Modular Refinement of Hierarchic Reactive Machines

Rajeev Alur Radu Grosu

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

Abstract

Scalable formal analysis of reactive programs demands integration of modular reasoning techniques with existing analysis tools. Principles such as abstraction, compositional refinement, and assume-guarantee reasoning are well understood for architectural hierarchy that describes the communication structure between component processes, and have been shown to be useful. In this paper, we develop the theory of modular reasoning for *behavior hierarchy* that describes control structure using hierarchic modes. From STATECHARTS to UML, behavior hierarchy has been an integral component of many software design languages, but only syntactically. We present the *hierarchic reactive modules* language that retains powerful features such as nested modes, mode reuse, exceptions, group transitions, history, and conjunctive modes, and yet has a *semantic* notion of mode hierarchy. We present an observational trace semantics for modes that provides the basis for mode refinement. We show the refinement to be compositional with respect to the mode constructors, and develop an assume-guarantee reasoning principle.

1 Introduction

The complexity and subtlety of programming reactive systems, such as telecommunications and avionics software, demands increased design automation and effective debugging tools. 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]. 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. Consequently, application of formal verification requires augmenting model checking with *modular reasoning* that allows decomposition of the analysis problem into smaller subproblems, or abstracting a component into a simpler one. Typically, such simplification is done manually, and requires considerable expertise. Much of today's research in formal verification aims to develop techniques to automate such modular reasoning.

To be able to exploit the design structure effectively for modular reasoning, the modeling language must support syntactically as well as semantically modular constructs. While modern programming languages offer a rich set of modular constructs (e.g. procedures, objects), the standard model checkers assume the model to be a state-transition graph (or a Kripke structure) with no structure. Our first attempt to enrich the modeling language resulted in the definition of *reactive modules* [AH99]. In this language, an atomic module is a state-machine whose variables are explicitly partitioned into input, output, and private variables. The operations of parallel composition, instantiation, and variable hiding allow building complex modules from atomic ones. The denotational semantics of each module consists of its input and output variables, together with the set of its traces, which captures the observable interaction of a module with its environment. The notion of *refinement* between two modules is based on inclusion of traces, and provides the basis for abstraction. The refinement relation is compositional with respect to the module operations. Thus, to show that the composite module $P_1\|P_2$ refines the module $Q_1\|Q_2$, it suffices to establish that P_1 refines Q_1 and P_2 refines Q_2. While the compositional proof rule decomposes the verification task of proving implementation between compound modules into subtasks, it may not always be applicable. In particular, P_1 may not implement Q_1 for all environments, but only if the environment behaves like P_2, and vice versa. For such cases, we must employ the *assume-guarantee proof rule* which asserts that in order to prove that $P_1\|P_2$ implements $Q_1\|Q_2$, it suffices to prove (1) $P_1\|Q_2$ implements Q_1, and (2) $Q_1\|P_2$ implements Q_2. The language of reactive modules, along with the assume-guarantee refinement checker, is supported by the model checker MOCHA [AHM+98], and the utility of the assume-guarantee reasoning has been demonstrated in analysis of a video-graphics image processor [HQR98]. The notion of compositional refinement based on observable behaviors is central to many concurrency formalisms such as CCS [Mil80], I/O automata [LT87], TLA [Lam91], but the circular assume-guarantee reasoning [Sta85, GL94, AL95, AH99, McM97] is valid only when the interaction of a module with its environment is non-blocking.

While the reactive modules language supports *architec-*

tural hierarchy, it offers little structure to express the behavior of individual modules. In this paper, we present the language of *hierarchic reactive modules* that supports both *architectural and behavioral hierarchy*, along with its compositional semantics and assume-guarantee proof calculus. The notion of behavior 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 lead to complex semantics [HPSS87, PS91, HN96, GSB98]. There have been several attempts to define a rigorous semantics of STATECHARTS alone. Typically the semantics is defined operationally by considering the global state of the system. In fact, multiple papers offer a *compositional* semantics that is congruent with the language constructs [US94]. However, the notion of *semantic equivalence* used in all these papers is structural isomorphism of underlying state-transition graphs (or bisimilarity but with most of the structure visible), rather than conventional observational equivalences. Also there have been attempts to exploit the hierarchic states for efficient reachability analysis [CAB+98, AY98, BLA+99]. However, there is no *observational* semantics that allows defining a refinement preorder on hierarchic states. Thus, we believe that the notion of *hierarchy* in behavior descriptions has been only syntactic so far, and our main contribution is to make it *semantic* with a supporting *refinement calculus*.

The central component of the behavioral description in our language is a *mode*. A mode consists of 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. Since the control can be simultaneously in multiple nested modes, the order in which the transitions are tried for execution is important. Unlike STATECHARTS, but as in ROOM, the operational semantics tries the transitions inside out, that is, we give priority to the internal transitions over the group transitions of the enclosing mode. This choice is also crucial for the clean denotational semantics. Our language allows mode instantiation and thus, reuse.

Our denotational semantics of a mode consists of its global variables, entry/exit points, and traces over global variables that capture a mode's behavior. The key step leading to such semantics involves a *closure* construction that adds transitions connecting the default points. This construction makes the transfer of control between a mode and its environment explicit. Consequently, the behavior of a mode can be viewed as a game in which the environment transfers control to the mode at one of its entry points, and the mode transfers the control back to the environment at one of its exit points. The *macro-transition* from an entry point to an exit point, thus, consists of multiple transitions, and can be constructed from the macro-transitions of the submodes together with the transition relation of the mode. The macro-transitions are then used to associate a set of executions and a corresponding set of traces with a mode. We show that the traces of a mode can be constructed from the traces of its submodes.

The denotational trace semantics naturally leads to a notion of refinement among modes based on inclusion of traces, and provides the basis for mode abstraction and substitution. We show that the constructors are compositional with respect to this refinement relation, and this leads to compositional proof rules for refinement. In particular, to establish that a mode M with a submode N refines a mode M' with submode N', it suffices to prove that (1) mode N refines N', and (2) mode M with N substituted by a "free" mode that allows most general update, refines mode M' with N' made free. Thus, compositional rule allows us to decouple the reasoning about a submode from the reasoning about its context. We also present a circular assume-guarantee proof rule in which the specification context M' can be assumed while establishing the first sub-goal, and the specification submode N' can be used while establishing the second sub-goal. For the validity of the rule, the refinement relation needs to be strengthened to make the points of transfer of control between a mode and the relevant submode visible.

Hierarchic languages such as STATECHARTS allow the notion of *conjunctive (parallel) states*. We argue that conjunction can be defined cleanly as a constructor over submodes such that a step of the constructed mode consists of a sequence of micro steps, one micro step for each submode. We establish that such a conjunctive constructor is compositional with respect to refinement and that its trace semantics is essentially the same as the trace semantics of the parallel composition constructor over modules. This suggests a schme for mixing modes and modules interchangeably in system descriptions.

The rest of the paper is organized as follows. In Section 2 we review reactive modules: their syntax, semantics and refinement rules. In Section 3 we follow a similar pattern for modes. We first introduce their syntax and then we define their semantics and refinement rules. Section 4 is devoted to conjunctive modes. First we define conjunction as a particular mode constructor. Then we show that conjunction is compositional with respect to refinement and relate it to module composition. Finally in Section 5 we draw some conclusions. During the entire paper, we use the specification of a small village telephone system, inspired from [BGG+98], as a working example to illustrate definitions.

