Chapter 3: Modeling for Verification

Sanjit A. Seshia, Natasha Sharygina, and Stavros Tripakis

System modeling is the initial, and often crucial, step in verification. The right choice of model and modeling language is important for both designers and users of verification tools. This chapter aims to provide a guide for system modeling in four stages. First, it provides an overview of the main issues one must consider in modeling systems for verification. These issues involve both the selection or design of a modeling language and the steps of model creation. Next, it introduces a simple modeling language, SML, for illustrating the issues involved in selecting or designing a modeling language. SML uses an abstract state machine formalism that captures key features of widely-used languages based on transition system representations. Our choice of introducing the simple modeling language is to simplify the connection between languages used by practitioners (such as Verilog, Simulink, or C) and various underlying formalisms (e.g., automata or Kripke structures) used in model checking. Third, the chapter demonstrates key steps in model creation using SML with illustrative examples. Finally, the presented modeling language SML is mapped to standard formalisms such as Kripke structures.

1 Introduction

A *model*, broadly speaking, can be thought of as the description of a system "on paper", or as a "virtual" system. This is in contrast to a "real" system, which can be thought of as a physical artifact: a car, a medical device, a Java program, or a stock market. Characterizing Java programs or stock markets as "physical" may seem

Sanjit A. Seshia
UC Berkeley, e-mail: sseshia@eecs.berkeley.edu

Natasha Sharygina
University of Lugano e-mail: natasha.sharygina@unisi.ch

Stavros Tripakis
UC Berkeley, e-mail: stavros@eecs.berkeley.edu

strange; however, philosophical considerations aside, most people would agree that these systems are concrete and real enough to affect our lives in a direct way. Models also affect our lives, but in a more indirect way, as we discuss below.

The type of model is defined by its purpose. Models of existing systems are often referred to simply as "models", whereas models of systems yet to be built may be termed "specifications" or "designs". Some models are written using informal notations that are open to interpretation, whereas others are written in languages with mathematical semantics and are called "formal models". Most importantly, models rarely capture an entire, complete system, since the sheer size and complexity of most systems make this an impossible task. As a result, models usually focus only on "relevant" parts of a system and/or only on specific aspects of a system. For example, a model of a system that involves both hardware (HW) and software (SW) may only focus on the SW part; or a model may only focus on the logical aspects of a communication system, e.g., the communication protocol, and ignore other aspects such as performance (e.g., throughput, latency) or energy consumption.

Models are essential for our lives. They likely always have been. Humans as well as many other organisms need to form in their brain internal representations of how the external world "works". These representations can be seen as models. Closer to the focus of this book, engineering and technology heavily rely on mathematical models. In fact, the tasks of designing and building a system are intimately linked with various modeling tasks. Specification models are used to communicate the goals and requirements of a system among different engineering teams. Detailed design models are built in order to estimate behavior of a system prior to its construction. This is essential in order to avoid the costs and dangers of building deficient systems. After a system is built, models are still essential in order to operate and maintain the system, calibrating and tuning it, monitoring abnormal behavior, and ultimately upgrading it.

The goal of this chapter is to illuminate the key issues in modeling systems for formal verification, in general, and model checking, in particular. Because there are so many different types of systems and application domains, with varying concerns, even within the field of formal verification there is a plethora of modeling languages, formalisms, and tools, as well as modeling techniques, uses, and methodologies. It is beyond the scope of this chapter to give a thorough account of these; such an account would probably require an entire book by itself. Rather, this chapter has three more modest aims. First, we seek to provide an overview of the main issues one must consider in modeling systems for verification. These issues involve both the selection or design of a modeling language and the steps of model creation. Second, we introduce a simple modeling language, SML, for illustrating the issues involved in selecting or designing a modeling language. SML uses an abstract state machine formalism that captures key features of widely-used languages based on transition system representations, and can serve to bind together the modeling languages introduced or used throughout the handbook. By introducing SML, we seek also to simplify the connection between real languages (say Verilog, Simulink, or C) used by practitioners and various underlying formalisms (i.e., automata or Kripke structures) used in model checking. Finally, the chapter demonstrates key steps in model creation using SML with illustrative examples drawn from three different domains: hardware, software, and cyber-physical systems.

There are at least two possible audiences for the material in this chapter. The first are *users* of verification tools who need to make decisions about the right modeling language and verification tool for their task, and how to best model the system so as to get useful results from the chosen tool. The second audience comprises *tool builders* who may want to choose the appropriate modeling constructs for their domain (e.g., a tool for synthetic biology) or for easing the verification of a particular class of problems (e.g., parameterized systems with large data structures). Researchers working in related areas, such as program synthesis, may also find the concepts discussed in this chapter useful for better understanding the characteristics of verification techniques that they build upon.

We begin this chapter, in Section 2, by discussing the main issues one must consider in modeling systems for verification. In Section 3, we present SML, a simple language that illustrates many of the aspects of formal modeling languages. Three illustrative examples of system modeling with SML are presented in Section 4. We relate SML to the well-known formalism of Kripke structures in Section 5, and conclude in Section 6.

2 Major Considerations in System Modeling

Models are built for different reasons. It is important to emphasize that models are primarily tools used to achieve a certain purpose. They are means to a goal, rather than the goal itself. As such, the notion of a model being "good" or "bad" has little meaning by itself. It is more appropriate to examine whether a model is good or not *with respect to a certain goal*. For example, a model may be good for estimating the throughput of a system but useless for checking whether the system has deadlocks, or vice versa.

While models are built with many different goals in mind, they generally support the system design process. Stakeholders in this process must choose the right modeling formalisms, languages, and tools, to achieve their respective goals. As in [17], we distinguish between a modeling *formalism* and a modeling *language*. Formalisms are mathematical objects consisting of an abstract syntax and a formal semantics. Languages are concrete implementations of formalisms. A language has a concrete syntax, may deviate from the formalism in the semantics that it implements, and may implement multiple semantics (e.g., changing the type of the numerical solver in a simulation tool may change the behavior of a model). Also, a language may implement more than one formalisms. Finally, a language usually comes together with a tool such as a compiler, simulator, or model-checker. As an example of the distinction between formalisms and languages, timed automata [4] is a formalism, whereas Uppaal timed automata [47] and Kronos timed automata [31] are languages.

In this section, we discuss some of the main factors one must consider while choosing a modeling formalism and the challenges in modeling. We also give a brief survey of some modeling languages used for model checking.

2.1 Selecting a Modeling Formalism and Language

Here are some of the main factors one typically considers when selecting a good modeling formalism and language, for formal verification in general and model checking in particular:

- Type of system;
- Type of properties;
- Relevant information about the environment;
- Level of abstraction;
- Clarity and modularity;
- Form of composition;
- Computational engines, and
- Practical ease of modeling and expressiveness.

We discuss each of these factors in more detail below.

2.1.1 Type of System

Different modeling formalisms have been developed based on the characteristic of the system being modeled. Some of the more common formalisms include:

- for *discrete(-time)* systems, formalisms such as finite state machines and push-down automata [38,44], extended state machines with discrete variables, hierarchical extensions such as Statecharts [35], as well as more declarative formalisms such as propositional temporal logics [52];
- for *continuous(-time)* systems, formalisms such as ordinary differential equations (ODEs) and differential algebraic equations (DAEs);
- for *concurrent processes*, formalisms such as communicating sequential processes (CSP) [36], a calculus of communicating systems (CCS) [53], the pi-calculus [54], Petri nets [56], marked graphs [26,43], etc.;
- for *compositional modeling*, formalisms such as process algebras [33,36,53], and Reactive Modules [6];
- for *dataflow* systems, formalisms such as Kahn process networks [42] and various subclasses such as synchronous dataflow (SDF) [48], Boolean dataflow (BDF) [21], scenario-aware dataflow (SADF) [59], etc.;
- *scenario-oriented* formalisms such as message sequence charts [40,41] and live sequence charts [28];

- for *timed* and *hybrid* systems, which combine discrete and continuous dynamics, formalisms such as timed and hybrid automata [3,4], or real-time temporal logics [2,5];
- *discrete-event formalisms* for timed systems, such as the denotational ones proposed in [7, 18, 50, 61], as well as operational ones inspired by discrete-event simulation and tools like Ptolemy [58,60];
- *probabilistic* variants of many of the above formalisms, such as Markov chains, Markov Decision Processes, stochastic timed and hybrid automata, etc., e.g., see [8,9,30,39,45];
- *game-* or *cost-theoretic* variants of some of the above formalisms, focusing on optimization and synthesis, instead of analysis, e.g., see [12,23,24].

In addition to the above formalisms which focus on somewhat specific classes of systems, the need for modeling *heterogeneity*, that is, capturing systems that combine semantically heterogeneous components, such as timed and untimed, discrete and continuous, etc., has resulted in heterogeneous modeling frameworks such as Focus [18], Ptolemy [32], or Metropolis [10, 29], and corresponding formalisms [13,18,60].

2.1.2 Type of Property

During verification, the system model is coupled with a specification of the property to be verified. Several classes of properties exist, each supported by corresponding specification languages. Examples of specification languages for reactive systems include computation tree logic, regular expressions, StateCharts diagrams, graphical interval logics, a modal mu-calculus and a linear-time temporal logic just to name a few (more details can be found in Chapt. 2 on Temporal Logic).

The choice of specification language and system model usually depends on the type of property one wishes to verify. For example, if one wishes to verify real-time properties of a system's execution over time, a real-time temporal logic might be the right choice of specification, and the system might be best represented as a timed automaton or timed CSP program. On the other hand, if for the same system one only wishes to verify Boolean properties such as absence of deadlock, then propositional temporal logic might suffice as the specification language.

