Deductive Verification of Modular Systems *

Bernd Finkbeiner, Zohar Manna and
Henny B. Sipma
finkbein|manna|sipma@cs.stanford.edu

Computer Science Department, Stanford University
Stanford, CA. 94305

Abstract. Effective verification methods, both deductive and algorithmic, exist for the verification of global system properties. In this paper, we introduce a formal framework for the modular description and verification of parameterized fair transition systems. The framework allows us to apply existing global verification methods, such as verification rules and diagrams, in a modular setting. Transition systems and transition modules can be described by recursive module expressions, allowing the description of hierarchical systems of unbounded depth. Apart from the usual *parallel composition*, *hiding* and *renaming* operations, our module description language provides constructs to *augment* and *restrict* the module interface, capablilities that are essential for recursive descriptions. We present proof rules for property inheritance between modules. Finally, *module abstraction* and induction allow the verification of recursively defined systems. Our approach is illustrated with a recursively defined arbiter for which we verify mutual exclusion and eventual access.

1 Introduction

In this paper we introduce a formal framework for the modular description and verification of parameterized fair transition systems. Our specification language is linear-time temporal logic (LTL) with past operators, a formalism widely accepted for the specification of reactive systems. We demonstrate that LTL can be successfully used to specify properties of system components in addition to specifying complete programs.

The objectives we want to achieve with our modular framework are the following. First, it should provide a system description language that allows concise, modular, possibly recursive and parameterized system descriptions, and that is sufficiently expressive to represent various modes of communication between modules. It should also allow the reuse of code, that is, transition modules can be described once and referred to multiple times in a system description. Second, the framework should allow analysis of individual components and combine

* This research was supported in part by the National Science Foundation under grant CCR-95-27927, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, ARO under grant DAAH04-95-1-0317, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

these analysis results in a meaningful way when components are composed. It should support a variety of analysis techniques currently applied to fair transition systems, such as verification rules, verification diagrams, model checking, abstraction and refinement, in such a way that the results can be seamlessly combined in the course of a single proof.

Our framework extends the principles for modular verification presented in [MP95b] and those formulated for I/O automata [LT89, LT87]. The basic building block of our system description language is a transition module, consisting of an interface that describes the interaction with the environment, and a body that describes its actions. Communication between a module and its environment can be asynchronous, through shared variables, and synchronous, through synchronization of transitions. More complex modules can be constructed from simpler ones by (recursive) module expressions, allowing the description of hierarchical systems of unbounded depth. Module expressions can refer to (instances of parameterized) modules defined earlier by name, thus enabling the reuse of code and the reuse of properties proven about these modules. Apart from the usual *hiding* and *renaming* operations, our module description language provides a construct to *augment* the interface with new variables that provide a summary value of multiple variables within the module. Symmetrically, the *restrict* operation allows the module environment to combine or rearrange the variables it presents to the module. As we will see later, these operations are essential for the *recursive* description of modules, to avoid an unbounded number of variables in the interface.

The basis of our proposed verification methodology is the notion of modular validity, as proposed in [Pnu85, Cha93, MP95b]. An LTL property holds over a module if it holds over any system that includes that module (taking into account variable renamings). That is, no assumptions are made about the module's environment. Therefore, modular properties are inherited by any system that includes the module. Many useful, albeit simple properties can be proven modularly valid. However, as often observed, not many interesting properties are modularly valid, because most properties rely on some cooperation by the environment.

The common solution to this problem is to use some form of *assumption-guarantee* reasoning, originally proposed by Misra and Chandy [MC81]. Here, a modular property is an assertion that a module satisfies a guarantee G, provided that the environment satisfies the assumption A. An assumption–guarantee property can be formulated as an implication of LTLformulas with past operators [BK84, GL94, JT95]. Thus in LTL there is no need for compositional proof rules dealing with the discharge of assumptions as for example in [AL93]. In our framework these rules are subsumed by *property inheritance* rules: systems that are composed of modules by parallel composition directly inherit properties of their components. In this way assumptions can be discharged either by properties of other components, or by the actual implementation of the composite module. If the assumption cannot be discharged, it is simply carried over to the composite module. This flexibility in our approach as to when and how

assumptions are discharged is similar to the one described by Shankar [Sha93]. In particular it does not require the verifier to anticipate assumptions that could be made on a module by other modules [Sha98].

Our verification methodology supports both composition and decomposition, as defined by Abadi and Lamport [AL93]. In *compositional* reasoning, we analyze a component without knowing the context it may be used in. We therefore state and prove properties that express explicitly under what assumptions on the environment a certain guarantee is given. This approach is taken by our modular proof rule and the *property inheritance* rules. In *decompositional* reasoning the composite system is analyzed by looking at one module at a time. In our experience both methods can be used during a system verification effort. Compositional reasoning is used to establish invariants and simple liveness properties about components. Then the system is analyzed from the top down, using the previously proven modular properties, and using abstraction to hide details that are irrelevant to the property at hand. We provide a modular inheritance rule that allows modules in expressions to be replaced with simpler modules, such that properties proven over the system containing the simpler module are also valid over the system containing the actual module. Alternatively this can also be used in the other direction, in design. Any (underspecified) module may be refined into a more detailed one, while preserving the properties proven so far.

A convenient abstraction, which can be constructed automatically, is the *interface abstraction*, which represents only the information in the interface and ignores all implementation details. Using the interface abstraction in place of a module is especially useful when we consider recursively described systems of unbounded depth: in this case the implementation details are in fact unknown. Such systems fit in naturally in our framework: we combine the decompositional interface abstraction with a compositional *induction rule*.

1.1 Example

We illustrate our description language and verification methodology with the verification of a recursively defined binary *arbiter* that guarantees mutual exclusion to a critical resource. A number of clients can each request access, and the arbiter gives a grant to one client at a time. Our design, shown in Figure 1, is based on a similar example in [Sta94]. The arbiter is described as a tree of nodes, where a tree consists of two subtrees and a node that guarantees mutual exclusion between the two subtrees. Thus while the simple algorithm represented in the nodes deals with two clients at a time, a tree of height h ensures mutual exclusion for 2^h clients.

