And/Or Hierarchies and Round Abstraction

Radu Grosu

Department of Computer and Information Science
University of Pennsylvania
Email: grosu@cis.upenn.edu
URL: www.cis.upenn.edu/~grosu

Abstract. Sequential and parallel composition are the most fundamental operators for incremental construction of complex concurrent systems. They reflect the temporal and respectively the spatial properties of these systems. Hiding temporal detail like internal computation steps supports temporal scalability and may turn an asynchronous system to a synchronous one. Hiding spatial detail like internal variables supports spatial scalability and may turn a synchronous system to an asynchronous one. In this paper we show on hand of several examples that a language explicitly supporting both sequential and parallel composition operators is a natural setting for designing heterogeneous synchronous and asynchronous systems. The language we use is SHRM, a visual language that backs up the popular and/or hierarchies of statecharts with a well defined compositional semantics.

1 Introduction

With the advent of very large scale integration technology (VLSI), digital circuits became too complex to be designed and tested on a breadboard. The hardware community introduced therefore languages like Verilog and VHDL [Ver,Vhdl] that allow to describe the architectural and the behavioral structure of a complex circuit in a very abstract and modular way. Architectural modularity means that a system is composed of subsystems using the operations of parallel composition and hiding of variables. Behavioral hierarchy means that a system is composed of subsystems using the operations of sequential composition and hiding of internal computation steps. Verilog allows the arbitrary nesting of the architecture and behavior hierarchies.

With the advent of object oriented technology, most notably UML [BJR97], combined visual/textual languages very similar in spirit to the hardware description languages [Har87,SGW94], gained a lot of popularity in the software community. Their behavior and block diagrams were rapidly adopted as a high level interface for Verilog and VHDL too (e.g. in the Renoir tool of Mentor Graphics and in the StateCad tool of Visual Software Solutions).

Recent advances in formal verification have led to powerful design tools for hardware (see [CK96] for a survey), and subsequently, have brought a lot of hope of their application to reactive programming. The most successful verification technique has been *model checking* [CE81,QS82]. In model checking, the system is described by a state-machine model, and is analyzed by an algorithm that explores the reachable state-space of the model. The state-of-the-art model checkers (e.g. Spin [Hol97] and Smv [McM93]) employ a variety of heuristics for efficient search, but are typically unable to analyze models with more than hundred state variables, and thus, *scalability* still remains a challenge.

A promising approach to address scalability is to exploit the modularity of the design. The input languages of standard model checkers (e.g., S/R in Cospan [AKS83] or Reactive modules in Mocha [AH99]) support architectural modularity but, unlike the hardware and the visual description languages, provide no support for modular description of the behaviors of individual components. In [AG00] we introduced the combined visual/textual language *hierarchic reactive modules* (Hrm) exhibiting both behavior and architecture modularity. This hierarchy is exploited for efficient search by the model checker Hermes [AGM00].

In this paper we introduce a synchronous version of the hierarchic reactive modules language (Shrm) that conservatively extends the reactive modules language. This language is used to model two very interesting abstraction operators of reactive modules: `next` and its dual `trigger`. They allow to collapse and delay arbitrary many consecutive steps of a module and environment respectively, and therefore to perform a temporal abstraction. This abstraction can be exploited efficiently in model checking because the states stored for the intermediate steps may be discarded. We argue that a language like Shrm and Verilog, supporting the arbitrary nesting of architecture and behavior hierarchies, is a natural setting for combined spatial and temporal abstraction. There is no need for special temporal operators because behavioral modularity does precisely the same thing. Moreover, by supporting sequential composition, choice, loops and preemption constructs, the combined setting allows to express complex structure in a more direct and intuitive way. To materialize this claim we reformulate the adder example in [AH99].

The rest of the paper is organized as follows. In Section 2 we introduce the modeling language Shrm. This language adds communication by events to the language Hrm presented in [AG00]. It also extends the reactive modules language both with behavior hierarchy and with a visual notation. In Section 3 we show that this language is a natural setting to perform spatial and temporal abstraction. As an application, we show how to encode the next operator of reactive modules. Finally in Section 4 we draw some conclusions.