It should be noted that the property specification language is not necessarily separate from the language used to model the system under verification. Often, the latter language provides mechanisms such as *assertions* or *monitors* that can be used to specify (usually safety) properties.

2.1.3 Modeling the Environment

One of the trickiest aspects of verification, which can both miss bugs and create spurious ones, is the task of modeling the environment of the system. The environment is usually much larger than the system, essentially incorporating everything other than the system under verification. On the other hand, it is also likely to be the part of the system that is least well-understood, since often times even a complete description of the environment is not available.

For example, in software model checking, one often needs to model the libraries that a piece of code uses, which forms its environment. If propositional temporal properties only involving the sequence of system calls is relevant, then environment models such as finite automata representing the language of system calls generated by a library component might be sufficient.

2.1.4 Level of Abstraction

The *level of abstraction*, i.e., the detail or faithfulness of a model is an essential consideration in modeling. A highly detailed model may be hard or impossible to build due to time and cost constraints. Even if it can be built, it may be too large or complex for it to be amenable to (manual or automated) analysis. *State explosion* is a well-known phenomenon that plagues many techniques such as model-checking. *Abstraction* methods are essential in building simpler and smaller models, by hiding unnecessary details. The difficulty is in understanding which details are truly irrelevant. Errors in this task often result in models that omit critical information. This may compromise the faithfulness of a model and ultimately render it useless.

As an example of successful use of abstraction, consider the process of modeling cache coherence protocols. Typically, one models the messages sent by the various processors as belonging to an abstract enumerated data type rather than represent the specific data formats actually used in the implementation of the protocol. Such abstraction is usually appropriate for the verification task, which depends only on the sequence and type of messages sent, rather than the specific bit-encoding of the message formats.

Verifiers that build models automatically from code usually rely heavily on automatic abstraction in the process. Selecting the right modeling formalism for such tools is usually critical to effective abstraction. Chapters 10 and 12 in this handbook provide more detail on techniques for automatic abstraction.

2.1.5 Clarity and Modularity

Models are often, although not always, designed to be viewed by humans. In such cases, the models must be clear and easy to understand. One way to ensure this is to use a modular approach in constructing the model. Typically a modeling language will include some notion of a *module* or *process*, and provide means to combine or compose such modules/processes into larger entities. Such notation usually also includes ways to hide internal details during the composition process, so as to retain only essential information in the interfaces of modules. In addition to making models easier to understand, modularity can some times also be exploited to make the verification task itself easier, e.g., by using compositional techniques (see Chapter

13 on compositional methods). A related point is the specific form of composition of modules, which we discuss next.

2.1.6 Form of Composition

Systems are rarely constructed monolithically. They are usually built by combining and modifying existing *components*. The form in which such components interact can determine the modeling formalism that is suitable for verification.

For example, consider a sequential circuit built up by connecting several modules, all of which share the same clock. Since all modules step on the same clock tick, a *synchronous* composition of these modules is a suitable choice, even when many of these modules are represented at a very coarse level of abstraction.

Similarly, consider modeling a distributed database that uses a protocol to ensure that replicated state is consistent. Different nodes connected through the Internet are unlikely to share a synchronized clock, and hence, for this problem, *asynchronous* composition is the appropriate form of composition.

For some systems, a hybrid of synchronous and asynchronous composition might be suitable; for example, processes might synchronize on certain input actions while stepping asynchronously otherwise. This is particularly the case in formalisms such as timed and hybrid automata, where processes synchronize in time (i.e., time elapses at the same rate for all processes) while their discrete actions may be asynchronous.

Notions such as synchronous/asynchronous composition are particularly relevant in formalisms with operational semantics, such as transition systems. In formalisms with denotational semantics, other forms of composition may be better suited. For instance, in Kahn Process Networks [42] processes are typically viewed as functions from streams to streams, composition is defined as functional composition, and fix-point theory is used to give semantics to feedback. In continuous-time formalisms processes may also be seen as functions manipulating continuous-time signals, and functional composition may be used here as well.

2.1.7 Computational Engines

A final consideration for modeling is the availability of suitable, scalable computational engines that power the verification tools for that class of models. For finite-state model checking, these include Binary Decision Diagrams (BDDs) [19] and Boolean satisfiability (SAT) solvers [51]. For model checking software and high-level models of hardware, satisfiability modulo theories (SMT) solvers [11] play a central role. (See also Chapters 4, 5 and 6 in this handbook for further details on BDDs, SAT, and SMT.)

Even within the realm of SAT and SMT solvers, modeling plays an important role in ensuring the scalability of verification. For example, hardware designs can be represented most naturally at the bit-vector level, where signals can take bit-vector values or be arranged into arrays (memories) of such bit-vector values. One strategy, that has proved highly-effective for control-dominated hardware designs, is to "bit-blast" the model to generate a SAT problem whose solution determines the result of verification. However, for many data-dependent properties, or for proving equivalence or refinement of systems, one might need a higher level of abstraction. It is here that techniques for automatically abstracting designs to a higher "term level" come in handy — functional blocks can be abstracted using uninterpreted or partially-interpreted functions, and data words can be represented as abstract terms without regard to their specific bit-encoding (see, e.g., [16,20]). Such representation then enables the use of SMT solvers in a rich set of theories including linear and non-linear arithmetic over the integers and reals, arrays, lists, uninterpreted functions, and bit-vector arithmetic.

2.1.8 Practical Ease of Modeling and Expressiveness

Even though theoretically two modeling languages may be equivalent in terms of expressiveness, in practice one may be much easier to use than another. For example, a language which provides no explicit notion of variables but requires users to encode the values of variables in the control states of an automaton is cumbersome to use except for toy systems. Also, a language which only provides Boolean data types is harder to use than a language which offers bounded integers or user-defined enumerations, even though the two languages are theoretically equivalent in terms of expressiveness. As a final example, a language which allows to declare process *types* and then to create multiple process *instances*, each with different parameters, is easier to use than a language which requires every process instance to be created in the model "by hand".

2.2 Modeling Languages

As mentioned earlier, it is useful to distinguish between formalisms, which are mathematical objects, and concrete modeling languages (and tools) which support such formalisms. A plethora of modeling languages exist, developed for different purposes, including:

- *Hardware description languages (HDLs).* These languages have been developed for modeling digital, analog, or so-called *mixed-signal* (combining digital and analog) circuits. Verilog, VHDL, and SystemC, are widespread HDLs. Tools implementing HDLs are provide features such as simulation, formal verification in some cases, and most importantly, automatic implementation such as logic synthesis and layout.
- *General-purpose modeling languages,* such as UML and SysML. These languages aim to capture many different aspects of software and systems in gen-

eral, and offer different sub-languages implementing various formalisms, from hierarchical state machines to sequence diagrams.

- *Architecture Description Languages (ADLs),* such as AADL. These languages aim to be system-level design languages, for software and other domain-specific systems (e.g., originally avionics systems, in the case of AADL).
- *Simulation-oriented languages and tools,* such as Matlab-Simulink or Modelica. These languages have their origin in modeling and simulation of physical systems and support ODE and DAE modeling. They have recently evolved, however, to encompass discrete models such as state machines, and to target the larger domain of control, embedded, and cyber-physical systems. Simulink and related tools provide primarily simulation, but also code generation and even formal verification in some cases.
- *Reactive programming languages,* such as the synchronous languages Lustre [22] and Esterel [14]. These languages were initially conceived as programming languages for reactive, real-time, and embedded systems. As a result, the tools that come with these languages are typically compilers and code generators, typically providing simulation for debugging purposes. However, synchronous languages and tools sometimes also provide exhaustive verification features, and include mechanisms for modeling environment assumptions and non-determinism. As a result, these languages can be used for more general modeling purposes, too.
- *Verification languages.* These languages have been developed specifically for formal verification purposes, using model-checking or theorem-proving techniques, including satisfiability solving. This class is the main focus of the present chapter.

It is beyond the scope of this chapter to provide a complete survey of modeling languages. As the topic of this book is formal verification, we focus on modeling languages developed specifically for verification purposes. However, even within this narrower domain, we can only list a small selection of languages, among all those proposed in the formal verification literature.

The list is presented in Table 1. Each language has been tailored to the particular kind of verification problem that it was designed to model, and the engines that underlie the corresponding tool. We have classified the languages along five dimensions: the supported formalism, data types, form of composition, properties (safety or liveness)[1], and the underlying computational engines. This listing is not meant to be exhaustive; rather the goal is to give the reader a flavor of the range of languages used in model checking today. We also note that several languages listed in the table were inspired by other formalisms; for example, the SAL language listed in Table 1 was inspired, in part, by the Reactive Modules formalism [6]. Further, it is important to remember that even verification tools that operate "directly" on programming languages such as C or Verilog extract some sort of formal model first, and this formal representation is typically very similar to modeling languages such as those listed in Table 1. Finally, bear in mind that this table lists characteristics of

[1] See Sec. 3.3 for definitions and a longer discussion of safety and liveness

modeling languages and tools at the time this article is being written, and that these may change over time.