The idea of constructing a n-way arbiter by combining 2-way arbiters in a tree goes back to [Sei80]. Local correctness proofs for implementations of the nodes are discussed in [Dil88], and safety properties of arbiter trees are verified in [GGS88].

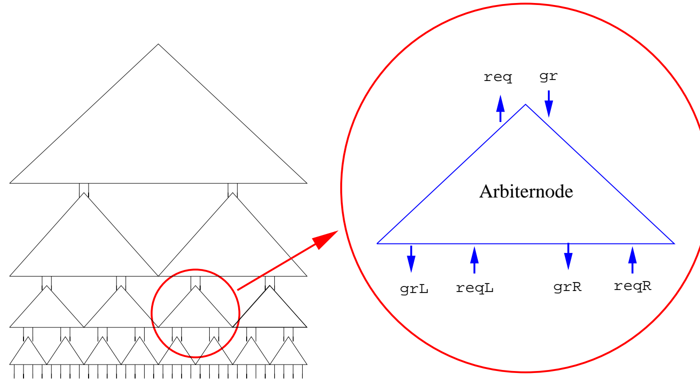

Fig. 1. A Hierarchical Arbiter.

1.2 STeP

Part of the framework presented here has been implemented in STeP (Stanford Temporal Prover), a tool for the deductive and algorithmic verification of reactive, real-time and hybrid systems [BBC+96, BBC+95, BMSU97]. STeP implements *verification rules* and *verification diagrams* for deductive verification. A collection of decision procedures for built-in theories, including integers, reals, datatypes and equality is combined with propositional and first-order reasoning to simplify verification conditions, proving many of them automatically. The proofs in the arbiter example, presented in Section 5, have been performed with STeP.

1.3 Outline

The rest of the paper is organized as follows. In Section 2 we introduce our computational model, fair transition systems, and specification language, LTL. In Section 3 we define transition modules and parameterized transition modules, and present the syntax and semantics of our module description language. Here we give a full description of the arbiter example. In Section 4 we propose a modular verification rule and devise verification rules for property inheritance across the operations of our module description language. We discuss modular abstraction and induction as techniques that can be used to prove properties over recursively defined modules. In Section 5 we verify mutual exclusion and eventual access for the arbiter using the rules presented in Section 4.

2 Preliminaries

2.1 Computational Model: Transition Systems

As the underlying computational model for verification we use *fair transition systems* (FTS) [MP95b].

Definition 1 Fair transition System. A fair transition system $\Phi = \langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ consists of

- V: A finite set of typed *system variables*. A *state* is a type-consistent interpretation of the system variables. The set of all states is called the *state space*, and is designated by Σ. We say that a state s is a p-state if s satisfies p, written $s \models p$.
- Θ: The *initial condition*, a satisfiable assertion characterizing the initial states.
- $\mathcal{T}$: A finite set of *transitions*. Each transition $\tau \in \mathcal{T}$ is a function

$$\tau : \Sigma \mapsto 2^{\Sigma}$$

  mapping each state $s \in \Sigma$ into a (possibly empty) set of τ-successor states, $\tau(s) \subseteq \Sigma$. Each transition τ is defined by a *transition relation* $\rho_\tau(V, V')$, a first-order formula in which the unprimed variables refer to the values in the current state s, and the primed variables refer to the values in the next state s'. Transitions may be parameterized, thus representing a possibly infinite set of similar transitions.
- $\mathcal{J} \subseteq \mathcal{T}$: A set of *just* transitions.
- $\mathcal{C} \subseteq \mathcal{T}$: A set of *compassionate* transitions.

Definition 2 Runs and Computations. A *run* of an FTS $\Phi = \langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ is an infinite sequence of states $\sigma : s_0, s_1, s_2, \ldots$, such that

- *Initiation*: s_0 is initial, that is, $s_0 \models \Theta$.
- *Consecution*: For each $j = 0, 1, \ldots$, s_{j+1} is a τ-successor of s_j, that is, $s_{j+1} \in \tau(s_j)$ for some $\tau \in \mathcal{T}$. We say that τ is *taken* at s_i if S_{i+1} is a τ-successor of s_i.

A *computation* of an FTS Φ is a run σ of Φ such that

- *Justice*: For each transition $\tau \in \mathcal{J}$, it is not the case that τ is continuously enabled beyond some point in σ without being taken beyond that point.
- *Compassion*: For each transition $\tau \in \mathcal{C}$, it is not the case that τ is infinitely often enabled beyond a certain point in σ without being taken beyond that point.

Definition 3 Parameterized Transition System. Let $\mathcal{F}$ be the class of all fair transition systems, and $P = \langle p_1, \ldots, p_n \rangle$ a tuple of parameters with type $t_1, \ldots, t_n$. Then a *parameterized transition system* $\mathbf{F} : t_{p1} \times \ldots \times t_{pn} \mapsto \mathcal{F}$ is a

function from the input parameters to fair transition systems. Given a parameterized fair transition system $\mathbf{F}$, and a list of values $a_1, \ldots, a_n$, type consistent with $p_1, p_2, \ldots, p_n$, then the *instance* $\mathbf{F}(a_1, a_2, \ldots, a_n)$ denotes the fair transition system where all the references to $p_1, p_2, \ldots, p_n$ have been replaced by $a_1, a_2, \ldots, a_n$.

An infinite sequence of states $\sigma: s_0, s_1, \ldots$ is a computation of a parameterized transition system $\mathbf{F}$ if σ is a computation of $\mathbf{F}(a)$ for some a. Thus the set of computations of a parameterized system is the union of the sets of computations of all its instances.

2.2 Specification Language

We use linear-time temporal logic (LTL) with past operators to specify properties of reactive systems. LTL formulas are interpreted over infinite sequences of states. The truth-value of a formula for a given model is evaluated at the initial position of the model. We say that a temporal formula holds at a particular state of a model if it is true of the sequence which starts at that state. Below we only define those temporal operators used in the rest of the example. For the full set, the reader is referred to [MP91].