Notation. Given a set V of typed variables, a *state* over V is a function mapping variables to their values. The set of states over V is denoted Q_V. Given a state s over V and a subset W of V, $s[W]$ denotes the state over W obtained by restricting s to the variables in W. The projection operator extends to sequences of states also. Given two sets V and W of variables, an *action* from V to W is a binary relation between the states over V and the states over W. An action α from V to W is said to be *enabled* at a state s over V if $(s,t) \in \alpha$ for some state t. An action α from V to W is said to be *non-blocking* if it is enabled at every state over V. The domain and range of an action can be expanded

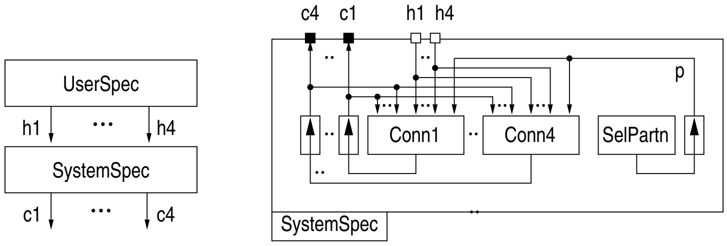

Figure 1: Architecture diagrams for the VTS

implicitly in a natural way: if α is an action from V to W, s is a state over $V' \supseteq V$ and t is a state over $W' \supseteq W$, then define $(s,t) \in \alpha$ if $(s[V], t[W]) \in \alpha$ and $t[v] = s[v]$ for $v \in (V' \cap W') \setminus W$. This implicit coercion is quite common and allows one to assume that the variables not explicitly occuring in a transition remain unchanged.

2 Modules

A module is defined by the set of its variables, rules for initializing the variables, and rules for updating the variables.

Variables. The variables of a module P are partitioned into three classes: *private* variables that cannot be read or written by other modules, *interface* variables that are written only by P, but can be read by other modules, and *external* variables that can only be read by P, and are written by other modules. Thus, interface and external variables are used for communication, and are called *observable* variables. The private and interface variables are written by the module, and are called *controlled* variables.

Initialization and update. The initialization specifies initial controlled states. The transition relation specifies how to change the controlled state as a function of the current state.

Definition 1 (Modules) *A* module P *consists of*

Variables. *A finite set V of typed variables that is partitioned into* private variables V_p, interface variables V_i *and* external variables V_e. *The variables in $V_c = V_p \cup V_i$ are called* controlled variables, *and the variables in $V_o = V_i \cup V_e$ are called* observable variables.

Initial states. *A non-empty subset I of states over V_c.*

Update relation. *A non-blocking action U from V to V_c.*

□

Example 1 (Village telephone system) *Consider a simple village telephone system that is able to establish a point-to-point connection between any two telephone lines that are disconnected and off hook. For simplicity, assume that the system has only four lines. The block (architecture) diagram for the system and and the users is shown in Figure 1, left.*

To define and analyze the behavior of the village telephone system, we associate each block in the diagram to a reactive module with the interface shown in the diagram. A high level definition of the environment is the `UserSpec` *module given below.*

```
type hookType is {on, off}

lazy module UserSpec is
  interface h1, h2, h3, h4 :  hookType;
  init
    [] true -> h1 := on; h2 := on;
         h3 := on; h4 := on;
  update
    [] h1 = on -> h1 := off;
    [] h1 = off -> h1 := on;
    [] h2 = on -> h2 := off;
    [] h2 = off -> h2 := on;
    [] h3 = on -> h3 := off;
    [] h3 = off -> h3 := on;
    [] h4 = on -> h4 := off;
    [] h4 = off -> h4 := on;
```

In each update round it may toggle one of the four lines between `on` *and* `off`*. The choice is nondeterministic. By making the module* lazy *it is also allowed to idle in a round, i.e., to arbitrarily delay toggling.* □

Semantics. For a module P, we use the notation $P.V_p$ to refer to the private variables of P, $P.U$ to refer to the transitions of P, etc. The semantics of a module is captured by defining its executions and traces:

Definition 2 (Traces) *An* execution *of a module P is a sequence $s_0 s_1 \cdots s_n$ of states over $P.V$ such that $s_0[P.V_c] \in P.I$ and $(s_i, s_{i+1}[P.V_c]) \in P.U$ for $0 \leq i < n$. If σ is an execution of P, then the corresponding sequence $\sigma[V_o]$ of observable states is called a* trace *of P. The set of all traces of P is called the* trace-language *of P, and is denoted L_P.* □

The *denotational semantics* of the module P is captured by its interface variables, external variables, and trace language. The requirement that I is non-empty and U is non-blocking ensures that the module can always take a step, and the trace-set L_P is infinite.

Remark 1 (Awaits dependencies and sub-rounds) *In our definition, the initial values of the controlled variables cannot depend on each other and in each update step, the new values of the controlled variables cannot depend on the new values of the external and controlled variables. Thus, modules are like Moore machines. The definition of reactive modules [AH99] allows specification of a partial order — called* awaits dependencies, *over variables such that if a variable x is greater than a variable y in this ordering, then the initial value of x can refer to the initial value of y and the update rule for x can refer to the updated value of y. This allows more complex forms of interaction between a module and its environment by splitting each update round into a* *fixed number of sub-rounds. In this paper, we have chosen a simpler model with no awaits dependencies for the sake of clarity of presentation.* □

Hierarchy. Operators on modules include instantiation, hiding of interface variables, and parallel composition. Here, we discuss the parallel composition operator that allows to combine two modules into a single one.

Definition 3 (Parallel Composition) *The modules P and Q are* composable *if $P.V_c \cap Q.V_c$ is empty. The* composition *R of composable modules P and Q, denoted $P \| Q$, is the module with:*

Variables. *The sets*

$$R.V_i = P.V_i \cup Q.V_i$$
$$R.V_e = (P.V_e \cup Q.V_e) \setminus R.V_i$$
$$R.V_p = P.V_p \cup Q.V_p$$

of interface, external and private variables.

Initial states. *The set $R.I = P.I \times Q.I$.*

Update relation. *The relation $R.U$ such that:*

$$(s,t) \in R.U \text{ iff } \begin{array}{l} (s[P.V], t[P.V_c]) \in P.U \wedge \\ (s[Q.V], t[Q.V_c]) \in Q.U \end{array}$$

□

Note that the denotational semantics of $P\|Q$ can be completely constructed from the denotational semantics of P and Q. This is because a sequence σ belongs to the trace language $L_{P\|Q}$ iff its corresponding projections belong to the trace languages L_P and L_Q.

Example 2 (Village telephone system) *A possible hierarchic decomposition of the module* `SystemSpec` *is obtained by introducing for each telephone line i a connection module* `Conn`*i. An additional module* `SelPartn` *is used to guide the selection of the communication partner. This defines the architecture shown by the block diagram in Figure 1, right. The blocks marked with a thick arrow represent registers. They separate the current state from the next state. The specification of the module* `SelPartn` *is trivial. It nondeterministically chooses one of the selection modes.*

```
type matchingType is
  {"1-2/3-4","1-3/2-4","1-4/2-3"}

module SelPartn is
  interface p : matchingType;
  init update
    true -> p := nondet;
```

