$$\texttt{AGEF}DIn_{16}$$

A similar property from the writer's perspective is that the writer can always eventually write data.

$$\texttt{AGEF}DOut_{32}$$

An interesting property showing the interplay between control and data is when we require a control state to be reachable upon the satisfaction of certain data constraints, or vice versa. For instance, one may require that when all IPs are at their idle state, the data counter `cntr` must be equal to 0, or that the corresponding data channel must be empty.

$$\texttt{AG}(\text{IPs in idle state } \Rightarrow (\texttt{cntr} = 0))$$

## 6.3 Converters: Description and Control

Given the data and control constraints in terms of CTL properties, the objective is to ensure that communicating on-chip protocols and the data channels conform to those constraints, thereby, ensuring the absence of protocol mismatches. In the event, the communicating protocols do not satisfy the constraints, a converter (an intermediary) is installed between the on-chip protocols to regulate the communication between the on-chip protocols such that the data and control constraints are satisfied. Without loss of generality, we will consider that there are two protocols, $P_1$ and $P_2$, and a converter $\mathscr{C}$, for the notational convenience. The parallel composition

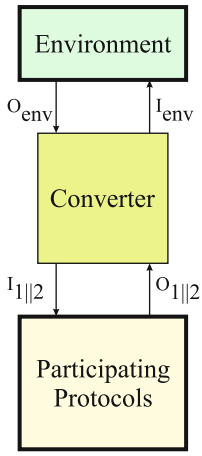



**Fig. 6.5** The I/O connections between environment, converter and protocols

of $P_1$ and $P_2$ represented by the synchronous composition (see Definition 4.2) of SKSs (see Definition 4.1).

In order to regulate the communication between protocols, the converter acts as an intermediary which can relay and buffer signals between on-chip protocols as necessary. If all signals are exchanged between the protocols, then their parallel composition is *closed*. However, often the environment can play a vital role by providing inputs to on-chip protocols; such systems are called *open* systems. The function of the converter is to relay signals between protocols, between environment and protocols (when the system is open), and generate new signals for the protocols if necessary. The relay of signals between protocols can be delayed (by buffering) and therefore, can change the behavior of the system. In the following, we will describe the relationship of the input/output protocol signals of the protocols, environment and converter and then discuss the different types of signals based on what a converter can do with them (relay, buffer, hide, and generate).

### 6.3.1 I/O Relationship Between Converter, Environment and On-Chip Protocols

Consider the Fig. 6.5 to better understand the input/output relations between the converter, environment and on-chip protocols. Output from the environment and on-chip protocols are inputs to converter, $I_{\mathscr{C}} = O_{env} \cup O_{1||2}$; while input to environment and on-chip protocols are produced as output from the converter $O_{\mathscr{C}} = I_{1||2} \cup I_{env}$. Of course, some of the outputs from the on-chip protocols are consumed by the environment, while some of the outputs from the environment are consumed by the on-chip protocols: $I_{env} \subseteq O_{1||2}, O_{env} \subseteq I_{1||2}$. This indicates that the SoC is an *open* system which uses a subset of its inputs and outputs

to interact with its environment (concepts of open and closed systems appear in Chap. 3).

Going back to our illustrative example in Fig. 6.2, consider that a converter is placed as an intermediary between the on-chip protocols (arbiter, master and the slave) and the environment (responsible for producing necessary signals, e.g., REQ1). The converter reads all the signals generated by the environment and the participating protocols.

$$I_{\mathscr{C}} = \{\texttt{REQ2}, \texttt{RDY2}, \texttt{GNT1}, \texttt{GNT2}, \texttt{SELR}, \texttt{REQ1}, \texttt{RDY1}\}$$

where signals REQ1 and RDY1 are generated in the environment ($O_{env}$= {REQ1,RDY1}) and all other signals are emitted by the protocols. Similarly, the converter emits all signals meant for the environment and the participating protocols.

$$O_{\mathscr{C}} = \{\texttt{REQ1}, \texttt{REQ2}, \texttt{RDY1}, \texttt{RDY2}, \texttt{GNT2}, \texttt{SELR}, \texttt{MORE}, \texttt{GNT1}\}$$

where the signal GNT1 is emitted by the converter to the environment ($I_{env} = \texttt{GNT1}$) and all other outputs are meant for the protocols. It can be seen that the signal GNT1 emitted by the converter to the environment is read by the converter from the on-chip protocols and that the signals REQ1 and RDY1 read by the converter from the environment are emitted as converter outputs to the protocols (Fig. 6.6).

### 6.3.2 Capabilities of the Converter

As mentioned in the previous sections, a converter can guide participating protocols by using the converter-environment-protocols I/O relationship. However, it should be noted that the converter has limited capabilities.

1. *Relay of signals*: Converter can simply act as a relay and transfers signals between environment and on-chip protocols. For example, in Fig. 6.2 the converter must relay the input signal RDY1 generated by the environment to the arbiter.
2. *Disabling of signals*: A converter may read an output signal from one on-chip protocol but hide it from another. This feature may help *disable* certain undesirable behaviour in the latter protocol. For example, in Fig. 6.2, a converter may disable the signal SELR, emitted by the consumer master from reaching the slave writer if there is not sufficient data contained in the shared memory.
3. *Buffering and event forwarding*: A converter may read and *buffer* a signal from one on-chip protocol and then forward it to another protocol at a later stage. This may be needed to synchronize the protocols that do not necessarily move in a lock-step fashion, i.e., when one of them is emits a signal, the other is not ready to consume. For example, a converter may buffer the signal REQ2 emitted by the consumer master and forward it to the arbiter at a later stage.

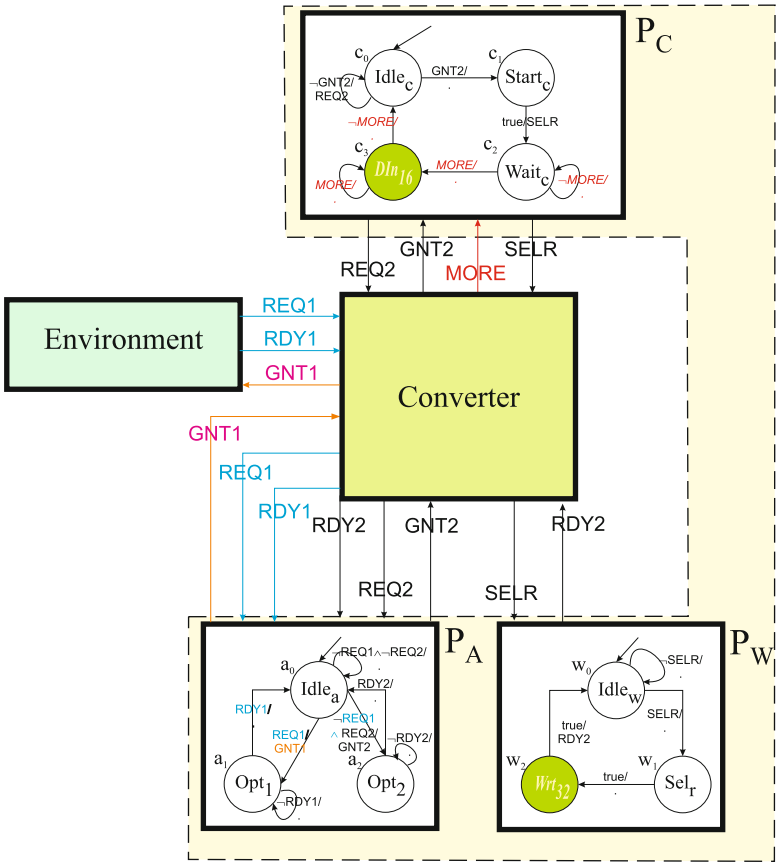



**Fig. 6.6** The I/O connections between on-chip protocols and the converter

4. *Generation of missing control signals*: A converter may *artificially generate* signals that are not emitted as outputs by any participating protocols or the environment, but are required by the protocols as inputs. For example, a converter may generate the missing control signal `MORE` to ensure progress in the consumer master protocol shown in Fig. 6.2.

### *6.3.3 Types of Input/Output Signals of Converter*

The capabilities of the converter is restricted to specific types of signals. For instance, a converter cannot delay and disable environment signals; it cannot artificially generate any signals that are produced by the environment or the protocols. As delaying and disabling can be viewed in similar light–disabling can be

**Table 6.1** Types of signals in a converter and notations

| | Environment → IP | IP→IP | IP→ Environment | →IP |
|---|---|---|---|---|
| Type | Uncontrollable | Buffered | Uncontrollable | Generated |
| Input | $I^{unc}_{\mathscr{C}} = O_{env}$ | $I^{buf}_{\mathscr{C}} = O_{1\|\|2} \cap I_{1\|\|2}$ | $I^{emit}_{\mathscr{C}} = I_{env} = O_{1\|\|2} \setminus I^{buf}_{\mathscr{C}}$ | |
| Output | $O^{unc}_{\mathscr{C}} = O_{env}$ | $O^{buf}_{\mathscr{C}} = I^{buf}_{\mathscr{C}}$ | $O^{emit}_{\mathscr{C}} = I^{emit}_{\mathscr{C}}$ | $O^{gen}_{\mathscr{C}} = I_{1\|\|2} - (O^{buf}_{\mathscr{C}} \cup O^{unc}_{\mathscr{C}})$ |

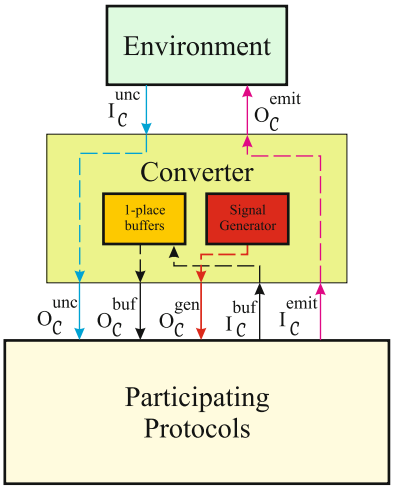



**Fig. 6.7** The different types of inputs and outputs of the converter

viewed as infinite time delay; we will consider broadly three types of input/outputs in a converter:

- *Uncontrollable* [RW89]: These signals cannot be delayed or disabled by the converter.
- *Bufferred*: These signals can be delayed or disabled by the converter.
- *Generated*: These signals are generated by the converter as neither the protocols nor the environment produce these.

The uncontrollable events are further partitioned into signals emitted by the environment and protocols. In short, the types closely follow the sources of the signals. Table 6.1 succinctly represents the types. The first row shows the source and destination of signals; the second row shows whether the signal can be controlled or generated. The last two rows shows the notations we use to represent the corresponding input and output signals in the converter $\mathscr{C}$. Figure 6.7 shows the I/O connections between converter, participating protocols and the environment. Buffered I/O is carried out when output signals emitted by the protocols are read in the converter's buffers and emitted at a later stage. Generated I/O happens when the converter artificially emits input signals for the protocols. Finally, all uncontrollable inputs are read by the converter from the environment and are provided to the protocols immediately without buffering, and vice versa.

### 6.3.4 Description of Converter

**Notations.** Given a boolean formula $b$ as a conjunction of literals $i$'s and their negations, we say that $i \in b$ if and only if $i$ is one of the literal appearing in a conjunct in $b$.

Given the types of converter inputs and outputs signals, we now define a converter as a finite state machine. Each state in the converter represents a configuration of the converter in terms of its control state and the set of the buffered signals. Each transition between converter states represents an evolution of the converter from one configuration to another. This converter SKS is described in Definition 6.1.

**Definition 6.1 (CSKS).** A converter SKS $\mathscr{C}$ is a $\langle S_{\mathscr{C}}, s_{\mathscr{C}0}, I_{\mathscr{C}}, O_{\mathscr{C}}, R_{\mathscr{C}}, clk\rangle$ where:

1. $S_{\mathscr{C}}$ is a finite set of states with $s_{\mathscr{C}0}$ being the initial state.
2. $I_{\mathscr{C}}$ is the set of inputs partitioned into the following sets:
   a. The set $I_{\mathscr{C}}^{unc}$ of uncontrollable environment inputs,
   b. The set $I_{\mathscr{C}}^{buf}$ of buffered inputs, and
   c. The set $I_{\mathscr{C}}^{emit}$ of uncontrollable protocol inputs.
3. $O_{\mathscr{C}}$ is the set of outputs partitioned into the following sets:
   a. The set $O_{\mathscr{C}}^{unc} = I_{\mathscr{C}}^{unc}$ of outputs to the protocols,
   b. The set $O_{\mathscr{C}}^{buf} \subseteq I_{\mathscr{C}}^{buf}$ of buffered events to the protocols,
   c. The set $O_{\mathscr{C}}^{emit} = I_{\mathscr{C}}^{emit}$ of outputs to the environment, and
   d. The set $O_{\mathscr{C}}^{gen}$ of generated outputs
4. $R_{\mathscr{C}} \subseteq [S_{\mathscr{C}} \times I_{\mathscr{C}}^{buf}] \times B(I_{\mathscr{C}}) \times 2^{O_{\mathscr{C}}} \times [S_{\mathscr{C}} \times I_{\mathscr{C}}^{buf}]$ is a total transition relation. For any transition $(\mathrm{c}, b) \xrightarrow{I/0} (\mathrm{c}', b')$, the following conditions hold:
   a. *Immediate emission of uncontrollable signals*:
      i. $\forall i \in I_{\mathscr{C}}^{unc} : (i \in I \Leftrightarrow i \in O)$
      ii. $\forall i \in I_{\mathscr{C}}^{emit} : (i \in I \Leftrightarrow i \in O)$
   b. *Correct buffering:*
      i. $O_{\mathscr{C}}^{buf} \cap O \subseteq b$
      ii. $b' = (b/[O_{\mathscr{C}}^{buf} \cap O]) \cup [O_{\mathscr{C}}^{buf} \cap \{o \mid o \in I\}]$
5. All transitions in $R_{\mathscr{C}}$ are synchronized with the ticks of the clock $clk$.

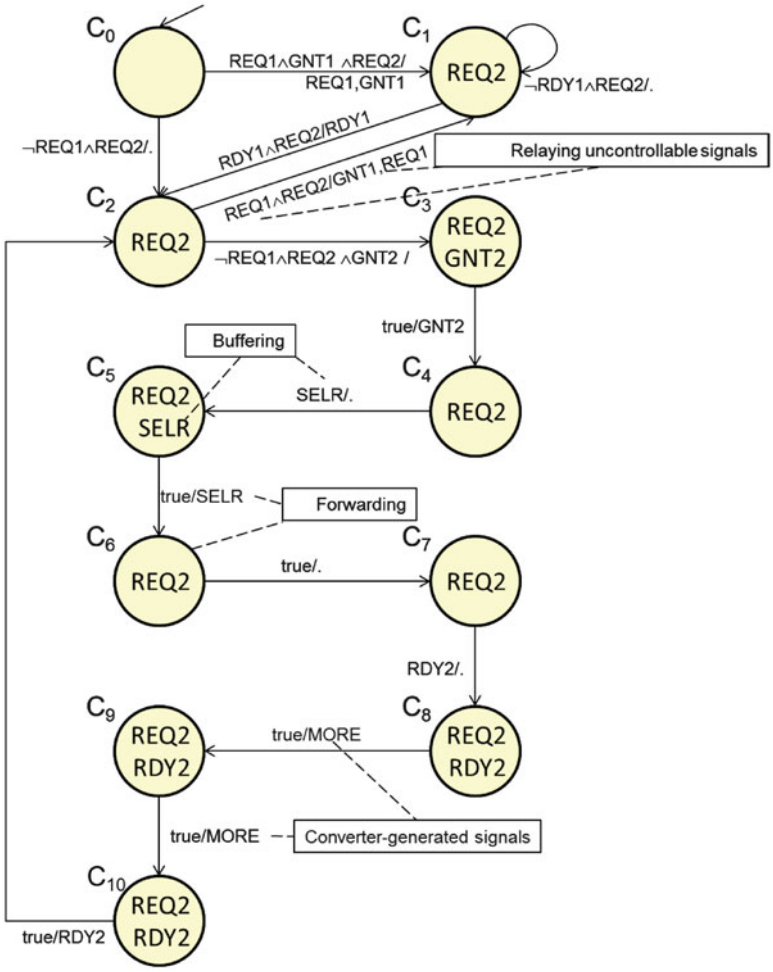



**Fig. 6.8** A converter for the SoC example presented in Fig. 6.2

The above definition captures the converter behavior as follows. The converter inputs are all possible signals from the environment and the protocols. The converter emits the uncontrollable signals that are exchanged between the environment and the protocols immediately (as soon as it consumes such signals). The converter produces outputs for a protocol from its buffer if such outputs can be generated by some other protocol. Figure 6.8 illustrates a sample converter for the SoC shown in Fig. 6.2. Using this as an illustrative example, we explain the above definition:

- *States and initial state (point 1)*: The converter in Fig. 6.8 has a finite set of states $\{C_0, \ldots, C_{10}\}$ with $C_0$ being the initial state.