Given a model $\sigma = s_0, s_1, \ldots$, the temporal operators $\Box$, $\Diamond$, $\ominus$ are defined as follows:

$\Box\, p$ holds at state s_j iff p is true for all states s_i, $i \geq j$;
$\Diamond\, p$ holds at state s_j iff p is true for some state s_i, $i \geq j$;
$\ominus\, p$ holds at state s_j iff s_j is not the first state and p holds at state s_{j-1};

$\Box$ and $\Diamond$ are called *future* operators, while $\ominus$ are called *past* operators. We will refer to formulas that contain only past operators as *past formulas*. In this paper we do not allow temporal operators to appear within the scope of a quantifier. A formula containing no temporal operators is called a *state-formula* or an *assertion*.

Given an FTS $\mathcal{S}$, we say that a temporal formula φ is $\mathcal{S}$-*valid* if every computation of $\mathcal{S}$ satisfies φ, and write $\mathcal{S} \models \varphi$.

3 Transition Modules and Systems

A transition system describes a closed system, that is, a system that does not interact with its environment. To enable reasoning about isolated components of a transition system, a transition system is described as a system of *transition modules*. A transition module has an *interface*, which describes its interaction with its environment, and a *body*, which is similar to a transition system, except that it must be consistent with the interface of the module it belongs to.

The interface of a module consists of a set of *interface variables* and a set of transition labels, that represent the *exported transitions*. We have four types of

interface variables: constants, input variables, output variables and shared variables. *Input variables* can only be changed by the environment, *output variables* only by the module. *Shared variables* can be modified by both the environment and the module.

Exported transitions can synchronize with other transitions with the same label in the environment. The result of synchronization is a new transition whose transition relation is the conjunction of the transition relations of the original transitions. One transition may have multiple labels, so it may synchronize with different transitions.

The module body has its own set of private variables that cannot be observed nor modified by the environment. The transitions in the body may modify private variables, output and shared variables, but not input variables.

To be able to prove properties about modules we associate with each module a transition system such that the set of computations of the associated transition system is a superset of the set of computations of any system that includes the module. Having these semantics for modules allows us to "lift" properties of modules to properties of the whole system, that is, if a property has been proven valid over a module, then a corresponding property can be inferred for any system that includes that module.

3.1 Transition Module: Definition

The basic building block of a transition module system is the transition module.

Vocabulary We assume that all variables in a transition module description are taken from a universal set of variables $\mathcal{V}$, called the *variable vocabulary*, and that all transition labels are taken from a universal set of labels called $\mathcal{T}_{id}$.

Definition 4 Transition Module. A *transition module* $\mathbf{M} = \langle I, B \rangle$ consists of an interface declaration $I = \langle V, T \rangle$ and a body $B = \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C} \rangle$. The interface components are

- $V \subseteq \mathcal{V}$: the set of interface variables, partitioned in four subsets as follows:
  - V_c: constants, possibly underspecified, which can be modified by neither the environment nor the module;
  - V_i: input variables, which can only be modified by the environment;
  - V_o: output variables, which can only be modified by the module;
  - V_s: shared variables, which can be modified by both the module and the environment.
- $T \subseteq \mathcal{T}_{id}$: a set of transition labels. The transitions corresponding to these labels may synchronize with transitions in the environment.

A transition module is called *closed* if both the set of shared variables and the set of exported transitions are empty.

The components of the body are:

- V_p: a set of private variables, which can neither be modified nor observed by the module's environment.
- Θ: the initial condition, an assertion over $V_p \cup V_o$.
- $\mathcal{T}$: a set of transitions, specified in the same way as described in Section 2; we require that

$$\rho_\tau \rightarrow \bigwedge_{v \in V_c} v' = v \qquad \text{for all } \tau \in \mathcal{T}$$

- $\lambda \subseteq \mathcal{T} \times \mathcal{T}_{id}$: a transition labeling relation, relating transitions to their labels. Note that multiple transitions can have the same label, and that a single transition may have multiple labels. We require that the labeling relation λ relates every label in the exported transitions T to at least one transition in $\mathcal{T}$, that is

$$\forall id \in T \;.\; \exists \tau \in \mathcal{T} \;.\; (\tau, id) \in \lambda$$

  For internal transitions, i.e., transitions that do not have a label in T, we require that they do not modify the input variables, that is,

$$\rho_\tau \rightarrow \bigwedge_{v \in V_i} v' = v \qquad \text{for all } \tau \in \{\tau \,|\, \forall id \in T \;.\; (\tau, id) \notin \lambda\}$$

- $\mathcal{J} \subseteq \mathcal{T}$: the set of just transitions.
- $\mathcal{C} \subseteq \mathcal{T}$: the set of compassionate transitions.

Modules can be *parameterized*, to represent, similar to the parameterized transition systems introduced in Section 2, functions from the parameters to transition modules.

Definition 5 Parameterized Transition Module. Let $\mathcal{M}$ be the class of all modules, and $P = \langle p_1, \ldots, p_n \rangle$ a set of parameters with type $t_1, \ldots, t_n$. Then a *parameterized transition module* (PTM) $\mathbf{M} : t_1 \times \ldots \times t_n \mapsto \mathcal{M}$ is a function from parameters to transition modules.

3.2 Example: Arbiternode

As described in Section 1, an arbiter is a device that guarantees mutual exclusion to a critical resource. Figure 1 shows the hierarchical design for an arbiter dealing with 2^n clients, which repeatedly uses the module `Arbiternode` (shown enlarged). An `Arbiternode` establishes mutual exclusion between two clients: its "left" and "right" client. In this section we only discuss the `Arbiternode` module; in Section 5 we will return to the complete arbiter design.