The specification of the connection module `Conn1` *is as follows. If the module is disconnected and its line is off hook, then it chooses a partner connection module as specified by the value of* p*, provided this partner module is also disconnected and its line is off hook. In this case it becomes connected. If the current communication partner goes on hook then the module goes in the drooping state. Finally, if its own line goes on hook, then it becomes disconnected.*

```
type connectionType is
  {disconnected, 1, 2, 3, 4, drooping}

module Conn1 is
  interface c1 : connType;
  external c2,c3,c4 : connType;
  external h1,h2,h3,h4 : hookType;
  external p : matchingType;
init
  [] true -> c1 := disconnected;
update
  [] h1 = on -> c1 := disconnected;

  [] c1 = 2 & h2 = on -> c1 := drooping;
  [] c1 = 3 & h3 = on -> c1 := drooping;
  [] c1 = 4 & h4 = on -> c1 := drooping;

  [] c1 = disconnected & h1 = off &
     c2 = disconnected & h2 = off &
     p = "1-2/3-4" -> c1 := 2;
  [] c1 = disconnected & h1 = off &
     c3 = disconnected & h3 = off &
     p = "1-3/2-4" -> c1 := 3;
  [] c1 = disconnected & h1 = off &
     c4 = disconnected & h4 = off &
     p = "1-4/2-3" -> c1 := 4;
```

The other connection modules are specified in a similar way. Composing the above modules gives the specification of the entire system, as shown below.

```
module SystemSpec is
  hide p in ( Conn1 ||..|| Conn4 || SelPartn )
module Spec is UserSpec || SystemSpec
```

□

Refinement. The notion of refinement between successive levels of abstraction is formalized by the definition of the implementation preorder:

Definition 4 (Implementation) *The module P* implements *the module Q, written $P \preceq Q$, if P and Q have identical interface variables, identical external variables, and $L_P \subseteq L_Q$.* □

The above notion of implementation can be generalized in a straightforward way to allow the implementation to have more interface variables than the specification.

A key property of the implementation relation is *compositionality* which ensures that the refinement preorder is congruent with respect to the module operations.

Proposition 1 (Compositionality) *If $P \preceq Q$ then $P\|R \preceq Q\|R$.*

By applying the compositionality rule twice and using the transitivity of refinement it follows that, in order to prove that a complex compound module $P_1\|P_2$ (with a large state space) implements a simpler compound module $Q_1\|Q_2$ (with a small state space), it suffices to prove (1) P_1 implements Q_1 and (2) P_2 implements Q_2. We call this the *compositional proof rule* for reactive modules. It is valid, because parallel composition and implementation behave like language intersection and language containment, respectively.

While the compositional proof rule decomposes the verification task of proving implementation between compound modules into subtasks, it may not always be applicable. In particular, P_1 may not implement Q_1 for all environments, but only if the environment behaves like P_2, and vice versa. For such cases, an assume-guarantee proof rule is needed [Sta85, GL94, AL95, AH99]. The *assume-guarantee proof rule* for reactive modules asserts that in order to prove that

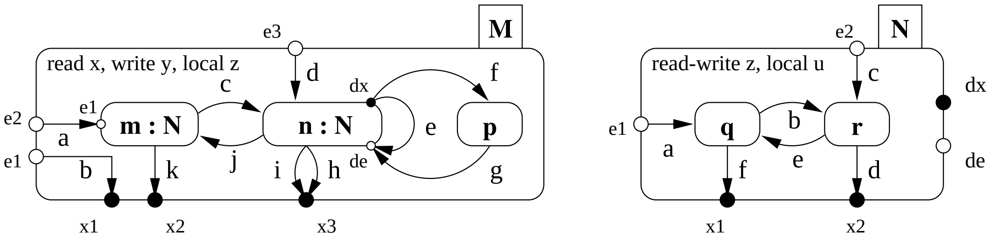

Figure 2: Scoping rules and transition types

$P_1 \| P_2$ implements $Q_1 \| Q_2$, it suffices to prove (1) $P_1 \| Q_2$ implements Q_1, and (2) $Q_1 \| P_2$ implements Q_2. Both proof obligations (1) and (2) typically involve smaller state spaces than the original proof obligation, because the complex compound module $P_1 \| P_2$ usually has the largest state space involved. The assume-guarantee proof rule is circular; unlike the compositional proof rule, it does not simply follow from the fact that parallel composition and implementation behave like language intersection and language containment. Rather the proof of the validity of the assume-guarantee proof rule proceeds by induction on the length of traces. For this, it is crucial that every trace of a module can be extended.

Proposition 2 (Assume-Guarantee) *If $P_1 \| Q_2 \preceq Q_1$ and $Q_1 \| P_2 \preceq Q_2$, then $P_1 \| P_2 \preceq Q_1 \| Q_2$.*

The right hand sides of the refinement relation in the above assume guarantee rule may be strengthen by using a small variation of the rule: if $P_1 \| Q_2 \preceq Q_1 \| Q_2$ and $Q_1 \| P_2 \preceq Q_1 \| Q_2$, then $P_1 \| P_2 \preceq Q_1 \| Q_2$.

3 Modes

3.1 Syntax

Hierarchy. 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 2 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. For example, the submode instances `m` and `n` in Figure 2 share the same mode `N`. Note that a mode resembles an *or* state in STATECHARTS but it has more powerful structuring mechanisms.

Variables. A mode may have *global* as well as *local variables*. The set of global variables V_g is used to share data with the mode's environment. The variables in V_g are classified into a set V_r of *read variables* and a set V_w of *write variables*. Hence, $V_g = V_r \cup V_w$. The set of *local variables* V_l of a mode is accessible only by its transitions and submodes.

Thus, the scoping rules for variables are as in standard structured programming languages. For example, the mode `M` in Figure 2 has the global read variable `x`, the global write variable `y` and the local read-write variable `z`. Similarly, the mode `N` has the global read-write variable `z` and the local read-write 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 2 is shared between the submode instances `m` and `n`. However, the value of the local variable `u` is not shared between `m` and `n`.

Entry/exit points. 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 2 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 2 the transition `a` connects the entry point `e2` of the mode `M` with the entry point `e1` of the submode instance `m`. The entry and exit points of a transition are attributes that a drawing tool may optionally show or hide to avoid cluttering. For example, the endpoints of the transition c are not shown explicitly in Figure 2.

According to the points they connect, the transitions of a mode may be classified into *entry, internal* and *exit* transitions. For example, in Figure 2, `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. Between these transitions there is a subtle difference. Entry transitions initialize the local variables by reading only the global variables. Exit transitions forget the values of the local variables and write only the global variables. The internal transitions only may both read and write the local variables.

Preemption. To model preemption each mode (instance) has a special, default exit point dx. 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 2, the transition `f` is a group transition for the submode `n`.

To achieve the preempting behavior we add for each internal exit point a *default* exit transition (from this point to dx) that is enabled when all other transitions starting in

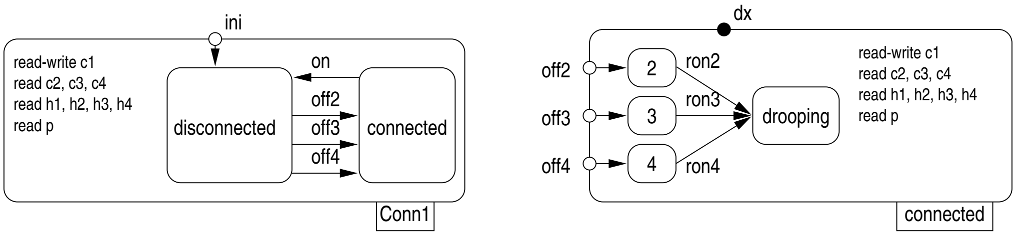

Figure 3: The mode Conn1

this point are disabled. These transitions are not explicitly drawn. They are implicit in the semantics of a mode.

For example, 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* (like the weak `kill` in Unix, versus the strong `kill -9`). This priority scheme facilitates a modular semantics.

History. To allow history retention, we use a special default entry point de. A transition entering the default entry point of a mode restores the values of all local variables along with the position of the control (a transition may enter a default entry of a mode only if the mode was most recently left along its default exit).

For example, both transitions `e` and `g` in Figure 2, 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).