Language	Supported formalisms	Data types	Composition	Properties	Engines
SMV	Discrete (finite-state systems)	Boolean, Finite-Domain	Synchronous (primarily)	Safety & liveness	BDDs, SAT
SPIN/Promela	Discrete (finite-state systems)	Enumerative, Channels, Records, ...	Asynchronous, message passing, rendez-vous	Safety & liveness	Explicit-state, partial order reduction, bit-state hashing
Murphi	Discrete (finite-state systems)	Boolean, finite-domain	Asynchronous	Safety & liveness	Explicit-state
SAL	Discrete and Hybrid systems (finite/infinite-state transition systems)	Boolean, Integer, Reals, Bitvectors, Arrays, ...	Synchronous & Asynchronous	Safety, liveness	BDDs, SAT & SMT
Alloy	Discrete (relational models)	Boolean, Integers, Bitvectors, Relations	Both (declarative relational modeling)	Safety	SAT
UCLID	Discrete (finite/infinite-state transition systems)	Boolean, Integers, Bitvectors, Arrays, Restricted Lambdas, ...	Synchronous & Asynchronous	Safety	BDDs, SAT & SMT
Uppaal	Timed (timed automata)	Enumerative, Arrays, Records, ...	Asynchronous with rendezvous	Safety & liveness	Difference-bound matrices, polyhedral reasoning
SpaceEx	Hybrid (hybrid automata)	Reals, discrete events	Asynchronous with synchronization on labels	Safety	Abstraction, polyhedral reasoning
Spec#/Boogie	Discrete (sequential programs)	Integer, Bitvectors, Arrays & other data structures	Object-oriented modules	Safety	Theorem proving, SMT

Table 1 A selection of verification languages and their main characteristics.

2.3 Challenges in Modeling

Beyond the computational difficulties of analyzing the models (e.g., state-explosion during model-checking, trace/time-explosion during simulation, etc.) there are other difficulties in modeling that may be viewed as being at a more high level, and thus harder to address. Some of these difficulties are as follows:

- Except in cases where the model is generated automatically (e.g., extracted from code), modeling is a creative process which can be quite difficult to get right.
- The choice of a modeling language/formalism is currently more of an art than a systematic science. There are few guidelines on how to go about this, and often the choice is dictated by historical or other reasons (e.g., company tradition, legacy models, etc.).
- Even after a model is constructed, it can be difficult to know whether the model is good, complete, or consistent. For specifications, this boils down to the problem often stated as: "have we specified enough properties?"
- Since models can themselves be incorrect or inconsistent, one must carefully interpret the results of verification. For example, in model checking, when a model fails to satisfy a property, is the model of the system wrong, or is the property incorrect? Such analysis usually requires some human insight.
- Constructing models of the environment, in particular, can be extremely tedious and error-prone. For instance, in automatic synthesis of systems from specifications (e.g., temporal logic), the process of writing down the specification, including constraints on the environment, is usually one of the hardest tasks. Similarly, for the problem of timing analysis of embedded software, many techniques involve having a human engineer painstakingly construct an abstract timing model of a microprocessor for use in software timing analysis (e.g., see [57] for a longer discussion). Automating the construction of environment models is an important challenge for extending the reach of formal verification and synthesis.

2.4 Scope of this Chapter

In the rest of this chapter, we seek to give a flavor of the above issues by introducing a simple modeling language, SML, and using it to model a small but diverse collection of systems. SML adopts many of the common features of modeling languages such as the idea of modeling a system as a *transition system* or an *abstract state machine* [34]. SML may not be the best fit, or even expressive enough, to model all types of systems and properties. With this in mind, we have chosen to make SML parametric with respect to its datatypes and the operations on those datatypes, illustrating with examples how different types of systems can be captured in the SML syntax. Some chapters in this handbook will introduce modeling formalisms of their own, which are similar to SML, but differ in their emphasis; for example, Chapter 16 on Software Verification introduces a transition system formalism that emphasizes aspects important in verifying safety properties of programs, such as having a special variable to model the program counter, and the notion of an error condition. We emphasize using SML the issues that arise across various system types, such as modularity, types of composition, and abstraction levels.

3 Modeling Basics

We define a language for modeling an abstract state machine called SML, which stands for "Simple Modeling Language". In this section, we present the syntax and semantics of SML.

3.1 Syntax

An SML program is made up of *modules*. Syntactically, an SML program is a list of *module* definitions. Each module definition comprises a module name followed by a module body. A module body, in turn, is made up of a list of definitions:

- a list of *input variable declarations*, $i_1 : \tau_1, i_2 : \tau_2, \ldots, i_k : \tau_k$;
- a list of *output variable declarations*, $o_1 : \tau'_1, o_2 : \tau'_2, \ldots, o_m : \tau'_m$;
- a list of a *state variable declarations*, $v_1 : \tau''_1, v_2 : \tau''_2, \ldots, v_l : \tau''_l$, and
- a list of *shared-variables declarations*, $u_1 : \tau'''_1, u_2 : \tau'''_2, \ldots, u_h : \tau'''_h$,
- a *behavior definition* which defines the transition and output relations of the module.

Some state variables may also be output variables. Shared variables are used for communication between asynchronously composed modules.

Declarations of variables may be omitted when the corresponding list is empty, e.g., a module with no shared variables will have no `sharedvars` section. Each variable has an associated type (domain of possible values), indicated above by the τ_i variables. We leave the types unspecified in this section; see Section 4 for examples.

The syntax of a module is given in Figure 1. It is useful to make a distinction between two kinds of modules:

- *primitive modules*, which are not made up of simpler modules, and thus omit the `composition`, `instances` and `connect` sections of the syntax,
- and *composite modules*, which are compositions of simpler modules, and thus include these sections.

A behavior definition of a primitive module comprises an *initial state definition* followed by a *transition relation definition*.

An initial state definition is a formula on state variables. A transition relation definition is a formula on input, output, state and "next-state" variables of the module. A next-state variable is of the form $\texttt{next}(v)$ where v is a state variable.

A composite module, in addition to the initial state and transition relation definitions, also includes:

- a *composition-instances declaration*, which declares the module instances that the composite module is made up of, and the form of composition, either *synchronous* or *asynchronous*, and

- a *connections definition*, which is a list (denoting conjunction) of binary equalities between two variables of the instances and of the composite module, or of a variable and a constant.

For a composite module, the transition relation section defines how variables defined locally in this module (i.e., not in sub-modules) evolve.

Given two instances `m1` and `m2` of a module M containing a variable `x`, we refer to the instances of `x` in `m1` and `m2` as `m1.x` and `m2.x`.

$$
\begin{array}{lll}
\texttt{module}\ M: & & \\
 & \texttt{inputs}: & i_1 : \tau_1; i_2 : \tau_2; \ldots, i_k : \tau_k; \\
 & \texttt{outputs}: & o_1 : \tau'_1; o_2 : \tau'_2; \ldots, o_m : \tau'_m; \\
 & \texttt{statevars}: & v_1 : \tau''_1; v_2 : \tau''_2; \ldots, v_l : \tau''_l; \\
 & \texttt{sharedvars}: & u_1 : \tau'''_1; u_2 : \tau'''_2; \ldots, u_h : \tau'''_h; \\
 & \texttt{init}: & \alpha; \\
 & \texttt{trans}: & \delta; \\
 & \texttt{composition}: & \texttt{synchronous} \mid \texttt{asynchronous}; \\
 & \texttt{instances}: & M_1, M_2, \ldots, M_n; \\
 & \texttt{connect}: & \gamma;
\end{array}
$$

Fig. 1 Module Syntax in SML. A primitive module does not have the `composition`, `sharedvars`, `instances` and `connect` sections.

For examples of primitive modules, see the Constant, Scale, Difference and DiscreteIntegrator modules in Figure 11. For examples of composite modules, see the Helicopter and System modules in Figure 11.

3.2 Dynamics

We give semantics to an SML program by viewing it as a *symbolic transition system* (STS). In Section 5, we will relate STSs to one of the classical modeling formalisms, Kripke structures.

An STS is a tuple $(I, O, V, U, \alpha, \delta)$ where:

- I, O, V, U are finite sets of input, output, state and shared variables, each variable also having an associated type;
- α is a formula over $V \cup U$ (the initial states predicate);
- δ is a formula over $I \cup O \cup V \cup U \cup V' \cup U'$ (the transition relation), where $V' = \{s' \mid s \in V\}$ is a set of primed state variables representing the "next state variables", and similarly with U'.

Given an SML module M as given in Figure 1, we define the STS for it as follows. If M is primitive, its syntax directly defines the tuple $(I, O, V, \emptyset, \alpha, \delta)$ (the set of shared variables is empty). If M is composite, then we define its STS in terms of the STSs for its constituent module instances. Suppose M is a composition of module instances $M_1, M_2, \ldots, M_n$, with corresponding STSs of the form $(I_i, O_i, V_i, U_i, \alpha_i, \delta_i)$, for $i = 1, ..., n$. Let β be the predicate derived from the initial state declaration of M (not its constituent modules). Let γ be the predicate derived from the connections declarations of M as a conjunction of each of the equations making up those declarations. Then, the STS for M is $(I, O, V, U, \alpha, \delta)$, where:

- I is $(\bigcup_{i=1}^{n} I_i) \setminus I_0$, where I_0 is the set of those input variables that are connected to an output variable in the connections declarations of M;
- O is $\bigcup_{i=1}^{n} O_i$;
- V is $\bigcup_{i=1}^{n} V_i$;
- U is $\bigcup_{i=1}^{n} U_i$;
- α is $\beta \wedge \bigwedge_{i=1}^{n} \alpha_i$;
- δ depends on the form of composition:
  - For synchronous composition, δ is defined as $\gamma \wedge \bigwedge_{i=1}^{n} \delta_i$.
  - For asynchronous composition there are a couple of choices. We will define here the choice of *interleaving semantics*, where a transition of the composition involves one module taking a transition while all others *stutter* with their state variables remaining unchanged. Under this choice, δ is $\gamma \wedge \bigvee_{i=1}^{n} (\delta_i \wedge \bigwedge_{j \neq i} \sigma_j)$, where $\sigma_j = \bigwedge_{v \in V_j} v = v'$ defines stuttering of module M_j.