2 Modeling Language

The central component of the modeling language is a *mode*. The attributes of a mode include global variables used to share data with its environment, local variables, well-defined entry and exit points, and submodes that are connected with each other by transitions. The transitions are labeled with guarded commands that access the variables according to the the natural scoping rules. Note that the transitions can connect to a mode only at its entry/exit points, as in

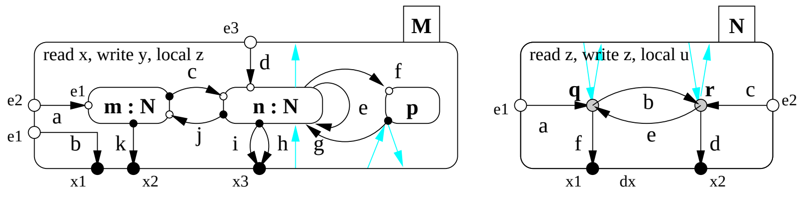

Fig. 1. Mode diagrams

Room but unlike statecharts. This choice is important in viewing the mode as a black box whose internal structure is not visible from outside. The mode has a *default exit* point, and transitions leaving the default exit are applicable at all control points within the mode and its submodes. The default exit retains the history, and the state upon exit is automatically restored by transitions entering the default *entry point*. Thus, a transition from the default exit is a group preemption transition and a transition from the default exit to the default entry is an group interrupt transition. While defining the operational semantics of modes, we follow the standard paradigm in which transitions are executed repeatedly until there are no more enabled transitions.

Modes A *mode* has a refined control structure given by a hierarchical state machine. It basically consists of a set of *submode instances* connected by *transitions* such that at each moment of time *only one* of the submode instances *is active*. A submode instance has an associated mode and we require that the modes form an *acyclic* graph with respect to this association. For example, the mode `M` in Figure 1 contains two submode instances, `m` and `n` pointing to the mode `N`. By distinguishing between modes and instances we may control the degree of *sharing* of submodes. Sharing is highly desirable because submode instances (on the same hierarchy level) are never simultaneously active in a mode. Note that a mode resembles an *or* state in statecharts but it has more powerful structuring mechanisms.

Variables and scoping A mode may have *global* as well as *local variables*. The set of global variables is used to share data with the mode's environment. The global variables are classified into *read* and *write variables.* The *local variables* of a mode are accessible only by its transitions and submodes. The local and write variables are called *controlled variables.* Thus, the scoping rules for variables are as in standard structured programming languages. For example, the mode `M` in Figure 1 has the global read variable `x`, the global write variable `y` and the local variable `z`. Similarly, the mode `N` has the global read-write variable `z` and the local variable `u`. Each variable x may be used as a register. In this case, the expression $p(x)$ denotes the value of x in the *previous top level round*[1] and the expression x denotes the *current* value of x.

[1] What previous top level round means will be made clear when discussing parallel modes.

The transitions of a mode may refer only to the declared global and local variables of that mode and only according to the declared read/write permission. For example, the transitions `a,b,c,d,e,f,g,h,i,j` and `k` of the mode `M` may refer only to the variables `x, y` and `z`. Moreover, they may read only `x` and `z` and write `y` and `z`. The global and local variables of a mode may be shared between submode instances if the associated submodes declare them as global (the set of global variables of a submode has to be included in the set of global and local variables of its parent mode). For example, the value of the variable `z` in Figure 1 is shared between the submode instances `m` and `n`. However, the value of the local variable `u` is not shared between `m` and `n`.

Control points and transitions To obtain a modular language, we require the modes to have well defined *control points* classified into entry points (marked as white bullets) and exit points (marked as black bullets). For example, the mode `M` in Figure 1 has the entry points `e1,e2, e3` and the exit points `x1,x2,x3`. Similarly, the mode `N` has the entry points `e1,e2` and the exit points `x1,x2`. The transitions connect the control points of a mode and of its submode instances to each other. For example, in Figure 1 the transition `a` connects the entry point `e2` of the mode `M` with the entry point `e1` of the submode instance `m`. The name of the control points of a transition are attributes and our drawing tool allows to optionally show or hide them to avoid cluttering.

According to the points they connect, we classify the transitions into *entry, internal* and *exit* transitions. For example, in Figure 1, `a,d` are *entry* transitions, `h,i,k` are *exit* transitions, `b` is an *entry/exit* transition and `c,e,f,g,j` are *internal* transitions. These transitions have different types. Entry transitions initialize the controlled variables by reading only the global variables. Exit transitions read the global and local variables and write only the global variables. The internal transitions read the global and the local variables and write the controlled variables.