Similarly to preemption, to achieve the above behavior we semantically add *default entry* transitions from the default entry point de of a mode m to its internal points. The default exit transitions save the current point in a local, history variable $m.h$ and the default entry transitions restore the current control point from this variable. A mode enriched with default entry and exit transitions is said to be *closed*.

Remark 2 (Mode instantiation) *A mode can be viewed as an* encapsulation *operator over its submodes, and thus, modes are constructed from leaf-modes using encapsulation repeatedly in a non-recursive manner. Mode instantiation allows reuse and sharing by permitting both to refer to the same mode and to rename a (subset of) entry points, exit points, read variables, and write variables. With mode instantiation, the mode structure is a directed acyclic graph and it can be exploited in an efficient way for model checking [AY98]. To simplify the formal definitions in the following we assume a tree like structure obtained by replacing each instance by its corresponding mode. Moreover, we assume that there are no name conflicts regarding local variables and entry/exit points across modes.* □

Now we are ready to present a formal definition of modes.

Definition 5 (Mode) *A* mode *consists of*

Control points. *A finite set E of entry points, and a finite set X of exit points. We also assume an additional default entry point de, and a default exit point dx, and define $dE = E \cup \{de\}$, and $dX = X \cup \{dx\}$.*

Variables. *A finite set V_r of* read *variables, a finite set V_w of write variables, and a finite set V_l of local variables. The variables $V_g = V_r \cup V_w$ are called global variables. We assume that the sets V_g and V_l are disjoint (but the sets V_r and V_w need not be).*

Submodes. *A finite set SM of* submodes. *If N is a submode in SM, then it is required that $N.V_r \subseteq V_r \cup V_l$ and $N.V_w \subseteq V_w \cup V_l$.*

Transitions. *A finite set T of transitions of the form (e, α, x), where e is in $E \cup SM.X$, x is in $X \cup SM.E$, and α is an action from V_r to $V_r \cup V_l$ if $e \in E$ and from $V_r \cup V_l$ to $V_w \cup V_l$ otherwise. We require that for each $e \in E$, the union $\cup\alpha$ such that $(e, \alpha, x) \in T$ for some x, is non-blocking.*

□

A *leaf mode* is a mode with no submodes and no local variables.

Example 3 (Village telephone system) *By using modes, the specification of the module* `UserSpec` *may be given as shown in Figure 4. The modes* `toggle1,...,toggle4` *are*

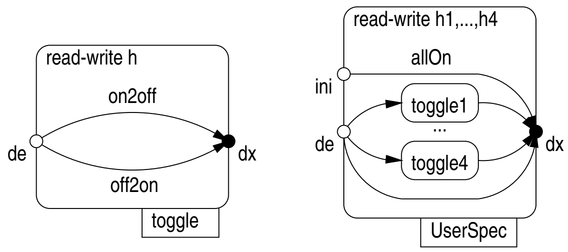

Figure 4: UserSpec for VTS

obtained from the mode `toggle` *by renaming the variable* `h` *with* `h1,...,h4` *respectively. The unmarked transition connecting de to dx is the identity transition expressing idling. The other transitions are defined as follows.*

$$\texttt{on2off} \stackrel{\text{def}}{=} \texttt{h = on -> h := off}$$

$$\texttt{off2on} \stackrel{\text{def}}{=} \texttt{h = off -> h := on}$$

$$\texttt{allOn} \stackrel{\text{def}}{=} \texttt{h1 := on; h2 := on; h3 := on; h4 := on}$$

The connection module `Conn1` *may be restated as a hierarchic mode as shown in Figure 3. Note that the default exit point* `dx` *of the mode* `connected` *is the source point of the transition* `on`. *The transitions are defined as follows.*

```
on    ≝  h1 = on -> c1 := disconnected

off2  ≝  h1 = off &
         h2 = off & c2 = disconnected &
         p = "1-2/3-4" -> c1 := 2
off3  ≝  h1 = off &
         h3 = off & c3 = disconnected &
         p = "1-3/2-4" -> c1 := 3
off4  ≝  h1 = off &
         h4 = off & c4 = disconnected &
         p = "1-4/2-3" -> c1 := 4

ron2  ≝  h2 = on -> c1 := drooping
ron3  ≝  h3 = on -> c1 := drooping
ron4  ≝  h4 = on -> c1 := drooping
```

□

Note that by distinguishing between control and data, mode diagrams are often more comprehensible than module specifications. This may have an important impact if the control structure is quite involved and this is the reason why hierarchic state transition diagrams are so popular in software engineering methods.

We use C to denote the set $dE \cup dX \cup SM.dE \cup SM.dX$ of all control points. Pairs of the form (c, s), where c is a control point and s is a state are called *configurations.* For notational convenience, we view the set T of transitions also as a binary relation over configurations: if $(e, \alpha, x) \in T$ and $(s, t) \in \alpha$, we write $((e, s), (x, t)) \in T$.

3.2 Operational Semantics

The priority among transitions. In Figure 5, 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.

Default exit transitions. In any mode, some transition leaving an entry point is guaranteed to be enabled, so execution can get stuck only at an exit point of a submode. In Figure 5 these points are explicitly drawn as black bullets. To make the transfer of control explicit, we add *default exit transitions* as follows. From an exit point x of a submode of M, we add a transition to the default exit point dx that is enabled if and only if, all the explicit outgoing transitions from x are disabled. If the actions are given by guarded commands, and if $g_1, \ldots, g_n$ are the guards of the explicit transitions, the guard of the default transition is $\neg(g_1 \vee \ldots \vee g_n)$. For example, in Figure 5, the default exit transitions starting in `q` and `r` have the guards $\neg(g_b \vee g_f)$ and $\neg(g_e \vee g_d)$ respectively, where g_b, g_d, g_e, g_f are the guards of the transitions `b,d,e,f`, respectively. Similarly, the default exit transition starting in `n.dx` has the guard $\neg(g_e \vee g_f)$ and the default exit transition starting in `p` has the guard $\neg g_g$. The other default exit transitions are not drawn to avoid cluttering. Each default exit transition saves the local state which is restored upon the subsequent entry to the default entry point. To remember the location of control, we add a new local variable h to a mode M and an action body to each default exit transition (from an exit point x to dx) that *saves* x in this history variable h.

Default entry transitions. The transitions entering the default entry point of a mode M restore the local state. Again, we introduce explicit default entry transitions to restore the location of control. For each default exit transition from an exit x of a submode of M, there is a *default entry transition* from de to x that is taken when the value of the local history variable h coincides with x. If x was a default exit point $n.dx$ of a submode n then, as shown in Figure 5, 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.

Part of the closure of the mode `M` of Figure 2 is shown in Figure 5. The closure construction is defined formally below.

Definition 6 (Closure) *Let $M = (E, X, V_r, V_w, V_l, SM, T)$ be a mode. The* closure $c(M)$ *of M is defined to be M if SM is empty, and is defined to be the mode $(E, X, V_r, V_w, V_l \cup \{h\}, c(SM), dT)$ containing a set of closed submodes $c(SM) = \{c(m) \mid m \in SM\}$ and a closed set dT of transitions obtained by adding, for each exit $x \in SM.dX$, the transitions (x, α_x, dx) and $(de, \beta_x, \tilde{x})$, where*

- *for $x \in SM.X$, $\tilde{x} = x$, and for $x = N.dx$, $\tilde{x} = N.de$,*
- *for states s and t, $(s, t) \in \alpha_x$ iff $t[h] = x$, $t[y] = s[y]$ for $y \neq h$, and for every transition (x, α, x') in T, α is disabled at s,*
- *for states s, $(s, s) \in \beta_x$ iff $s[h] = x$.*

□

Now we proceed to define the operational semantics. 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.