3.3 Modeling Concepts

We now discuss concepts that are fundamental to creating and understanding models for formal verification.

3.3.1 Open and Closed Systems

A closed system is one that has no inputs. Any system with one or more inputs is open. In verification, we typically deal with closed systems, obtained by composing the system under verification with (a model of) its environment. In SML, we typically model interacting open systems as individual SML modules, and their composition, which is to be verified, is a closed system.

3.3.2 Safety and Liveness

Modeling formalisms for verification must be designed for the class of properties to be verified. In this regard, the most general categorization of properties is into *safety* and *liveness*. Formally, a property ϕ is a *safety property* if, for any infinite trace (execution) of the system, it does not satisfy ϕ if and only if there exists a finite prefix of that trace that cannot be extended to an infinite trace satisfying ϕ. We say that ϕ is a *liveness property* if every finite-length execution trace can be extended to an infinite trace that satisfies ϕ.[2] Chapter 2 gives several examples of safety and liveness properties, expressed in temporal logic.

Models encode safety or liveness concerns in different ways. Safety properties are defined by the *transition relation* of the model. With suitable encoding, the violation of a safety condition can viewed as taking one of a set of "bad" transitions. Thus, by allowing certain transitions and disallowing others, a model can restrict its permitted executions to those adhering to a safety property. On the other hand, liveness properties are usually represented using *fairness conditions* on infinite execution paths of a model. We discuss fairness in more depth below, but, in essence, fairness constraints can be used to rule out ways in which a finite-length execution of the model can be extended to an infinite execution violating the desired liveness condition.

3.3.3 Fairness

An important concept in execution of composed modules is fairness. Fairness constraints block infinite executions that are not realistic for concurrent systems, and are often required to demonstrate liveness properties. In other words, the fairness constraints are needed to ensure a proper resolution of the nondeterministic decisions taken during the execution of a concurrent system, especially in the asynchronous (interleaving) composition case: no module gets neglected and each module always makes progress.

There are two major types of fairness: *strong* and *weak*. Weak (Büchi-type) fairness establishes that a step of a module cannot be enabled *forever* after some point in the execution of the system, without being taken. For example, consider a simple system with two processes, A and B, and two integer variables, x and y. A has a loop in which it repeatedly increments x. B has a single transition where it sets y to 1. Without weak fairness, an infinite execution of the system is possible where B never executes, and A keeps updating x forever. This execution is not fair according to the weak fairness definition, because B's single transition is always enabled but never taken in that execution.

Weak fairness is useful, but only goes that far. Sometimes, a stronger notion of fairness is needed. For instance, consider a modified version of the above example, where B's transition is guarded by the condition that x is odd. Then, the execution where only A moves is weakly fair, because B's transition is not always enabled: it is only enabled once every two steps that A takes. Strong fairness can be used to eliminate this type of unfair executions. Strong (Streett-type) fairness guarantees that a step cannot be enabled *infinitely often* without being taken.

[2] See papers by Lamport [46] and Alpern and Schneider [1] for a detailed treatment of safety and liveness.

The above examples show how fairness can be used to eliminate unrealistic behaviors resulting from the asynchronous (interleaved) composition of a set of modules. But fairness can be useful for modeling behavior even within a single module. An example of this setting is presented in Sec. 4.1.5.

3.3.4 Encapsulation

The module interface is defined in terms of its inputs and outputs, i.e., the environment communicates with the module by updating its input variables, which in response reacts by updating its output variables. A good model will expose *all* internal state that can (should) be accessed by its environment and *only* that state: this is important for both ensuring that the verifier does not miss bugs or generate spurious error reports.

3.3.5 Moore and Mealy machines

The SML notation can be easily used both for Mealy and Moore machines, the standard modeling formalisms. For Moore machines, the output relation simply takes the form $\bigwedge_{j=1}^{m} o_j = f_j(V)$, where f_j denotes the output function for output variable o_j. Similarly, for Mealy machines, the output relation would take the form $\bigwedge_{j=1}^{m} o_j = f_j(V, I)$.

4 Examples

In order to illustrate modeling with SML, we present three examples from three different problem domains: digital circuits (Sec. 4.1), control systems (Sec. 4.2), and concurrent software (Sec. 4.3). Each example is a simplified version of a design artifact arising in practice. In each case, we begin by presenting this design, its simplification, and the corresponding SML model. We then describe how the SML model might need to be transformed in order to succeed at various verification tasks associated with the model. Our goal is to illustrate the various modeling considerations discussed in Sec. 2.1 in the particular context of each example.

4.1 Synchronous Circuits

Our representative example of digital circuits is a simple chip multiprocessor (CMP) router. First, we present a brief description of this example, and then describe it in SML notation.

4.1.1 Router Design

Network-on-chip (NoC) architectures are the backbone of modern, multicore processors and system-on-chip (SoC) designs, serving as the communication fabric between processor cores and other on-chip devices such as controllers for memory and input-output devices. It is important to prove that individual routers and networks of interconnected routers operate as per specification. The CMP router design [55] we focus on is part of an on-chip interconnection network that connects processor cores with memory and with each other. The main function of the router is to direct incoming packets to the correct output port. Each packet is made up of smaller components called *flits*. There are three kinds of flits: a *head flit*, which reserves an output channel, one or more *body flits*, which contain the data payload, and a *tail flit*, which signals the end of the packet. The typical data layout of a flit is depicted in Figure 2. The two least-significant bits represent the flit type, the next 6 most-significant bits represent the destination address, and the 24 most significant bits contain the data payload.

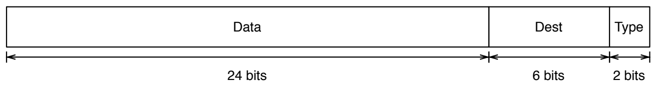


Fig. 2 Anatomy of a flit. A flit contains a 24-bit data payload, a 6-bit destination address, and a 2-bit type.

The CMP router consists of four main modules, as shown in Figure 3. The *input controller* buffers incoming flits and interacts with the *arbiter*. Upon receipt of a head flit, the input controller requests access to an output port based on the destination address contained in the head flit. The arbiter grants access to the output ports in a fair manner, using a simple round-robin arbitration scheme. The remaining modules are the *encoder* and *crossbar*. When the arbiter grants access to a particular output port, a signal is sent to the input controller to release the flits from the buffers, and at the same time, an allocation signal is sent to the *encoder* which in turn configures the *crossbar* to route the flits to the appropriate output port.

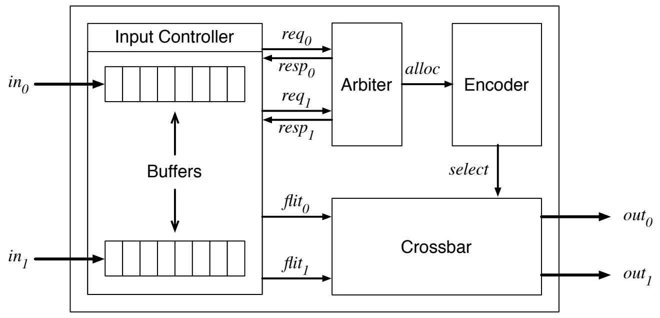


Fig. 3 Chip-Multiprocessor (CMP) Router. There are four main modules: the input controller, the arbiter, the encoder, and the crossbar.

4.1.2 Simplifications and SML Model

Since the original router design is quite large, we make several simplifications to create a small example for this introductory chapter. First, we assume that each flit that makes up the packet raises a request to the arbiter, not just the head flit; this eliminates some logic tracking the type of flit. Second, we assume that the destination address, which indicates the output port on which a flit must be directed, to be exactly two bits. Specifically, bits 2 and 3 of each flit encode the address, with `01` encoding output port 0 (bit 2 is `1` and bit 3 is `0`), and `10` encoding output port 1. The destination address is directly copied to the request lines of the arbiter; the encoding `00` for a request line indicates the absence of a request. Finally, we eliminate the Encoder module, directly using the `alloc` signals to route flits through the crossbar to the output ports of the router.

