Efficient Reachability Analysis of Hierarchical Reactive Machines

R. Alur, R. Grosu, and M. McDougall

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

Abstract. Hierarchical state machines is a popular visual formalism for software specifications. To apply automated analysis to such specifications, the traditional approach is to compile them to existing model checkers. Aimed at exploiting the modular structure more effectively, our approach is to develop algorithms that work directly on the hierarchical structure. First, we report on an implementation of a visual hierarchical language with modular features such as nested modes, variable scoping, mode reuse, exceptions, group transitions, and history. Then, we identify a variety of heuristics to exploit these modular features during reachability analysis. We report on an enumerative as well as a symbolic checker, and case studies.

1 Introduction

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 design. Modern software engineering methodologies such as UML [BJR97] exhibit two kinds of modular structures, *architectural* and *behavioral.* Architectural modularity means that a system is composed of subsystems using the operations of parallel composition and hiding of variables. The input languages of standard model checkers (e.g., S/R in Cospan [AKS83] or Reactive modules in Mocha [AH99]) support architectural modularity, but provide no support for modular description of the behaviors of individual components. In this paper, we focus on exploiting the *behavioral hierarchy* for efficient model checking.

The notion of behavioral hierarchy was popularized by the introduction of STATECHARTS [Har87], and exists in many related modeling formalisms such as

Modecharts [JM87] and Rsml [LHHR94]. It is a central component of various object-oriented software development methodologies developed in recent years, such as Room [SGW94], and the Unified Modeling Language (Uml [BJR97]). Such hierarchic specifications have many powerful primitives such as exceptions, group transitions, and history, which facilitate modular descriptions of complex behavior. The conventional approach to analyze such specifications is to compile them into input languages of existing model checkers. For instance, Chan et al [CAB+98] have analyzed Rsml specifications using Smv, and Leue et al have developed a hierarchical and visual front-end to Spin. While the structure of the source language is exploited to some extent (e.g., [CAB+98] reports heuristics for variable ordering based on hierarchical structure, and [BLA+99] reports a way of avoiding repeated search in same context), compilation into a *flat* target language loses the input structure. In terms of theoretical results concerning the analysis of such descriptions, verifying linear properties of sequential hierarchical machines can be done efficiently without flattening [AY98], but in presence of concurrency, hierarchy causes an exponential blow-up [AKY99].

The input language to our model checker is based on *hierarchic reactive modules* [AG00]. This choice was motivated by the fact that, unlike Statecharts and other languages, in hierarchic reactive modules, the notion of hierarchy is *semantic* with an observational trace-based semantics and a notion of refinement with assume-guarantee rules. Furthermore, hierarchic reactive modules support *extended* state machines where the communication is via shared variables. The first contribution of this paper is a concrete implementation of hierarchic reactive modules. Our implementation is *visual* consistent with modern software design tools, and is in Java. The central component of the description 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 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 default exit to entry models a group transition applicable to all control points inside. 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. Our language distinguishes between a mode *definition* and a mode *reference*, and this allows sharing and reuse.

Our model checker checks invariants by reachability analysis of the input model. The model is parsed into an internal representation that directly reflects the hierarchical structure, and the analysis algorithms, symbolic and enumerative, attempt to exploit it in different ways:

Transition indexing. The transition relation is maintained indexed by the modes and their control points. In the enumerative setting, this is beneficial for quick access to potentially enabled transitions, and also due to shared mode definitions. In the symbolic setting, this provides a generalization of the traditional *conjunctively partitioned* representation [McM93].

State-space representation. In the enumerative setting, states are represented as *stacks* of vectors rather than vectors, and this is useful in handling priorities of transitions. In symbolic search, we maintain the state-space as a forest of binary decision diagrams indexed by control points. The resulting search has, consequently, a mixture of enumerative and symbolic strategies.

Typing. Each mode explicitly declares the variables that it reads and writes, thus, providing different types for different transitions. This information is used in symbolic search for heuristics such as early quantification.

Variable scoping. The pool of variables is not global. For instance, the state can consist of variables x and y in one mode, and x and z in another. This information, available statically, is exploited by both the searches. This optimization is possible due to the encapsulation provided by our language.