- *Input and output classifications (points 2 and 3)*: As discussed in Table 6.1, the converter is able to read uncontrollable signals from both the environment and the protocols, as well as buffered inputs from the protocols. For the converter shown in Fig. 6.8, `REQ1` and `RDY1` are uncontrollable environment inputs, `GNT1` is an uncontrollable protocol input, and `REQ2`, `RDY2` and `SELR` are buffered inputs from the protocols. Similarly, the converter emits uncontrollable signals to the environment, and uncontrollable, buffered and generated signals to the protocols. For the converter shown in Fig. 6.8, `REQ1` and `RDY1` are uncontrollable outputs to protocols, `GNT1` is an uncontrollable output to the environment, `REQ2`, `RDY2` and `SELR` are buffered outputs to the protocols, and `MORE` is a generated output to the protocols.
- *Immediate relay of uncontrollable signals (point 4a)*: The converter immediately relays any signal that is to be exchanged between the on-chip protocols and the environment. For example, in the transition from state $\mathrm{C}_2$ to $\mathrm{C}_1$, the converter reads the uncontrollable signals `REQ1` and `GNT1`, emitted by the environment and the arbiter protocol respectively. The converter immediately emits these signals as outputs of the same transition.
- *Correct buffering (point 4b)*: The converter maintains a one-place buffer for all signals that are exchanged between the protocols. This allows representing the buffered signals at any state $\mathrm{C}$ in the converter as a set $b$. Converter transitions are of the form $(\mathrm{C}, b) \xrightarrow{I/O} (\mathrm{C}', b')$, where $b$ is the set of buffered signals at state $\mathrm{C}$ and $b'$ is the set of buffered signals at state $\mathrm{C}'$. In Fig. 6.8, the state of the buffer at a converter state is shown as the label of the state. For example, in state $\mathrm{C}_0$, the buffer is empty, and at state $\mathrm{C}_1$, the buffer is $\{\texttt{REQ2}\}$. Point 4b of Definition 6.1 constrains every transition of the form $(\mathrm{C}, b) \xrightarrow{I/O} (\mathrm{C}', b')$ in the converter as follows. Firstly, we require a buffered signal to be present in the buffer $b$ for it to be emitted during the transition from $(\mathrm{C}, b)$ (point 4b.i). For example, the transition from state $\mathrm{C}_5$ to $\mathrm{C}_6$ in Fig. 6.8 involves the emission of the buffered signal `SELR`. This transition is allowed only because the buffer at state $\mathrm{C}_5$ contains `SELR`. Secondly, point 4b.ii requires that when a transition is taken from state $(\mathrm{C}, b)$ to state $(\mathrm{C}', b')$, the buffer $b'$ at state $\mathrm{C}'$ must contain any retained buffered signals from $b$ (those that were not emitted during the transition) and all buffered signals read from the protocols during the transition. For example, in Fig. 6.8, the buffer is updated from $\emptyset$ to $\{\texttt{SELR}\}$ to include the buffered signal `SELR` read during the transition from state $\mathrm{C}_4$ to $\mathrm{C}_5$.

  Using sets to contain buffered signals means that a converter can remember only a single instance of a signal. In other words, we use converters with *one-place* buffers for control signals. One-place buffers guarantee that the converter buffer size is at most equal to the total number of signals that can be exchanged by the protocols, which in turn, ensures that the converter SKS model size is finitely bounded.
- The converter can generate missing control signals that are needed by the on-chip protocols during any transition without restriction. For example, in Fig. 6.8, the converter generates the missing `MORE` during the two transitions from state $\mathrm{C}_8$ to $\mathrm{C}_9$ and $\mathrm{C}_9$ to $\mathrm{C}_{10}$.

### 6.3.5 *Lock-Step Composition of Converter and On-Chip Protocols*

A converter acts as an intermediary between protocols, and between protocols and the environment. It relays environment signals that it cannot enable or disable, suppresses/buffers some signals and generates signals as appropriate. The interaction between the converter and the protocols result in a new SKS model, whose inputs are the signals that are produced by the environment (for the protocols) and the outputs are the signals that are consumed by the environment (produced by the protocols). The interaction is described as a *lock-step composition*, the name suggests that for every moves in the protocols behavior, there exists a move in the converter. Figure 6.9 shows the lock-step composition of the converter $\mathscr{C}$ shown in Fig. 6.8 and the synchronous composition (Definition 4.2) of the on-chip protocols shown in Fig. 6.2. As Fig. 6.9 illustrates, the lock-step composition of a converter and protocol (which can be a composition of protocols) is a SKS.

Each state in the lock-step composition corresponds to a state $\mathrm{C}$ in the converter and a state $s$ in the synchronous composition. For example, the initial state of the lock-step composition SKS shown in Fig. 6.9 corresponds to the initial states $\mathrm{C}_0$ of the converter and $(a_0, c_0, w_0)$ of the synchronous composition of the three on-chip protocols. The labels of every state $(\mathrm{C}, s)$ in the lock-step composition are the same as the labels of the constituent protocol state $s$. For example, state $\mathrm{C}_1, (a_1, c_0, w_0)$ has the same labels $Opt_1, Idle_c$ and $Idle_w$ as state $(a_1, c_0, w_0)$. Finally, each transition $(\mathrm{C}, s) \xrightarrow{b/O} (\mathrm{C}', s')$ between any two states $(\mathrm{C}, s)$ and $(\mathrm{C}', s')$ in the lock-step composition is described in terms of the transitions between states $\mathrm{C}$ and $\mathrm{C}'$ in the converter and $s$ and $s'$ in the protocols. Specifically,

1. A transition from $(\mathrm{C}, s)$ to $(\mathrm{C}', s')$ exists if and only if state $\mathrm{C}$ has a transition to $\mathrm{C}'$ and state $s$ has a transition to $s'$. For example, in Fig. 6.9, state $\mathrm{C}_0, (a_0, c_0, w_0)$ has a transition to state $\mathrm{C}_1, (a_1, c_0, w_0)$ because states $\mathrm{C}_0$ and $(a_0, c_0, w_0)$ have the following transitions to states $\mathrm{C}_1$ and $(a_1, c_0, w_0)$ :

$$\begin{aligned} (\mathrm{C}_0, \emptyset) &\xrightarrow{\texttt{REQ1}\wedge\texttt{GNT1}\wedge\texttt{REQ2}/\texttt{REQ1,GNT1}} (\mathrm{C}_1, \{\texttt{REQ2}\}) \\ (a_0, c_0, w_0) &\xrightarrow{\texttt{REQ1}/\texttt{GNT1}} (a_1, c_0, w_0) \end{aligned} \tag{6.3.1}$$

   Henceforth, due to one-to-one correspondence between converter states and control buffers, we omit the reference to the buffers along converter transitions, and note only the converter states. For example, $(\mathrm{C}_0, \emptyset)$ is noted as just $\mathrm{C}_0$.
2. The transition from $(\mathrm{C}, s)$ to $(\mathrm{C}', s')$ contains an input signal $i$ if:
   (a) $i$ is an uncontrollable signal read by the transition from $s$ to $s'$.
   (b) $i$ is read as an input as well as emitted as an output signal in the converter transition from $\mathrm{C}$ to $\mathrm{C}'$.

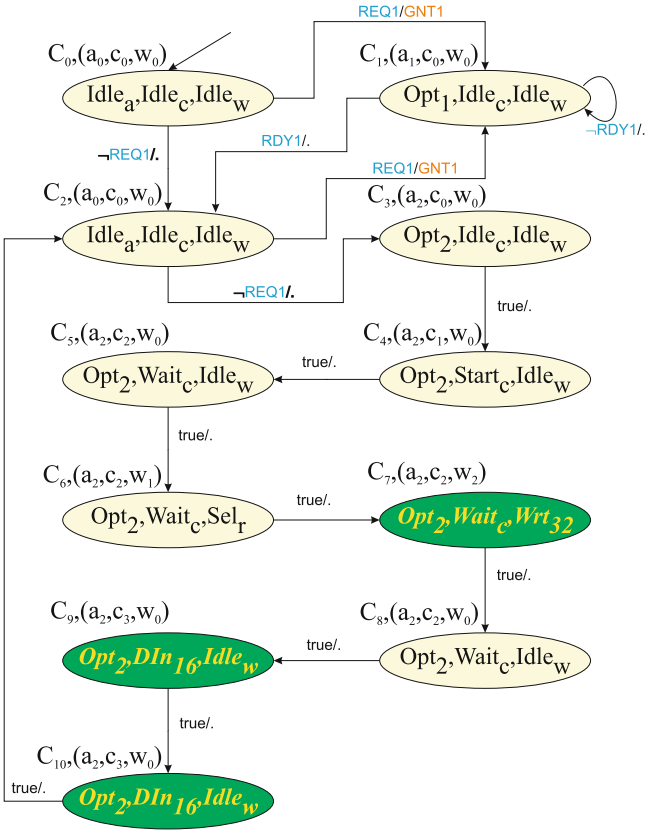



**Fig. 6.9** The lockstep composition of the converter shown in Fig. 6.8 and the composition of the oversampled protocols shown in Fig. 6.2

The two items above ensure that the converter relays uncontrollable signals immediately from the environment to the protocols. In the absence of any uncontrollable signal, the input label of the transition from $s$ to $s'$ becomes `true` (Recall that input labels are boolean functions). For example, the input label of the lock-step transition from $\mathrm{C}_0, (a_0, c_0, w_0)$ to $\mathrm{C}_1, (a_1, c_0, w_0)$ contains the input signal `REQ1` which is read as an input in the protocol transition and relayed by the converter transition (see Eq. 6.3.1).

3. The transition from $(\mathrm{C}, s)$ to $(\mathrm{C}', s')$ contains an output signal $o$ if
   (a) $o$ is an uncontrollable signal emitted in the transition from $s$ to $s'$, and
   (b) $o$ is read as an input as well as emitted as an output signal in the converter transition from $\mathrm{C}$ to $\mathrm{C}'$.

   The two items above ensure that the lock-step composition immediately relays all uncontrollable signals emitted by the protocols to the environment. For

example, the lock-step transition from $C_0, (a_0, c_0, w_0)$ to $C_1, (a_1, c_0, w_0)$ contains the uncontrollable output signal GNT1 which is emitted by the protocol transition and relayed by the converter transition (see Eq. 6.3.1 above).

4. The transition from $(C, s)$ to $(C', s')$ does not contain an input signal $i$ if $i$ is a bufferable signal emitted by the transition from $s$ to $s'$ and read by the converter in the transition from $C$ to $C'$. For example, the transition from state $C_0, (a_0, c_0, w_0)$ to $C_1, (a_1, c_0, w_0)$ in Fig. 6.9 does not contain the bufferable signal REQ2 because it is buffered by the converter.
5. The transition from $(C, s)$ to $(C', s')$ does not contain an output signal $o$ if
   - $o$ is a buffered signal emitted during the transition from $C$ to $C'$, and read as an input in the transition from $s$ to $s'$. For example, in Fig. 6.9, the transition from state $C_5, (a_2, c_2, w_0)$ to $C_6, (a_2, c_3, w_0)$ does not contain the (buffered) signal SELR emitted by the converter in the transition from $C_5$ to $C_6$ (Fig. 6.8).
   - $o$ is a generated signal emitted by the converter during the transition from $C$ to $C'$, and read as an input in the transition from $s$ to $s'$. For example, in Fig. 6.9, the transition from state $C_8, (a_2, c_2, w_0)$ to $C_9, (a_2, c_3, w_0)$ does not contain the (generated) signal MORE emitted by the converter in the transition from $C_8$ to $C_9$ (Fig. 6.8).

The lock-step composition of the protocols in Fig. 6.2 and the converter shown in Fig. 6.8, as shown in Fig. 6.9, executes as follows. From its initial state $(C_0, (a_0, c_0, w_0))$, the converter lets the arbiter respond to the uncontrollable signal REQ1 to move to $(C_1, (a_1, c_0, w_0))$. In this transition, the converter makes REQ1 available to the protocols, and then emits the external output GNT1 (read from the arbiter) to the environment and buffers the signal RDY2 (read from the consumer master). In $(C_1, (a_1, c_0, w_0))$, the system waits for the uncontrollable signal RDY1 to reset to state $(C_2, (a_0, c_0, w_0))$. If however in state $(C_0, (a_0, c_0, w_0))$, the uncontrollable signal REQ1 is not read, a transition to $(C_1, (a_0, c_0, w_0))$ is triggered by the converter. During this transition, the converter reads and buffers the signal RDY2 (generated by the consumer master).

From $(C_2, (a_0, c_0, w_0))$, in the next clock tick, if the uncontrollable signal REQ1 is available, the system moves to state $(C_2, (a_1, c_0, w_0))$. Otherwise, a transition to state $(C_3, (a_2, c_0, w_0))$ is triggered. During this transition, the converter emits the previously buffered signal REQ2 (to be read by the arbiter). It then reads the signals GNT2 , REQ2 from the protocols (emitted by the arbiter and the consumer master respectively) and buffers them. From $(C_3, (a_2, c_0, w_0))$ the converter passes the previously buffered GNT2 to enable a transition to $(C_4, (a_2, c_1, w_0))$. In the next tick, the converter reads and buffers the signal SELR (emitted by the consumer master) by triggering a transition to state $(C_5, (a_2, c_2, w_0))$. Then, the converter passes the buffered SELR signal (to be read by the writer slave) and the system makes a transition to state $(C_6, (a_2, c_2, w_1))$.

The converter then allows the protocols to evolve without providing any inputs. During this transition, the system moves to state $(C_7, (a_2, c_2, w_2))$. Note that $(C_7, (a_2, c_2, w_2))$ is a data-state because it is labelled by $Wrt_{32}$. This means that

when the system reaches $(c_7, (a_2, c_2, w_2))$, the data-bus of the system contains 32-bits resulting in the data counter `cntr` being incremented to 32 (see Sect. 6.2.1 for details).

From $(c_7, (a_2, c_2, w_2))$, the system moves to $(c_8, (a_2, c_2, w_0))$ and the converter reads and buffers the input `RDY2` (emitted by the writer slave). From $(c_8, (a_2, c_2, w_0))$, the converter generates the signal `MORE`, to allow a transition to $(c_9, (a_2, c_3, w_0))$. The converter then again generates `MORE` to move the system to state $(c_{10}, (a_2, c_3, w_0))$. Note that states $(c_9, (a_2, c_3, w_0))$ and $(c_{10}, (a_2, c_3, w_0))$ are data states as they are both labelled by $DIn_{16}$, which represents a 16-bit read operation. Hence, the counter is adjusted (decremented) to have values 16 and 0 respectively. Finally, from $(c_{10}, (a_2, c_3, w_0))$, the converter passes the previously buffered signal `RDY2` (to the arbiter) allowing the system to move to state $(c_2, (a_0, c_2, w_0))$.

## 6.4 Generating Converters Using Module Checking

We have discussed three important aspects in SoC design so far. We have described how the protocols are composed, which represents all possible behavior of the participating protocols (see Definition 4.2 in Sect. 4.1.2). We use temporal logic to formally represent the desired behavior of the protocols with respect to the control signals and data-width (see Sects. 4.2 and 6.2). We have also described how a converter interacts with the protocols and can potentially regulate the behavior in the interaction (see Sect. 6.3).

The final piece in our setup is the method that can be used to *generate* converters (if possible) automatically such that the generated converter interacts with the protocols and satisfy the desired behavior as specified using temporal logic. The method directly follows from the module checking algorithm discussed in Sect. 3.2. Under special cases, when there are no uncontrollable signals and the composition of on-chip protocols is closed, model checking may be used [SRB08a].

Recall that, the central theme for module checking as described in Sect. 3.2 is that

1. A module does not satisfy a property if and only if there exists some restricted behavior in the module which does not satisfy the property (or satisfies the negation of the property) under consideration, and
2. Existence of such a restricted behavior can be verified by generating a "environment" which forces such a restriction.

In Sect. 3.2, we have discussed how such an environment can be automatically generated using a tableau-based algorithm. The same idea can be applied in SoC and protocol conversion. The objective is to restrict the behavior of the protocols in such a way that the restricted behavior satisfies certain desired property. The restriction is realized by generating a converter (if possible). Therefore, the correspondence is as follows:

- The composition of protocols is the module in a module checking.
- The lock-step composition of protocols and the converter (to be generated) corresponds to the parallel composition between the module and its "environment".
- The desired properties of the protocols to be enforced by the converter correspond to the negation of the properties satisfied by the module in some environment.
- The environment identified as part of module checking corresponds to the converter in the context of protocol conversion.