Fig. 4 shows the SML representation of the router. We use the usual short form of declarations where variables of the same type are grouped together in the same declaration statement. The top-level module is termed `System`. Note that it has two inputs, and therefore is an open system. The top-level module is a synchronous composition of three modules: the input controller, the arbiter, and the crossbar. The input controller module in turn is a synchronous composition of two instances of the module modeling a FIFO buffer, plus some additional control logic to request access to an output port from the arbiter. Note how the buffers are defined as arrays mapping bitvectors to bitvectors, and implemented as circular queues. The arbiter uses a single priority bit to mediate access to an output port when both input ports request that same output port; if there is no conflict between output port requests, both can be granted simultaneously. The alloc signals generated by the arbiter are used in the crossbar to direct the flit at the head of the input buffers to the corresponding output port, if one is granted. If no flit is directed to an output port, the value of that output is set to a default invalid value `NAF` (standing for “not a flit”).

```
module InputController {
  inputs:  in0, in1 : bitvec[32];
           resp0, resp1 : bitvec[1];
  outputs: req0, req1 : bitvec[2];
           flit0, flit1 : bitvec[32];
  wires: deq0, deq1: bool;
  composition: synchronous
  instances:
     B0 : Buffer;  B1 : Buffer;
  connect:
     B0.flit_in = in0;
     B1.flit_in = in1;
     B0.dequeue = deq0;
     B1.dequeue = deq1;
     B0.flit_out = flit0;
     B1.flit_out = flit1;
  trans:
      deq0 = (resp0 != 0)
   && deq1 = (resp1 != 0)
   && req0 = flit0[3:2]
   && req1 = flit1[3:2];
}

module Buffer {
  inputs:  flit_in : bitvec[32];
           dequeue : bool;
  outputs: flit_out : bitvec[32];
  statevars:
      buf[SZ] : array bitvec[32] -> bitvec[32];
      num : bitvec[4];
      head : bitvec[4];
      tail : bitvec[4];
  init:
      num = 0x0 && head = 0x0 && tail = 0x0
      && for (i in 0x0..0x7):
           buf[i] = NAF;
  trans:
     flit_out = buf[head]
  && (flit_in != NAF) =>
         (num < SZ) =>
                next(tail) = (tail+1 mod SZ)
             && next(buf[tail]) = flit_in
             && next(num) = num+1
      && (dequeue && num > 0x0) =>
                next(head) = (head+1 mod SZ)
             && next(num) = num-1;
}
```

```
module Arbiter {
  inputs: req0, req1 : bitvec[2];
  outputs: resp0, resp1 : bitvec[1];
           alloc0, alloc1 : bitvec[2];
  statevars: priority : bool;
  init: priority = false;
  trans:
     req0 = 0 && req1 = 0 =>
       resp0 = 0 && resp1 = 0
  && req0 != 0 && req1 = 0 =>
       resp0 = 1 && resp1 = 0
  && req0 = 0 && req1 != 0 =>
       resp0 = 0 && resp1 = 1
  && req0 != 0 && req1 != 0 =>
        req0 = req1 =>
             priority =>
               resp0 = 0 && resp1 = 1
           && (!priority) =>
              resp0 = 1 && resp1 = 0
           && next(priority) = !priority
     && req0 != req1
           resp0 = 1 && resp1 = 1
  && (req0 = 01 && resp0 = 1) => alloc0 = 00
  && (req0 = 10 && resp0 = 1) => alloc0 = 01
  && (req1 = 01 && resp1 = 1) => alloc1 = 00
  && (req1 = 10 && resp1 = 1) => alloc1 = 01
  && (resp0 = 0 => alloc0 = 11)
  && (resp1 = 0 => alloc1 = 11);
}

module Crossbar {
  inputs: in0, in1 : bitvec[32];
          alloc0, alloc1 : bitvec[2];
  outputs: out0, out1 : bitvec[32];
  statevars: flit0, flit1 : bitvec[32];
  init: flit0 = NAF && flit1 = NAF;
  trans:
     out0 = flit0 && out1 = flit1
  && alloc0 = 00 => next(flit0) = in0
  && alloc0 = 01 => next(flit1) = in0
  && alloc1 = 00 => next(flit0) = in1
  && alloc1 = 01 => next(flit1) = in1
  && alloc0 = 11 & alloc1 = 00 => next(flit1) = NAF
  && alloc0 = 11 & alloc1 = 01 => next(flit0) = NAF
  && alloc1 = 11 & alloc0 = 00 => next(flit1) = NAF
  && alloc1 = 11 & alloc0 = 01 => next(flit0) = NAF;
}
```

```
module System {
  inputs: in0, in1 : bitvec[32];       outputs: out0, out1 : bitvec[32];
  composition: synchronous;
  instances: InpCtrl : InputController, Arb : Arbiter, Xbar : Crossbar,
  connect:
    in0 = InpCtrl.in0; in1 = InpCtrl.in1; out0 = Xbar.out0; out1 = Xbar.out1;
    InpCtrl.req0 = Arb.req0; InpCtrl.req1 = Arb.req1;
    InpCtrl.resp0 = Arb.resp0; InpCtrl.resp1 = Arb.resp1;
    Xbar.in0 = InpCtrl.flit0; Xbar.in1 = InpCtrl.flit1;
    Arb.alloc0 = Xbar.alloc0; Arb.alloc1 = Xbar.alloc1;
}
```

Fig. 4 A simplified chip multiprocessor router modeled in SML. `NAF` stands for “not a flit”, and is implemented as the bitvector value `0x00000003`. We use a `wires` declaration to introduce new names for expressions. `SZ` is a parameter denoting the size of queues.

4.1.3 Verification Task: Progress through the Router

Consider the following verification problem stated in English below:

> Any incoming flit on an input port is routed to its correct output port, as specified by its destination address, within L clock cycles.

This is a typical *latency bound property* that every router must satisfy. The bound L on the latency is left parametric for now.

A latency bound property with a fixed bound L falls in a broader class of properties known as *quality-of-service* properties.

It is common, however, to use an abstraction of this property that only requires L to be finite, but places no other requirement on its value. In essence, such a property specifies that any incoming flit is to be *eventually* routed to the correct output port.

Both forms of the latency bound property can be expressed in *temporal logic*, a specification formalism that will be explored in greater depth in Chapter 2 on temporal logic.

4.1.4 Data Type Abstraction

Consider a simple environment that injects exactly one packet onto each input port, with the destination of the packets modeled by an symbolic destination field in the respective head flit. Each packet comprises a head flit, several body flits, and a tail flit. By making the destination field symbolic, we can model both interesting scenarios: the case where the two injected packets are destined for different output ports as well as the case where they are headed for the same output port (resulting in contention to be resolved by the arbiter).

Figure 5 gives the SML code for the above environment model. Note that the outputs generated by this environment are the inputs to the top-level module `System` in the router design. These outputs are modeled as bit-vector variables.

The choice of how to model the datatype of the flit can have a big impact on the scalability of verification. In work by Brady et al. [15], a router design almost identical to the one given in this chapter was considered for verification.[3] Two types of verification tasks were performed. In both cases, bounded-model checking (BMC) (covered in Chapter 10) was used to check that starting from a reset state, the router correctly forwards both packets to their respective output ports within a fixed number of cycles that depends on the length of the packet (`PKT_SZ`). The difference was in how the data component of flits in all modules of the design were represented. In one case, the data component of each flit was modeled as a bitvector 28 bits wide (as in Fig. 5), while in the second case, this was modeled as an *abstract term*, which can be encoded with fewer bits by an underlying decision procedure for a combination of uninterpreted functions and equality and bit-vector arithmetic (see Chapter

[3] The differences have to do with modeling the crossbar and routing logic more accurately than we have in this chapter, and are not significant for the discussion herein.

```
module SimpleRouterEnv {
  inputs:  iflit0, iflit1 : bitvec[32]; // flits from router
  outputs: oflit0, oflit1 : bitvec[32]; // flits to router
  statevars:
      counter : bitvec[32];
      data0 : bitvec[28]; data1 : bitvec[28]; // changes non-determinisically
      dest0 : bitvec[2]; dest1 : bitvec[2]; // arbitrary value fixed for packet
  init:
      counter = 0x0 && (dest0 = 10 || dest0 = 01)
   && (dest1 = 10 || dest1 = 01) // non-deterministically assigned 01 or 10
  trans:
      next(counter) = counter + 1          // 32-bit finite-precision addition
   && next(dest0) = dest0 && next(dest1) = dest1 // stays unchanged for the packet
   && (counter = 0) =>
           oflit0 = (data0 @ dest0 @ 10) // head flit
        && oflit1 = (data1 @ dest1 @ 10) // head flit
   && (counter >= 1 || counter < PKT_SZ-1) =>
           oflit0 = (data0 @ dest0 @ 00) // body flit
        && oflit1 = (data1 @ dest1 @ 00) // body flit
   && (counter = PKT_SZ-1) =>
           oflit0 = (data0 @ dest0 @ 01) // tail flit
        && oflit1 = (data1 @ dest1 @ 01) // tail flit
   && counter >= PKT_SZ => oflit0 = NAF && oflit1 = NAF  // not a flit
}
```

Fig. 5 A simple environment model for the chip multiprocessor router modeled in Fig. 4. `PKT_SZ` is a parameter denoting the size of the packet (i.e., the number of flits per packet).

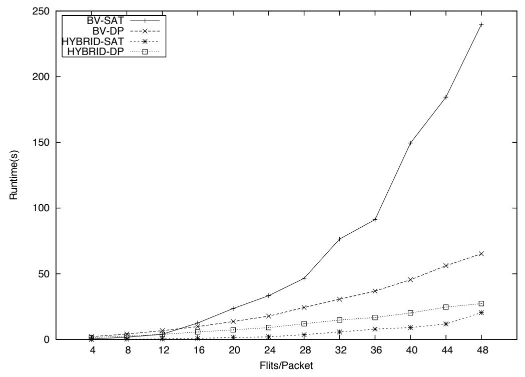


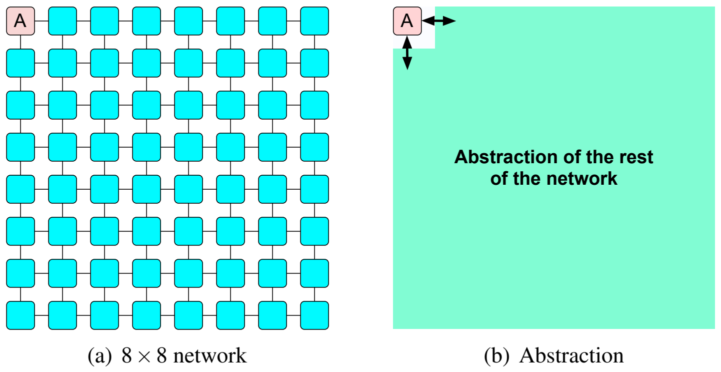
Fig. 6 Runtime comparison for increasing packet size in CMP router model. The runtimes for the bitvector-level and term-level designs are compared for increasing packet size. Runtimes for the underlying decision procedure are broken up into that required for generating a SAT solver encoding ("DP") and that taken for the SAT solving ("SAT").