Note that the above heuristics are quite natural to the hierarchical representation, and has advantages with respect to the flat representation of the same model. Another advantage of the language is that the granularity of steps of interacting components can be controlled as desired. This is because a *macro-step* of a mode corresponds to executing its *micro-steps* repeatedly until there are no more enabled transitions, and parallel composition corresponds to interleaving macro-steps.

We have implemented an enumerative checker based on depth-first search, and a symbolic search that uses BDD packages from VIS [BHSV$^+$96]. We report on two case studies. As a first example, we modeled the `tcp` protocol. Our visual interface allowed a direct mapping of the block-diagram description from [PD96], and our enumerative checker found a deadlock bug in that description. Second, we constructed an example that is illustrative of nesting and sharing of modes, and scoping of variables. The performance of both enumerative and symbolic checkers is significantly superior compared to the respective traditional checks.

2 Modeling Language

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.

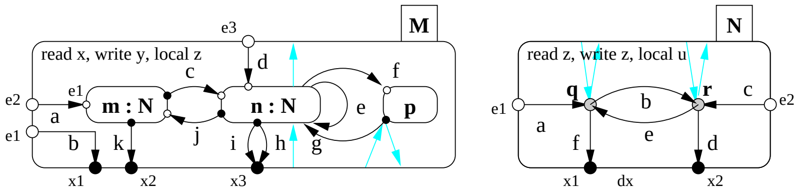


Fig. 1. Mode diagrams

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`.

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.

Semantics 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.

Parallel composition by interleaving A mode having only two points, the default entry and the default exit, is called a mode in *top level form*. These modes can be used to explicitly model all the parallel composition operators found in the theory of reactive systems [AG00]. For simplicity we consider in this paper only the *interleaving semantics* of parallel composition. In this semantics, a round (macro step) of a composed mode is a round of one of its submodes. The choice between the submodes is arbitrary. We can easily model this composition, by overriding the default entry transions of the submodes if these are in top level form. Note also that, the state of the supermode is given, as expected, as a tuple of the states of the submodes.

3 Search Algorithms

3.1 Enumerative Search

The enumerative search algorithm takes as input a set of top-level modes and a set of global variables that these modes can read and modify. We are also given an invariant that we want this system to satisfy. The invariant is a boolean

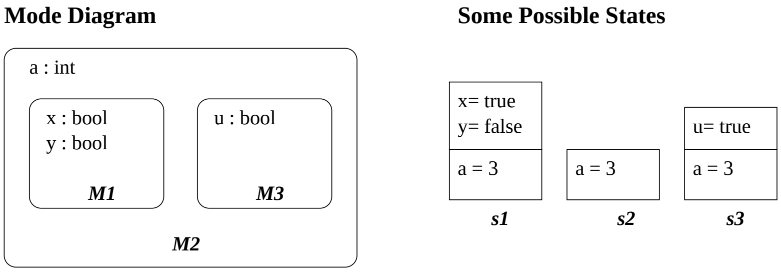


Fig. 2. Local variables conserve state size

expression defined on the global variables. For the enumerative search we assume that each of the top-level modes is sequential; each top-level mode represents a single thread of execution. These top-level modes are run concurrently by interleaving their macro-steps. A state of the system consists of the values of all the global variables and the state of each of the top-level modes. In each round one of the modes may modify the variables and change its own internal state yielding a new state. The set of states of the system can therefore be viewed as a directed graph where, if s and t are states of the system, (s, t) is an edge in the graph if and only if s yields t after one round of execution.

Searching all states is straightforward; beginning from an initial state we perform a depth-first search on the graph. For each state we encounter during the search we check that the desired invariant holds. If the invariant doesn't hold for some state then the depth-first search algorithm supplies us with a path in the graph from the initial state to the state which violates the invariant. This path forms a counter-example which is returned to the user.

The set of states that have been visited is stored in a hash table so that we can check if a given state is in the set in constant time. The set of successors of a state is computed by examining the modes. The hierarchical structure of the modes is retained throughout the search. This structure allows us to optimize the search in a number of ways.