The two clients of the `Arbiternode` can request the grant by setting their request bits, `reqL` and `reqR` for the left and right client, respectively. If the `Arbiternode` owns the grant, that is, if the `gr` bit is set, it can pass the grant on to a client by setting the client's grant bit, `grL` or `grR`. The client can subsequently release the grant by resetting its request bit, which causes the arbiter

```
Module Arbiternode:

external in gr, reqL, reqR : bool
out req, grL, grR : bool where !req /\ !grL /\ !grR

Transition RequestGrant Just:
    enable !gr /\ (reqL \/ reqR)
    assign req:=true

Transition GrantLeft Just:
    enable gr /\ req /\ reqL /\ !grR
    assign grL:=true

Transition GrantRight Just:
    enable gr /\ req /\ reqR /\ !grL /\ !reqL
    assign grR:=true

Transition ReleaseLeft Just:
    enable gr /\ req /\ !reqL /\ grL
    assign grL:=false, grR:=reqR, req:=reqR

Transition ReleaseRight Just:
    enable gr /\ req /\ !reqR /\ grR
    assign grR:=false, req:=false

EndModule
```

Fig. 2. `Arbiternode` module

to reset the grant bit (`grL`, `grR`) and either give the grant to its other client, or release its own grant by resetting `req`.

Figure 2 shows the description of the `Arbiternode` module in STeP input format. In STeP, variables declared as **external in**, **out**, **external out** refer to input, output and shared variables, respectively. The keywords **enable** and **assign** allow a description of the transition relation in a programming-like notation: the transition relation is the conjunction of the enabledness condition, a relation `a' = b` for each assignment `a := b`, and `c' = c` for any variable `c` that is not explicitly assigned a new value.

The left client enjoys a slightly higher priority than the right client: if the node has the grant, and both the left and the right client request it, the grant will be given to the left client, by transition `GrantLeft`. On the other hand, the node releases the grant after it is released by the right client, even if the left client requests it. This is to make sure that the node does not keep the grant forever: the grant is given at most once to each client before it is released again.

3.3 Associated Transition System

As mentioned before, it is our objective to reason about modules and use the results as lemmas in the proof of properties that use these modules. To do so, we relate modules to transition systems: we define the associated transition system of a module, such that the set of computations of the associated transition system is a superset of the set of computations of any system that includes the module. We say that the set of computations of a module is equal to the set of computations of its associated transition system.

To ensure that the set of computations of a module includes all computations of a system including that module, we cannot make any assumptions about the environment of the module, except that it will not modify the module's private and output variables, and that it will not synchronize with the module's internal transitions. We model this environment by a single transition, called the *environment transition*, τ_{env} with transition relation

$$\rho_{\tau_{env}} : \bigwedge_{v \in V_c \cup V_o \cup V_p} v' = v$$

where V_c are the constants of the module, and V_o and V_p are its output and private variables, respectively.

Given a transition module $\mathbf{M} = \langle I, B \rangle$, with $I = \langle V, T \rangle$ and $B = \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C} \rangle$ we define its *associated transition system*

$$S_{\mathbf{M}} = \langle V_p \cup V, \Theta, \mathcal{T}^*, \mathcal{J} - \mathcal{T}_{exp}, \mathcal{C} - \mathcal{T}_{exp} \rangle$$

where $\mathcal{T}^*$ denotes the set of associated transitions, i.e.,

$$\mathcal{T}^* = \left\{ \tau^* \;\middle|\; \exists \tau \in \mathcal{T}_{int} \,.\, \rho_{\tau^*} = \rho_\tau \wedge \bigwedge_{v \in V_i} v' = v \right\} \cup \mathcal{T}_{exp} \cup \{\tau_{env}\}$$

and $\mathcal{T}_{exp}$ is the set of exported transitions, i.e., transitions in $\mathcal{T}$, that have an exported label

$$\mathcal{T}_{exp} = \{\tau \mid \exists id \in T \,.\, (\tau, id) \in \lambda\}$$

$\mathcal{T}_{int}$ is the set of internal transitions, i.e., transitions in $\mathcal{T}$, that have no exported label

$$\mathcal{T}_{int} = \{\tau \mid \forall id \in T \,.\, (\tau, id) \notin \lambda\}$$

The transition relation of the internal transitions is modified to account for the fact that these transitions, in contrast to the exported transitions, are guaranteed to preserve the values of the input variables, since they cannot synchronize with the environment. The fairness conditions are removed from the exported transitions, because we cannot make any assumptions about the enabling condition of the transitions with which they may synchronize (the enabling condition may be false), and therefore we can no longer assume that a just transition must eventually be taken as long as the local enabling condition continues to hold.

Example: The `Arbiternode` presented in the previous section has three output variables: `grL`, `grR` and `req`, and no private variables. All transitions are internal. The associated transition system therefore consists of the module transitions shown in Figure 2 and the environment transition with the transition relation

$$\rho_{\tau_{env}} : \ \mathtt{grL} = \mathtt{grL}' \wedge \mathtt{grR} = \mathtt{grR}' \wedge \mathtt{req} = \mathtt{req}'$$

Parameterized transition modules. Parameterized transition modules have associated parameterized transition systems: Let $\mathbf{M} : P \mapsto \mathcal{M}$ be a parameterized module, then the associated parameterized transition system $S_{\mathbf{M}} : P \mapsto \mathcal{M}$ maps each parameter value p to the transition system associated with $\mathbf{M}(p)$.

3.4 Module Systems: Definition

We defined the notion of transition modules. In modular verification, however, we are not interested in single modules, but rather in a collection of modules that, together, describe the system behavior. For this purpose we define *module systems.*

Vocabulary We assume a universal set of module identifiers M_{id}. Let $\mathcal{M}$ denote the set of all modules.

Definition 6 Module System. A *module system* $\Psi = \langle \mathbf{Menv}, \mathbf{M_{main}} \rangle$ consists of a module environment $\mathbf{Menv} : M_{id} \mapsto \mathcal{M}$ and a designated main module $\mathbf{M_{main}}$.

3.5 Module Systems: Syntax

Module systems are described by a list of module declarations that define the module environment, followed by a module expression that defines the main module. The module declarations assign modules, also defined by module expressions, to module identifiers.

Module expressions

If $\mathcal{E}$, $\mathcal{E}_1 \ldots \mathcal{E}_n$, are well-formed module expressions, then so are the following:

- $\langle I, B \rangle$, a *direct* module description, defining the interface I and the body B of a transition module.
- $id(e)$, where id is a module identifier, and e is a (possibly empty) list of expressions over constant symbols and variables, indicating a *reference* to another module.
- $(g_1 : \ \mathcal{E}_1); \ldots ; (g_n : \mathcal{E}_n)$, where $g_1 \ldots g_n$ are first-order formulas, denoting a *case distinction*: This allows to describe differently structured modules for different parameter values.