Definition 7 (Operational semantics) *For a mode $M = (E, X, V_r, V_w, V_l, SM, T)$, the set V_p of private variables is defined to be the set $V_l \cup SM.V_p$.*

The set mT of macro-transitions consists of transitions of the form (e, α, x) with $e \in dE$, $x \in dX$, and α is the action from V_r to $V_w \cup V_p$ if $e \in E$ and from $V_r \cup V_p$ to $V_w \cup V_p$ otherwise, defined as follows.

Given the macro-transitions of the submodes of M, a micro-execution *of M is a sequence of the form*

$$(e_0, s_0) \rightarrow (e_1, s_1) \rightarrow \cdots \rightarrow (e_n, s_n)$$

of control points $e_i \in C$ and states s_i over $V_g \cup V_p$ such that

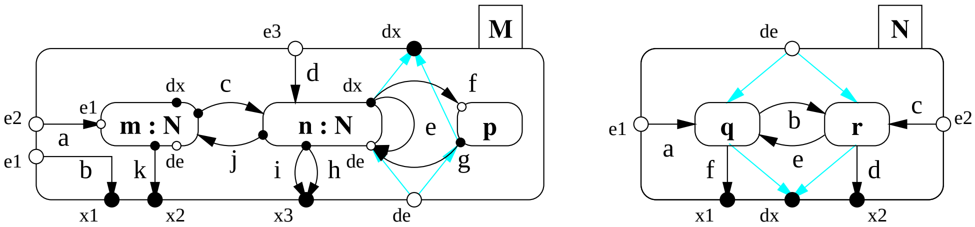

Figure 5: Closed behavior diagrams

- *for even i, the transition $((e_i, s_i), (e_{i+1}, s_{i+1}))$ is in dT, and*
- *for odd i, the transition $((e_i, s_i), (e_{i+1}, s_{i+1}))$ is in $SM.mT$.*

Given such an execution for an entry point e_0 and an exit point e_n of M, the macro-transition relation mT contains $((e_0, s_0), (e_n, s_n))$. □

In the above definition of micro-executions of a mode, the states s_i are valuations to the variables $V_g \cup V_p$, but only a subset of these influence each step.

The *operational semantics* of a mode M consists of is its control points, global variables, private variables, and its macro-transitions.

Remark 3 (Consistency of modes) *There are two well-formedness requirements to ensure consistency. First, the control should reach the default entry point of a submode only when the submode was previously visited and exited along its default exit point. Second, the number of micro-steps constituting a macro-step should be bounded. The second assumption is needed to ensure that there are no infinite loops within a mode, so the control always is passed back to the environment. These requirements can be enforced by syntactic restrictions that can be checked by static analysis.* □

A *top-level* mode is a mode M with a single entry point e and no exit points. Such a mode can be viewed as a module with private variables V_p, interface variables V_w, external variables $V_r \backslash V_w$, initialization specified by macro-transitions from e to dx, and update specified by macro-transitions from de to dx. For example, the mode `Conn1` is a top-level mode.

3.3 Trace 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 private ones. Finally, it gives the state back to the mode along one of its entry points.

Definition 8 (Denotational semantics) *An* execution *of a mode M is a sequence*

$$(e_0, s_0) \rightarrow (x_0, t_0) \rightarrow (e_1, s_1) \rightarrow (x_1, t_1) \rightarrow \cdots \rightarrow (x_n, t_n)$$

of control points $e_i \in dE$, $x_i \in dX$ with e_0 in E and states s_i and t_i over $V_g \cup V_p$ such that for all i, $((e_i, s_i), (x_i, t_i)) \in mT$ and $s_{i+1}[V_p] = t_i[V_p]$. Given such an execution, the corresponding trace *of M is obtained by projecting each state to the set V_g of global variables. The set of* traces *of M is denoted L_M. The* denotational semantics *of a mode M consists of its control points, global variables, and the set L_M of traces.* □

Note that, for a top level mode, the environment is another reactive module. For a lower level mode, the environment may be a regular or a group transition.

In order to show that our trace semantics is compositional, we need to be able to define the semantics of a mode only in terms of the trace semantics of its submodes. Recall that an execution of a mode is obtained by alternating between its macro-transitions and environment transitions. A macro-transition of a mode is obtained by compressing a micro-execution which in turn is obtained by alternating the transitions of M with the macro-transitions of its submodes (see Figure 6).

Definition 9 (Trace extraction) *Given a sequence $\sigma = (e_0, s_0)(e_1, s_1) \ldots (e_n, s_n)$ of control points and states, and a mode N, the restriction $\sigma \Uparrow N$ is the sequence obtained from σ replacing each s_i by $s_i[N.V_g]$, and by deleting pairs (e_i, s_i) if $e_i \notin N.dE \cup N.dX$.* □

Lemma 1 (Trace construction) *For a mode M, a sequence*

$$(e_0, s_0) \rightarrow (x_0, t_0) \rightarrow (e_1, s_1) \rightarrow (x_1, t_1) \rightarrow \cdots \rightarrow (x_n, t_n)$$

of control points $e_i \in dE$, $x_i \in dX$ with e_0 in E and states s_i and t_i over V_g, is a trace iff for all i, there exits a sequence $(e_{i,0}, s_{i,0})(e_{i,1}, s_{i,1}), \ldots (e_{i,n_i}, s_{i,n_i})$ of states $s_{i,j}$ over $V_g \cup V_p$ and control points $e_{i,j}$ in C such that

1. *$e_{i,0} = e_i$, $s_{i,0}[V_g] = s_i$,*
2. *$e_{i,n_i} = x_i$, $s_{i,n_i}[V_g] = t_i$,*
3. *$s_{i+1,0}[V_p] = s_{i,n_i}[V_p]$,*
4. *for all j, $((e_{i,2j}, s_{i,2j}), (e_{i,2j+1}, s_{i,2j+1})) \in dT$, and*
5. *for all submodes N, the restriction of the concatenated sequence*

$$\begin{array}{l}(e_{0,0}, s_{0,0})(e_{0,1}, s_{0,1}) \ldots (e_{0,n_0}, s_{0,n_0}) \\ (e_{1,0}, s_{1,0})(e_{1,1}, s_{1,1}) \ldots (e_{1,n_1}, s_{1,n_1}) \\ \ldots\end{array}$$

to N is a trace of N.

The above lemma is used to prove the following theorem:

Theorem 1 *The set of traces of a mode M can be computed from the set of traces of its submodes and its closed transition relation dT.*

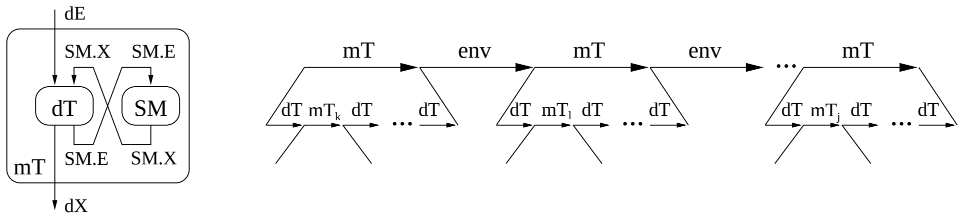

Figure 6: The traces of M

3.4 Refinement

The trace semantics leads to a natural notion of refinement between modes: a mode M refines N if it has the same global variables and control points, and every trace of M is a possible trace of N.