Transition Indexing To determine if there is an edge from a state s to a state t we need to examine all the possible sequences of micro-steps that are enabled in s. Each micro-step corresponds to a transition from the active control point to a destination point (which becomes the active point in the next micro-step). In the mode representation transitions are indexed by their starting points. If we want to find all enabled transitions we only need to examine those that start from the active point.

Local Variables Modes have local variables which are only visible to submodes. The internal state of a mode can be stored as a stack of sets of variables. This stack resembles the control stack present during the execution of a program in C or Java. Each element of the stack contains the variables that are local to the corresponding level in the mode hierarchy.

Since local variables of a mode are available only to the submodes of that mode the size of a mode's state is smaller than a system where all variables are global. Figure 2 shows a simple mode diagram and three possible states of the mode. Modes $M1$ and $M3$ are submodes of $M2$. States $s1, s2$ and $s3$ are the respective possible states when $M1, M2$ and $M3$ are active. If $M1$ is active the x, y and a variables are in scope and therefore must be present in the state $s1$, The variable u is not in scope if $M1$ is active and therefore it is not present in state $s1$. If $M2$ is active then x, y and u are not scope and not present in $s2$. Similarly, $s3$ does not contain the x and y variables. This means the total size of the state of a mode is proportional to the depth of the hierarchy.

State Sharing The stack structure of a mode's state also allows us to conserve memory by sharing parts of the state. Two states which are distinct may nevertheless contain some stack elements which are identical. We can construct the states in such a way that the equivalent elements of the stack are actually the same piece of memory.

For example, states s and t have two levels of hierarchy corresponding to local and global variables. If all the global variables have the same values in s and t then both s and t can refer to the same piece of memory for storing the values of the global variables.

State Hashing The tool gives the user the option of storing a hash of a state instead of the entire state. Since the hashed version of a state occupies less memory than the state itself we can search more states before we run out of memory. The problem with this technique is that the enumerative search will skip states whose hashes happen to be the same as previously visited states. As a result the enumerative search may falsely conclude that an invariant is true for all states. On the other hand, any counter-example that is found *is* valid. By varying the amount of information lost through hashing we can balance the need for accuracy with the need to search large state spaces. The SPIN model-checker uses this hashing technique [Hol91] for states which have a fixed number of variables. However, our tool must hash states which vary in size and structure (because of the local variables).

3.2 Symbolic Search

Similarly to the enumerative search algorithm, the symbolic search algorithm takes as input a set of top-level modes and a set of variables that these modes can read and modify and an invariant that we want this system to satisfy. However, in contrast to the enumerative search we do not need to assume that the top-level modes are sequential. The reason is that a state in this case is not a stack, but rather a map (or context) of variables to their values. This context varies *dynamically*, depending on the currently accessible variables.

In order to perform the symbolic search for a hierarchic mode we could proceed as follows. (1) Obtain a flat transition relation associated to the hierarchic mode. (2) Represent the reached states and the transition relation by ordered multi-valued binary decision diagrams (mdds). (3) Apply the classic symbolic search algorithm. This is the current model checking technology.

We argue however, that such a flat representation is not desirable. The main reason is its inefficient use of memory. One can do much better by keeping the above mdds in a decomposed way, as suggested by the modular structure. In particular, a natural decomposition is obtained by keeping and manipulating the control points outside the mdds.

Reached set representation. Keeping the control points outside the state allows us to *partition* the state space in regions, each containing all states with the same control point. This decomposition has not only the advantage that any partition may be considerably smaller than the entire set but also that it is very intuitive. It is the way mode diagrams, and in general extended state machines, are drawn. Hence, we represent the *reached state space* by a mapping of the currently reached control points to their associated reached region mdd. The *region mdd* of a control point is minimized by considering only the variables visible at that control point. This takes advantage of the natural scoping of variables in a hierarchic mode.