Default control points To model preemption each mode (instance) has a special, default exit point *dx*. In mode diagrams, we distinguish the default exit point of a mode from the regular exit points of the mode, by considering the default exit point to be represented by the mode's border. A transition starting at *dx* is called a *preempting* or *group transition* of the corresponding mode. It may be taken whenever the control is inside the mode and no internal transition is enabled. For example, in Figure 1, the transition `f` is a group transition for the submode `n`. If the current control point is `q` inside the submode instance `n` and neither the transition `b` nor the transition `f` is enabled, then the control is transferred to the default exit point *dx*. If one of `e` or `f` is enabled and taken then it acts as a preemption for `n`. Hence, inner transitions have a higher priority than the group transitions, i.e., we use *weak preemption*. This priority scheme facilitates a modular semantics. As shown in Figure 1, the transfer of control to the default exit point may be understood as a *default exit transition* from an exit point *x* of a submode to the default exit point *dx* that is enabled if and only if, all the explicit outgoing transitions from x are disabled. We exploit this intuition in the symbolic checker.

History and closure To allow history retention, we use a special default entry point *de*. As with the default exit points, in mode diagrams the default entry point of a mode is considered to be represented by the mode's border. A transition entering the default entry point of a mode either restores the values of all local variables along with the position of the control or initializes the controlled variables according to the read variables. The choice depends on whether the last exit from the mode was along the default exit point or not. This information is implicitly stored in the constructor of the state passed along the default entry point. For example, both transitions `e` and `g` in Figure 1, enter the default entry point *de* of `n`. The transition `e` is called a *self* group transition. A self group transition like `e` or more generally a self loop like `f,p,g` may be understood as an interrupt handling routine. While a self loop may be arbitrarily complex, a self transition may do simple things like counting the number of occurrences of an event (e.g., clock events). Again, the transfer of control from the default entry point *de* of a mode to one of its internal points x may be understood as a *default entry transition* that is taken when the value of the local history variable coincides with x. If x was a default exit point $n.dx$ of a submode n then, as shown in Figure 1, the default entry transition is directed to $n.de$. The reason is that in this case, the control was blocked somewhere inside of n and default entry transitions originating in $n.de$ will restore this control. A mode with added default entry and exit transitions is called *closed*. Note that the closure is a semantic concept. The user is not required to draw the implicit default entry and exit transitions. Moreover, he can override the defaults by defining explicit transitions from and to the default entry and exit points.

Operational semantics: macro-steps In Figure 1, the execution of a mode, say `n`, starts when the environment transfers the control to one of its entry points `e1` or `e2`. The execution of `n` terminates either by transferring the control back to the environment along the exit points `x1` or `x2` or by "getting stuck" in `q` or `r` as all transitions starting from these leaf modes are disabled. In this case the control is implicitly transferred to `M` along the default exit point `n.dx`. Then, if the transitions `e` and `f` are enabled, one of them is nondeterministically chosen and the execution continues with `n` and respectively with `p`. If both transitions are disabled the execution of `M` terminates by passing the control implicitly to its environment at the default exit `M.dx`. Thus, the transitions within a mode have a higher priority compared to the group transitions of the enclosing modes.

Intuitively, a round of the machine associated to a mode starts when the environment passes the updated state along a mode's entry point and ends when the state is passed to the environment along a mode's exit point. All the internal steps (the *micro steps*) are hidden. We call a round also a *macro step*. Note that the macro step of a mode is obtained by alternating its closed transitions and the macro steps of the submodes.

Denotational semantics: traces The execution of a mode may be best understood as a game, i.e., as an alternation of moves, between the mode and its environment. In a *mode move*, the mode gets the state from the environment along its entry points. It then keeps executing until it gives the state back to the environment along one of its exit points. In an *environment move*, the environment gets the state along one of the mode's exit points. Then it may update any variable except the mode's local ones. Finally, it gives the state back to the mode along one of its entry points. An *execution* of a mode M is a sequence of macro steps of the mode. Given such an execution, the corresponding *trace* is obtained by projecting the states in the execution to the set of global variables. The *denotational semantics* of a mode M consists of its control points, global variables, and the set of its traces.

Atoms and parallel modes An *atom* is a mode having only two points, the default entry point and the default exit point. A *parallel mode* is a very convenient abbreviation for a particular mode consisting of the parallel composition of atoms. To avoid race conditions, the parallel composition of atoms is defined only if (1) the atoms write disjoint sets of variables and (2) there is no cyclic dependency among the variables of different atoms (this similar to [AH99] and it can be statically checked). A weaker form of cyclic dependency is however allowed: for any write variable x in an atom A, another atom B may safely refer to $p(x)$, the *previous value of* x. If the atom B refers to x, than it refers to the last value of x, i.e., the value of x produced at the end of the subround of A. The atom B has therefore to *await* the atom A.