Of course there are some important distinguishing factors—for instance, in module checking, we consider that a state emitting uncontrollable signals does not emit controllable ones (system state) and vice versa (environment state); on the other hand, in protocol conversion, the protocols can produce and consume both controllable and uncontrollable signals from the same state. The difference is due to the difference in the semantics of the parallel composition between module and its environment and the lock-step composition between the protocols and their converter. However, the basic principles of the converter in protocol conversion and the environment in module checking remain identical, i.e., control the controllable signals, relay/allow the uncontrollable ones.

The converter shown in Fig. 6.8 can be generated using our algorithm such that it can control the composition of the protocols shown in Fig. 6.2 such that all mismatches are resolved, and all system-level control and data constraints are satisfied. The resulting lock step composition is shown in Fig. 6.9. We have adapted the module checking approach and developed a converter synthesis approach that is presented in Appendix A.

## 6.5 Concluding Remarks

This chapter presented a unifying approach towards performing protocol conversion with control and data mismatches. Earlier works could only handle control or data mismatches separately or handle both in a restricted manner. The fully automated approach is based on CTL module checking, allowing temporal logic specifications for both control and data constraints. Converters are capable of handling different types of I/O signals such as uncontrollable signals, buffered signals and missing signals in the given protocols. Appendix A provides the details of the converter generation algorithm, with an example to illustrate its operation. The next chapter provides a detailed review of related work for SoC design and system-level verification of SoCs.

# Chapter 7
# Related Work and Outlook

The amount of functionality available on a single chip has grown according to Moore's law. At present, a single chip can contain the equivalent functionality of hundreds of processors. However, even though the available functionality has grown exponentially, the improvements in the methods to support the design of complex systems that can make use of such vast resources have considerably lagged behind. Improving levels of integration such as large scale integration (LSI), very large scale integration (VLSI) and ultra-large scale integration (ULSI) have allowed for the design of increasingly complex systems [BM06]. ULSI allows for the construction of SoCs where all components of a computer system are integrated onto a single chip. However, in order to support the construction of complex SoCs, there is a need to improve the techniques used for SoC design. This chapter provides a summary of various SoC design techniques that are relevant to the issues addressed in this monograph.

This chapter is organized as follows. Section 7.1 discusses the main approaches used in system-level validation of SoCs using formal techniques. This section discusses how correctness requirements of SoCs are specified (Sect. 7.2), how the interfaces of IPs are formalized (Sect. 7.3) and how the correctness of these models is analysed with respect to the requirements (Sect. 7.4). Section 7.5 presents the SoC design process. Finally, concluding remarks are presented in Sect. 7.6.

## 7.1 System-Level Verification

Hardware and software designers have used the notion of abstraction [Par77] to tackle the increasing complexity of systems. System-on-a-chips (SoCs) are ideal examples that illustrate successful design reuse. Here, a range of pre-designed intellectual property blocks, or IPs are appropriately integrated to achieve the functionality of an overall system in a single chip. While the SoC design process [KB02] is quite well established, major challenges still remain to be tackled, in the system-level verification phase. This phase is especially crucial as SoCs are

R. Sinha et al., *Correct-by-Construction Approaches for SoC Design*,
DOI 10.1007/978-1-4614-7864-5_7, 

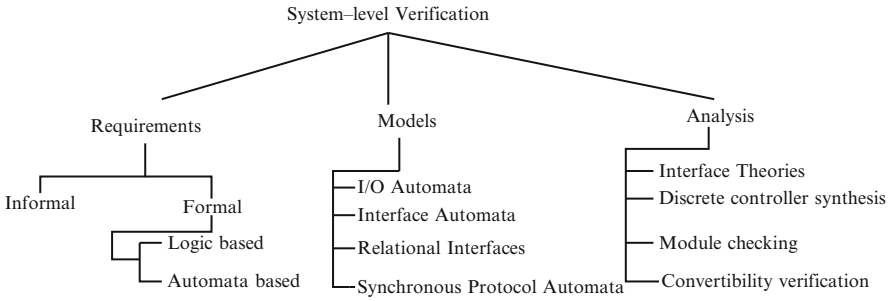



**Fig. 7.1** A classification of related literature

penetrating market segments that are inherently safety critical [CJ09]. With current verification practices, this step is consuming a considerable proportion of the overall effort of the design cycle (any where between 50% and 8%). This is one of the motivations for this monograph.

The work of this monograph, however, is closely related to many research efforts that have either inspired the proposed approach or are orthogonal but closely related to it. Figure 7.1 summarizes all these topics that are related to the work presented in this monograph. For SoC verification at the system-level, we need some correctness requirements, models of the protocols of the individual IPs and their compositions and finally some algorithms to check of the specified requirements are satisfied at the system level. Hence, we classified related research along these three dimensions, namely *requirements, models and algorithms for analysis*.

## 7.2 Requirements

We design systems to meet requirements. Requirements originate from the various stakeholders involved in a design like users, parts manufacturers, designers, sales and management staff, etc. A requirement captures *what the system must do* (or must not do). Requirements may capture the functional and non-functional aspects of a design, and may also include operational, technical and organizational constraints on the design cycle [vL01]. The field of requirements engineering involves three tasks: *discovering*, *documenting*, and *managing* requirements [SS97]. Requirements discovery happens during an early *elicitation* phase, resulting in the forming of a System requirements specification (SRS). However, requirements can change and arrive even very late in the design cycle. Requirements may be documented using natural language or using sophisticated documenting languages provided by management systems such as IBM DOORS [Wie99]. A requirements management system stores each requirement, manages its relationships with other requirements and traces its satisfaction in the design or its components.

A very important task in documenting requirements is to ensure that a documented requirement captures all aspects of the intended requirement without generalising or constraining it. This is where formal temporal logics, such as CTL, can be extremely useful. However, it is a fallacy to assume that every requirement can be captured formally. In fact, most requirements are stored informally in natural language form and their satisfaction in a design is checked using manual methods (such as checklists), even for safety-critical systems like avionics [LVLG90]. While capturing every requirement formally is impossible, human barriers to the use of formal methods [Par10], such as lack of expertise, mean that formal methods are not used for even those requirements that can be formally captured. In Chap. 4, we propose SoC boilerplates to reduce the impact of such human barriers. SoC boilerplates allow designers to write properties using constrained natural language, and these properties are converted to CTL automatically.

Formal requirements themselves may be captured using temporal logic, or using finite-state automata [PdAHSV02]. Temporal logic requirements describe the desired sequences of states (state labels) in a system. Automata-based requirements are typically variants of Büchi automata [SGGR13], and describe the desired sequences of input and output events leading to *marked* or *final* states. A Büchi automaton-based requirement is satisfied if a final state is visited infinitely often by a system. Both temporal logic and automata can therefore capture orthogonal requirements, based on states or signals sequencing, respectively. However, CTL-based solutions such as SoC boilerplates presented in this book can capture restricted signal sequencing requirements in temporal logic. Similarly, some temporal logics, such as LTL, can be reduced to Büchi automata [SB00], allowing us to capture LTL requirements using either temporal logic or automata.

## 7.3 Models and Compositions

In addition to correctness requirements, system-level verification requires models of protocols of IPs and their compositions. We address these concerns in the following.

### *7.3.1 Interface Modelling*

We argued in this monograph that we need to augment existing SoC design flow with mathematical descriptions of the interfaces of IPs (Chap. 1). This proposal is actually motivated by many years of related research, which proposed different types of mathematical models to formalize these interfaces. A recent paper by Li et al. [LXB+11] makes a poignant statement about the need for such formalization. Here, the authors examined the current practice in the design of device drivers. Like the SoC domain, English is the main vehicle for describing the hardware-software interfaces used. This paper states that: “English does not have formal semantics;

therefore these specifications commonly contain ambiguities and inconsistencies.... We have formalized four types of HW/SW interfaces, such as Ethernet controller devices/drivers and USB devices/drivers. We detected fifteen issues in the English specifications." All these drivers were already exhaustively tested before this study and none of the reported inconsistencies were detected earlier. This example, thus, clearly motivates the need for formal modelling techniques. In the following, we present related work on the formalization of interface descriptions using mathematical models.

Abstract interfaces of Parnas [Par77] proposed the separation of the interface aspect of a component from its behaviour to facilitate composition. This pioneering idea is a precursor to many significant ideas for component-based development of systems. Formalization of such notions started with Lynch and Tutle's I/O automata [LT87] for interface modelling of a component with respect to an *uncontrollable environment*. In an I/O automata, in every state of the model all input actions are enabled. De Alfaro and Henzinger [dAH01a] classify this as a *pessimistic* strategy. They argue that software components are often designed in conjunction with their environment. Hence the system designer may make certain optimistic assumptions regarding the environment. This *optimistic view* enables more succinct models of components using the concept of *interface automata* [dAH01a].

While the above finite state models have been developed for generic applications, many FSM variants have been developed for modelling the on-chip communication protocols of SoCs. D'silva et al. [DRS04a, DRS05a] proposed the concept of *synchronous protocol automata* (SPA) to formalize on-chip communication protocols. SPA is an automata with an explicit clock event that triggers every transition of the system. SPA is capable of modelling the control part, the bounds on data exchanges and the clocks of the different components. Synchronous Statechart [MR01] style hierarchy and concurrently is also supported to facilitate very compact and readable descriptions of SoC communication protocols. Some recent work [Fuj11, SYY+10] also use finite state machines (very similar to SPA but without any explicit clocks) for modelling of on-chip communication protocols. They demonstrate that such formal modeling not only facilitates automated analysis for correctness but also assists in post-silicon debug.

In a different domain, Li et al. [LXB+11] worked on the formalization of interface descriptions of device drivers. Such drivers involve the modelling of the hardware and the associated software. Authors illustrate the need for modelling of the stack for such systems. Hence, they use Buchi automata (automata that accept infinite input strings) for hardware modelling while they propose the use of labelled pushdown systems for software modelling. The proposed approach provides a formal framework for the modelling of generic components that involve both hardware and software. They also suggest a number of strategies such as the use of non-determinism and abstraction to deal with state-space explosion issues, similar to those developed by us for SoCs in Chap. 4.

### 7.3.2 Composition

Once a set of interface models are available using some mathematical framework, it is essential to define suitable compositions of such models so as to define the interaction between these components. Broadly, such interaction could use either asynchronous [Hoa85, Mil89] or synchronous composition [BG92, BCE+03] depending on the problem domain.

Asynchronous compositions are well known in many mathematical models such as Kahn process networks (KPNs) [Kah74], which use point-to-point, unbounded FIFO channels for communication between data-flow actors composed in a network. Due to these channels, KPNs use non-blocking writes and blocking reads. KPNs have been shown to be deterministic due to a restriction that once a process tests an empty channel, it can't context switch to test another non-empty channel to continue execution. Process algebras such as CCS [Mil89] and CSP [Hoa85] also use point-to-point channels for communication among threads (also known as processes). However, they perform a rendezvous over the channel (i.e., a blocking read and a blocking write). Unlike KPN, where processes operate over data-streams, process algebras capture event-triggered nature of processes and point-to-point synchronization using rendezvous using one-place channels. Asynchronous composition of processes using rendezvous leads to *interleaving* of actions from the participating processes. Hence, such asynchronous composition is non-deterministic.

Synchronous compositions are used in a family of languages for embedded systems called synchronous languages [BCE+03]. Here, a system is modelled using a set of concurrent threads such that every thread has a shared notion of time relative to the *ticking* of a global clock. The main basis for design of systems using the synchronous approach is known as the *synchrony hypothesis*, which states that the system operates infinitely fast relative to its environment. Such an abstraction helps in eliminating the inherent non-determinism of asynchronous compositions that happen due to interleaving of actions from the participating threads. However, such determinism comes at a price. Synchronous programs may exhibit instantaneous feedback cycles, known as non-causal programs [BCE+03]. Hence, compilers must be designed to detect and subsequently reject such programs.

In the context of SoCs and interfaces, earlier papers used asynchronous compositions [PdAHSV02] while recent work based on SPA [DRS04a, DRS05a] and synchronous Kripke structures (SKS) [SRBS08, SRBS09, SRSB12] use synchronous compositions. Synchronous composition is natural in the SoC setting since IPs progress relative to specific clocks. However, the entire system may support multiple clocks and as long as the system has a fastest clock from which other clocks can be derived. Hence, the well known idea of *oversampling* [Hal94] may be used to convert such a multiclock system into a fully synchronous system. Our recent work [SRSB12] is based on this idea from synchronous languages to formalize component composition in SoCs.

Orthogonal to SoCs, more generic compositions that are primarily asynchronous are developed in [LXB+11] to define the composition of hardware and software components, which are used in device drivers. Such compositions are both more complex to analyse and are inherently non-deterministic.

## 7.4 Analysis

In the hardware design community, there have been many attempts for automated matching of incompatible protocols [Bor88, COB95, NG95, IDOJ96, BMM98], the starting point being the pioneering approach for *automatic transducer synthesis* [Bor88]. Here, timing diagrams of the two custom hardware were used and the approach generated the logic specification of the required *glue logic* automatically. This approach primarily worked on bridging control signal mismatches and data-width mismatches were not considered. This limitation was overcome by Narayan and Gajski [NG95]. This work checked for a *duality condition* to create an appropriate glue logic. The data width mismatches were also bridged by latching data values within the local memory of the interface protocol and supplying the combined data values appropriately. These early approaches, pre-dated the complexity of SoCs and lacked several features that are needed to tackle such complexity. They also relied on informal specifications and ad-hoc analysis techniques to solve certain aspects of system-level verification. For example, key questions such as the following were never addressed:

1. What kinds of models of protocols are needed to support rigorous analysis?
2. What kinds of compositions are need to be supported?
3. Given formal models of protocols and their compositions, how to decide that the protocols are compatible?
4. When a set of protocols are incompatible, how to determine if a suitable glue logic exists such that these incompatibilities can be bridged using such glue logic?

We have discussed existing techniques that address the first two questions in Sect. 7.3. In the following, we will examine a set of techniques that address the last two questions, which are based on automated techniques for analysis and verification of on-chip communication protocols. Passerone et al. [PdAHSV02] formalized these two questions as *convertibility verification* and *converter synthesis* respectively. In the following, we will use this terminology.

All recent analysis techniques based on formal models construct some kind of product automata of the protocols and then prune infeasible paths from this product [KN96, PdAHSV02, ADS+09]. Infeasible paths are those that violate any compatibility requirement (such as a buffer overflow due to one of the protocol writing data into the bus that is more than its capacity, for example) or may violate a desired specification (which may require that a control signal is emitted only after another one has been received). These approaches, can thus be broadly classified as the problem of *discrete controller synthesis* [RW89, GMW12]. Here a plant is controlled suitably using an appropriate controller such that the composition of the plant and controller always meets the desired specification. The role of the controller is to prune infeasible paths that violate the desired specification. Hence, the convertibility verification problem may be viewed as the problem of controller synthesis, as formulated in [KNM97]. In this paper, the models of the protocols

and the specification are expressed as FSMs (called labelled transition systems or LTSs). LTSs are not suitable for capturing all aspects of on-chip bus protocols such as control-flow, data-flow and clocks.

Another class of techniques, based on assume guarantee reasoning [CGP00], known as interface theories [dAH01b], are also somewhat related to the system-level verification problems in SoCs. These theories rely on certain assumptions regarding the environment of components and check that the compositions guarantee certain properties. There has also been some work relating these to interface automata and convertibility verification in [PdAHSV02].