Update relation representation. The update relation of a hierarchic mode d is not flattened. It is kept in d by annotating each transition of d with the mdd corresponding to the transition. This has the following advantages. First, all instances of d at the same level of the hierarchy and connected only at their regular points may share these transitions. The reason is that their local variables are never simultaneously active. Second, working with scoped transition relations and knowing the set of variables U updated by a transition t (broadly speaking working with typed transition relations) we may compute the image $image(R, t)$ of a region R in an optimal way as $(\exists U.R \wedge t)[U/U']$ and not as $(\exists V.R \wedge t \wedge id_{V \setminus U}))[V/V']$ where V is the set of all variables and $id_{V \setminus U}$ is the conjunction of all relations $x' = x$ with $x \in V \setminus U$. The second, more inefficient representation is to our knowledge the way the image is computed in all current model checkers. Moreover, while the internal and default transitions have this optimized form, the entry and exit transitions allow even further optimization via quantification of variables that are no longer needed.

- **Entry transitions.** $image(R, t) = (\exists(U \setminus V_l).R \wedge t)[U/U']$ because R and t is not allowed to reference the unprimed local variables in V_l.
- **Exit transitions.** $image(R, t) = (\exists(U \cup V_l).R \wedge t)[U/U']$ because t is not allowed to reference the primed local variables in V_l' and the unprimed local variables in V_l are hidden.

Variable ordering The variable ordering is naturally suggested by the partial ordering between modes. Considering that variable names are made disjoint by prefixing them with the names of the submode instances along the path to the referencing submode instance, then the ordering is nothing but the lexicographic ordering of the prefixed names. This ordering makes sure that variables defined at the same level in the mode hierarchy are grouped together.

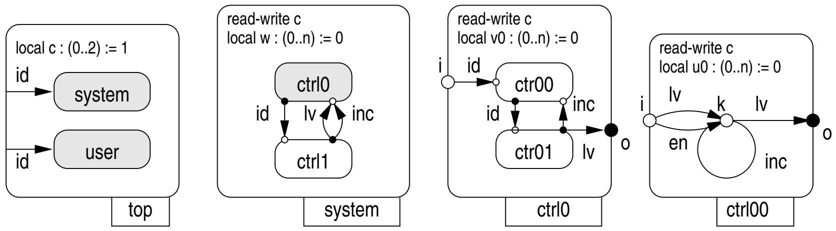


Fig. 3. A generic hierarchic example

Initialization. The initial state is a mapping of the history variables to a special, bottom value. Passing this state along the default entry point of the top-level mode, all the way down in the mode hierarchy, assures the selection of an initialization transition that updates the local variables according to their initialization statement in the mode diagram. The initial reached set maps the default entry point of the top level mode to this state.

Image computation. The main loop of the image computation algorithm is as usual. It starts with the initial *macro onion ring* (the initial reached set) and computes in each iteration (macro-step) a new macro onion ring by applying the image computation to the current macro onion and (the update relation of) the top level mode. The algorithm terminates either if the new macro onion ring is empty or if its intersection with the target region (containing the "bad" states) is nonempty.

The image computation of the next macro onion ring is the secondary loop. It starts with a *micro onion ring* having only one control point: the pair consisting of the default entry point of the top level mode and the macro onion ring mdd. Each micro step computes a new onion ring by applying the image computation to all points in the current micro onion ring and for each point to all outgoing transitions in a breadth first way. A destination point is added to the new micro onion ring if the difference between the computed mdd and the mdd associated to that point in the reached set is not empty. The mdd in the reached set is updated accordingly. The loop terminates when the new micro onion ring contains again only one control point, the default exit point of the top level mode and its associated mdd. The new macro onion ring is then the difference between the set of states corresponding to this mdd and the reached set associated to the top level default entry point.

Generic hierarchic system. In Figure 3 we show a generic hierarchic system with three levels of nesting. The mode `user` nondeterministically sets a control variable `c` to 1 or 2 meaning `increment` and respectively `leave`. The mode `system` consists of the nested modes $\mathtt{ctrl}_0$ and $\mathtt{ctrl}_1$, that are further decomposed in the modes $\mathtt{ctrl}_{00}$, $\mathtt{ctrl}_{01}$ and $\mathtt{ctrl}_{10}$, $\mathtt{ctrl}_{11}$, respectively. Each has a local variable ranging between zero and a maximum value n that is incremented when `c` is 1 and the value of the local variable is less than n. When `c` is 2 or when `c` is one and the local variable reached n the mode is left. To ensure only one increment per macro-step, after performing a transition the mode sets `c` to 0. This blocks it until the user sets the next value. The identity transition `id` is the same on all levels and it is defined as follows:

```
id ≙  true -> skip
```