Since a mode may update a controlled variable x several times, we have to make sure that $p(x)$ is well defined, no matter how many times the variable is updated. In the following, we consider $p(x)$ to be the value of x at the *end of the previous top level round.* A top level round is the round of the *top level atom* containing x. Syntactically, a top level atom is an atom prefixed by the keyword `top`. Semantically, a top level atom makes sure that at the end of each round, $p(x)$ is updated to the current value of x.

Top level atoms fix the granularity of interaction and therefore they may be used only in the parallel composition of other top level atoms (parallel composition does not alter this granularity). Modes and parallel modes also fix the spatial and temporal granularity of computation. Modes and top level atoms in Shrm closely correspond to tasks and modules in Verilog. Tasks are programming units whereas modules are simulation units.

Semantics of parallel modes The semantics of a parallel mode is very similar to the semantics of modules in [AH99]. As shown in [AG00] this semantics can be completely defined in terms of modes as follows. Take an arbitrary linearization of the await dependency among the atoms of a parallel mode (since the await dependency is a partial order, this is always possible). Construct a mode by connecting the atoms with identity transitions, as required by the linearization.

If the parallel mode is a top level atom, update $p(x)$ to the last value of x[2]. The language generated by this mode, defines the semantics of the parallel mode.

By definition, a parallel mode is a particular atom. As a consequence it may be freely used inside a mode as a submode. Hence, SHRM allows the arbitrary nesting of the architecture and behavior hierarchies. When conveniently, we will draw a parallel mode as a block diagram with atoms as boxes and shared variables as arrows. The entry/exit point information is not very informative for parallel modes (and atoms).

Events The shared variables communication paradigm and the notion of top level round allows us to model *events* as toggling boolean variables. Sending an event e is the action $e := \neg p(e)$ and receiving an event e is the boolean expression $e \neq p(e)$. These are abbreviated by $e!$ and $e?$ respectively. Note that, no matter how many times a mode sends an event inside a top level round, only one event is sent to the other modes.

Renaming of modes Similarly to modules in [AH99], modes may be renamed. Given a mode m and a renaming $x_1, \ldots, x_n := y_1, \ldots, y_n$ where x_i are global variables and y_i are fresh variables, the mode $m[x_1, \ldots, x_n := y_1, \ldots, y_n]$ is a mode identical with m excepting that the variables x_i are replaced with the variables y_i, for $1 \leq i \leq n$.

3 Temporal and Spatial Abstraction

In order to reduce the complexity of a system, [AH99] introduce the abstraction operator **next**. Given a module m and a subset Y of its interface (write) variables, **next** Y **for** m collapses consecutive rounds of m until one of the variables in Y changes its value.

A *controlled state* of m is a valuation for the controlled variables of m and an *external state* of m is a valuation for the external (read) variables of m. For two external states s and t of m, an *iteration of m from s to t* is a finite sequence $s_0 \ldots s_n$ of controlled states of m such that $n \geq 1$ and for all $0 \leq i < n$ the state $s_{i+1} \cup t$ is an successor of the state $s_i \cup s$. In other words, along an iteration the controlled variables are updated while the external variables stay unchanged.

The iteration $s_0 \ldots s_n$ *modifies* the set Y of controlled variables if $s_n[Y] \neq s_0[Y]$ and for all $0 \leq i < n, s_i[Y] = s_0[Y]$, where $s[Y]$ is the projection of the state s on the variables in Y. If the iteration modifies Y then the state $s_n \cup t$ is called the *Y-successor* of the state $s_0 \cup s$. A *round marker* for the module m is a nonempty set Y of interface variables such that for all states s and t of m, there are nonzero and finitely many Y-successors u of s such that u and t agree on the values of the external (read) variables of m.

[2] This is a simpler definition than in [AG00]. This is because we use here the notion of top level atoms.

If Y is a *round marker* for the module m, then the abstraction **next** Y **for** m is a module with the same declaration as m and a single atom A_m^Y. The update relation of A_m^Y contains pairs (s, t) where t is a Y-successor of s.