Most formal techniques for convertibility verification rely on the composed protocols and a set of correctness criteria. The correctness criteria may be expressed in the form of an automata or using some temporal logic. We classify formal analysis techniques for convertibility verification as follows:

- Game theoretic [PdAHSV02, SGGR13]: The problem of convertibility verification is developed as two-payer game in [PdAHSV02]. Here the game is played between the protocols and their correctness requirements on one side (protocol compositions) and the adversary (the converter) on the other side. Converter synthesis amounts to the implementation of a winning strategy. In this formulation, the protocols are described using simple FSMs and the correctness specification is also a FSM. This approach only considers correctness of control-flow. A recent game-theoretic formulation [SGGR13] uses Buchi automata? for the specification of correctness criteria and uses labelled transition systems with bounded integer counters (similar to SKS used in this book) for protocol description. The convertibility verification problem is solved using a Buchi game. This formulation supports many nice features such as *incrementality*, which is not handled by earlier approaches. An incremental approach allows the addition of new components into an existing design by making *incremental* changes to this design rather than designing from scratch. This approach is quite generic and supports, control, data-width mismatches effectively while also dealing with *uncontrollable* events. Hence the verification used in this paper considers an open system rather than a closed system (see Chap. 3 for the distinction).
- Transaction relation and refinement [DRS05a, ADS+08, RGSG09, ADS+09, SYY+10]: These approaches are based on checking of compatibility between protocols so that protocols can make progress collectively and deadlocks can be avoided. A transaction relation between the states of a pair of protocols is used to formalize the convertibility verification question. When a transaction relation does not exist, the two protocols are incompatible. Later approaches [ADS+08, ADS+09] based on this notion of compatibility can deal with control, data and clock mismatches effectively. A key feature of the above solutions is that correctness criteria is not explicitly specified but correctness is implicitly ensured through the compatibility checking process. A similar approach is also employed in [SYY+10] for compatibility checking of EFSMs. However, this approach considers additional correctness specification as a spec FSM. Roop

et al. [RGSG09] proposed a LTS-based representation of protocols and the desired specification. They formulated the convertibility verification question as a refinement relation between the desired specification and the protocol composition. This formulation allowed the converter to engage in the usual controllable actions of disabling and could also perform additional forcing actions [RSR01] to enforce the specification when needed.

- Model checking [SRB08b, CN09]: While considering protocol compatibility, we may take either the optimistic approach of interface automata [dAH01a] or the pessimistic approach of I/O automata [LT87]. If we consider that all IPs are already available before the convertibility verification process, then the system may be considered to be closed. In this case, the problem of convertibility verification may be addressed as a model checking [CGP00] question by using the optimistic view similar to [dAH01a]. Sinha et al. [SRB08b] formulate a model checking based solution by assuming that the protocols are specified as SKS and correctness requirements are specified in CTL.
- Module checking [SRBS08, SRBS09, SRSB12]: In general, a system may be designed incrementally [SGGR13]. In this case, some IPs may be available and a converter may be generated by considering that the system is partially-closed i.e., closed with respect to the set of available IPs but open to new inputs from future IPs and other environmental components. Such an approach is in between the pessimistic approach [LT87] (where the system is completely open) and optimistic approach [dAH01a] (where the system is closed). Module checking [KVW01] is needed to verify correctness of open systems. Hence, in [SRBS08, SRBS09, SRSB12] a module checking-based approach is developed for addressing the convertibility verification problem when the system is partially closed. These approaches can also deal with control, data-width and clock mismatches. A recent approach of [SRSB12] can also deal with the question of incremental converter synthesis.

### *7.4.1 Advanced Techniques*

In addition to the above techniques, there have been some proposals [CN09] to deal with additional *non-functional* constraints such as power-consumption during convertibility verification. This approach generalizes the approach of [SRBS09, ADS+09] to capture such non-functional aspects during converter synthesis. *Sub-system* reuse [LP13] is another related field of research which involves integrating the hardware as well as the software levels of subsystems (clusters of IP blocks) is studied. Finally, safety and associated certification is a major concern for SoCs used in the medical domain. In [PMN+09], a SoC design approach is presented for the design of mixed-criticality [BLS10] systems. Here, the system consists of a set of tasks of varying levels of criticality i.e., life critical, mission critical and non-critical. Life critical tasks must never fail as their failure may be life threatening. Mission critical tasks rarely fail and such failures, while not life threatening, may degrade

system performance and are not desirable. Finally, non-critical tasks are allowed to fail, as long as such failures can be isolated and do not propagate into other tasks. This paper formulates an approach for the design of such mixed-criticality systems by designing appropriate *wrappers* that perform the task of isolation of faults by monitoring resource usage and communication. This paper presents the case study of a pace maker example, to illustrate the proposed design approach.

## 7.5 The SoC Design Process

SoC design approaches can be broadly categorized as system-based, platform-based, or component-based design techniques.

### *7.5.1 System-Level Design*

System-level design is a top-down approach that involves automatically transforming high-level specifications to an implementation by automating most of the synthesis and validation stages of the design cycle. In this paradigm, the designer is required to completely specify the behaviour of the system using formal specifications. Following this, a series of transparent steps are used to transform the specifications into an implementation that can be synthesized [SJ04]. The transformation steps usually require minimal user guidance.

The motivation behind system-level design is the fact that for complex systems, gate-level or even component-level specifications are too low-level to be written easily by a designer. System-level design attempts to bridge the gap between the available technology and the SoC complexity that can be synthesized. Typically, only a fraction of the amount of computation possible on a single chip is utilized by SoC designs as design and validation techniques do not support fast and efficient design of complex systems [Hen03]. By allowing abstract system-level specifications, user effort is minimized while the design time is also reduced due to the automation of most of the design cycle.

Most synchronous languages like Esterel and Lustre can be translated directly to hardware and/or software, and hence provide a good framework for system-level design [Ber91]. Some approaches support the synthesis of an SoC from FSM-based high-level specifications [GWC+00]. Other approaches introduce the use of application specific instruction set processors (ASIPs) that can be extended by adding new instructions, inclusion/exclusion of pre-defined computation blocks and parametrization to customize them for the given application [Hen03]. Popular tools such as SystemC [GTC01], SystemVerilog [Ric03] and Esterel Studio [BBBD02] are built to allow system-level design and synthesis of SoCs.

Although system-based design is ideal from the perspective of a short design cycles, it is usually difficult to achieve because of the complex capabilities required

of the SoC design tool. Furthermore, due to minimal user input during the later stages of the design cycle, any errors at the specification stage are only found out during prototyping and are much more expensive to address at that stage.

### 7.5.2 Component-Based Design

In 20th century systems, computation was expensive while communication overheads were minimal. However, in complex SoCs, that can contain potentially contain a large number of cores on a single chip [MP03], computation expenses are minimal while the cost of communication between IPs has become the main focus [BM06]. Component-based design attempts to shift the focus of design tools from computation to communication aspects of SoCs.

Component-based design is a bottom-up approach which requires the user to select the IPs to be used in the design and specify their combined behaviour. The task of generating the interconnections between the IPs is then automated by the design tool [SVSL00]. This paradigm is also known as communication-based design [SSM+01] and is used especially for MPSoCs (multi-processor SoCs) [CLN+02].

Networks-on-a-chip (NoCs) are SoCs built using a component-based design paradigm where the interconnections between the IPs is viewed as a network [BDM02]. Although existing buses can be used for most designs, for multi-core designs the communication overheads over buses can be significant. Furthermore, as more IPs are connected to a bus, the overall power consumption of the system increases. Even for multi-master buses, arbitration becomes a major problem. At the same time point-to-point links between IPs are optimal but their number grows exponentially as more IPs are added. NoC treats the interconnection problem as a network routing problem [GG00]. A number of application-specific network generation techniques that optimize power consumption and other performance constraints have been proposed [BdM00, Ben06, RSG03].

### 7.5.3 Platform-Based Design

With increasing time-to-market pressures, it is difficult to build ad-hoc low volume systems by generating application-specific integrated circuits (ASICs) as in component-based design paradigm [KMN+00]. At the same time, system-level design requires complex capabilities from the design tools and is over dependent on the specification stage of the design process.

Platform-based design involves designing SoCs on a pre-selected architecture. The architecture, called the platform, is a library of components and a set of composition (communication) rules for IPs. It is a layer of abstraction that can hide lower levels of architectural details from the user [SVCBS04]. This abstraction helps speed up validation as it theoretically limits the size of the system (by

abstracting details of the architecture). If needed, the abstracted layers can be provided to the user in order to support bottom-up designs. At the same time, higher abstractions can be used to support top-down design approaches [CDBSVS02]. Platform-based design has found wide acceptance in industry and is being used to develop complex SoCs. For example, the TI OMAP platform is popularly used to build mobile phone chips [SVCBS04].

Platform-based design can help reduce the gap between increasing system complexities and design productivity [GWC+00]. A universal SoC platform, to which processors and IPs can be added, speeds up the development of SoCs considerably [GWC+00].

### *7.5.4 Design of Multi-clock SoCs*

Many SoCs contain IPs that execute on different clocks. The main integration challenge for such SoCs is the communication between IPs. A number of approaches have been proposed for the construction of multi-clock systems from such IPs.

The globally asynchronous, local synchronous (GALS) paradigm supported by synchronous languages such as Esterel is popularly used to model and synthesize multi-clock SoCs. Multi-clock Esterel [BS01] allows modelling multi-clock SoCs by allowing the partitioning of systems into decoupled modules each having its own clock. Suhaib [Suh07] presents a number of approaches to compose components that execute using different clocks. Synchronous protocol automata [DRS05a] allow for the modelling of clocks by representing them as inputs to each automaton that represents an IP. The relationship between clocks is represented as another automaton which applies the given clock ratios by emitting the clock signals to be consumed by other automata.

However, the above methods of representing clocks as inputs has a major disadvantage as it results in an exponential blow-up in state space when automata are composed [RSR07]. This is due to the fact that clocks are treated as inputs and with the addition of each new clock (or input), the number of input combinations doubles [RSR07]. MCcharts, presented in [RSR07], prevents this problem by coupling each input to a specific clock. An input can only be sampled when its corresponding clock ticks (has a rising edge). This ensures that as new inputs are added, the number of input combinations only grows linearly. Communication between IPs that are unsynchronized is modelled using *software rendezvous* which forces an IP to wait until the other communicating IP is ready.

GALDS, a GALS based design approach that generates a local clock generator and a global clock generator for a multi-clock SoC, is presented in [CZ05]. Problems such as clock skews and heat dissipation are taken into account and communication between IPs is facilitated via an asynchronous FIFO. The use of a digital clock generator reduces power consumption.

A few works investigate the problem of synchronizing communication between components with different clocks. The use of synchronizers, which are used to

ensure that data is sent only in pre-determined safe clock cycles, have been studied in [Sei80, Lef05]. Approaches like [CG03, Gre93], on the other hand, look at adjusting the delays between sending and receiving components to achieve synchronization. In the protocol-conversion setting, [MCS+06] presents a solution where control-flow and synchronization issues between multi-clock systems are bridged using an automatically constructed interface. A significant contribution of this work is the hardware synthesis of the interface that handles clock-mismatches comprehensively. It assumes that clocks of communicating components are *rational*, which means that all clocks are derived by dividing a common base clock. This is a fair assumption as clocks on a single chip are usually derivatives of a common clock.

## 7.6 Conclusions

The approach developed in this monograph is based on existing techniques that propose approaches for the formalization of on-chip communication protocols [Fuj11, DRS04b, DRS05a] and requirements for hardware-software interfaces [LXB+11]. The synchronous Kripke structure (SKS) model developed in Chap. 4 is most closely related in spirit to a set of techniques [DRS05a, ADS+09] developed at UNSW, Sydney. Both our approach and the UNSW-approach are inspired by the synchronous approach for embedded system design [BCE+03]. The synchronous approach is widely used in industry for the design of complex, safety critical systems and is supported by many industry strength tools such as SCADE [SCA]. We used this approach in the SoC domain for two reasons. First, the multiclock nature of SoCs is very natural to model using the synchronous approach and associated oversampling as illustrated in Chap. 5. Second, the synchronous approach has many benefits such as determinism and deadlock freedom, that will be very helpful in reducing the system-level verification effort of SoCs. Moreover, these features are essential for achieving correct-by-construction designs as expounded in this monograph and will be valuable for safety-critical SoCs.

Our SKS models enhance earlier models by correctly capturing control-flow, data-flow and clocks along with correctness specifications to facilitate model/module checking-based analysis. This is unlike earlier approaches such as SPA that require implicit notion of correctness. For capturing correctness requirements explicitly, techniques such as FSMs [Fuj11,SYY+10,PdAHSV02], Buchi automata [SGGR13], linear temporal logic [LXB+11] and branching time temporal logic [SRB08b, SRBS09, SRSB12] have been used. This book develops the concept of SoC boiler plates in Chap. 4 to further simplify the requirement analysis process for SoCs. These requirements are, unlike plain English, much more structured while being quite high-level for designers. Also, these requirements can be automatically translated to formal requirements, so that problems encountered in standard English-based requirements (see [LXB+11]) can be overcome.

The problem of convertibility verification is equivalent to module checking in the general setting. This is because, not all system components may be available at the

start of the design process and design may have to proceed in an incremental fashion. Hence, the complexity of convertibility verification, in the worst case, is equivalent to the complexity of module checking [KVW01]. See [KVW01] for a detailed comparison of the complexity issues for answering the module checking question for CTL, LTL, and CTL*. Kumar's group have also shown that the problem of discrete controller synthesis for CTL and CTL* is equivalent to the module checking problem [JK06]. However, when the system components are known in advance, the worst case complexity is that of model checking [CGP00].

This monograph tries to build on many years of research, primarily in academia, on the need for formalization of requirements and the on-chip protocols for SoCs. In describing a set of coherent methodology to aid the system-level validation process of SoCs, we believe that we have initiated a process to start dialogue between SoC designers and practitioners on one hand and academic researchers on the other. Hence, feedback on any aspect of this monograph on how to enhance the content and its applicability or otherwise is always welcome. The success of this manuscript lies in whether a dialog could be started between these two diverse worlds.

# Appendix A
# Converter Generation Algorithm

The algorithm to automatically generate converters in the proposed setting is based on module checking, and can address both control and data mismatches. When the system is completely closed (has no uncontrollable signals), the problem reduces to that of model checking. The converter generation algorithm can handle multiple protocols at once and allows explicit handling of uncontrollable signals.

To illustrate the converter generation algorithm, we reproduce Fig. 6.2 as Fig. A.1 which shows the SKS for the three on-chip protocols of a simple SoC. The three protocols, namely arbiter, consumer master, and memory slave, suffer from mismatches and their composition does not satisfy user-specified requirements (as discussed in Sect. 6.1). We show how our algorithm generates a converter to address mismatches and to ensure that the system (composed of the converter and the protocols) satisfies user-specified requirements.

## A.1 Initialization

The main inputs to the algorithm are:

- The parallel composition $P_1||P_2$ of the protocols $P_1$ and $P_2$. $P_1$ and $P_2$ can themselves be the parallel compositions of two or more protocols. In case protocols execute using different clocks, they must be *oversampled* before the composition can be computed (see Sect. 5.2 for details).
- A set of counters (one counter for each data communication medium as discussed in Sect. 6.2.1).
- A set $\Psi$ of CTL formulas to be satisfied over the SKS description of the protocols and the above data counters. In addition, it is required that an additional formula $\mathtt{AG}\,true$ is also included in $\Psi$. The need for this additional requirement is discussed later in Sect. A.6.
- A classification of I/O: As per Sect. 6.3.3, the user must classify each I/O signal of the protocols as uncontrollable (from environment or protocols), buffered

R. Sinha et al., *Correct-by-Construction Approaches for SoC Design*,
DOI 10.1007/978-1-4614-7864-5, © Springer Science+Business Media New York 2014

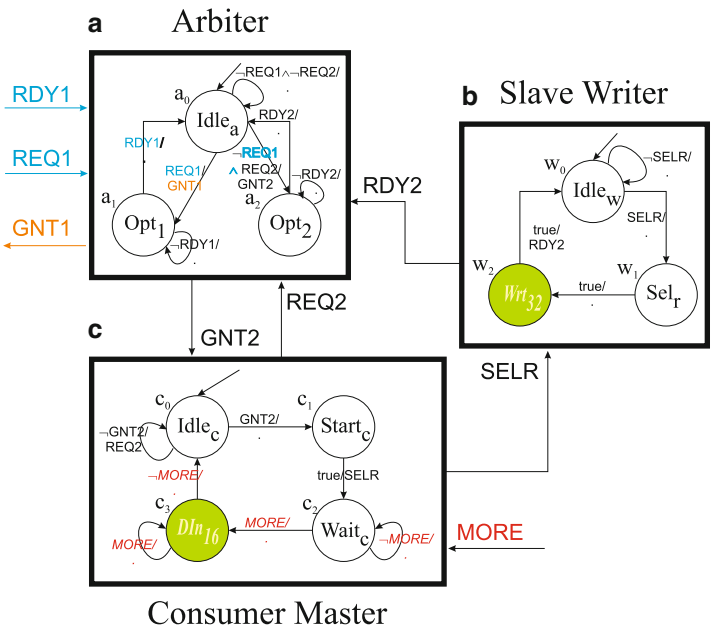



**Fig. A.1** Representation of the on-chip protocols of the IPs in Fig. A.1

(shared between protocols), and generated (by the converter). For example, the user identifies a subset $I_{\mathscr{C}}^{unc}$ of the input set $I_{1||2}$ of $P_1||P_2$ that are marked as uncontrollable environment signals (as discussed in Sect. 6.3.3).