- $(\mathcal{E}_1 \mid\mid \mathcal{E}_2)$. The *parallel composition* operator merges two modules into one module, keeping private variables apart, merging the interfaces, and synchronizing transitions that have the same label.
- **Hide**$(X, \mathcal{E})$, where X is a set of variables, or a set of transition labels. The **Hide** operator removes variables or transition labels from a module's interface. Removing variables from the interface makes them unavailable for reading or writing by the module's environment. Removing transition labels makes the corresponding transitions unavailable for synchronization under that label (a single transition may have multiple labels, so it may still synchronize under other labels).
- **Rename**$(\beta, \mathcal{E})$, where β is a variable substitution $\beta : \mathcal{V} \mapsto \mathcal{V}$, or a transition label substitution $\beta : \mathcal{T}_{id} \mapsto \mathcal{T}_{id}$. The **Rename** operator renames variables or transition labels in the interface.
- **Augment**$(\beta, \mathcal{E})$ where β is a mapping from variables to expressions over variables and constants.
  The purpose of the augmentation operation is to create new variables in the interface that maintain the same value as the corresponding value of the expression. To ensure that the module can maintain these values, the expression can be defined over private and output variables only.
- **Restrict**$(\beta, \mathcal{E})$, where β is a mapping from variables to expressions over variables and constants.
  The purpose of the restrict operation is to replace input variables in the interface by expressions over other input variables.

Module Systems

If $\mathcal{E}_1 \ldots \mathcal{E}_n, \mathcal{E}_{main}$ are well-formed module expressions, $id_1 \ldots id_n$ are module identifiers and $P_1 \ldots P_n$ are (possibly empty) lists of formal parameters, then

$$id_1(P_1) = \mathcal{E}_1 \; ; \; \ldots \; ; \; id_n(P_n) = \mathcal{E}_n \; ; \; \mathcal{E}_{main}$$

is a well-formed module system.

3.6 Module Systems: Semantics

A description of a module system defines both a modular environment **Menv** and a main module. We will first define the semantics of module expressions, assuming the existence of a modular environment **Menv**. The semantics of a module system will be defined at the end of this section.

Module Expressions

A module expression $\mathcal{E}$ denotes a transition module. To be able to resolve references to other modules and to evaluate guards, the meaning of module expressions is relative to a module environment **Menv** and variable environment **Venv**. In the following we assume that these are given.

Direct descriptions The semantics of module expressions is defined inductively. As the base case we have the expression that describes a module directly; in this case

$$[\![\mathcal{E}]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T\rangle, \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C}\rangle\rangle$$

Reference If the expression is a reference to another module, that is $\mathcal{E} = id(A)$, then the expression is well-defined if the module environment **Menv** assigns a parameterized module **M** to *id*, and A is type consistent with **M**'s parameters. Then:

$$[\![id(A)]\!]_{\mathbf{Menv},\mathbf{Venv}} = \mathbf{M}([\![A]\!]_{\mathbf{Venv}})$$

Definitions and Conventions Before we define the semantics of the other operations we will introduce some definitions and conventions. To ensure that the result of composing two modules is again a transition module, we have to impose some conditions on their interfaces, in particular that they do not make inconsistent assumptions about their environments. We also require that transitions that will be synchronized in the composition have the same fairness conditions.

Definition 7 Compatible Modules. Two modules are compatible if:

1. Their interfaces $I_1 = \langle V_1, T_1\rangle$ and $I_2 = \langle V_2, T_2\rangle$ are compatible, that is, an output variable of one module is not a shared or output variable of the other module.
2. Their exported transitions $\mathcal{T}_{exp,1}$ and $\mathcal{T}_{exp,2}$ have compatible fairness conditions. That is, for $\tau_1 \in \mathcal{T}_{exp,1}$ and $\tau_2 \in \mathcal{T}_{exp,2}$ with the same label, we require that $\tau_1 \in \mathcal{J}_1 \leftrightarrow \tau_2 \in \mathcal{J}_2$ and $\tau_1 \in \mathcal{C}_1 \leftrightarrow \tau_2 \in \mathcal{C}_2$.

In the definition of the operators we frequently will have to rename part or all of the variables. In the definitions we will use the following convention. Given an expression $\mathcal{E}(v_1, \ldots, v_n)$ and a variable renaming function $\alpha : \mathcal{V} \mapsto \mathcal{V}$, we denote by $\alpha(\mathcal{E}(v_1, \ldots, v_n)$ the expression $\mathcal{E}(\alpha(v_1), \ldots, \alpha(v_2))$. We assume that for every $v \in \mathcal{V}$ if α maps v into $\hat{v}$, then it also maps v' into $\hat{v}'$. We will occasionally write $\alpha(\mathcal{T})$ to represent the set of transitions such that all variables and primed variables in the transition relation are renamed according to α.

Case distinction Let $\mathcal{E}_1 \ldots \mathcal{E}_n$ be well-defined module expressions denoting the modules

$$[\![\mathcal{E}_1]\!]_{\mathbf{Menv},\mathbf{Venv}}, \ldots, [\![\mathcal{E}_n]\!]_{\mathbf{Menv},\mathbf{Venv}} = \mathbf{M}_1, \ldots, \mathbf{M}_n$$

and $g_1, \ldots, g_n$ be first-order formulas. The expression $g_1 : \mathcal{E}_1 \ldots g_n : \mathcal{E}_n$ is well-defined if

- $\mathbf{M}_1 \ldots \mathbf{M}_n$ have identical interfaces, and
- the free variables of $g_1 \ldots g_n$ do not appear in the input, output, shared or private variables of $\mathbf{M_1} \ldots \mathbf{M_n}$, and

- for every variable environment **Venv** there exists exactly one i, $1 \leq i \leq n$ such that $[\![g_i]\!]_{\mathbf{Venv}}$ is true.