Within the language SHRM, the next abstraction is a simple but important case of sequential control on top of parallel modes. Given a (parallel) mode m and a round marker Y, the mode corresponding to **next** Y **for** m is shown in Figure 2. The game semantics of modes provides exactly the meaning of next above. The state (token) s is passed by the environment to the mode **next** Y

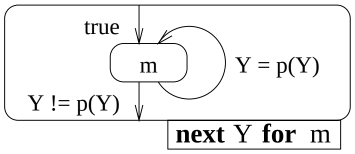

Fig. 2. Next abstraction

for m along its default entry point *de*. The state t is passed back by the mode to the environment along its default exit point *dx* only if t is a Y-successor of s (in this case $Y \neq p(Y)$). As long as the state token is inside `next` the environment does not have any chance to modify it. As a consequence, the states $s_0 \ldots s_{n-1}$ computed by repeatedly traversing the loop are an iteration for this mode. None of these states are Y-successors of s because of the loop guard $Y = p(Y)$. Since the set Y is a round marker for m there is always the possibility for the loop to terminate. The textual variant of Figure 2 is shown below[3].

```
atom next (m, Y) is
  read m.read;
  write m.write;
  submode m;

  transition from de to m.de is
    true -> skip;
  transition from m.dx to m.de is
    Y = p(Y) -> skip;
  transition from m.dx to dx is
    Y != p(Y) -> skip;
```

A generalization of the next operation above is the *reuse* of a (parallel) mode. In this case additional control is needed to prepare the input and store the output of the reused mode. For example, consider a one bit *adder* implemented as a parallel mode as shown in Figure 3.

[3] The mode m and the set Y are considered parameters in this specification. The selectors *m.read* and *m.write* return the read and write variables of m.

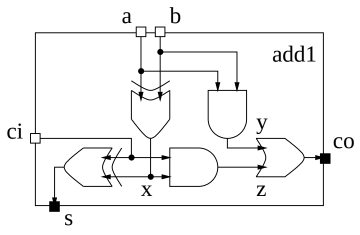

Fig. 3. One bit adder

Its textual equivalent is given below. It reproduces the circuit in Figure 3.

```
atom add1 is
  read a,b,ci : bool;
  write s, co : bool;
  local x, y, z : bool;
  || xor[in1, in2, out := a, b, x]
  || and[in1, in2, out := a, b, y]
  || xor[in1, in2, out := ci, x, s]
  || and[in1, in2, out := ci, x, z]
  || or[in1, in2, out := y, z, co]
```

Suppose now that we want to define a two bit adder by using in parallel two one bit adders, i.e., by decomposing the two bit addition *spatially.* The *spatial scaling involves a local variable (wire)*, that passes the the carry bit from the lower bit adder to the higher bit adder. Hence, spatial abstraction involves hiding of local variables as shown in Figure 4 left. The textual equivalent is given below. Note that the spatial abstraction does not change the notion of a round (or clock cycle). This remains the same for all modes (circuits) constructed in this way. Combinational cycles are prohibited by the parallel composition operation.

```
atom pAdd2 is
  read x, y : array (0..1) of bool; cIn : bool;
  write z : array (0..1) of bool; cOut : bool;
  local c : bool;
  || add1[a, b, s, ci, co := x[0], y[0], z[0], cIn, c]
  || add1[a, b, s, ci, co := x[1], y[1], z[1], c, cOut]
```

Suppose now that we want to define the two bit adder by reusing the one bit adder, i.e., by decomposing the two bit addition *temporally.* This implementation splits each computation step into two micro-steps. In the first micro-step the one bit adder is used to add the lower bits. In the second micro-step the one bit adder is used to add the higher order bits. Similarly, an n-bit adder can be implemented in n micro-steps. To capture the *micro step* intuition, we have to *hide (or compress) the micro steps* into one computation step. But this exactly what mode encapsulation is about.

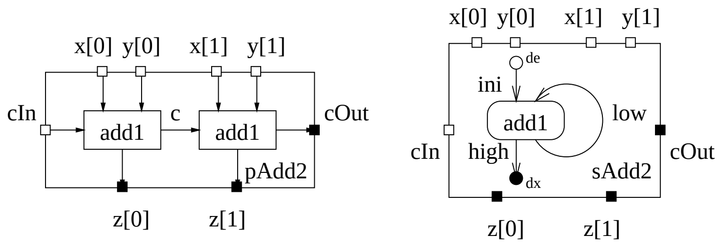

Fig. 4. Two bit adders

In contrast to the simple next operation defined before, in this case we also have to prepare the input for the one bit adder and to store the partial results. We also need a local counter to count the number of micro steps. This implementation is shown visually in Figure 4 right. Its textual definition is given below. The reader is urged to compare it with the less intuitive and much more involved implementation given in [AH99].

```
atom sAdd2 is
  read x, y : array (0..1) of bool; cIn : bool;
  write z : array (0..1) of bool; cOut : bool;
  local a, b, s, ci, co, r : bool;
  transition ini from de to add1.de is
    true -> r := 0; a := x[0]; b := y[0]; ci := cIn;
  transition low from add1.dx to add1.de is
    r = 0 -> r := 1; z[0] := s; a := x[1]; b := y[1]; ci := co;
  transition high from add1.dx to dx is
    r = 1 -> z[1] := s; cOut := co;
```