### Illustration

For the SoC example presented in Fig. A.1, the following inputs are provided to the algorithm:

- The parallel composition $P_A||(P_C||P_W)$ of the arbiter, consumer master and slave writer protocols.
- A set of counters. The set contains the counter `cntr` that describes the correct data-communication behaviour between the master and slave over the SoC data-bus.
- The set of CTL formulas $\Psi$ contains the following formulas:
  - AGEF$DIn_{16}$: The consumer master can always eventually read data from the system data bus.
  - AGEF$DOut_{32}$: The memory writer protocol can always eventually write data.
  - AGEF$Opt_2$: From every reachable state in the system, there must be a path that reaches a state where the arbiter grants access to the consumer master.

  - $\texttt{AG}\neg Idle_c \Rightarrow Opt_2$: The master cannot be in an active state (any state other than its idle state), without the arbiter granting it bus access (by entering state $Opt_2$).
  - $\texttt{AG}(0 \leq \texttt{cntr} \leq 32)$: The data bus cannot contain more than 32 bits of data at any time.
- Signals `REQ1` and `RDY1` are marked as uncontrollable environment signals, `GNT1` is marked as an uncontrollable signal from the protocols, `RDY2`, `REQ2`, `SELR`, and `GNT2` are marked as buffered signals, while `MORE` is a generated signal.

## A.2 Data Structure and Initialization

As presented in Chap. 3, a tableau is implemented as an acyclic directed graph which contains nodes and links. It is defined as follows.

**Definition A.1 (Tableau).** Given $P_1||P_2$, and a set of CTL formulas $\Psi$, a tableau $Tab$ is a labelled acyclic graph $\langle N, n_0, L\rangle$ where:

- $N$ is a finite set of nodes of the tableau. Each node $n \in N$ has 4 attributes:
  - A state $\texttt{s} \in S_{1||2}$,
  - A set of formula `FS` where each formula $\varphi \in F$ is a sub formula of some formula in $\Psi$, and
  - A unique valuation `I` which assigns an integer value to each counter.
  - A set of buffered signals $\texttt{E} \subseteq I_{\mathscr{C}}^{buf}$.
- $n_0 \in N$ is the *root node* of the tableau which corresponds to the initial state $s_{0_{1||2}}$ of $P_1||P_2$, the counter valuation $\texttt{I}_0$ which assigns the value 0 to each counter in `I`, and the original set of formulas $\Psi$. The set of buffered signals `E` is empty for the root node.
- $L \subseteq N \times N$ is the set of all links (edges) of the tableau. A link $L(n, n')$ represents a directional edge from node $n$ to $n'$ in the tableau.

The proposed approach tries to generate a tableau (and eventually a converter) which allows only those paths in the protocols, such that for each path:

- All CTL properties are satisfied. Hence, the algorithm must monitor the CTL (sub) formulas that must be satisfied by each state along the path. For any node $n$ in the tableau, $n.\texttt{FS}$ is used to hold the formulas to be satisfied at state $n.s$.

- All data channels (tracked using counters) remain within the bounds specified by the user. The counter valuation $n.\mathtt{I}$ at each node $n$ in the tableau tracks the amount of data contained in each data channel. The tableau may not contain any node $n$ where a counter goes out of bounds.
- Any transitions triggered by uncontrollable signals are not disabled. Each state in the set of paths allowed by the tableau (converter) must be able to react to uncontrollable signals and take relevant transitions. This is handled in our algorithm by creating links that ensure uncontrollable signals are handled correctly. We describe this aspect later in Sect. A.3.
- Buffered signals are handled correctly, i.e., all buffered signals are emitted by the protocols before they can be read by the protocols as inputs. The set $n.\mathtt{E}$ contains the buffered inputs emitted by the protocols along the path from the initial state $n_0.s$ to state $n.s$. $\mathtt{E}$ therefore contains all buffered signals that can be emitted to enable transitions in the protocols. Details appear later in Sect. A.3.

**Initialization.** During initialization of the proposed algorithm, only the root node $n_0$ of the tableau is created.

For the protocols shown in Fig. A.1, the initial node $n_0$ contains the following:

- The state $n_0.\mathtt{s} = (a_0, c_0, w_0)$ (the initial state of the composition of the three protocols).
- The set of formulas

$$s_0.\mathtt{FS} = \{\mathtt{AGEF}\mathit{DIn}_{16}, \mathtt{AGEF}\mathit{DOut}_{32}, \mathtt{AGEF}\mathit{Opt}_2, \mathtt{AG}\neg\mathit{Idle}_c \Rightarrow \mathit{Opt}_2, \mathtt{AG}(0 \leq \mathtt{cntr} \leq 32)\}.$$

- The unique valuation $n_0.\mathtt{I} = \{0\}$ where the 0 represents the initial value of the counter `cntr` associated with the data-bus (the only data channel in the case study).
- The set of buffered signals $n_0.\mathtt{E} = \emptyset$ (initially the buffer is empty).

## A.3 Tableau Generation Algorithm

The algorithm to identify whether there exists a converter that can regulate given protocols to ensure satisfaction of desired properties is presented in Algorithm 1. The proposed algorithm takes as argument the following:

1. A state $s_e$ of the protocols $P_1||P_2$.
2. The set of counter valuations `I` to keep track of the status of each data communication channel.
3. The set of formulas `FS` (referred to as obligations) representing the desired properties (control and data constraints) to be satisfied at that state.
4. An event set `E` which contains buffered events that can be relayed to the protocols by a converter (to be generated). This is used for checking if a given input has been *buffered* by the converter to be relayed later. Buffering is needed to

**Algorithm 1** NODE isConv(s, I, FS, E, H)

```
1: curr = createNode(s, I, FS, E);
2: if anc ∈ H = curr then
3:    if FS contains AU or EU formulas then
4:       return FALSE_NODE
5:    else
6:       curr.type = LEAF_NODE, curr.link = anc
7:       return curr
8:    end if
9: end if
10: H_1 = H ∪ {curr};
11: if FS contains a formula F which is neither of type AX nor EX then
12:    FS_1 := FS − F, Node ret := FALSE_NODE
13:    if F = TRUE then
14:       ret := isConv(s, I, FS_1, E, H_1)
15:    else if F = p (p ∈ AP or p is a counter-constraint) then
16:       if p is satisfied in s, I then
17:          ret := isConv(s, I, FS_1, E, H_1)
18:       end if
19:    else if F = ¬p (p ∈ AP or p is a counter-constraint) then
20:       if p is not satisfied in s,I then
21:          ret := isConv(s, I, FS_1, E, H_1)
22:       end if
23:    else if F = φ ∧ ψ then
24:       ret := isConv(s, I, FS_1 ∪ {φ, ψ}, E, H_1)
25:    else if F = φ ∨ ψ then
26:       ret := isConv(s, I, FS_1 ∪ {φ}, E, H_1)
27:       if ret = FALSE_NODE then
28:          ret := isConv(s, I, FS_1 ∪ {ψ}, E, H_1)
29:       end if
30:    else if F = AGφ then
31:       ret := isConv(s, I, FS_1 ∪ {φ ∧ AXAGφ}, E, H_1)
32:    else if F = EGφ then
33:       ret := isConv(s, I, FS_1 ∪ {φ ∧ EXEGφ}, E, H_1)
34:    else if F = A(φ U ψ) then
35:       ret := isConv(s, I, FS_1∪{ψ ∨ (φ∧AXA(φ U ψ))}, E, H_1)
36:    else if F = E(φ U ψ) then
37:       ret := isConv(s, I, FS_1∪{ψ ∨ (φ∧EXE(φ U ψ))}, E, H_1)
38:    end if
39:    if ret != FALSE_NODE then
40:       curr.addChild(ret), ret := curr
41:    end if
42:    return ret
43: end if
44: curr.type := X_NODE
45: FS_AX = {φ | AXφ ∈ FS},FS_EX = {φ | EXφ ∈ FS}
46: for each conforming subset Succ of the successor set of s do
47:    for Each possible distribution of FS_EX (stored in FS_EX' for each s') do
48:       for each state s′ in Succ do
49:          E′ = E - buffered i/p required + internal o/p emitted. (in transition from s to s')
50:          I' = adjust counters (I,s') (add to I the weights of data labels at s' for each channel)
51:          if (N:=isConv(s', I', FS_AX ∪ FS_EX', E', H_1)) ≠ FALSE_NODE then
```

```
52:            curr.addEdge(o1, o2/i1, i2); curr.addChild(N);
53:          else
54:            Remove all children of curr and select another distribution of FS_EX for Succ
55:          end if
56:        end for
57:        return curr
58:      end for
59:  end for
60:  return FALSE_NODE
```

desynchronize the generation of an event by one protocol and its consumption by another protocol.

5. A history set H which keeps track of visited state formula pairs to ensure termination of the algorithm (see termination conditions, described later in Sect. A.4).

The first four arguments represent a node in the tableau while the fifth argument (H) contains the set of all nodes that have been visited along the path from the root node to the current node. The function returns a NODE which is the root of a tableau witnessing whether or not the behavior starting from state s can be regulated/converted to ensure the satisfaction of formulas in FS. The state s is said to be convertible if and only if the return of the function isConv is a non false node.

### *A.3.1 Description of the Tableau Generation Algorithm*

Algorithm 1 operates as follows. It first creates a node from the arguments passed to it and then checks if a similar node has already been created (lines 1–9). If this node is already present, no further processing is required. The termination conditions corresponding to this part of the algorithm are presented later in Sect. A.6.

At line 10, a new node is created using s and FS information and the history set is updated to H_1 by inserting this new node. Then, the algorithm removes on formula F from FS and processes it further (line 12). All AX or EX type formulas are not processed further until line 44 (because they correspond to *next*-state commitments).

Each F is processed as follows. If F is the propositional constant *true* then converter existence is checked against s with respect to FS_1 as any state can satisfy *true* (lines 13, 14). Similarly, if F is a proposition $p$ (or a negated proposition $\neg p$), s must be labelled by $p$ (not labelled by $p$); otherwise the algorithm returns a false-node. If F is a counter constraint $p$ (or a negated counter constraint $\neg p$), then the set of counter valuations I must satisfy $p$ (not satisfy $p$) (lines 15–22). If F is a conjunction of formulas, s must satisfy all remaining formulas (in FS_1) and all the conjuncts of F (lines 23, 24). A false-node is returned if this call fails. For disjunctive formulas, it is checked if s satisfies remaining formulas (in FS_1) and one of the disjuncts (lines 25–29). A false-node is returned if all the above calls fail.

If F is AG$\varphi$, the obligations on s is updated to include $\varphi \wedge$ AXAG$\varphi$, denoting $\varphi$ must be satisfied at s and all its (converter-enabled) successors must satisfy AG$\varphi$ (lines 30, 31). Similarly, if F is EG$\varphi$, the obligation on s is updated to include $\varphi \wedge$ EXEG$\varphi$, denoting $\varphi$ must be satisfied at s and at least on of its permitted successors must satisfy EG$\varphi$ (lines 32, 33). Similarly, if F is equal to A($\varphi$ U $\psi$), the obligation on s is updated to include $\psi \vee (\varphi \wedge$ AXA($\varphi$ U $\psi$) (lines 34, 35). Formulas of type E($\varphi$ U $\psi$) are handled in a similar manner (lines 36, 37).

The control reaches line 39 when one of the above checks has been carried out or F is equal to *false*. Here, the algorithm checks if the return value of the recursive call to isConv returned a false node. If this is the case, it returns a false-node because the success of the node curr depended on the result of the recursive call (which returned failure). On the other hand, if the recursive call returned a non-false node, it adds the returned node as a child of curr and return curr (lines 39–43).

The control reaches line 44 only when none of the other rules are applicable, i.e., FS only contains formulas of the form AX or EX, which capture the *next state* obligations of s. The node is labelled as an X_NODE.

**Illustration of Tableau Unrolling**

Before presenting how X_NODE are processed, we present how lines 1–43 of Algorithm 1 operate. Figure A.2 shows how Algorithm 1 operates on the initial node $n_0$ which corresponds to the initial state $(a_0, c_0, w_0)$ of the protocol composition, the initial counter status $\{0\}$ which shows that the data-bus (associated with the lone counter cntr) is initially empty, the CTL formula set $\{\texttt{AGEF}DIn_{16}, \texttt{AGEF}Opt_2, \texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ to be satisfied (we use only three formulas to simplify the tableau illustration), and empty control buffer and history sets. The counter status, control buffer set and history set do not change in the illustration, so we omit these details. Each row in Fig. A.2 illustrates the application of some rule in Algorithm 1 (as per the Description column in Fig. A.2). We also underline the changes in the formula set FS made by the algorithm along each step. At the end of the illustration in Fig. A.2, $n_0$ contains only AX and EX formulas, and can therefore be classified as a X_NODE (line 44).

### A.3.1.1 Handling Next-State Commitments

When Algorithm 1 reaches line 44, the current node contains only *next-state* AX and EX formulas. The algorithm operates as follows from this point. Firstly, two sets of next state commitments are computed. The set FS_AX aggregates all the formulas that must be satisfied in all destinations of (converter-permitted) transitions from s. Similarly, the set FS_EX aggregates all formulas that must be satisfied by at least one permitted successor of s (line 45).

| FS | Description |
|---|---|
| $\{\texttt{AGEF}DIn_{16},\texttt{AGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Initialnodeofthetableau |
| $\{\texttt{EF}DIn_{16}\wedge\texttt{AXAGEF}DIn_{16},\texttt{AGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving AG (lines 30–31, Alg. 1) |
| $\{\texttt{EF}DIn_{16}\wedge\texttt{AXAGEF}DIn_{16},\texttt{AGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving $\wedge$ (lines 23–24, Alg. 1) |
| $\{DIn_{16}\vee(true \wedge \texttt{EXEF}DIn_{16}),\texttt{AXAGEF}DIn_{16},\texttt{AGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving EF (lines3 6–37, Alg. 1) |
| $\{DIn_{16},\texttt{AXAGEF}DIn_{16},\texttt{AGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving $\vee$ (lines 25–26 (case1), Alg. 1) |
| $\{true\wedge\texttt{EF}DIn_{16},\texttt{AXAGEF}DIn_{16},\texttt{AGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving$\vee$(lines 27–29 (case2), Alg. 1) (case 1 returns false because $(a0,c0,w0)\neg\models DIn16$ |
| $\{true\texttt{EXEF}DIn_{16},\texttt{AXAGEF}DIn_{16},\texttt{AGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving $\wedge$ (lines 23–24, Alg. 1) |
| $\{\texttt{EXEF}DIn16,\texttt{AXAGEF}DIn16,\texttt{AGEF}Opt2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving *true* (lines 13–14, Alg. 1) |
| $\{\texttt{EXEF}DIn_{16},\texttt{AXAGEF}DIn_{16},\texttt{EF}Opt_2\wedge\texttt{AXAGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving AG (lines 30–31, Alg. 1) |
| $\{\texttt{EXEF}DIn_{16},\texttt{AXAGEF}DIn_{16},\texttt{EF}Opt_2,\texttt{AXAGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving $\wedge$ (lines 23–24, Alg. 1) |
| $\{\texttt{EXEF}DIn_{16},\texttt{AXAGEF}DIn_{16},Opt_2\vee(true\wedge\texttt{EXEF}Opt_2),\texttt{AXAGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving EF (lines 36–37, Alg. 1) |
| $\{\texttt{EXEF}DIn_{16},\texttt{AXAGEF}DIn_{16},Opt_2,\texttt{AXAGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving $\vee$ (lines 25–26 (case1), Alg. 1) |
| $\{\texttt{EXEF}DIn_{16},\texttt{AXAGEF}DIn_{16},true\wedge\texttt{EXEF}Opt_2,\texttt{AXAGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving $\vee$ (lines 27–29 (case2), Alg. 1) (case 1 returns false because $(a0,c0,w0)\neg\models Opt2$ |
| $\{\texttt{EXEF}DIn_{16},\texttt{AXAGEF}DIn_{16},true,\texttt{EXEF}Opt_2,\texttt{AXAGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving $\wedge$ (lines 23–24, Alg. 1) |
| $\{\texttt{EXEF}DIn_{16},\texttt{AXAGEF}DIn_{16},\texttt{EXEF}Opt_2,\texttt{AXAGEF}Opt_2,\texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving *true* (lines 13–14, Alg. 1) |
| $\{\texttt{EXEF}DIn_{16},\texttt{AXAGEF}DIn_{16},\texttt{EXEF}Opt_2,\texttt{AXAGEF}Opt_2,(0 \leq \texttt{cntr} \leq 32)\wedge\texttt{AXAG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving AG (lines 30–31, Alg. 1) |
| $\{\texttt{EXEF}DIn_{16},\texttt{AXAGEF}DIn_{16},\textbf{EX}\texttt{EF}Opt_2,\texttt{AXAGEF}Opt_2,(0 \leq \texttt{cntr} \leq 32),\texttt{AXAG}(0 \leq \texttt{cntr} \leq 32)\}$ | Resolving $\wedge$ (lines 23–24, Alg. 1) |
| $\{\texttt{EXEF}DIn_{16},\texttt{AXAGEF}DIn_{16},\texttt{EXEF}Opt_2,\texttt{AXAGEF}Opt_2,\texttt{AXAG}(0 \leq \texttt{cntr} \leq 3\ 2)\}$ | Resolving counter-constraint (lines 15–14, Alg. 1) $((a0,c0,w0) \models (0 \leq \texttt{cntr} \leq 32)$ |