Definition 10 (Refinement) *A mode M and a mode N are said to be* compatible *if $M.V_r = N.V_r$, $M.V_w = N.V_w$, $M.E = N.E$ and $M.X = N.X$. Given two compatible modes M and N, M refines N, denoted $M \preceq N$, if $L_M \subseteq L_N$.* $\square$

For a finite index set I, we write $\{M_i \mid i \in I\} \preceq \{N_i \mid i \in I\}$ if $M_i \preceq N_i$ for each $i \in I$. The refinement operator is compositional with respect to the encapsulation:

Theorem 2 (Submode Compositionality) *Given a mode $M = (E, X, V_r, V_w, V_l, SM, T)$ and suppose $SM \preceq SN$. Then for $N = (E, X, V_r, V_w, V_l, SN, T)$, $M \preceq N$.*

The refinement rule is shown in a visual way in Figure 7, left.

Example 4 (Village telephone system) *A refinement of the mode* `toggle` *is the mode* `ctoggle` *shown in Figure 8, left. While in* `toggle`*, the switch to* `on` *is enabled whenever* `h` *is* `off`*, in* `ctoggle` *the switch to* `on` *is enabled whenever* `c` *is* `disconnected`*. The transitions of* `ctoggle` *are given*

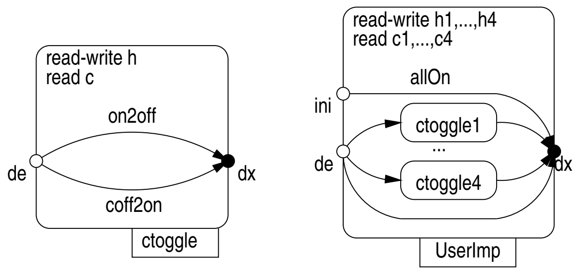

Figure 8: An implementation of `UserSpec`

below.

```
on2off  ≝  h = on -> h := off
coff2on ≝  c != disconnected -> h := on
```

To prove refinement, the additional read variable `c` *may be added to the mode* `toggle` *without any harm because it is not read by this mode. In a more general setting, the refinement rules would take additional variables into account.*

Since `ctoggle` $\preceq$ `toggle` *we obtain by compositionality and the fact that renaming does not change refinement that* `UserImp` $\preceq$ `UserSpec`*.* $\square$

If we consider a submode N within a mode M, the remaining submodes of M and the transitions of M can be viewed as an environment for N. If we replace N by another submode N', let the resulting mode be denoted $M[N']$. To show that $M[N']$ refines $M[N]$, by the above theorem, it suffices to establish that N' refines N. A dual question concerns replacing the context M of N by some other context M' with possibly different transitions and/or other submodes. To establish that $M'[N]$ refines $M[N]$, we can replace N by any submode that it refines,

Theorem 3 (Context Compositionality) *Let M and M' be compatible modes with a common submode N. If $M[N] \preceq M'[N]$ and $N' \preceq N$, then $M[N'] \preceq M'[N']$.*

This suggests the following compositional proof method. To establish that $M[N]$ refines $M'[N']$, we first show that N refines N'. Then, we can choose the most general mode with the interface of N, namely, the one that puts no constraints on the update of write-variables.

Definition 11 (Most General Submodes) *Given the global variables $V_g = (V_r, V_w)$, entry points E, and exit points X, the mode $G(V_r, V_w, E, X)$ is defined to be the mode with no private variables, and whose transitions contain all pairs of the form $((e, s), (x, t))$ for $e \in dE$, $x \in dX$, s is any state over V_r and t is any state over V_w.* $\square$

Note that the mode $G(V_r, V_w, E, X)$ is the most general mode with the given data and control: verify that every mode mode M refines $G(M.V_r, M.V_w, M.E, M.X)$. Thus, if $M[G]$ refines $M'[G]$, for $G = G(N.V_r, N.V_w, N.E, N.X)$, we can conclude that $M[N]$ refines $M'[N]$. A visual representation of this rule is shown in Figure 7, middle.

The compositionality rules allow us to decompose the proof obligation into refinement of submodes in the most general context, and refinement of context under the most general submodes. Can we allow circular assume-guarantee reasoning in the style of Proposition 2. Unfortunately, if $M[N'] \preceq M'[N']$ and $M'[N] \preceq M'[N']$, we cannot conclude that $M[N] \preceq M'[N']$. This is because in the trace-semantics of the mode $M[N]$, the interaction between the context M and submode N is not observable. A modified definition of refinement preserves interactions with a specified submode.

Definition 12 (Submode sensitive refinement) *Let M be a mode with a submode N. The trace-language L_M^N is defined so that for every trace*

$$(e_0, s_0)(x_0, t_0)(e_1, s_1)(x_1, t_1)\ldots(x_n, t_n)$$

of M, and for all micro-executions ρ_i of M from (e_i, s_i) to (x_i, t_i), the sequence

$$(e_0, s_0)(\rho_0 \Uparrow N)(x_0, t_0)(e_1, s_1)(\rho_1 \Uparrow N)(x_1, t_1)\ldots(x_n, t_n)$$

is in L_M^N. Let M and M' be compatible modes, and let N and N' be submodes of M and M', respectively, compatible with each other. Then, $M \preceq^N M'$ iff $L_M^N \subseteq L_{M'}^{N'}$. $\square$

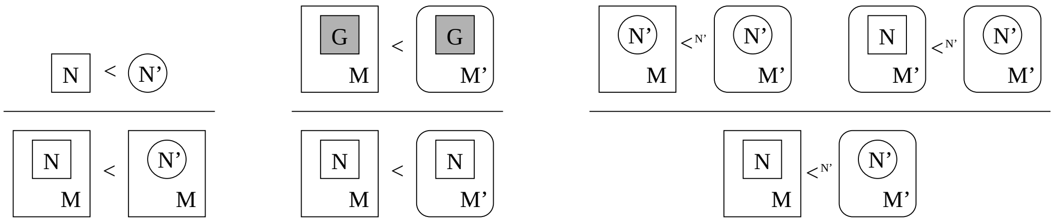

Figure 7: Compositional and assume/guarantee rules

Observe that if $M \preceq^N M'$, then $M \preceq M'$. This stronger refinement preorder supports circular assume-guarantee reasoning:

Theorem 4 (Assume Guarantee) *Let M and M' be compatible modes, and let N and N' be submodes of M and M', respectively, compatible with each other. Then, if $M[N'] \preceq^{N'} M'[N']$ and $M'[N] \preceq^{N'} M'[N']$, then $M[N] \preceq^{N'} M'[N']$.*

A visual representation of this rule is shown in Figure 7, right.

4 Conjunctive Modes

The closure semantics of modes presented in the previous sections is a very expressive setting in which we can define all the interesting parallel composition operators found in the literature. Moreover, the user may define himself the desired composition and can even mix different ways of compositions in the same setting. The main characteristics of the semantics that allow us to do this are hierarchy, control interfaces and the fact that closed modes never get "stuck" and take "no time" to execute. In this setting, a module is a top-like mode (with *ini* as the only regular control point) that communicates with its environment via shared variables. In the following subsection we discuss the lock-step parallel composition.

Lock-step parallel composition. In the lock-step composition, a round (macro step) of a composed mode consists of a sequence of rounds of the submodes, one round for each submode. The order of execution of the submodes is arbitrary and has no consequence on the trace semantics of the composed mode. As a consequence, each linearization of the submodes is a valid translation of the corresponding composed module. A linearization that gives priority to the left mode is shown in Figure 9, right. The associated module diagram is given in Figure 9, left. Note that the module diagram explicitly shows the communication variables. To

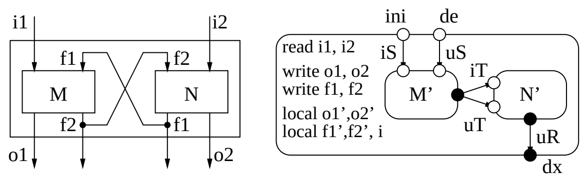

Figure 9: lock-step parallel composition

make sure that each mode in the sequence accesses the same values of the variables, namely the values at the beginning of the round, we have to latch the write variables. This is easily accomplished as follows.