This example illustrates the points in Sec. 2.1 related to the right level of abstraction required to ensure that the underlying computational engine (a SAT-based decision procedure — also known as an SMT solver — in this case) can efficiently verify the problem at hand.

4.1.5 Environment Modeling

In the previous section, we considered the verification of a router with a rather contrived environment model that simply injects one packet into the routers input ports. In reality, such a router will be interconnected with other network elements in a specified topology. For example, consider an 8×8 grid of interconnected nodes shown in Fig. 7(a), where each node typically represents a router and network interface logic (e.g., connecting the router with a core or memory element). For this section, we will assume that each node is simply a router design similar to that given in Fig. 4, but with five input and output ports — four of these can be connected to neighboring nodes on the grid, and one can be connected to the processor or memory element associated with that node.

Next, consider a specific node in the grid labeled A. Suppose that we want to verify the property that every packet traveling through A spends no more than 15 cycles within A. One approach is to model the overall network as a synchronous composition of 64 routers, one for each node. However, this results in a model with tens of thousands of Boolean state variables, which is beyond the capacity of the best current formal verification tools.



(a) 8×8 network (b) Abstraction

Fig. 7 Verifying latency through a router in an on-chip network. We wish to verify a bound on the latency through node A.

An obvious alternative approach involves abstraction and modeling with non-determinism. Specifically, as depicted in Fig. 7(b), one can abstract all nodes in the network other than A into an environment model E, where E is a transition system which, at each cycle, non-deterministically selects a port to send an input packet to A, sets the destination port for that packet, and chooses a data payload. If we apply this technique to the problem of verifying a latency bound of 15 through the router, it fails to prove the latency bound using a sequence of input packets that causes contention for the same output port within the router. The question is, however, whether such a pattern can arise in this particular network for the traffic patterns that the architect has in mind. Purely non-deterministic modeling of traffic sources does not permit us to model traffic sources in a more precise manner.

Thus, as mentioned in Sec. 2.1, the environment model must be tailored to the type of property to be verified. In this case, in order to prove a particular latency bound, we must come up with a suitable model of traffic sources (and sinks).

One formalism for more fine-grained modeling of any channel of NoC traffic (including sources and sinks) is a *bounded channel regulator*, introduced by Holcomb et al. [37] based on the regulator model described by Cruz [27]. A *channel* in an NoC is any link between components in the network. A bounded channel regulator $\mathscr{T}_{\mathscr{R}}$ is a monitor on a channel that checks three constraints: the *rate* ρ, *burstiness* σ, and bound B on the traffic that traverses the channel. Figure 8 illustrates a bounded channel regulator. The buffer of size σ begins filled with tokens, and one token is removed whenever a head flit passes through the channel being monitored. Src_ρ adds a token to the buffer once every ρ cycles, unless the buffer is already full. During a simulation, Src_ρ becomes inactive once it has produced a total of $B-\sigma$ tokens. If a head flit ever passes through the channel when the regulator queue is empty, then a Boolean flag a_i is set to signal that the channel traffic does not conform to the constraints of regulator i. We refer to a regulator with rate ρ, burstiness σ, and bound B as $\mathscr{T}_{\mathscr{R}}(\rho,\sigma,B)$. It is easy to model a bounded regulator in SML using non-deterministic assignment for generating packets, a FIFO buffer for storing tokens, and a counter for enforcing the rate of the traffic regulator.

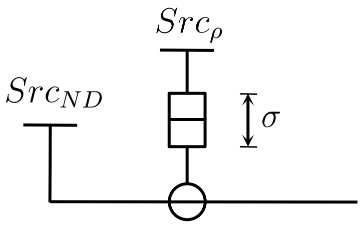


Fig. 8 Traffic regulator. The bounded channel regulator $\mathscr{T}_{\mathscr{R}}(\rho,\sigma,B)$ models all traffic patterns with average rate constrained by ρ, burstiness by σ, and total number of packets bounded by B.

The regulator can be applied to any channel in a network that is viewed as a generator of packets, such as a neighboring router, a processing element, or a memory element. Such packet generators can be modeled as a completely non-deterministic source Src_{ND} (as shown in Fig. 8). Combining Src_{ND} with the regulator provides a way to restrict behaviors of the non-deterministic source that are overly-adversarial compared to the actual design that this model abstracts away. It is also possible to model traffic sinks in a similar fashion.

Since router A in Fig. 7(b) is at a corner of the grid, it sees less incoming traffic than other routers in the center. With suitable parameters ρ, σ, and B for the bounded regulator model, one can verify the latency bound of 15 through the router

`A`, as reported by Holcomb et al. [37]. The parameters can be generated manually or inferred automatically from traces generated from program executions.

4.1.6 Summary

In this section, we discussed the latency-bound verification problem for a CMP router. Several of the considerations outlined in Sec. 2.1 arose. First, since the system was a digital circuit with a single clock, a discrete state machine formalism based on synchronous composition was suitable. Next, additional datatype abstraction was necessary to reduce the search space for the underlying SAT-based computational engines. A suitable environment model had to be created to reason about a latency-bound property. We started with a simplistic environment model, but then, in order to consider the more realistic whole-network scenario, had to formulate suitable models for sources (and sinks) of network traffic. This environment model also incorporates relevant abstractions to reduce the search space for verification.

4.2 Synchronous Control Systems

In the previous section we presented an example of synchronous digital circuit modeled in SML. Synchronous digital circuits are an important class of systems that can be modeled using the synchronous model of computation. Another important class of systems which can be often viewed as synchronous systems are control systems, implemented either in hardware or software. Control systems typically consist of a *controller* which interacts with a *plant*, that is, a physical process to be controlled, in a closed-loop manner. The controller typically samples certain observable variables of the plant periodically through sensors, and issues commands to the plant through actuators.

In this section we present a toy feedback control system consisting of a simple proportional controller for a simple helicopter model. The example is an adapted version of the one described in Section 2.4 of [49]. There, the model is a *continuous-time* model. Here, we present a *discrete-time* (synchronous) version with the aim of illustrating how it could be captured in SML.

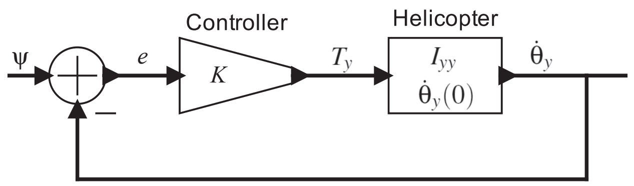


Fig. 9 A simple feedback control system. A proportional controller for a helicopter model (taken from [49]).

The system is shown graphically in a block-diagram notation in Figure 9. It contains, at the top level, the Controller and Helicopter modules connected in feedback. The input of the Helicopter module is torque, denoted by T_y, and its output is angular velocity, which is the time derivative $\dot{\theta}_y$ of the angle θ_y. The input of the Controller is the error e, that is, the difference between the target angular velocity ψ (which we will take to be constant) and the actual angular velocity.

In this toy example the Controller is a simple proportional controller which multiplies the error by a constant K, that is, the controller implements $T_y = K \cdot e$. The Helicopter consists of two sub-modules, a Scale and an Integrator, as shown in Figure 10.

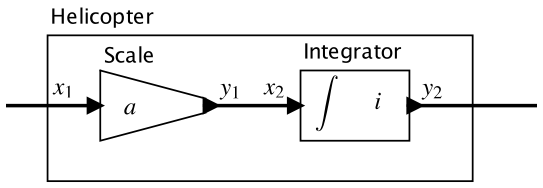


Fig. 10 A simple helicopter model. The Helicopter block of Figure 9 (from [49]).

The entire system, modeled in SML, is shown in Figure 11. The top-level System module instantiates and connects four sub-modules, a Helicopter, a Proportional-Controller, a Constant (modeling the target angular velocity ψ) and a Difference module (computing the error e). The Helicopter module consists of a Scale and a DiscreteIntegrator (replacing the continuous integrator of Figure 10). The ProportionalController consists simply of a Scale. The Constant, Scale, Difference and DiscreteIntegrator modules are primitive modules, while the rest are composite modules.

Properties of interest in control systems are stability, robustness, and control performance. Such properties can sometimes be expressed in formal specification languages such as temporal logic. For instance, in the helicopter example above, we may wish the error e to eventually become almost zero, and remain close to zero forever after. We may also want e to become almost zero within a certain deadline. Finally, we may also want e not to exceed certain upper and lower bounds as it converges to zero (i.e., bounded "overshoot" and "undershoot"). All these properties can be formalized in temporal logic.