If well-defined, the module expression $g_1 : \mathcal{E}_1 \dots g_n : \mathcal{E}_n$ denotes the module

$$[\![g_1 : \mathcal{E}_1 \dots g_n : \mathcal{E}_n]\!]_{\mathbf{Menv},\mathbf{Venv}} = \begin{cases} \mathbf{M}_1 & \text{if } [\![g_1]\!]_{\mathbf{Venv}} \text{ is true} \\ \dots \\ \mathbf{M}_n & \text{if } [\![g_n]\!]_{\mathbf{Venv}} \text{ is true} \end{cases}$$

Parallel composition Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two well-defined module expressions denoting

$$[\![\mathcal{E}_1]\!]_{\mathbf{Menv},\mathbf{Venv}} = \mathbf{M}_1 = \langle\langle V_1, T_1\rangle, \langle V_{p,1}, \Theta_1, \mathcal{T}_1, \lambda_1, \mathcal{J}_1, \mathcal{C}_1\rangle\rangle$$
$$[\![\mathcal{E}_2]\!]_{\mathbf{Menv},\mathbf{Venv}} = \mathbf{M}_2 = \langle\langle V_2, T_2\rangle, \langle V_{p,2}, \Theta_2, \mathcal{T}_2, \lambda_2, \mathcal{J}_2, \mathcal{C}_2\rangle\rangle$$

Then $[\![\mathcal{E}_1 \mid\mid \mathcal{E}_2]\!]_{\mathbf{Menv},\mathbf{Venv}}$ is well-defined if the interfaces $\langle V_1, T_1\rangle$ and $\langle V_2, T_2\rangle$ are compatible. If well-defined, the expression $[\![\mathcal{E}_1 \mid\mid \mathcal{E}_2]\!]_{\mathbf{Menv},\mathbf{Venv}}$ denotes

$$[\![\mathcal{E}_1 \mid\mid \mathcal{E}_2]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T\rangle, \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C}\rangle\rangle$$

where

- *Interface Variables:* $V = V_1 \cup V_2$. The partitioning of V into input, output, and shared variables is shown in Figure 3, where **X** denotes combinations that are not allowed.

	$v_2 \in$		
	$V_{i,2}$	$V_{o,2}$	$V_{s,2}$
$V_{i,1}$	V_i	V_o	V_s
$v_1 \in V_{o,1}$	V_o	**X**	**X**
$V_{s,1}$	V_s	**X**	V_s

Fig. 3. Combination of interface variables.

- *Exported transitions:* $T = T_1 \cup T_2$
- *Private variables:* Since we do not require $V_{p,1}$ and $V_{p,2}$ to be disjoint, we have to rename private variables to ensure that private variables of different modules are not identified with each other. To do so we let V_p be the disjoint union of $V_{p,1}$ and $V_{p,2}$ and define α_1 to be the mapping that maps every variable from $V_{p,1}$ into the corresponding variable of V_p, and maps all other variables to themselves; α_2 is defined similarly. We assume that $V_p \cap V = \emptyset$. So we have
$$V_p = V_{p,1} \dot{\cup} V_{p,2} = \alpha_1(V_{p,1}) \cup \alpha_2(V_{p,2})$$

- *Initial Condition:* Let α_i be the renaming functions defined before. The initial condition of the composition is the conjunction of the two initial conditions after appropriately renaming the private variables:

$$\Theta = \alpha_1(\Theta_1) \wedge \alpha_2(\Theta_2)$$

- *Transitions:* The new set of transitions is given by

$$\mathcal{T} = \mathcal{T}_{1,p} \cup \mathcal{T}_{2,p} \cup \mathcal{T}_{syn}$$

where $\mathcal{T}_{1,p}$, or $\mathbf{M}_1$'s private transitions, are the transitions from $\mathbf{M_1}$ that do not synchronize with transitions of module $\mathbf{M}_2$, and similarly for $\mathcal{T}_{2,p}$, and $\mathcal{T}_{syn}$ contains the result of synchronizing those transitions in $\mathbf{M}_1$ and $\mathbf{M}_2$ whose labels appear in both interfaces. Variables in the transition relations are renamed according to the renaming functions α_i as before. For internal transitions, a conjunct is added to the transition relation stating that the transition does not modify the private variables originating from the other module. Formally, for $(i,j) = (1,2),(2,1)$:

$$\mathcal{T}_{i,p} = \left\{ \tau \;\middle|\; \begin{array}{c} \exists \tau^* \in \mathcal{T}_i \,.\, \exists id \in \mathcal{T}_{id} \,. \\ ((\tau^*, id) \in \lambda_i \wedge id \notin id_{syn}) \\ \wedge \\ \rho_\tau = \alpha_i(\rho_{\tau^*}) \wedge \alpha_j(\bigwedge_{v \in V_{j,p}} v' = v) \end{array} \right\}$$

where id_{syn} is the set of labels that are exported by both modules, that is

$$id_{syn} = T_1 \cap T_2$$

The set of synchronized transitions is described by

$$\mathcal{T}_{syn} = \left\{ \tau \;\middle|\; \begin{array}{c} \exists id \in id_{syn}, \tau_1 \in \mathcal{T}_1, \tau_2 \in \mathcal{T}_2 \,. \\ (\tau_1, id) \in \lambda_1 \wedge (\tau_2, id) \in \lambda_2 \wedge \rho_\tau = (\alpha_1(\rho_{\tau_1}) \wedge \alpha_2(\rho_{\tau_2})) \end{array} \right\}$$