First, we add for each write variable of a submode a local primed variable to the composed mode. The primed variable is used to keep the current value, i.e., the value computed by the submode in the current round. The entry transition uS of the composed mode initializes the primed variables with the value of the corresponding unprimed variables. The exit transition uR assigns the value of the primed variable back to the unprimed one.

Second, since the submodes were defined in terms of unprimed variables, they have to be renamed such that each occurrence of an unprimed write variable (even in read declarations) is replaced by a primed variable. Note that only one submode is renamed with respect to a write variable because composability assures that the two submodes write to disjoint sets of variables. In Figure 9 the renamed modes are marked as primed.

To be able to distinguish between the initialization and update rounds we also add a local variable i that is true in the initialization rounds and false otherwise. This is achieved by setting i to true and false in iS and uS respectively and by guarding iT and uT by this value. Formally, this is defined as follows.

Definition 13 (Lock-step composition) *Given two top-level modes M and N such that $M.V_w \cap N.V_w = \emptyset$. The lock-step composition $M \| N$ of the modes M and N is defined as follows.*

Control points. *The sets of entry and exit points $dE = \{ini, de\}$ and $dX = \{dx\}$.*

Variables. *The sets of V_w, V_r and V_l of write, read and local variables defined as below.*

$$\begin{aligned} V_w &= M.V_w \cup N.V_w, \\ V_r &= (M.V_r \cup N.V_r) \setminus V_w, \\ V_l &= M.V_w' \cup N.V_w' \cup \{i\} \end{aligned}$$

Submodes. *The set $\{M', N'\}$ where:*

$$\begin{aligned} M' &= M[M.V_w'/M.V_w] \\ N' &= N[N.V_w'/N.V_w] \end{aligned}$$

That is, M' is obtained from M by replacing, for each variable x in $M.V_w$, every reference to x in M by x'.

Transitions. *The set T of entry, internal and exit transitions:*

$$\begin{aligned} T = \quad & \{(ini, iS, M'.ini), (de, uS, M'.de)\} \cup \\ & \{(M'.dx, iT, N'.ini), (M'.dx, uT, N'.de)\} \cup \\ & \{(N'.dx, uR, dx)\} \end{aligned}$$

where iS, uS, iT, uT and uR are defined as follows.

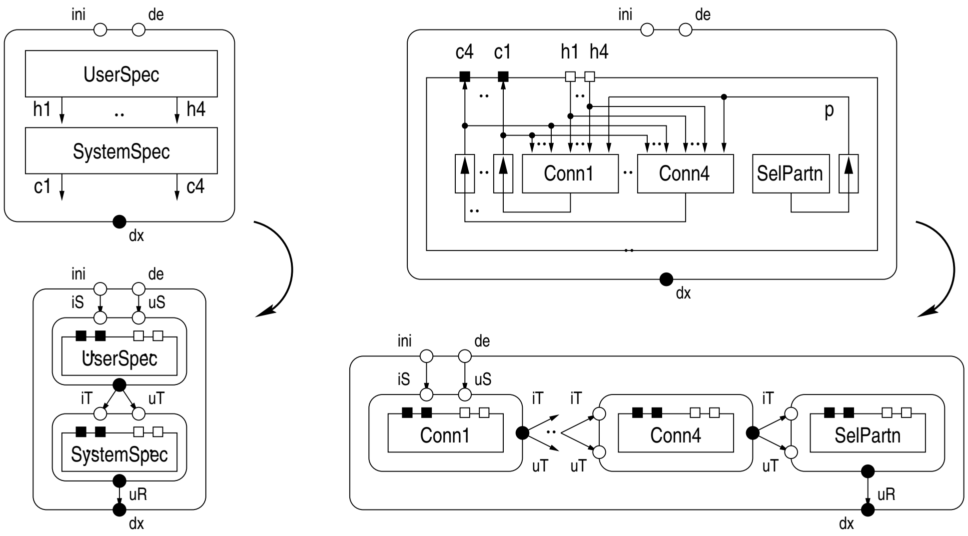

Figure 10: Recursive translation of modules to modes

$$
\begin{aligned}
iT &\stackrel{\text{def}}{=} \texttt{i = true -> skip}\\
uT &\stackrel{\text{def}}{=} \texttt{i = false -> skip}\\
iS &\stackrel{\text{def}}{=} \texttt{true -> i := true}\\
uS &\stackrel{\text{def}}{=} \texttt{true -> i := false; } \texttt{x}'_1 \texttt{ := x}_1\texttt{; .. ; x}'_n \texttt{ := x}_n\\
uR &\stackrel{\text{def}}{=} \texttt{true -> i := false; } \texttt{x}_1 \texttt{ := x}'_1\texttt{; .. ; x}_n \texttt{ := x}'_n
\end{aligned}
$$

where $\texttt{x}_1 .. \texttt{x}_n$ *are the variables in* V_w*.*

□

From the above definition it follows easily that $M \preceq M \| N$ and that parallel composition is compositional with respect to refinement.

Theorem 5 *If* $M_1 \preceq N_1$ *and* $M_2 \preceq N_2$ *are top-like modes, then* $M_1 \| M_2 \preceq N_1 \| N_2$*.*

Remark 4 (Await dependencies) *In the above theorem we considered only Moore-like modules and their translation to Moore-like modes. In general, reactive modules in* MOCHA *allow to define a partial order (await dependencies) between atomic modules called atoms such that the values computed by an atom in a round may influence the values computed by any atom greater in the partial order in the same round. We can easily handle this in our setting by considering that each submode in the composition has both an unprimed and a primed copy of each variable. Similarly to the transitions of a parallel mode, the submodes may then use both primed and unprimed variables in their transitions. The partial order, is nothing but a constraint that has to be satisfied by the linearization of modes. Note that in this setting, one may define events analogously to reactive modules. For example, testing for the occurrence of an event* $e?$ *is translated to* $e' \neq e$*. Similarly, issuing an event* $e!$ *is translated to* $e' := \neg e$*.* □

The lock-step composition of two top-level modes is *essentially the same* as the parallel composition of the modules defined by these modes. The only difference is of technical nature: the modules assume that the environment is not writing on their interface variables while the modes do not make this assumption for their write variables. This mismatch can be easily handled by defining two operators: *mode2module* and *module2mode* that convert top-level modes to modules and reciprocally modules to modes.

The *mode2module* operation only adds the environment assumption. The *module2mode* operation works recursively. First it replaces each submodule by a submode that does not impose the environment assumption. Then it replaces the parallel composition of modules by the lock-step composition of modes. Since any atom in reactive modules is also a mode, this gives an algorithm to completely convert any module to a mode.

Example 5 (Village telephone system) *The recursive conversion of the village telephone system module (architecture diagram) to a mode is shown in Figure 10.* □

Hence, a module diagram may be regarded as a convenient shorthand for a particular mode diagram. However, since a module diagram shows the observable variables and their sharing among modes, a module diagram is more convenient for the representation of parallel modes. The fact that a module diagram represents a mode allows us to integrate module diagrams inside of mode diagrams and the other way around. As a consequence, we can construct arbitrary and/or hierarchies of mixed synchronous or asynchronous components.