The game semantics of modes makes the **trigger** construct from [AH99] superfluous. As long as the top level atom does not pass the state token, the environment cannot modify it. As a consequence, it cannot work faster than the atom itself and this is exactly the purpose of **trigger**.

4 Conclusions

In this paper we have introduced a synchronous visual/textual modeling language for reactive systems that allows the arbitrary nesting of architectural and behavioral hierarchy. We have shown that such a language is the natural setting for spatial and temporal scaling and consequently for the modeling of heterogenous synchronous and asynchronous (stuttering) systems. This language is more expressive than reactive modules because it allows to define behavior hierarchy. It is more expressive than hierarchic reactive modules because it supports communication by events. In a nutshell, it has much of the expressive power of Verilog and VHDL and yet it has a formal semantics that supports the efficient application of formal verification techniques, especially of model checking.

The additional expressive power with respect to reactive and hierarchic reactive modules does not come however, for free. When applying symbolic search (e.g. invariant checking) we have to introduce an additional fresh variable px for each each variable x addressed as $p(x)$. To avoid this waste, we could classify the variables like in VHDL, into proper variables and signals and disallow the repeated updating of signals. By insisting that only signals x can be are addressed as $p(x)$ no additional space is required. In conclusion, even though experimental data is small so far, conceptual evidence suggests that a language supporting the arbitrary nesting of behavior and architecture hierarchy could be beneficial both for modeling and for analysis.

Acknowledgments We would like to thank Rajeev Alur for reading a draft of this paper and providing valuable feedback. We would also like to thank Tom Henzinger for fruitful discussions and enthusiasm for a language supporting both hierarchies. This work was supported by the DARPA/NASA grant NAG2-1214.

References

[AG00] R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Proceedings of the 27th Annual ACM Symposium on Principles of Programming Languages*, pages 390–402, 2000.

[AGM00] R. Alur, R. Grosu, M. McDougall. Efficient Reachability Analysis of Hierarchical Reactive Machines. In *Proceedings of the 12th Conference on Computer Aided Verification*, Chicago, USA, 2000.

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[AHM$^+$98] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proceedings of the 10th International Conference on Computer Aided Verification*, LNCS 1427, pages 516–520. Springer-Verlag, 1998.

[AKS83] S. Aggarwal, R.P. Kurshan, and D. Sharma. A language for the specification and analysis of protocols. In *IFIP Protocol Specification, Testing, and Verification III*, pages 35–50, 1983.

[AKY99] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *Automata, Languages and Programming, 26th International Colloquium*, pages 169–178. 1999.

[AY98] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Proceedings of the Sixth ACM Symposium on Foundations of Software Engineering*, pages 175–188. 1998.

[BHSV$^+$96] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentell, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *Proceedings of the Eighth Conference on Computer Aided Verification*, LNCS 1102, pages 428–432. 1996.

[BJR97] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide.* Addison Wesley, 1997.

[BLA$^+$99] G. Behrmann, K. Larsen, H. Andersen, H. Hulgaard, and J. Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. In *TACAS '99: Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Software*, 1999.

[CAB$^+$98] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–519, 1998.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[CK96] E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[JM87] F. Jahanian and A.K. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C-36(8):961–975, 1987.

[LHHR94] N.G. Leveson, M. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process control systems. *IEEE Transactions on Software Engineerings*, 20(9), 1994.

[McM93] K. McMillan. *Symbolic model checking: an approach to the state explosion problem*. Kluwer Academic Publishers, 1993.

[PD96] L. Peterson and B. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 1996.

[Pet81] G. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3), 1981.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *Proceedings of the Fifth International Symposium on Programming*, LNCS 137, pages 195–220. Springer-Verlag, 1982.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-time object oriented modeling and design*. J. Wiley, 1994.

[Ver] IEEE Standard 1364-1995. Verilog Hardware Description Language Reference Manual, 1995.

[Vhdl] IEEE Standard 1076-1993. VHDL Language Reference Manual, 1993.