The transitions `lv` and `inc` of the mode `system` are defined as below:

```
lv  ≙  c = 2 | (c = 1 & w = n) -> c := 0; w := 0;
inc ≙  c = 1 & w < n           -> c := 0; w := w + 1;
```

Except the local variable to be tested and incremented, the transition `inc` has the same definition in all modes. The exit transition `lv` has a simpler body in the submodes. For example in mode $\texttt{ctrl}_{00}$:

```
lv ≙  c = 2 | (c = 1 & u0 = n) -> skip
```

Finally, the entry transitions `en` and `lv` of the leaf modes have the same definition modulo the local variable. For example, in mode $\texttt{ctrl}_{00}$:

```
lv ≙  c = 2  -> skip
en ≙  c != 2 -> c := 0; u0 := 0;
```

The mdd associated to the point `k` of the mode $\texttt{ctrl}_{00}$ is a boolean relation $R_{ctrl_{00}.k}(c, w, v_0, u_0)$. Similarly, the mdds associated to the exit point `o` of the modes $\texttt{ctrl}_{00}$ and $\texttt{ctrl}_{0}$ are boolean relations $R_{ctrl_{00}.o}(c, w, v_0)$ and $R_{ctrl_{0}.o}(c, w)$ respectively. Note that variables are not quantified out at the default exit points because we have to remember their value. Additionally, at these points we have to save the active submode information. Hence, the mdds associated to the default exit points dx of the modes $\texttt{ctrl}_{00}$ and $\texttt{ctrl}_{0}$ are boolean relations $R_{ctrl_{00}.dx}(c, w, v_0, u_0)$ and $R_{ctrl_{0}.dx}(c, w, v_0, u_0, h_0)$ respectively. The history variable h_0 is 0 if the active submode is $\texttt{ctrl}_{00}$ and 1 if the active submode is $\texttt{ctrl}_{01}$.

The mdd associated to a transition is a relation constructed, as usual, by considering primed variables for the next state values. For example, the transition `inc` of the mode $\texttt{ctrl}_{00}$ is defined by the relation

$$inc(c, c', u_0, u_0') \mathrel{\hat{=}} c = 1 \wedge u_0 < n \wedge c' = 0 \wedge u_0' = u_0{+}1$$

The image of the region $R_{ctrl_{00}.k}$ under the transition inc is computed as below:

$$(\exists c, u_0.\ R_{ctrl_{00}.k}(c, w, v_0, u_0) \wedge inc(c, c', u_0, u_0'))\ [c, u_0 / c', u_0']$$

Lacking typing information, most model checkers use the more complex relation:

$$\begin{aligned}(\exists c, w, v_0, u_0.\ & R_{ctrl_{00}.k}(c, w, v_0, u_0) \wedge inc(c, c', u_0, u_0') \wedge \\ & w' = w \wedge v_0' = v_0)\ [c, w, v_0, u_0 / c', w', v_0', u_0']\end{aligned}$$

The exit transition `lv` of the mode $\texttt{ctrl}_{00}$ is defined by the following relation:

$$lv(c, u_0) \mathrel{\hat{=}} c = 2 \vee (c = 1 \wedge u_0 = n)$$

The image of the region $R_{ctrl_{00}.k}$ under this transition is computed as below:

$$\exists u_0.\ R_{ctrl_{00}.k}(c, w, v_0, u_0) \wedge lv(c, u_0)$$

It quantifies out the local variable u_0. It is here where we obtain considerable savings compared to classic model checkers. The image of the region $R_{ctrl_{00}.i}$ under the entry transition:

$$en(c, c', u_0') \hat{=}\ \ c \neq 2 \wedge c' = 0 \wedge u_0' = 0$$

does not quantify out the local variable u_0 even if u_0 was updated, because the transition is not allowed to used the unprimed value of u_0.

$$(\exists c.\ R_{ctrl_{00}.i}(c, w, v_0) \wedge lv(c, c', u_0'))\ [c, u_0/c', u_0']$$

The ordering of the unprimed variables in the generic hierarchic system is defined as follows: $c < w < v_0 < u_0 < u_1 < v_1 < u_2 < u_3$.