**Fig. A.2** Resolution of different types of formula by Algorithm 1 for the initial node $n_0 = ((a_0, c_0, w_0), \{0\}, \{\texttt{AGEF}DIn_{16}, \texttt{AGEF}Opt_2, \texttt{AG}(0 \leq \texttt{cntr} \leq 32)\}, \emptyset, \emptyset)$ of the tableau for the case study shown in Fig. A.1

Next, in line 46, a *conforming* subset of the set of successors of `s` is created. A conforming subset is computed in the following manner:

1. For each $b_{unc} \in B(I^{unc}_{\mathscr{C}})$ over the set of uncontrollable inputs such that `s` has an outgoing transition w.r.t. $b_{unc}$, the set of successor states $Succ_{b_{unc}}$ of `s` is computed as follows

$$Succ_{b_{unc}}(s) = \{s' : (s \xrightarrow{b/o} s') \wedge (b = b_{unc} \wedge b_{rem})\}$$

where

$$\text{If } b_{rem} = i_1 \wedge i_2 \wedge \ldots \wedge i_n, \text{ then, } (i_j \in I^{buf}_{\mathscr{C}}) \Rightarrow (i_j \in \texttt{E})$$

$Succ_{b_{unc}}(s)$ is a set of outgoing transitions of $s$. For each transition $s \xrightarrow{b/o} s'$ in $Succ_{b_{unc}}(s)$, we divide the input expression $b$ into a conjunction of two sub-expressions $b_{unc}$ and $b_{rem}$. While $b_{unc}$ is the uncontrollable part of the input expression, while $b_{rem}$ is the controllable expression over buffered and generated signals (see Table 6.1 for signal classifications). $Succ_{b_{unc}}(s)$ contains a transition $s \xrightarrow{b_{unc} \wedge b_{rem}/o} s'$ *iff* all buffered signals needed for the transition are present in `E`. All remaining signals in $b_{rem}$ can be generated by the converter.

Without the loss of generality, we assume above that input expression $b$, and its sub-expressions $b_{unc}$ and $b_{rem}$ as conjunctions over input signals. In other words, we represent them as *complete monomials* [MR01], which allows us to collect the uncontrollable signals that must be present for $b$ (or $b_{unc}$) to evaluate to true in a set.

2. A conforming subset *Succ* is a set of successors of $s_e$ that contains exactly one element of each successor set corresponding to each complete monomial over uncontrollable signals. Since there are $2^m$ subsets corresponding to the complete monomials over uncontrollable inputs ($m = |I^{unc}_{\mathscr{C}}|$)), the size of *Succ* cannot exceed $2^m$.

Once a conforming subset *Succ* is selected, the future obligations of `s` are distributed amongst its elements. All `AX` commitments contained in `FS_AX` are to be checked against each element of *Succ*. The formulas in `FS_EX` are distributed amongst the various elements of *Succ*, such that at least one state in *Succ* is required to satisfy each formula in `FS_EX` (lines 47–55).

It recursively calls `isConv` for each state `s'` in *Succ* to check if it satisfies all commitments (all `AX` commitments and some `EX` commitments) passed to it and adjust the buffer signal set `E'` for each recursive call. If any element of *Succ* returns a failure, it moves to select a different distribution of the formulas in `FS_EX` for the elements of *Succ* (line 47). If a distribution that allows the satisfaction of all future commitments is found, the current node is returned (signifying success). On the other hand, if no distribution of the formulas in `FS_EX` returns success, another conforming subset is chosen (line 46). If no conforming subset that satisfies the future commitments of `s` under any possible distribution can be found, the procedure returns failure (line 59).

**Illustration of Handling Next State Commitments**

For the initial node $n_0$ of the tableau for the protocols shown in Fig. A.1, the conforming subset *Succ* is computed as follows. The only uncontrollable environment signal state $(a_0, c_0, w_0)$ can read (has outgoing transitions using) is REQ1. Therefore, we create a set of enabled transitions when REQ1 is available, and another set of enabled transitions when REQ1 is absent because REQ1 and ¬REQ1 are the only two complete monomials over uncontrollable signals in this case.

When REQ1 is present, state $(a_0, c_0, w_0)$ has outgoing transitions to states $(a_1, c_0, w_0)$, $(a_1, c_1, w_0)$, $(a_1, c_1, w_1)$, and $(a_1, c_0, w_1)$ (see Fig. A.3). However, transitions to the last three states require at least one of the buffered inputs SELR and/or GNT2. Since the initial node $n_0$ of the tableau has no buffered signals in its control buffer set ($n_0.\mathtt{E} = \emptyset$), these three transitions cannot fire. Hence, when REQ1 is present, the only enabled successor is state $(a_1, c_0, w_0)$ reached via the transition $(a_0, c_0, w_0) \xrightarrow{\mathtt{REQ1} \wedge \neg \mathtt{SELR} \wedge \neg \mathtt{GNT2} / \{\mathtt{GNT1}, \mathtt{REQ2}\}} (a_1, c_0, w_0)$. In other words, $Succ_{\mathtt{REQ1}} = \{(a_1, c_0, w_0)\}$. Similarly, when REQ1 is absent, given that no buffered signals are available, the only reachable successor is state $(a_0, c_0, w_0)$ via the transition $(a_0, c_0, w_0) \xrightarrow{\neg \mathtt{REQ1} \wedge \neg \mathtt{REQ2} \wedge \neg \mathtt{SELR} \wedge \neg \mathtt{GNT2} / \{\mathtt{REQ2}\}} (a_1, c_0, w_0)$. In this case, $Succ_{\neg \mathtt{REQ1}} = \{(a_0, c_0, w_0)\}$. Only one conforming subset $Succ = \{(a_1, c_0, w_0), (a_0, c_0, w_0)\}$ is possible in this case. Conforming subsets are computed in Algorithm 1 on line 46.

The next step is to create new nodes in the tableau for the states in the conforming subset (line 47). In our case, we will create two nodes $n_1$ and $n_2$, corresponding to states $(a_1, c_0, w_0)$ and $(a_0, c_0, w_0)$ in the conforming subset, as shown in Fig. A.5a, b. We then distribute the AX and EX formulas of the parent node $n_0$ to nodes $n_1$ and $n_2$. From Fig. A.2, we can see that node $n_0$ contains AXAGEF$DIn_{16}$, AXAGEF$Opt_2$, and AXAG($0 \leq$ cntr $\leq 32$) AX-type formulas, and EXEF$DIn_{16}$ and EXEF$Opt_2$ EX-type formulas. Since there are two EX-type formulas, there are four possible distributions for distributing AX and EX formulas to the two nodes, as shown in Fig. A.4.

Algorithm 1 will create nodes $n_1$ and $n_2$ in pairs (based on every distribution) according to line 47. We adjust control signal buffer set if the transition from $n_0.s$ to $n_1.s$ (or $n_2.s$) results in the protocols emitting any buffered signals (line 49). The transitions from state $(a_0, c_0, w_0)$ to states $(a_1, c_0, w_0)$ and $(a_0, c_0, w_0)$ both result in the protocol emitting the buffered signal REQ2. Hence, for both $n_1$ and $n_2$, $\mathtt{E}' = \{\mathtt{REQ2}\}$ (as shown in Fig. A.5b). The algorithm also adjusts the counter values if the transition from or the resulting states performing some data operations. In our case, since entering state $(a_1, c_0, w_0)$ or $(a_0, c_0, w_0)$ does not result in a data operation, the counter is kept at 0.

Nodes $n_1$ and $n_2$ are then further processed by Algorithm 1 using tableau unrolling until each of them become an X_NODE. Then, the process of creating new nodes to handle the next state commitments of each of these nodes is repeated.

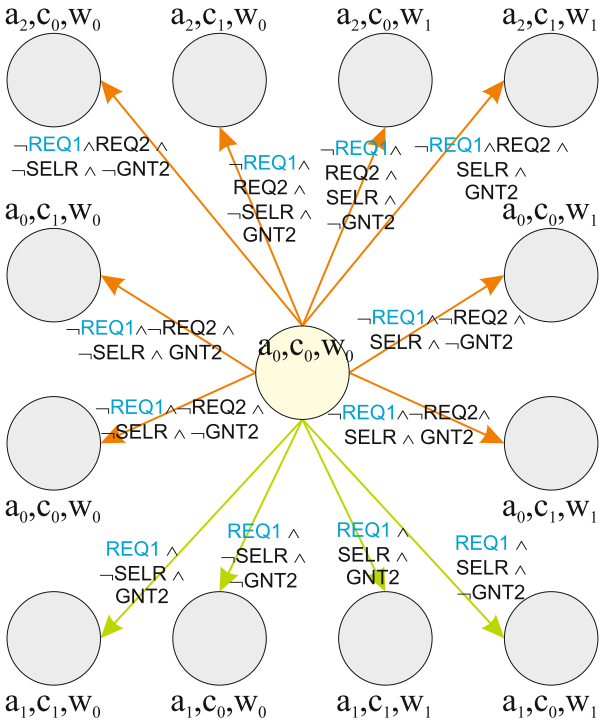



**Fig. A.3** Transitions of the initial state $(a_0, c_0, w_0)$ of $P_A||(P_C||P_W)$ (transitions contain only the input conditions)

| No. | Node $n_1$ ($n_1.s = (a_1,c_0,w_0)$) | Node $n_2$ ($n_2.s = (a_1,c_0,w_0)$) |
|---|---|---|
| 1 | AXAGEF$DIn_{16}$, AXAGEF$Opt_2$, AXAG($0 \leq$ cntr $\leq 32$), EXEF$Opt_2$, EXEF$DIn_{16}$ | AXAGEF$DIn_{16}$, AXAGEF$Opt_2$, AXAG($0 \leq$ cntr $\leq 32$) |
| 2 | AXAGEF$DIn_{16}$, AXAGEF$Opt_2$, AXAG($0 \leq$ cntr $\leq 32$), EXEF$Opt_2$ | AXAGEF$DIn_{16}$, AXAGEF$Opt_2$, AXAG($0 \leq$ cntr $\leq 32$), EXEF$DIn_{16}$ |
| 3 | AXAGEF$DIn_{16}$, AXAGEF$Opt_2$, AXAG($0 \leq$ cntr $\leq 32$), EXEF$DIn_{16}$ | AXAGEF$DIn_{16}$, AXAGEF$Opt_2$, AXAG($0 \leq$ cntr $\leq 32$), EXEF$Opt_2$ |
| 4 | AXAGEF$DIn_{16}$, AXAGEF$Opt_2$, AXAG($0 \leq$ cntr $\leq 32$) | AXAGEF$DIn_{16}$, AXAGEF$Opt_2$, AXAG($0 \leq$ cntr $\leq 32$), EXEF$Opt_2$, EXEF$DIn_{16}$ |

**Fig. A.4** Distribution of EX and AX formulas over nodes $n_1$ and $n_0$

## A.4 Termination

Observe that, the recursive process *isConv* may not terminate as formulas are expanded at lines 34, 36, 38 and 40. To ensure termination, it relies on fixed point characterization of CTL (lines 1–9). The finitization for recursive formulas is

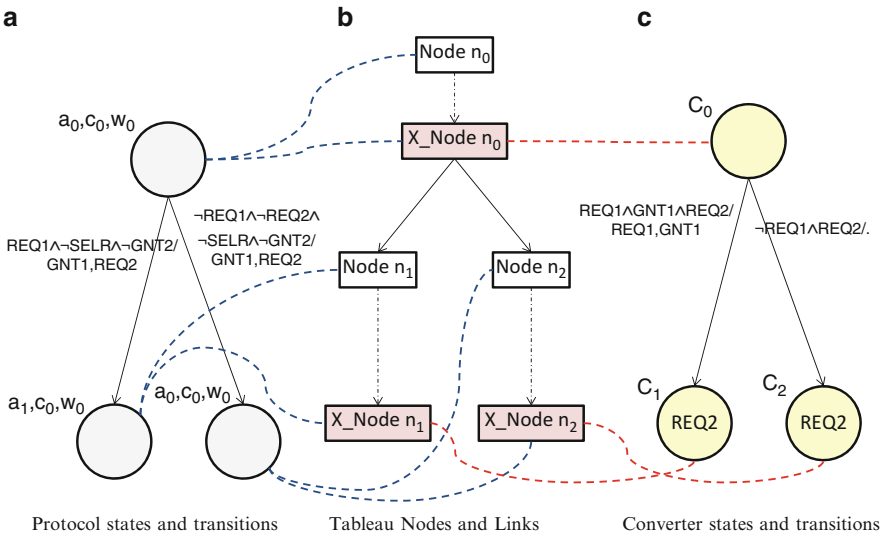



**Fig. A.5** Illustration of protocol, tableau and converter. (**a**) Protocol states and transitions. (**b**) Tableau nodes and links. (**c**) Converter states and transitions

identical as the finitization for CTL discussed in Chap. 3, Sect. 3.2.1.5. Specifically, semantics of $\mathtt{AG}\varphi$ is the greatest fixed point of $Z =_\nu \varphi \wedge \mathtt{AX}Z$ as AG properties are infinite path properties. As a result, proof of whether a state satisfies an AG property can depend on itself (similar for EG formulas). On the other hand, the semantics of $\mathtt{A}(\varphi\ \mathtt{U}\ \psi)$ is the least fixed point of $Z =_\mu \psi \vee (\varphi \wedge \mathtt{AX}Z)$ as $\mathtt{A}(\varphi\ \mathtt{U}\ \psi)$ properties are finite path properties where $\psi$ must be satisfied after finite number of steps (similar for EU formulas). From the above observations, if a state `s` with a formula set `FS` is revisited in the recursive call (line 5), it is checked whether there exists any least fixed point obligation. If this is the case, this recursive path is not a witness to the satisfaction of the obligations in `FS` by `s` and false node is returned; otherwise the current node is returned (lines 6–11).

Another important aspect affecting the termination of the algorithm is the presence of counter valuations. Even if the counters can potentially take infinite number of valuations (which will also lead to non-termination of the recursive procedure), the valuations of the counters that are of interest are always partitioned finitely. For example for data-width requirements of the form $\mathtt{AG}(0 \leq \mathtt{counter} \leq j)$, any `counter` valuations less than 0 or greater than $j$ results in violation of the property when `isConv` terminates and returns false-node. As such, there are exactly $j+3$ partitions of `counter` valuations; $j+1$ partitions each recording desired valuations between 0 and $j$ and the rest recording undesirable valuations.

For example, the tableau containing the initial node $n_0$ and its children nodes $n_1$ and $n_2$ shown in Fig. A.5 will grow only to a finite size. All $\mathtt{AG}, \mathtt{EG}, \mathtt{E}(\ \mathtt{U}\ ), \mathtt{A}(\ \mathtt{U}\ )$ properties are finitized so that nodes are not duplicated. Similarly, the value of the

counter `cntr` contained in the set `I` does not grow beyond the range $[0,32]$ (as constrained by the data requirement $0 \leq$ `cntr`$32$), resulting in a finite number of nodes in the tableau.