Note that if a transition τ has a label that synchronizes and a label that does not synchronize, the composed module will contain the synchronized transition as well as the unsynchronized version.
- *Labeling function:*

$$\lambda = \left\{ (\tau, id) \;\middle|\; \left(\begin{array}{c} \exists \tau^* \left[\begin{array}{c} ((\tau^*, id) \in \lambda_1 \wedge \rho_\tau = \alpha_1(\rho_{\tau^*}) \wedge \alpha_2(\bigwedge_{v \in V_{2,p}} v' = v)) \\ \vee \\ ((\tau^*, id) \in \lambda_2 \wedge \rho_\tau = \alpha_2(\rho_{\tau^*}) \wedge \alpha_1(\bigwedge_{v \in V_{1,p}} v' = v)) \end{array} \right] \\ \wedge \;\; id \notin id_{syn} \\ \vee \left(\begin{array}{c} \exists \tau_1, \tau_2 \,.\, (\tau_1, id) \in \lambda_1 \wedge (\tau_2, id) \in \lambda_2 \wedge \\ \rho_\tau = (\alpha_1(\rho_{\tau_1}) \wedge \alpha_2(\rho_{\tau_2})) \wedge id \in id_{syn} \end{array} \right) \end{array} \right) \right\}$$

- *Fairness conditions:* Since we are assuming that transitions can synchronize only if their fairness conditions are the same, we can take the union of the two sets, accounting for the renaming of the transition relations:

$$\mathcal{J} = \alpha_1(\mathcal{J}_1) \cup \alpha_2(\mathcal{J}_2)$$
$$\mathcal{C} = \alpha_1(\mathcal{C}_1) \cup \alpha_2(\mathcal{C}_2)$$

Hiding Let $\mathcal{E}$ be a well-defined module expression denoting

$$[\![\mathcal{E}]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle I = \langle V, T\rangle, B = \langle V_p, \ldots\rangle\rangle$$

and X a set of variables. Then

$$[\![\mathbf{Hide}(X, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V - X, T\rangle, \langle V_p \cup X, \ldots\rangle\rangle$$

If X is a set of transition labels then

$$[\![\mathbf{Hide}(X, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T - X\rangle, B\rangle$$

Renaming Let $\mathcal{E}$ be a well-defined module expression denoting

$$[\![\mathcal{E}]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T\rangle, \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C}\rangle\rangle$$

and $\beta : \mathcal{V} \mapsto \mathcal{V}$ a function that maps variables into variables. α is a renaming on the private variables that ensures that V_p and V are still disjoint after the renaming. If for some interface variable $v \in V$ and private variable $w \in V_p$, $\beta(v) = w$, then $\alpha(w) = z$, where z is a new variable, not present in the interface or in the private variables. Then

$$\begin{aligned}&[\![\mathbf{Rename}(\beta, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \\ &\quad \langle\langle \beta(\alpha(V)), T\rangle, \langle V_p, \beta(\alpha(\Theta)), \beta(\alpha(\mathcal{T})), \lambda, \beta(\alpha(\mathcal{J})), \beta(\alpha(\mathcal{C}))\rangle\rangle\end{aligned}$$

If $\beta : \mathcal{T}_{id} \mapsto \mathcal{T}_{id}$ is a function that maps transition labels into transition labels, then

$$[\![\mathbf{Rename}(\beta, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, \beta(T)\rangle, \langle V_p, \Theta, \mathcal{T}, \beta(\lambda), \mathcal{J}, \mathcal{C}\rangle\rangle$$

where $(\tau, id) \in \beta(\lambda)$ iff $\exists id^* \,.\, id = \beta(id) \wedge (\tau, id^*) \in \lambda$.

Augmentation Let $\mathcal{E}$ be a well-defined module expression denoting

$$[\![\mathcal{E}]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T\rangle, \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C}\rangle\rangle$$

and β a partial function mapping variables into expressions over output variables. Again α is a renaming of the private variables that keeps V and V_p disjoint.

$$[\![\mathbf{Augment}(\beta, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V \cup dom(\beta), T\rangle, \langle \alpha(V_p), \Theta^*, \mathcal{T}^*, \lambda, \mathcal{J}^*, \mathcal{C}^*\rangle\rangle$$

where the variables in $dom(\beta)$ are added to the output variables. A constraint on the new variables is added to the initial condition:

$$\Theta^* = \alpha(\Theta) \wedge \bigwedge_{v \in dom(\beta)} v = \beta(v)$$

and all transition relations are augmented to update of the newly added variables, that is

$$\mathcal{T}^* = \left\{ \tau^* \;\middle|\; \exists \tau \in \mathcal{T} \,.\, \rho_{\tau^*} = \left(\alpha(\rho_\tau) \wedge \bigwedge_{v \in dom(\beta)} v' = \beta(v') \right) \right\}$$

$\mathcal{J}^*$ and $\mathcal{C}^*$ are defined analogously.

Restriction Let $\mathcal{E}$ be a module expression denoting $[\![\mathcal{E}]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T\rangle, \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C}\rangle\rangle$, V_n a set of fresh variables, and β a partial function, mapping input variables into expressions over variables in V_n. Then

$$[\![\mathbf{Restrict}(\beta, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \\ \langle\langle (V - dom(\beta) \cup V_n, T\rangle, \langle \alpha(V_p), \beta(\alpha(\Theta)), \beta(\alpha(\mathcal{T})), \lambda, \beta(\alpha(\mathcal{J})), \beta(\alpha(\mathcal{C}))\rangle\rangle$$

where the variables in V_n are added to the input variables. β is applied to the initial condition and all transition relations. As before, α denotes the renaming of the private variables necessary to keep V_p and V disjoint.

Module Systems

A module system is described by a list of equations of the form $id(P_i) = \mathcal{E}_i$ defining the modular environment, followed by an expression $\mathcal{E}_{\text{main}}$ defining the *main* module. The modular environment **Menv** is defined as follows,

$$\mathbf{Menv} = \text{lfp}\left(\lambda\, \mathbf{Menv}^* \,.\, \begin{cases} id_1 \mapsto \lambda X \,.\, [\![\mathcal{E}_1]\!]_{\mathbf{Menv}^*,\mathbf{Venv}[P_1 \backslash X]} \\ id_2 \mapsto \lambda X \,.\, [\![\mathcal{E}_2]\!]_{\mathbf{Menv}^*,\mathbf{Venv}[P_2 \backslash X]} \\ \ldots \end{cases}\right)$$

where lfp denotes the least fixpoint. The main module $\mathbf{M}_{\text{main}}$ is the interpretation of $\mathcal{E}_{\text{main}}$ in this environment:

$$\mathbf{M}_{\text{main}} = [\![\mathcal{E}_{\text{main}}]\!]_{\mathbf{Menv},\mathbf{Venv}}$$

In the remainder we assume that a given module system is well-defined, that is, the environment has a unique least fixed point.