4 Experimental Results

Mutual Exclusion Our smallest non-trivial experiment involved Peterson's algorithm for two party mutual exclusion [Pet81]. We implemented the algorithm using three modes that run concurrently. Modes p_1 and p_2 represented the two parties that want to use the shared resource. A special mode called *clock* was used to toggle a variable *tick* that the other modes consume whenever a step of the algorithm is executed. This use of a *tick* variable ensures that each macro-step of a mode corresponds to exactly one step of the algorithm that we are modeling. Once the *tick* variable is consumed a mode is blocked until the next macro-step. This technique allows a programmer to control the number of micro-steps that occur within one macro-step. Our tool performed an enumerative search of all possible executions. The search revealed that the model has 276 distinct states, all of which preserve mutual exclusion. The tool also verified that the algorithm is free of deadlock by checking that each state of the model leads to at least one successor state. Both searches took about 4 seconds to complete on an Intel Celeron 333 mhz.

TCP The Transmission Control Protocol (TCP) is a popular network protocol used to ensure reliable transmission of data. TCP connections are created when a client opens a connection with a server. Once a connection is opened the client and server exchange data until one party decides to close the connection. When a connection is opened or closed the client and server exchange special messages and enter a series of states.

TCP is designed to work even if some messages get lost or duplicated. It is also designed to work if both parties simultaneously decide to close a connection. A desirable property of protocol like TCP is that it cannot lead to deadlock; it should be impossible for both client and server to be waiting for the other to send a message.

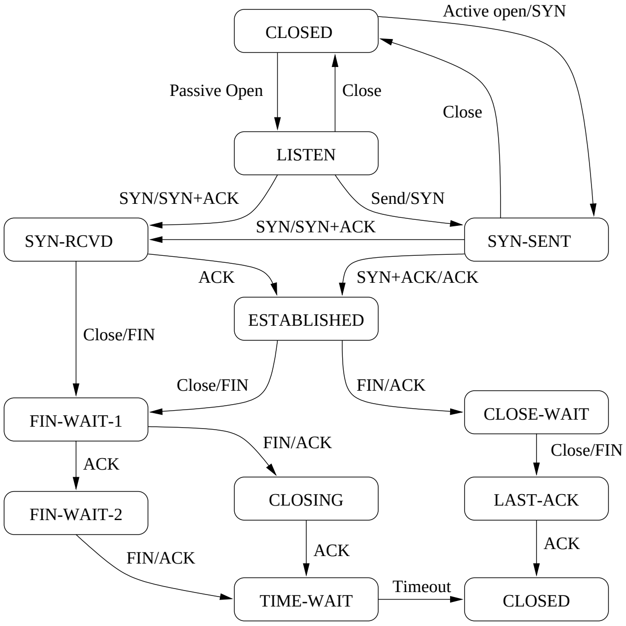


Fig. 4. TCP state-transition diagram

There is a concise description of the messages and states of TCP in [PD96] which is reproduced in Figure 4. This description is given as a state-transition diagram which makes it very easy to model it as a mode in our tool. We set out to verify that the TCP protocol, as described in [PD96], is free of deadlock under certain assumptions.

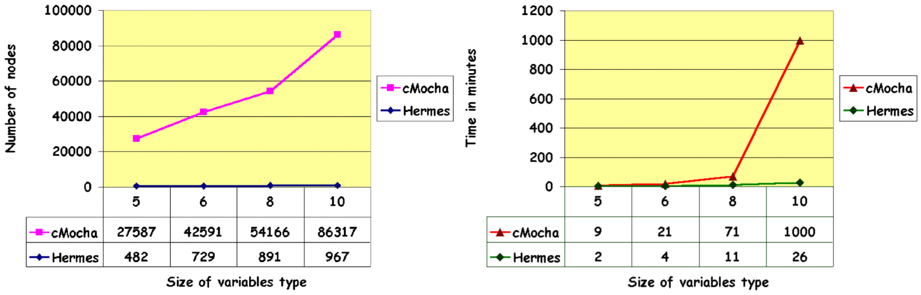
In our first experiment we simulated a client and server opening and closing a TCP connection. In our model we assumed that the network never lost or duplicated a message. We also assumed the client and server both had a one cell queue for storing incoming messages; if a party received a second message before it had a chance to process the first message then the first message would be lost. Our tool performed an enumerative search of possible execution sequences and discovered a bug in the description of TCP after searching 2277 states. If both parties decide simultaneously to close the connection while in the *established* state then they will both send *FIN* messages. One of the parties, say the client, will receive the *FIN* message and respond by sending an ACK message and entering the *closing* state. If this ACK arrives at the server before the other *FIN* is processed the second *FIN* will get lost. When the ACK is read the server will move to the *fin-wait-2* state. Now the protocol is deadlocked; the client is waiting for an ACK message while the server is waiting for a *FIN* message.