4.3 Concurrent Software

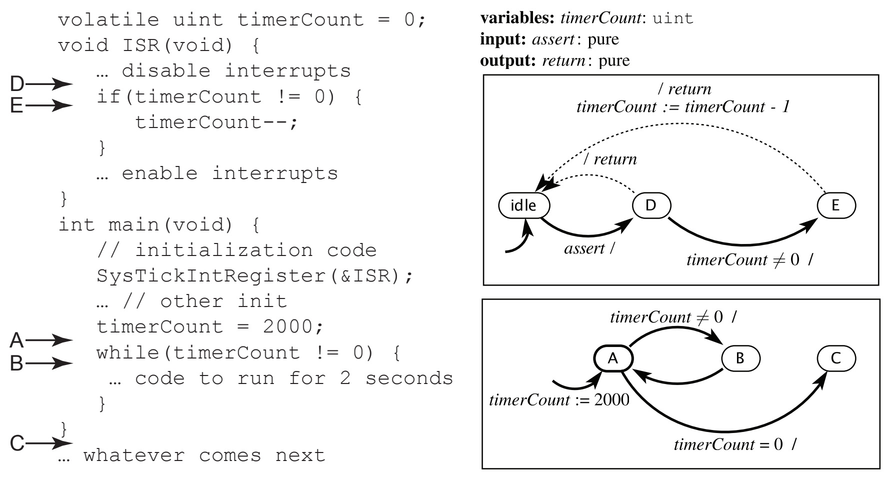
Concurrent software has long been a key application domain for model checking. In this section, we show how modeling a concurrent program for verification can be subtle, even when there are small programs with limited concurrency.

```
module Constant {
  inputs:  value : real;
  outputs: out   : real;
  trans:
    out = value;
}

module Scale {
  inputs:    in    : real;
  outputs:   out   : real;
  statevars: coeff : real; // const
  trans:
    out = in * coeff;
}

module Difference {
  inputs:  in1 : real,
           in2 : real;
  outputs: out : real;
  trans:
    out = in1 - in2;
}

module DiscreteIntegrator {
  inputs:    in  : real;
  outputs:   out : real;
  statevars: sum : real;
     // initialized externally
  trans:
    out = sum &&
    next(sum) = in + sum;
}
```

```
module ProportionalController {
  inputs:    in    : real;
  outputs:   out   : real;
  statevars: coeff : real;
  composition: synchronous;
  instances: S : Scale;
  connect:
    S.coeff = coeff;
    S.in    = in;
    out     = S.out;
}


module Helicopter {
  inputs:    in    : real;
  outputs:   out   : real;
  statevars: coeff : real;
  init:
    I.sum = 0.0;
  composition: synchronous;
  instances: S : Scale,
             I : DiscreteIntegrator;
  connect:
    S.coeff = coeff;
    S.in    = in;
    I.in    = S.out;
    out     = I.out;
}
```

```
module System {
  composition: synchronous;
  instances: Heli : Helicopter,
             Ctrl : ProportionalController,
             Trgt : Constant,
             Diff : Difference;
  connect:
    Trgt.value = 0.0;
    Diff.in1   = Trgt.out;
    Diff.in2   = Heli.out;
    Ctrl.coeff = 10.0;
    Ctrl.in    = Diff.out;
    Heli.coeff = 10.0;
    Heli.in    = Ctrl.out;
}
```

Fig. 11 A toy synchronous control system modeled in SML.



```
    volatile uint timerCount = 0;
    void ISR(void) {
       … disable interrupts
D──➤   if(timerCount != 0) {
E──➤      timerCount--;
       }
       … enable interrupts
    }
    int main(void) {
       // initialization code
       SysTickIntRegister(&ISR);
       … // other init
A──➤   timerCount = 2000;
B──➤   while(timerCount != 0) {
        … code to run for 2 seconds
       }
C──➤ }
    … whatever comes next
```

variables: *timerCount*: uint
input: *assert*: pure
output: *return*: pure
/ *return*
timerCount := *timerCount* - 1
/ *return*
idle
D
E
assert /
timerCount ≠ 0 /
timerCount ≠ 0 /
A
B
C
timerCount := 2000
timerCount = 0 /

Fig. 12 Modeling a program that does something for two seconds and then continues to do something else (figure from [49]).

Consider the program outlined in Figure 12.[4] The main function seeks to perform some action (at the program location denoted B) for two seconds and then do something else (at location C). To keep track of time, the program uses a timer interrupt. The function `ISR` is registered as the interrupt service routine for this timer interrupt. The interrupt is raised every millisecond, so `ISR` will be invoked 2000 times in two seconds. The program seeks to track the number of invocations of `ISR` by using the shared variable `timerCount`.

Consider verifying the following property:

> The `main` function of the program will always reach position C.

In other words, will the program eventually move beyond whatever computation it was to perform for two seconds?

A natural approach to answer this question is to model both the `main` and `ISR` functions as state machines. In Fig. 12, we show two finite state machines that model `ISR` and `main`. The states of the FSMs correspond to positions in the execution labeled A through E, as shown in the program listing. Let us further assume that the `main` function loops in location C.

Note that these state machine models incorporate some important assumptions. One of these is on the atomicity of operations in the program. The program locations A through E are between C statements, so implicitly we are assuming that each C statement is an atomic operation — a questionable assumption in general. However, for simplicity, we will make this assumption here.

Another modeling assumption concerns the frequency with which interrupts can be raised. The raising of an interrupt is modeled in Fig. 12 using the input `assert`,

[4] This example is taken from a textbook on Embedded Systems [49].

which is a "pure" signal meaning that it is an event that is either present or absent at any time step (this can be modeled in SML as a Boolean). Assuming that interrupts can occur infinitely often, we can model the environment that raises interrupts as a state machine that non-deterministically raises an interrupt (sets `assert`) at each step.

The next question becomes how to compose these state machines to correctly model the interaction between the two pieces of sequential code in the procedures `ISR` and `main`. As first glance, one might think of *asynchronous composition* as a suitable choice. However, that choice is incorrect in this context because the interleavings are not arbitrary. In particular, `main` can be interrupted by `ISR`, but `ISR` cannot be interrupted by `main`. Asynchronous composition fails to capture this asymmetry.

Synchronous composition is also not a good fit. When `ISR` is running, the `main` function is not, and vice-versa, unlike synchronous composition where the transition systems move in lock step.

Therefore, to accurately compose `main` and `ISR`, we need to combine them with a *scheduler* state machine. The scheduler has two modes: one in which `main` is executing, and one in which `ISR` is executing after it pre-empts `main`. This model also requires minor modifications to the state machines `main` and `ISR`. All three state machines are then composed together hierarchically and synchronously. Fig. 13 shows the SML model for the combination.

The resulting machine is composed with its environment state machine, which models the raising of an interrupt. This final composition is asynchronous, reflecting the modeling assumption that the environment and the concurrent program do not share a common clock. However, we probably want to rule out the interleaving in which only the environment steps without ever giving the concurrent program a chance to execute. A fairness specification allows us to impose this modeling constraint.

With this model, we are able to verify that, in fact, the program does not satisfy the desired property. One counterexample involves an interrupt being repeatedly raised by `Env` and the program spends all its time in invocations of `ISR` with `main` unable to make any progress.

To summarize this example, we note that even the simplest concurrent programs, such as the interrupt-driven program of this section, can be tricky to model. Although conventional wisdom holds that asynchronous composition is the "right" choice for concurrent software verification, one must be careful to take into consideration relative priorities of tasks/processes and scheduling policies.

5 Kripke Structures

This section introduces Kripke Structures, perhaps the most common formalism for specifying system models. It then defines the mapping between the constructs of the modeling language defined in Section 3 and the elements of the Kripke Structures.

```
module Main {
  inputs: enable: bool;
  statevars: pc : {A, B, C};
  sharedvars: timerCount : integer;
  init:
      timerCount = 2000 && pc = A
  trans:
    enable =>
        pc = A =>
             timerCount = 0  => next(pc) = C
          && timerCount != 0 => next(pc) = B
     && pc = B =>
          next(pc) = A
     && pc = C =>
          next(pc) = C
  && !enable =>
          next(pc) = pc
}

module ISR {
  inputs: enable : bool;
  outputs: return : bool;
  statevars: pc : {idle, D, E};
  sharedvars: timerCount : integer;
  init:
      pc = idle
  trans:
     enable =>
        pc = idle =>
           next(pc) = D && !return
     && pc = D && timerCount != 0 =>
           next(pc) = E && !return
     && pc = D && timerCount = 0 =>
           next(pc) = idle && return
     && pc = E =>
           next(timerCount) = timerCount-1
           && next(pc) = idle && return
  && !enable =>
        next(pc) = pc && !return
}
```

```
module System {
  inputs: assert : bool;
  statevars: mode : {main, ISR};
  wires: M_enable, I_enable: bool;
  composition: synchronous;
  instances:
     M : main; I : ISR;
  connect:
     M.timerCount = I.timerCount;
     M.enable = M_enable;
     I.enable = I_enable;
  init:
      mode = main
  trans:
     M_enable = (mode = main)
  && I_enable = (mode = ISR)
  && (mode = main || next(mode) = main)
     && assert =>
         next(mode) = ISR
  && mode = ISR && I.return =>
       next(mode) = main
}


module Environment {
  outputs: assert : bool;
  init: true
  trans: true
}


module System_and_Env {
  composition: asynchronous;
  instances:
     Sys : System,
     Env : Environment;
  connect:
     Sys.assert = Env.assert
}
```

Fig. 13 A simple interrupt-driven program modeled in SML.

5.1 Transition Systems

Transition systems are the most common formalism used in formal verification since they naturally capture the dynamics of a discrete system. Transition systems are directed graphs where nodes model *states*, and the edges represent *transitions* denoting the state changes. A state encapsulates information of the system (i.e., values of the system variables) at a particular moment in time during its execution. For instance, a state of the mutual exclusion protocol can indicate the critical or noncritical sections of the system processes. Similarily, for example in hardware circuits executing synchronously, a state can represent the register values in addition to the values of the input bits. Transitions encapsulate the gradual changes that the system parameters exhibit at each execution step of the system. In the case of the mutual exclusion protocol a transition may indicate that a process moves from its non-critical section to a waiting or critical section state. In software, on the other hand, the transition may correspond to the execution of a program statement (say, an assignment operation) which may result in the change of the values of some program variables together with a program counter. While in hardware, a transition models the update of the registers and the outputs bits in responce to the updated set of inputs.