## A.5 Converter Extraction

If a successful tableau is constructed, the converter is extracted by traversing the nodes of the tableau as shown in Algorithm 2. Firstly, a map `MAP` containing nodes of the given tableau as keys and mapping each such key to a state of the converter is created. This map is initially empty. It then passes the tableau's root node to the recursive procedure Algorithm 3 which processes a given node as follows:

**Algorithm 2** `STATE` extractConverter(Tableau t)

1: Create new map `MAP`
2: initialState = extractState(t.rootnode);
3: **return** initialState

**Algorithm 3** `NODE` extract(`NODE`)

1: **if** `NODE` is present in `MAP` **then**
2: **return** map.get(`NODE`)
3: **else if** `NODE` is of type `INTERNAL_NODE` **then**
4: `MAP`.put(`NODE`,`extract`(`NODE`.child))
5: **return** `MAP`.get(`NODE`)
6: **else if** `NODE` is of type `LEAF_NODE` **then**
7: `MAP`.put(`NODE`,`extract`(`NODE`.link))
8: **return** `MAP`.get(`NODE`)
9: **else if** `NODE` is of type `X_NODE` **then**
10: create new state of the converter `C`
11: `MAP`.put(`NODE`,`C`)
12: **for** each linked `NODE`' of `NODE` **do**
13: State `C`' = `extract`(`NODE`')
14: add transition $\texttt{C},\texttt{NODE.E} \xrightarrow{b_{unc} \wedge b_{emit} \wedge b_{buf} / O_{emit} \cup O_{unc} \cup O_{buf} \cup O_{gen}} \texttt{C}',\texttt{NODE}'.\texttt{E}'$ given $\texttt{NODE.S} \xrightarrow{b'/o'} \texttt{NODE}'.\texttt{S}$ such that

- $b_{unc} = i_{u1} \wedge \ldots \wedge i_{um}$ is the uncontrollable sub-formula of $b'$.
- $b_{emit} = o_{e1} \wedge \ldots \wedge o_{es}$ where each $o_{e1} \ldots o_{es}$ are the uncontrollable outputs in $o'$
- $b_{buf} = i_{b1} \wedge \ldots \wedge i_{bn}$ and $i_{b1}, \ldots, i_{bn}$ are buffered signals in $o'$..
- $O_{unc} = \{i_{u1}, \ldots, i_{um}\}$ is the set of uncontrollable signals to the protocols (as $b_{unc}$).
- $O_{emit} = \{o_{e1} \ldots o_{es}\}$ is the subset of uncontrollable outputs to the environment $o'$ (read as $b_{emit}$).
- $O_{buf} = \{i_{b1}, \ldots, i_{bt}\}$ is the set of buffered signals emitted by the converter.
  Also, $\{i_{b1}, \ldots, i_{bt}\} \subseteq \texttt{NODE.E}$ and $\texttt{NODE}'.\texttt{E}' = (\texttt{NODE.E} \backslash O_{buf}) \cup \{i_{b1}, \ldots, i_{bn}\}$ where $i_{b1}, \ldots, i_{bn}$ are the bufferable signals read from the protocol (as $b_{buf}$).
- $O_{gen}$ is the subset of generated signals such that the presence of signals in $O_{emit} \cup O_{emit} \cup O_{buf} \cup O_{gen}$ result in the satisfaction of $b'$.

15: **end for**
16: **return** `MAP`.get(`NODE`)
17: **end if**

1. If NODE is already present in MAP as a key, return the state of the converter associated with it (lines 1, 2).
2. If NODE is an internal node (which only has one child), the state of the converter corresponding to this node is the state of the converter extracted with respect to its child (lines 3–5). Internal nodes are formed as intermediate nodes leading to either a X_NODE or a LEAF_NODE. Figure A.5 shows how node $n_0$, all intermediate nodes (formed by unrolling the tableau), and the eventual X_NODE are all related to the converter state $C_0$.
3. If NODE is of type LEAF_NODE, the state of the converter corresponding to the node is the same as the state of the converter corresponding to its linked ancestor (lines 6–8). Leaf nodes are finitized nodes that simply link back to another existing node in the tableau.
4. If NODE is of type X_NODE, the algorithm creates a new state C of the converter corresponding to the node (line 10). This association between NODE and C is saved in MAP (line 11). Then, for each child node NODE′, extract is called to create a state C′ in the converter (line 13). The state C contains a transition to C′ (line 4). The inputs and outputs of the converter transition depend on the inputs and outputs of the transition from the protocol state NODE.$s$ to NODE′.$s$. Essentially the converter transition reads as input all uncontrollable environment and protocol signals (as $b_{unc}$) and emits them immediately to the protocols (as $O_{emit}$) and the environment (as $O_{emit}$). All buffered inputs read from the protocols (as $b_{buf}$) are added to the converter control buffer E′. Similarly, any buffered signals generated by the converter (as $O_{buf}$) must be present in the control buffer E at C and are removed from the control buffer E′ at C′. Finally, the converter may generate any generated signals (as $O_{gen}$) which are required to fire the transition in the protocol.

   Figure A.5c shows the transition from converter state $C_0$ to $C_1$, which correspond to nodes $n_0$ and $n_1$ in the tableau shown in Fig. A.5b. The converter reads the uncontrollable signals REQ1 and GNT1 generated by the environment and protocols (transition from state $(a_0, c_0, w_0)$ to $(a_1, c_0, w_0)$ in Fig. A.5a). It also reads the buffered signal REQ2 generated by the protocol transition. This signal is buffered in the converter - shown as the label REQ2 of state $C_1$. As outputs, the converter emits the uncontrollable signals GNT1 and REQ1 to the environment and protocols. It generates neither a buffered signal nor a generated signal in this transition.

Figure A.5 shows the process of creating a tableau from states in the protocols, and the eventual generation of the converter. The full converter is shown in Fig. 6.8.

## A.6 The Reason for the Inclusion of AG*true* in $\Psi$

As noted in Sect. A.1, the formula AG*true* is included in the formula set to be satisfied by the converted system. We use this formula to ensure that the generated converter ensures that the converted system (lockstep composition of converter and protocols) has a *total* transition relation (every state has outgoing transitions).

When the initial (root) node of the tableau is created during initialization, the formula set `root.FS` contains at least AG*true*, which is eventually expanded into the next-state commitment AXAG*true* (line 31). Being an AX commitment, the commitment AG*true* is eventually passed to all children of the `root` node along with other AX and EX formulas obtained from other formulas in $\Psi$ (lines 44–58). The recursive expansion of the AG formula ensures that every node in the tableau has at least one next-state commitment AXAG*true*. Consequently, for every node NODE in the tableau, `NODE.FS` is never empty.

If at a node NODE, there are no future commitments (AX or EX formulas) to be met, a state of the converter corresponding to NODE would enable all transitions in the state `NODE.s` (and all its successors and so on). However, this means that the converter may enable multiple (or no) transitions for each unique complete monomial over uncontrollable environment inputs. By ensuring that every node in the tableau has at least the commitment AXAG*true*, the algorithm terminates only by finitizing the tableau as discussed above in Sect. A.4. Under this restriction, the converter state obtained from each X_NODE has transitions that allow the satisfaction of the conversion refinement relation.

## A.7 Complexity

The complexity of the algorithm can be obtained from the number of recursive calls. It is of the order

$$O(|\mathtt{I}| \times 2^{|S|} \times 2^{|\Psi|} \times 2^{|\mathtt{E}|})$$

where $|\mathtt{I}|$ is the size of the counter set, $|S|$ is the size of the state space of the parallel composition of participating protocols, $|\Psi|$ is the size of the set of formulas to be satisfied by the initial state of the parallel composition, and $|\mathtt{E}|$ is the maximum size of the buffered signal set contained in the converter.

For a system with no data counters, $|\mathtt{I}| = 1$. For a system with a single counter, $|\mathtt{I}|$ is the number of all possible counter valuations (counter range). For systems with multiple-data counters, $|\mathtt{I}|$ is the product of all counter ranges.

The exponential factors appear due to the following reasons:

- The complexity is exponential in the size of the state-set of the parallel composition due to the requirement of selecting a conforming subset of the set of successors of a given state. As the conforming subset is a subset of the set of successors (which is finite and not more than $|S_{1||2}|$ in size), the total number of conforming subsets is equal to or less than $2^{|S_{1||2}|}$.
- The complexity is exponential in the size of the subformulas of the initial formula set $\Psi$ because of the presence of EX formulas, which makes the problem an application of module checking [KVW01]. Given $n$ EX-type formulas to be distributed among $m$ states of a conforming subset, there are $n^m$ distributions.

However, the process can be optimized to not repeat a combination that has failed previously. For example, in Fig. A.4, if distribution 2 failed due to the failure of state $(a_0, c_0, w_0)$ to satisfy its commitments $\texttt{AXAGEF}DIn_{16}$, $\texttt{AXAGEF}Opt_2$, $\texttt{AXAG}(0 \leq \texttt{cntr} \leq 32)$, $\texttt{EXEF}DIn_{16}$, distribution 4, which includes the same commitments for the state will not be repeated.

Under this condition, each recursive call corresponding to a current distribution either rules out a given distribution over a given element, or results in failure. Given $n$ EX formulas and $m$ elements in the conforming subset, here are $2^n \times (m-1)$ such calls in the worst case (the factor $m-1$ , where $m \leq S_{1||2}$ can be ignored due to the presence of the higher order expression $2^{|S|}$ as shown previously).

- The complexity is exponential in the size of the number of signals that can be buffered because each node in the algorithm maintains a buffer of signals currently stored. The termination part of the algorithm (lines 5–11) depends on folding back a node to another node with the same buffer (as well as state and CTL formulas). For $m$ signals that can be buffered, a node may correspond to one of the $2^m$ subsets of the set of bufferable signals.

## A.8 Soundness and Completeness

The following theorem proves that the approach comprehensively handles the questions of converter correctness and existence. It can also be proved that the given algorithm always terminates even in the presence of bounded counters and recursive CTL formulas.

**Theorem A.1 (Sound and Complete).** *Given the parallel composition* $P_1||P_2 = \langle AP_{1||2}, S_{1||2}, s_{0_{1||2}}, I_{1||2}, O_{1||2}, R_{1||2}, L_{1||2}, clk\rangle$ *of two deterministic* SKS $P_1$ *and* $P_2$*, an initial set of counter valuations* I*, an empty set* E *of buffered signals, a set* FS *of CTL formulas, and the identification of all I/O signals as bufferable, non-bufferable or uncontrollable, a converter that can control the parallel composition to satisfy all properties in* FS *exists* iff *the call* isConv($s_{0_{1||2}}$, I, FS, $\emptyset$) *does not return* FALSE_NODE.

*Proof.* The above theorem holds for a fixed buffer size (given by the user) for each data medium shared between participating protocols.

The proof of this theorem follows from the soundness and correctness of the local module checking theorem. In addition to these results, the algorithm adds the following features:

- Data counters: It can be seen that the resulting converter always ensures that all data-constraints are satisfied. A node in a successful tableau exists its corresponding state (under the converter's control) satisfies all formulas assigned to it. As all data constraints are always checked on a node (line 18, 22) before returning success, it can be seen that the converter cannot contain any nodes

where data counters are violated. Also, as data counters must have a finite range, the termination of the algorithm is also guaranteed.

- `Conversion refinement relation`: When a node contains only next-state commitments, instead of checking every subset of its successors, the algorithm checks only those subsets that are conforming (see definition). This ensures that the converter does not attempt to enable transitions in a state in the composition such that the conversion refinement relation is violated.

It can be seen that the algorithm comprehensively checks all possible conforming subsets of a state. Hence, it is not possible to have a conforming subset of a given state that can satisfy given commitments but cannot be found by the algorithm (completeness). □

## References


[ARM] ARM, AMBA Specification (Rev 3.0) (2007), http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html

[ARM99] ARM, AMBA Specification (Rev 2.0) (1999), http://www.arm.com/products/solutions/AMBA_Spec.html

[AZ10] P. Ateshian, D. Zulaica, *ARM Synthesizable Design with Actel FPGAs: With Mixed-Signal SoC Applications (set 3)*, 1st edn. (McGraw-Hill Inc., New York, 2010)

[ADS+08] K. Avnit, V. D'Silva, A. Sowmya, S. Ramesh, S. Parameswaran, A formal approach to the protocol converter problem. *Design, Automation and Test in Europe, 2008. Date '08*, pp. 294–299, Mar 2008

[ADS+09] K. Avnit, V. D'Silva, A. Sowmya, S. Ramesh, S. Parameswaran, Provably correct on-chip communication: a formal approach to automatic protocol converter synthesis. ACM Trans. Design Autom. Electr. Syst. **14**(2) (2009)

[BLS10] S. Baruah, H. Li, L. Stougie, Towards the design of certifiable mixed-criticality systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010* (IEEE, Stockholm, 2010), pp. 13–22

[BMM98] A. Basu, R.S. Mitra, P. Marwedel, Interface synthesis for embedded applications in a co-design environment. In *11th IEEE International Conference on VLSI Design*, pp. 85–90, 1998

[BRS07] S. Basu, P.S. Roop, R. Sinha, Local module checking for CTL specifications. Electron. Notes Theor. Comput. Sci. **176**(2), 125–141 (2007)

[Ben06] L. Benini, Application specific NoC design. In *DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe* (European Design and Automation Association, Belgium, 2006), pp. 491–495

[BdM00] L. Benini, G. de Micheli, System-level power optimization: techniques and tools. ACM Trans. Design Autom. Electr. Syst. **5**(2), 115–192 (2000)

[BDM02] L. Benini, G. De Micheli, Networks on chips: a new SoC paradigm. Computer **35**(1), 70–78 (2002)

[BCE+03] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, R. de Simone, The synchronous languages 12 years later. Proc. IEEE **91**(1), 64–83 (2003)

[Ber91] G. Berry, A hardware implementation of pure Esterel. In *International Workshop on Formal Methods in VLSI Design*, Jan 1991

[BG92] G. Berry, G. Gonthier, The ESTEREL synchronous programming language. Sci. Comput. Program. **19**, 87–152 (1992)

[BS01] G. Berry, E. Sentovich, Multiclock esterel. In *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (Springer, London, 2001), pp. 110–125

[BBBD02] G. Berry, L. Blanc, A. Bouali, J. Dormoy, Top-level validation of system-on-chip in Esterel studio. *7th IEEE International High-Level Design Validation and Test Workshop*, pp. 36–41, Oct 2002

[BM06] T. Bjerregaard, S. Mahadevan, A survey of research and practices of network-on-chip. ACM Comput. Surv. **38**(1), 1 (2006)

[Bor88] G. Borriello, A new interface specification methodology and its application to transducer synthesis. Ph.D. thesis, University of California, Berkeley, 1988

[JEK+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: $10^{20}$ states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science* (IEEE Computer Society Press, Washington, 1990), pp. 1–33

[CT92] J.C. Candy, G.C. Temes, Oversampling methods for a/d and d/a conversion. *Oversampling Delta-Sigma Data Converters* ( IEEE Press, New York, 1992), pp. 1–25

[CN09] J. Cao, A. Nymeyer, Formally synthesising a protocol converter: a case study. In *Proceedings of the 14th International Conference on Implementation and Application of Automata, CIAA '09* (Springer, Berlin, 2009), pp. 249–252

[CDBSVS02] L.P. Carloni, F. De Bernardinis, A.L. Sangiovanni-Vincentelli, M. Sgroi, The art and science of integrated systems design. In *Proceedings of the 28th European Solid-State Circuits Conference (ESSCIRC 2002)*, pp. 25–36, Sep 2002

[CLN+02] W.O. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya, L. Gauthier, M. Diaz-Nava, Multiprocessor SoC platforms: a component-based design approach. Design Test Comput. IEEE **19**(6), 52–63 (2002)

[CG03] A. Chakraborty, M.R. Greenstreet, Efficient self-timed interfaces for crossing clock domains. In *ASYNC '03: Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems* (IEEE Computer Society, Washington, 2003), p. 78

[CZ05] A. Chattopadhyay, Z. Zilic, GALDS: a complete framework for designing multiclock ASICs and SoCs. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **13**(6), 641–654 (2005)

[CJ09] Y.-Y. Chen, T.-Y. Juang, *Vulnerability Analysis and Risk Assessment for SoCs Used in Safety-Critical Embedded Systems* (InTech, 2009)

[COB95] P. Chou, R.B. Ortega, G. Borriello, Interface co-synthesis techniques for embedded systems. In *ICCAD*, pp. 280–287, 1995

[CGP00] E.M. Clarke, O. Grumberg, D. Peled, *Model Checking* (MIT Press, Massachusetts, 2000)

[CP03] J.-L. Colaço, M. Pouzet, Clocks as first class abstract types. In *EMSOFT*, ed. by R. Alur, I. Lee. Lecture Notes in Computer Science, vol 2855 (Springer, New York, 2003), pp. 134–155

[Cou96] P. Cousot, Abstract interpretation. ACM Comput. Surv. (CSUR) **28**(2), 324–328 (1996)

[CC10] P. Cousot, R. Cousot, A gentle introduction to formal verification of computer systems by abstract interpretation. In *Logics and Languages for Reliability and Security*, ed. by J. Esparza, B. Spanfelner, O. Grumberg. NATO Science for Peace and Security Series - D: Information and Communication Security, vol 25 (IOS Press, Amsterdam, 2010), pp. 1–29