For our second experiment we modified the model so that the client and server had queues that could hold two messages instead of one. A message would only get lost if a third message arrived before the first was processed. Our tool found a deadlock state in this model after searching 3535 states. Once again the deadlock occurs after both client and server decide simultaneously to close the connection. In this case, however, the server decides to close the connection *before* it has been established. This can lead to a state where the server's queue gets filled and a message gets dropped. The deadlock occurs when the server is in the *fin-wait-2* state and the client is in the *closing* state, which is the same deadlock state that we saw in the first TCP experiment.

Both experiments were performed using an Intel Celeron 333mhz machine. The first experiment ran for 74 seconds and the second experiment ran for 138 seconds.

Generic hierarchic system The hierarchical structure of the example in Figure 3 makes it a good candidate for the state sharing optimization described in Section 3.1. The three levels of local variables allow memory to be re-used when storing states; if some levels are identical in two distinct states the enumerative search algorithm makes an effort to share the memory used to store the levels that both states have in common.

Each state contains a set of objects called *environments*. Each environment keeps track of one level of the hierarchy. An enumerative search of this example found 3049 distinct states in 138 seconds. These states contained 14879 environment objects, but because of state sharing only 8103 environment objects needed to be allocated. The technique yields a 45% reduction in the number of objects needed to store the set of visited states.



	5	6	8	10
cMocha	27587	42591	54166	86317
Hermes	482	729	891	967

	5	6	8	10
cMocha	9	21	71	1000
Hermes	2	4	11	26

The symbolic search takes advantage of the existential quantification of local variables at all regular exit points. This leads, as shown in Figure 4, to a significant saving both in space (total number of nodes in the mdd pool) and time (for the reachability check) compared to the (C version) of the Mocha model checker [AHM+98]. The comparison was done for different values for n.

Since the concurrency is in this example only on the top level and all the variables (excepting `c`) are local, one can heavily share the transition relations. In fact, only the modes `ctr0` and `ctr00` are necessary and all references may point to these modes. As expected, using this sharing, we obtained the same results with respect to the reached set.

5 Conclusions

We have reported on an implementation of a visual hierarchical language for modeling reactive systems, and enumerative and symbolic checkers that work directly on the hierarchical representation attempting to exploit the modularity. While hierarchical specifications seem more convenient to express complex requirements, in terms of the efficiency of analysis, two questions are of interest. First, given a verification problem, should one use a hierarchical notation hoping for tractable analysis? We don't have adequate experimental evidence yet to answer this. In fact, given that the modeling languages of different tools differ so much, and different tools implement many different heuristics, parameters of a scientific comparison are unclear. Second, if the input specification is hierarchical, should one use the proposed solution over compilation into a non-hierarchical checker? Even though experimental data is small so far, we believe that there is adequate conceptual evidence suggesting a positive answer. A lot of work remains to be done to optimize the two checkers, and apply them to more and substantial examples.

Acknowledgments This work was partially supported by NSF CAREER award CCR97-34115, by DARPA/NASA grant NAG2-1214, by SRC contract 99-TJ-688, by Bell Laboratories, Lucent Technologies, by Alfred P. Sloan Faculty Fellowship, and by DARPA/ITO MARS program. The implementation of our enumerative and symbolic model checker uses components from Mocha, and the help of all members of the Mocha team is greatfully acknowledged.

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.

[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.