There exist many various classes of transition systems. The choice of a particular transition system depends on the nature of the system being modeled. This chapter presents the most common formalism, called Kripke Structures, which is suitable for modeling of most hardware and software systems. Kripke structures are transition systems which are specified by constructs called *atomic propositions*. Atomic propositions express facts about the states of the system, for example, "**Heli.coeff = 10.0**" for variable **Heli.coeff** from the helicopter control system example.

Definition 1. (**Kripke Structure**) (cf. [25]) Let AP be a non-empty set of atomic propositions. A Kripke structure is transition system defined as a four tuple $M = (S;S_0;R;L)$, where S is a finite set of states, $S_0 \in S$ is a finite set of initial states ($S_0 \subseteq S$), $R \subseteq S \times S$ is a transition relation, for which it holds that $\forall s \in S : \exists s' \in S : (s;s') \in R$, and $L : S \to 2^{AP}$ is the labeling function which labels each state with the atomic propositions which hold in that state.

A path of the Kripke structure is a sequence of states $s_0, s_1, s_2, ...$ such that $s_0 \in S_0$ and for each $i \leq 0$, $s_{i+1} = R(s_i)$.

The word on the path is a sequence of sets of the atomic propositions $\omega = L(s_1), L(s_2), L(s_3), ...$, which is an ω-word over alphabet 2^{AP}.

The program semantics is defined by its language which is a set of finite (infinite) words of all possible paths which the system can take during its execution.

Kripke structures are the models defining semantics (definition of when a specified property holds) for the most widely used specification languages for reactive systems, namely temporal logics.

Kripke structures can be seen as describing the behavior of the modeled system in an modeling language independent manner. Therefore, temporal logics are really modeling formalism independent. The definition of atomic propositions is the only thing that needs to be adjusted for each formalism.

A fair execution of the program modeled by a Kripke structure is ensured by fairness constraints which rule out the unrealistic paths. A fair path ensures that certain fairness constraints hold. In general, a strong fairness constraint can be defined as a temporal logic formula of the following form: $\mathbf{GF}\, AP \Longrightarrow \mathbf{GF}\, AP'$, where AP, AP' are sets of atomic propositions of interest. The formula states that if some atomic propositions AP are true infinitely often (corresponding to the fact that a process is ready to execute, or that some transition is enabled) then atomic propositions AP' will be true infinitely often as well (the process will execute, or the transition will be taken). A weak fairness constraint is can be defined as $\mathbf{FG}\, AP \Longrightarrow \mathbf{GF}\, AP'$, stating that if AP holds continuously after some point on, then AP' holds infinitely often.

5.2 *From* SML *Programs to Kripke Structures*

For purposes of verification, we must close the model of the system under verification with the model of its environment. For such a closed SML program, suppose that the corresponding STS is $(\emptyset, O, V, \alpha, \delta)$. From this STS, we obtain the corresponding Kripke structure as (S, S_0, R, L), where $S = 2^{V \cup O}$, $S_0 = \{s \in S \,|\, \alpha(s) = \mathbf{true}\}$, $R = \delta$, and L is such that $L(s)$ is the value of variables in $V \cup O$ in state s.

6 Summary

This chapter has reviewed some of the fundamental issues in system modeling for verification. We introduced SML, a simple modeling language for abstract state machines or transition systems, and illustrated its use on three examples from different domains. We also gave semantics to SML using the well-known formalism of Kripke structures. The rest of this handbook will cover several concepts in model checking, many of which are based on domain-specific modeling formalisms. SML captures the essential features of those formalisms and potentially provides a basis for better understanding the connections between chapters and even creating new connections.

References

1. Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
2. R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, 104(1):2–34, 1993.
3. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
4. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
5. R. Alur and T. Henzinger. Logics and models of real time: a survey. In *Real Time: Theory in Practice*, LNCS 600, 1992.
6. R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
7. C. Baier and M. E. Majster-Cederbaum. Denotational semantics in the CPO and metric approach. *Theoretical Computer Science*, 135(2):171–220, 1994.
8. Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Performance evaluation and model checking join forces. *Commun. ACM*, 53(9):76–85, 2010.
9. Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors. *Validation of Stochastic Systems - A Guide to Current Research*, volume 2925 of *Lecture Notes in Computer Science*. Springer, 2004.
10. F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36:45–52, 2003.

11. Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.
12. Gerd Behrmann, Kim Guldstrand Larsen, and Jacob Illum Rasmussen. Priced timed automata: Algorithms and applications. In *Third International Symposium on Formal Methods for Components and Objects (FMCO)*, pages 162–182, 2004.
13. A. Benveniste, P. Caspi, R. Lublinerman, and S. Tripakis. Actors without Directors: a Kahnian View of Heterogeneous Systems. In *HSCC'09: Proceedings of the 12th International Conference on Hybrid Systems: Computation and Control*, LNCS, pages 46–60. Springer, 2009.
14. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
15. Bryan Brady, Randal Bryant, and Sanjit A. Seshia. Abstracting RTL designs to the term level. Technical Report UCB/EECS-2008-136, EECS Department, University of California, Berkeley, Oct 2008.
16. Bryan A. Brady, Randal E. Bryant, Sanjit A. Seshia, and John W. O'Leary. ATLAS: automatic term-level abstraction of RTL designs. In *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2010.
17. D. Broman, E.A. Lee, S. Tripakis, and M. Törngren. Viewpoints, Formalisms, Languages, and Tools for Cyber-Physical Systems. In *6th International Workshop on Multi-Paradigm Modeling (MPM'12)*, 2012. http://avalon.aut.bme.hu/mpm12/papers/paper%205.pdf.
18. Manfred Broy and K. Stolen. *Specification and Development of Interactive Systems*, volume 62 of *Monographs in Computer Science*. Springer, 2001.
19. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
20. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.
21. J. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California, Berkeley, 1993.
22. P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symp. POPL*. ACM, 1987.
23. Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Quantitative languages. In *Proc. Computer Science Logic (CSL)*, volume 5213 of *LNCS*, pages 385–400, 2008.
24. Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Alternating weighted automata. In *Fundamentals of Computation Theory (FCT)*, volume 5699 of *LNCS*, pages 3–13, 2009.
25. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.
26. F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journ. Computer and System Science 5*, pages 511–523, 1971.
27. R. L. Cruz. A calculus for network delay, part I. network elements in isolation. *IEEE Transactions on Information theory*, 37(1):114 –131, Jan 1991.
28. Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
29. A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu. A next-generation design framework for platform-based design. *Conference on Using Hardware Design and Verification Languages (DVCon)*, 152, 2007.
30. M.H.A. Davis. *Markov Models and Optimization*. Chapman & Hall, London, 1993.
31. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The Tool KRONOS. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III: Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer, 1996.

32. J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.
33. Wan Fokkink. *Introduction to Process Algebra*. Springer, 2nd edition, 2007.
34. Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors. *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings*, volume 1912 of *Lecture Notes in Computer Science*. Springer, 2000.
35. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8:231–274, 1987.
36. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
37. Daniel Holcomb, Bryan Brady, and Sanjit A. Seshia. Abstraction-based performance analysis of NoCs. In *Proceedings of the Design Automation Conference (DAC)*, pages 492–497, June 2011.
38. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
39. Jianghai Hu, John Lygeros, and Shankar Sastry. Towards a theory of stochastic hybrid systems. In *Hybrid Systems: Computation and Control (HSCC)*, volume 1790 of *Lecture Notes in Computer Science*, pages 160–173. Springer, 2000.
40. ITU. Z.120 – Message Sequence Chart (MSC). Available at http://www.itu.int/rec/T-REC-Z.120, 02/2011.
41. ITU. Z.120 Annex B: Formal semantics of Message Sequence Charts. Available at http://www.itu.int/rec/T-REC-Z.120, 04/1998.
42. G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, Proceedings of IFIP Congress 74*. North-Holland, 1974.
43. R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, November 1966.
44. Z. Kohavi. *Switching and finite automata theory, 2nd ed.* McGraw-Hill, 1978.
45. M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS*, pages 220–270. Springer, 2007.
46. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.
47. K. Larsen, P. Petterson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1/2), October 1997.
48. E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
49. Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems – A Cyber-Physical Systems Approach*. 2011. http://LeeSeshia.org.
50. X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, 409(1):110–125, 2008.
51. Sharad Malik and Lintao Zhang. Boolean satisfiability: From theoretical hardness to practical success. *Communications of the ACM (CACM)*, 52(8):76–82, 2009.
52. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
53. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.
54. Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.
55. Li-Shiuan Peh. *Flow Control and Micro-Architectural Mechanisms for Extending the Performance of Interconnection Networks*. PhD thesis, Stanford University, August 2001.

56. W. Reisig. *Petri Nets: An Introduction*. Springer, 1985.
57. Sanjit A. Seshia. Quantitative analysis of software: Challenges and recent advances. In *7th International Workshop on Formal Aspects of Component Software (FACS)*, October 2010.
58. C. Stergiou, S. Tripakis, E. Matsikoudis, and E. A. Lee. On the Verification of Timed Discrete-Event Models. In *FORMATS 2013*. Springer, 2013.
59. B.D. Theelen, M.C.W. Geilen, S. Stuijk, S.V. Gheorghita, T. Basten, J.P.M. Voeten, and A.H. Ghamarian. Scenario-aware dataflow. Technical Report ESR-2008-08, Eindhoven University of Technology, July 2008.
60. S. Tripakis, C. Stergiou, C. Shaver, and E. A. Lee. A modular formal semantics for Ptolemy. *Mathematical Structures in Computer Science*, 23:834–881, July 2013.
61. R. K. Yates. Networks of real-time processes. In E. Best, editor, *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*, volume LNCS 715. Springer-Verlag, 1993.