[dAH01a] L. de Alfaro, T.A. Henzinger, Interface automata. In *ESEC / SIGSOFT FSE*, pp. 109–120, 2001

[dAH01b] L. de Alfaro, T.A. Henzinger, Interface theories for component-based design. In *EMSOFT*, ed. by T.A. Henzinger, C.M. Kirsch. Lecture Notes in Computer Science, vol 2211 (Springer, New York, 2001), pp. 148–165

[DRS04a] V. D'Silva, S. Ramesh, A. Sowmya, Bridge over troubled wrappers: automated interface synthesis. In *VLSI Design* (IEEE Computer Society, Washington, 2004), pp. 189–194

[DRS04b] V. D'Silva, S. Ramesh, A. Sowmya, Synchronous protocol automata: a framework for modelling and verification of SoC communication architectures. In *DATE* (IEEE Computer Society, Washington, 2004), pp. 390–395

[DRS05a] V. D'Silva, S. Ramesh, A. Sowmya, Synchronous protocol automata: a framework for modelling and verification of SoC communication architectures. IEE Proc. Comput. Digital Tech. **152**(1), 20–27 (2005)

[DKW08] V. D'Silva, D. Kroening, G. Weissenbacher, A survey of automated techniques for formal software verification. IEEE Trans. CAD Integr. Circuits Syst. **27**(7), 1165–1178 (2008)

[Due04] C.A.M. Dueñas, Verification and test challenges in SoC designs. In *SBCCI '04: Proceedings of the 17th Symposium on Integrated Circuits and System Design* (ACM, New York, 2004), p. 9

[Fly97] D. Flynn, Amba: enabling reusable on-chip designs. Micro, IEEE **17**(4), 20–27 (1997)

[Fuj11] M. Fujita, Synthesizing, verifying, and debugging SoC with FSM-based specification of on-chip communication protocols. In *ATVA 2011*, ed. by T. Bultan, P.-A. Hsiung. Lecture Notes in Computer Science, vol 6996 (Springer, Berlin, 2011)

[GDWL92] D. Gajski, N. Dutt, A. Wu, S. Lin, *High Level Synthesis: Introduction to Chip and System Design* (Kluwer Academic Publishers, Boston, 1992)

[GWC+00] D.D. Gajski, A.C.-H. Wu, V. Chaiyakul, S. Mori, T. Nukiyama, P. Bricaud, Essential issues for IP reuse. In *ASP-DAC 2000: Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 37–42, 2000

[GLK+99] M. Genoe, C. Lennard, J. Kunkel, B. Bailey, G. de Jong, G. Martin, K. Hashmi, S. Ben-Chorin, A. Haverinnen, How standards will enable hardware/software codesign. In *7th International Workshop on Hardware/Software Codesign* (ACM Press, New York, 1999)

[GTC01] A. Ghosh, S. Tjiang, R. Chandra, System modeling with systemc. In *Proceedings of the 4th International Conference on ASIC*, pp. 18–20, 2001

[GMW12] C. Gierds, A.J. Mooij, K. Wolf, Reducing adapter synthesis to controller synthesis. IEEE Trans. Services Comput. **5**(1), 72–85 (2012)

[GL00] A. Goel, W.R. Lee, Formal verification of an IBM coreconnect processor local bus arbiter core. In *DAC '00: Proceedings of the 37th Conference on Design Automation* (ACM, New York, 2000), pp. 196–200

[Gre93] M.R. Greenstreet, STARI: a technique for high-bandwidth communication. Ph.D. thesis, Princeton University, 1993

[GG00] P. Guerrier, A. Greiner, A generic architecture for on-chip packet-switched interconnections. In *DATE '00: Proceedings of the Conference on Design, Automation and Test in Europe* (ACM, New York, 2000), pp. 250–256

[GZ97] R.K. Gupta, Y. Zorian, Introducing core-based system design. IEEE Design Test Comput. **14**(4) (1997)

[Gut99] R.J. Gutmann, Advanced silicon IC interconnect technology and design: present trends and rf wireless implications. IEEE Trans. Microwave Theor. Tech. **47**(6), 667–674 (1999)

[Hal94] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer International Series in Engineering and Computer Science (Kluwer, Princeton, 1994)

[HC02] A. Hall, R. Chapman, Correctness by construction: developing a commercial secure system. Software, IEEE **19**(1), 18–25 (2002)

[Hen03] J. Henkel, Closing the SoC design gap. Computer **36**(9), 119–121 (2003)

[Hoa85] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice Hall International, Saddle River, 1985)

[HD02] R. Hofmann, B. Drerup, Next generation coreconnect processor local bus architecture. In *15th Annual IEEE International ASIC/SOC Conference, 2002* (IEEE, Rochester, 2002), pp. 221–225

[HLP+10] K. Huang, J. Lu, J. Pang, Y. Zheng, H. Li, D. Tong, X. Cheng, FPGA prototyping of an AMBA-based windows-compatible SoC. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (ACM, New York, 2010), pp. 13–22

[JD11] E. Hull, K. Jackson, J. Dick, *Requirements Engineering* (Springer, New York, 2011)

[IDT04] IDT, Idt peripheral bus. *White Paper* (IDT, Santa Clara, 2004)

[IEC] IEC, Iec61508 - functional safety standard for electrical/electronics/programmable electronics systems, http://www.iec.ch/functionalsafety/

[IDOJ96] T.B. Ismail, J.M. Daveau, K. O'Brien, A.A. Jerraya, A system level communication approach for hardware/software systems. Microprocess. Microsyst. **20**(3), 149–157 (1996)

[JK06] S. Jiang, R. Kumar, Supervisory control of discrete event systems with CTL* temporal logic specifications. SIAM J. Control Optim. **44**(6), 2079–2103 (2006)

[CWCS11] F. Johnson, C. Chong-White, M. Millar, S. Shaw, Validating the realism and representativeness of SCATS in simulation. In *18th World Congress on ITS*, 2011

[Kah74] G. Kahn, The semantics of a simple language for parallel programming. *Information Processing* (Elsevier, North Holland, 1974)

[KB02] M. Keating, P. Bircaud, *Reuse Methodology Manual for System-On-A-Chip Designs* (Springer, New York, 2002)

[KG99] C. Kern, M.R. Greenstreet, Formal verification in hardware design: A survey. ACM Trans. Design Autom. Electron. Syst. **4**, 123–193 (1999)

[KMN+00] K. Keutzer, S. Malik, R. Newton, J. Rabaey, A. Sangiovanni-vincentelli, System-level design: orthogonalization of concerns and platform-based design. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **19**, 1523–1543 (2000)

[KN96] R. Kumar, S.S. Nelvagal, Convertibility verification using supervisory control techniques. In *IEEE International Symposium on Computer-Aided Control System Design* (IEEE Computer Society, Washington, 1996), pp. 32–37

[KNM97] R. Kumar, S. Nelvagal, S.I. Marcus, A discrete event systems approach for protocol conversion. Discrete Event Dyn. Syst. **7**(3), 295–315 (1997)

[KV96] O. Kupferman, M.Y. Vardi, Module checking [model checking of open systems]. In *Computer Aided Verification, 8th International Conference, CAV '96* (Springer, Berlin, 1996), pp. 75–86

[KV97] O. Kupferman, M.Y. Vardi, Module checking revisited. In *Computer-Aided Verification, 7th International Conference, CAV '97* (Springer, Berlin, 1997), pp. 36–47

[KVW01] O. Kupferman, M.Y. Vardi, P. Wolper, Module checking. Inform. Comput. **164**(2), 322–344 (2001)

[Lef05] J. Lefebvre, *Esterel v7 Reference Manual-Initial Standardization Proposal*, 2005

[LXB+11] J. Li, F. Xie, T. Ball, V. Levin, C. McGarvey, Formalizing hardware/software interface specifications. In *ASE*, ed. by P. Alexander, C.S. Pasareanu, J.G. Hosking (IEEE, New York, 2011), pp. 143–152

[LP13] M. Lindwer, M.R. Pedersen, High-performance imaging subsystems and their integration in mobile devices. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '13* (EDA Consortium, San Jose, 2013), pp. 170–170

[LVLG90] C.D. Locke, D.R. Vogel, L. Lucas, J.B. Goodenough, Generic avionics software. specification. Technical Report, DTIC Document, 1990

[LT87] N.A. Lynch, M.R. Tuttle, Hierarchical correctness proofs for distributed algorithms. In *PODC*, ed. by F.B. Schneider (ACM, New York, 1987), pp. 137–151

[MP03] P. Magarshack, P.G. Paulin, System-on-chip beyond the nanometer wall. *Proceedings of the Design Automation Conference*, pp. 419–424, June 2003

[MR01] F. Maraninchi, Y. Rémond, Argos: an automaton-based synchronous language. Comput. Lang. **27**(1–3), 61–92 (2001)

[MCS+06] J. Mekie, S. Chakraborty, D.K. Sharma, G. Venkataramani, P.S. Thiagarajan, Interface design for rationally clocked gals systems. In *12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'06)* (IEEE Computer Society, Washington, 2006), pp. 160–171

[Mil89] R. Milner, *Communication and Concurrency* (Prentice Hall International, Upper Saddle River, 1989)

[NG95] S. Narayan, D. Gajski, Interfacing incompatible protocols using interface process generation. In *32nd Design Automation Conference*, pp. 468–473, 1995

[OSB11] Z.J. Oster, G.R. Santhanam, S. Basu, Identifying optimal composite services by decomposing the service composition problem. In *ICWS* (IEEE Computer Society, Washington, 2011), pp. 267–274

[Par77] D.L. Parnas, Use of abstract interfaces in the development of software for embedded computing systems. Technical Report 8047 (Naval research lab, Washington, 1977)

[Par10] D.L. Parnas, Really rethinking 'formal methods'. Computer **43**(1), 28–34 (2010)

[PdAHSV02] R. Passerone, L. de Alfaro, T.A. Henzinger, A.L. Sangiovanni-Vincentelli, Convertibility verification and converter synthesis: two faces of the same coin. In *International Conference on Computer Aided Design ICCAD*, 2002

[PMN+09] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, L. Sha, Handling mixed-criticality in soc-based real-time embedded systems. In *Proceedings of the Seventh ACM International Conference on Embedded Software, EMSOFT '09* (ACM, New York, 2009), pp. 235–244

[Pet99] W. Peterson, Design philosophy of the wishbone soc architecture. *Silicore Corporation*, 1999

[RSR07] I. Radojevic, Z. Salcic, P.S. Roop, Mccharts and multiclock fsms for modelling large scale systems. In *Fifth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2007)* (IEEE Computer Society, Washington, 2007), pp. 3–12

[RSG03] V. Raghunathan, M.B. Srivastava, R.K. Gupta, A survey of techniques for energy efficient on-chip communication. In *DAC '03: Proceedings of the 40th Conference on Design Automation* (ACM, New York, 2003), pp. 900–905

[RW89] P.J.G. Ramadge, W.M. Wonham, The control of discrete event systems. Proc. IEEE **77**, 81–98 (1989)

[ITR96] Recommendation Z ITU-T, Z Recommendation. 120, *Message Sequence Chart (MSC)* (ITU-T, Geneva, 1996), p. 27

[Ric03] D.I. Rich, The evolution of system verilog. IEEE Design Test Comput. **20**(4), 82–84 (2003)

[RSR01] P.S. Roop, A. Sowmya, S. Ramesh, Forced simulation: a technique for automating component reuse in embedded systems. ACM Trans. Design Autom. Electr. Syst. **6**(4), 602–628 (2001)

[RGSG09] P.S. Roop, A. Girault, R. Sinha, G. Goessler, Specification enforcing refinement for convertibility verification. In *International Conference on Application of Concurrency to System Design, ACSD'09* (IEEE Computer Society, Augsburg, 2009), pp. 148–157

[RMK03] A. Roychoudhury, T. Mitra, S.R. Karri, Using formal techniques to debug the amba system-on-chip bus protocol. In *DATE '03: Proceedings of the Conference on Design, Automation and Test in Europe* (IEEE Computer Society, Washington, 2003), p. 10828

[oNSW] RTA Government of New South Wales, http://www.scats.com.au/. Accessed 11 Feb 2013

[SJ04] I. Sander, A. Jantsch, System modeling and transformational design refinement in forsyde [formal system design]. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **23**(1), 17–32 (2004)

[SV02] A. Sangiovanni-Vincentelli, Defining platform-based design. *EEDesign of EETimes*, 2002

[SVSL00] A.L. Sangiovanni-Vincentelli, M. Sgroi, L. Lavagno, Formal models for communication-based design. In *CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory* (Springer, London, 2000), pp. 29–47

[SVCBS04] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, M. Sgroi, Benefits and challenges for platform-based design. In *DAC '04: Proceedings of the 41st Annual Conference on Design Automation* (ACM, New York, 2004), pp. 409–414

[SR07] M. Satpathy, S. Ramesh, Test case generation from formal models through abstraction refinement and model checking. In *A-MOST* (ACM, New York, 2007), pp. 85–94

[SCC00] W. Savage, J. Chilton, R. Camposano, IP reuse in the system on a chip era. In *ISSS '00: Proceedings of the 13th International Symposium on System Synthesis* (IEEE Computer Society, Washington, 2000), pp. 2–7

[SCA] SCADE, Scade suite, http://www.esterel-technologies.com/products/scade-suite/. Accessed 30 Apr 2013

[Sei80] C.L. Seitz, System timing. *Introduction to VLSI Systems*, Chapter 7 (Addison-Wesley, Reading, 1980)

[SSM+01] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, A. Sangiovanni-Vencentelli, Addressing the system-on-a-chip interconnect woes through communication-based design. In *DAC '01: Proceedings of the 38th Conference on Design Automation* (ACM, New York, 2001), pp. 667–672

[SYY+10] C.-H. Shih, Y.-C. Yang, C.-C. Yen, J.-D. Huang, J.-Y. Jou, Fsm-based formal compliance verification of interface protocols. Inform. Sci. Eng. **26**, 1601–1617 (2010)

[SRB08a] R. Sinha, P.S. Roop, S. Basu, A model checking approach to protocol conversion. Electron. Notes Theor. Comput. Sci. **203**(4), 81–94 (2008)

[SRBS08] R. Sinha, P.S. Roop, S. Basu, Z. Salcic, A module checking based converter synthesis approach for SoCs. In *IEEE International Conference on VLSI Design* (IEEE Computer Society, Washington, 2008), pp. 492–501

[SRB08b] R. Sinha, P.S. Roop, S. Basu, Soc design approach using convertibility verification. EURASIP J. Embed. Syst. **2008** (2008)

[SRBS09] R. Sinha, P.S. Roop, S. Basu, Z. Salcic, Multi-clock SoC design using protocol conversion. In *Design and Test Europe (DATE)* (IEEE, New York, 2009), pp. 123–128

[SRSB12] R. Sinha, P.S. Roop, Z. Salcic, S. Basu, Correct-by-construction multi-component soc design. In *DATE*, ed. by W. Rosenstiel, L. Thiele (IEEE, New York, 2012), pp. 647–652

[SGGR13] R. Sinha, A. Girault, G. Goessler, P.S. Roop, Formal system-on-chip design using incremental converter synthesis. Technical Report TBA, INRIA, Rhone-Alps, 2013

[SB00] F. Somenzi, R. Bloem, Efficient büchi automata from ltl formulae. In *Computer Aided Verification* (Springer, New York, 2000), pp. 248–263

[SS97] I. Sommerville, P. Sawyer, *Requirements Engineering: A Good Practice Guide*, 1st edn. (Wiley, New York, 1997)

[Suh07] S.M. Suhaib, Formal methods for intellectual property composition across synchronization domains. Ph.D. thesis, Virginia Polytechnic Institute and State University, 2007

[vL01] A. van Lamsweerde, Goal-oriented requirements engineering: a guided tour. In *Proceedings of 5th IEEE International Symposium on Requirements Engineering, 2001*, pp. 249–262, 2001

[Web00] W.-D. Weber, Enabling reuse via an IP core-centric communications protocol: Open core protocoltm. *Proceedings of the IP*, pp. 20–22, 2000

[Wie99] K.E. Wiegers, Automating requirements management. Software Development **7**(7), 1–5 (1999)

[XZ03] N. Xu, Z. Zhou, Avalon bus and an example of SOPC system. Semicond. Technol. **28**(2), 17–20 (2003)

# Index



R. Sinha et al., *Correct-by-Construction Approaches for SoC Design*,
DOI 10.1007/978-1-4614-7864-5, © Springer Science+Business Media New York 2014