The construction of such hierarchies is an important step towards modeling modern concepts like dynamical network reconfiguration or mobility. For example, creating a new process can be easily accomplished by a transition that takes a module diagram into another module diagram that has an additional module. To model mobility we also need to distinguish between location of computation modules and software modules. It is important to note that this expressive power comes together with a clean semantics that can be used successfully in analysis.

Example 6 (Hot-lines) *A possible implementation of the module* `SystemSpec` *is by hot lines, as shown in Figure 11, right. The mode definition of the module* `Line1` *is shown in Figure 11, left. The associated transitions are given below. The definition of the module* `Line2` *is similar.*

$$\texttt{all0} \stackrel{\text{def}}{=} \texttt{true ->}$$

```
c1 := disconnected; c2 := disconnected;
```

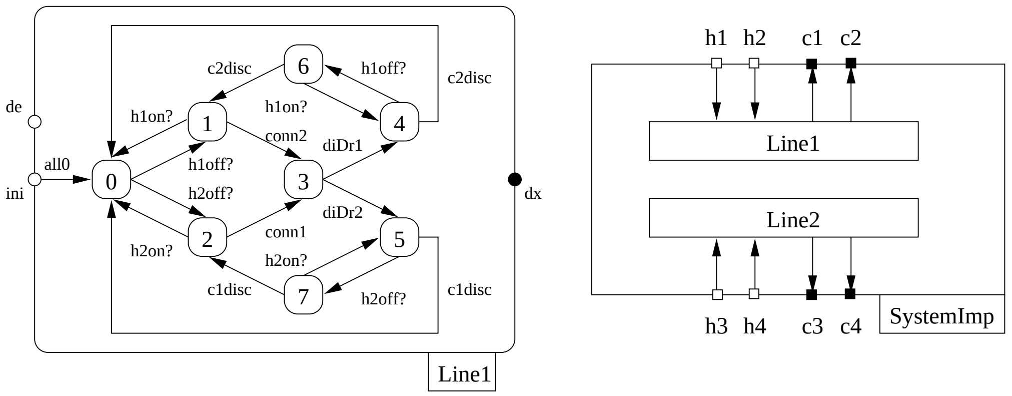

Figure 11: Hot lines implementation

`h1off?` $\stackrel{\text{def}}{=}$ `h1 = off -> skip;`

`h2off?` $\stackrel{\text{def}}{=}$ `h2 = off -> skip;`

`h1on?` $\stackrel{\text{def}}{=}$ `h1 = on -> skip;`

`h2on?` $\stackrel{\text{def}}{=}$ `h2 = on -> skip;`

`conn2` $\stackrel{\text{def}}{=}$ `h2 = off -> c1 := 2; c2 := 1;`

`conn1` $\stackrel{\text{def}}{=}$ `h1 = off -> c1 := 2; c2 := 1;`

`diDr1` $\stackrel{\text{def}}{=}$ `h1 = on -> c1 := disconnected; c2 := drooping;`

`diDr2` $\stackrel{\text{def}}{=}$ `h2 = on -> c2 := disconnected; c1 := drooping;`

`c2disc` $\stackrel{\text{def}}{=}$ `h2 = on -> c2 := disconnected;`

`c1disc` $\stackrel{\text{def}}{=}$ `h1 = on -> c1 := disconnected;`

We would like now to prove by assume/guarantee reasoning that `UserImp` $\|$ `SystemImp` $\preceq$ `UserSpec` $\|$ `SystemSpec`. *This can be done in a mixed module/modes setting or solely in a modes setting by converting the above modules to modes. In this case, we can use the assume/guarantee rule for modes.*

Let `SystemImpPar` *be the top level mode for the parallel composition together with the mode for* `SystemImp` *and* `SystemSpecPar` *be the top level mode for the parallel composition together with the mode for* `SystemSpec`. *Then we have to prove that*

$$\texttt{SystemImpPar}[\texttt{UserSpec}] \preceq \texttt{SystemSpecPar}[\texttt{UserSpec}]$$
$$\texttt{SystemSpecPar}[\texttt{UserImp}] \preceq \texttt{SystemSpecPar}[\texttt{UserSpec}]$$

It is easy to see that in this case the assume/guarantee rule for modes is the same as the one for modules. □

5 Conclusions

The notion of hierarchy is useful for structuring architecture of component connections as well as for describing behavior of individual components. While architectural hierarchy has been well understood in context of modular reasoning, there has been no basis for modular reasoning about behavior hierarchy. Existing languages for hierarchic state-machines have complex operational semantics and no notion of observational refinement. We show that hierarchy can be preserved in observational trace semantics even in presence of powerful features such as mode hierarchy, exceptions, history retention, conjunctive modes, and mode reuse. Our language has powerful rules for refinement of modes, and should provide a basis for systematic development and formal analysis of hierarchic descriptions.

The current proposal builds on our previous work on the language of reactive modules and the toolkit Mocha that supports assume-guarantee refinement checks. The operations of building a mode by connecting submodes, scoping of local variables, and mode instantiation, are direct analogs of parallel composition of modules, variable hiding, and module instantiation, respectively. Indeed, the same graphical-user-interface can be used for both the module diagrams and mode diagrams. We have already started work on building the GUI and a model checker for the hierarchic modules.

In the reactive modules language behavior is described using the so-called *atoms* which roughly correspond to logic-gates, and is already quite satisfactory for hardware applications. The hierarchic reactive modules extensions makes it well suited for software applications as well. In fact, our framework provides a coherent integration of the two types of specifications, making it suitable for hardware/software codesign.

Acknowledgments

We thank Manfred Broy, Carl Gunter, Tom Henzinger, Michael McDougall, Amir Pnueli, Bran Selic, Gheorghe Stefanescu and Mihalis Yannakakis for fruitful discussions and suggestions. Rajeev Alur is supported by DARPA/NASA grant NAG2-1214, NSF CARRER award CCR-9734115, SRC award 99-688, Sloan Faculty Fellowship, and Bell Laboratories. Radu Grosu is supported by DARPA/NASA grant NAG2-1214.

References

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999. Invited submission to FLoC'96 special isuue. A preliminary version appears in *Proc. 11th LICS, 1996.*

[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.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM TOPLAS*, 17:507–534, 1995.

[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.

[BGG+98] K. Bhargavan, C.A. Gunter, E.L. Gunter, M. Jackson, D. Obradovic, and Pamela Zave. The village telephone system: A case study in formal software engineering. In *Proceedings of 11th International Conference on Theorem Proving in Higher-Order Logics TPHOLs98*, 1998.

[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.

[GL94] O. Grümberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[GSB98] R. Grosu, T. Stauner, and M. Broy. A modular visual model for hybrid systems. In *Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'98)*, LNCS 1486. Springer-Verlag, 1998.

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

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Software Engin. Methods*, 5(4):293–333, 1996.

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

[HPSS87] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proc. 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64, 1987.

[HQR98] T.A. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV 98: Computer-aided Verification*, LNCS 1427, pages 521–525, 1998.

[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.

[Lam91] L. Lamport. The temporal logic of actions. Technical report, DEC Systems Research Center, Palo Alto, California, 1991.

[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.

[LT87] N.A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

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

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In *Computer-Aided Verification CAV'97*, pages 24–35, 1997.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer-Verlag, 1980.

[PS91] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Proc. Symposium on Theoretical Aspects of Computer Software*, LNCS 526, pages 244–264, 1991.

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

[Sta85] E.W. Stark. A proof technique for rely-guarantee properties. In *FST & TCS 85: Foundations of Software Technology and Theoretical Computer Science*, LNCS 206, pages 369–391, 1985.

[US94] A. Uselton and S. Smolka. A compositional semantics for statecharts using labeled transition systems. pages 2–17, 1994.