3.7 Example: Arbiter

Continuing the arbiter example, we now describe the full hierarchical arbiter. The `Arbiter` module is composed from an `Arbitertree` and a module named `Top` that gives and takes grants. `Arbitertree` is a tree of `Arbiternode`s that were defined in Figure 2. Both `Arbiter` and `Arbitertree` are parameterized by their height h.

An `Arbitertree` of height h communicates with 2^h clients, who can each request access by setting a *request* bit. One client at a time will be given access to the resource, and the `Arbiter` informs the client about its granted access by setting the client's grant bit. The leafs of the tree are defined by an expression over `Arbiternode` (++ denotes bit-vector concatenation):

```
Leafnode = Hide(grL, grR,
               Augment(grants = grL ++ grR,
                  Restrict(reqL = requests[0], reqR = requests[1],
                     Arbiternode))
```

The **Restrict** operation instantiates its input variables `reqL` and `reqR` with the actual request bits of the clients, and the `Augment` operation combines the two output variables `grL` and `grR` into a single variable `grants`. After the augmentation, the output variables `grL` and `grR` can be hidden. Thus the interface of `Leafnode` is

V_i:	gr: **bool** requests: **bitvector**[**2**]
V_o:	req: **bool** grants: **bitvector**[**2**]

The parameterized module `Arbitertree` is described by the module expression

```
Arbitertree(h)=

    h = 1: LeafNode

    h > 1: Hide(grantsL, grantsR, grL, grR, reqL, reqR,
              Augment(grants = grantsL ++ grantsR,
                 (Restrict(requests = requests [0 : 2^{h-1} − 1],
                    Rename(gr = grL, req = reqL, grants = grantsL,
                       ArbiterTree(h-1)))
                 ||
                 Arbiternode
                 ||
                 Restrict(requests = requests [2^{h-1} : 2^h − 1],
                    Rename(gr = grR, req = reqR, grants = grantsR,
                       Arbitertree(h-1))))))
```

Each instance has the interface

V_i:	gr: **bool** requests: **bitvector**$[0..2^h - 1]$
V_o:	req: **bool** grants: **bitvector**$[0..2^h - 1]$

For any given h, the parameterized module `Arbitertree` describes a tree of height h. The module expression is illustrated by Figure 4, which shows the three modules that are composed and their input and output variables. Note that the **Augment** and **Restrict** operations are necessary to obtain identical interfaces for the cases $h = 1$ and $h > 1$.

We complete our description of a hierarchical arbiter by defining the `Arbiter` module, the main module of the system. An `Arbitertree` of height h guarantees mutual exclusion among the 2^h clients, but the tree will only give a grant to

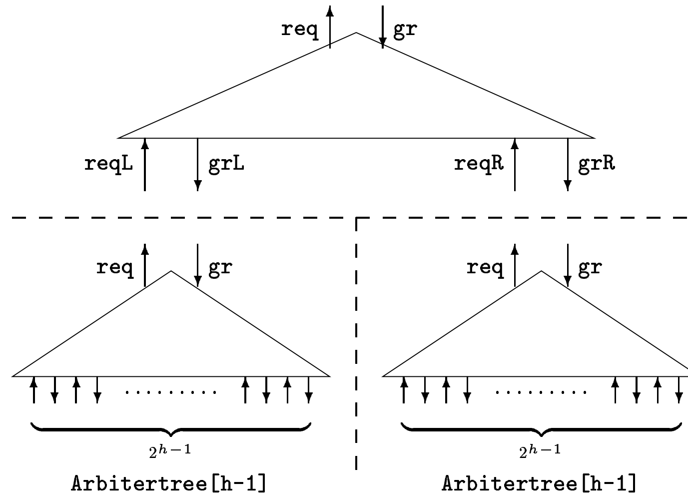

Fig. 4. Composition of `Arbiternode` and a left and right subtree.

```
Module Top:

out gr : bool where gr=false
external in req :bool

Transition Grant Just:
    enable !gr /\ req
    assign gr:=true

Transition Retract Just:
    enable gr /\ !req
    assign gr:=false

EndModule
```

Fig. 5. Module `Top`.

some client if it has received the grant from its parent entity. This parent entity is represented by the module `Top`, shown in Figure 5.

`Top`'s only actions are to award a grant when one is requested and retract a grant when one is released.

The main module is described as an instance `Arbiter`(h) of the parameterized `Arbiter`module, which is defined as follows:

`Arbiter(h)` =	**Hide**(`req, gr`, (`Arbitertree`(h) || `Top`))

and has interface

V_i:	`requests`: **bitvector**$[2^h]$
V_o:	`grants`: **bitvector**$[2^h]$

4 Deductive Verification

In the previous sections we have developed a formalism for modular system descriptions. In this section we now move on to the verification of such systems. We begin with an introduction to available formalisms for the *global* verification of transition systems in Section 4.1. Next, we extend the notion of (global) program validity of temporal formulas to *modular* validity. For systems with non-recursive descriptions we give a proof rule in Section 4.2. While this is a feasible approach to establish the modular validity, we are interested in methods that make use of the structure given by module descriptions. In Section 4.3 we discuss how modular properties can be *inherited* by other modules, and in Section 4.4 we define module abstraction. Finally, in Section 4.5, we discuss the verification of recursively described modules.

4.1 Verification of Transition Systems

The classical deductive framework for the verification of fair transition systems is based on *verification rules*, which reduce temporal properties of systems to first-order or simpler temporal premises [MP95b].

For a past formula φ,

$$\begin{array}{l} 1.\ \mathcal{S} \models \Theta \rightarrow \varphi \\ 2.\ \mathcal{S} \models \{\varphi\}\, \mathcal{T}_\mathcal{S}\, \{\varphi\} \\ \hline \quad \mathcal{S} \models \Box\, \varphi \end{array}$$

Fig. 6. Invariance rule INV.

Figure 6 presents the *invariance rule*, INV, which can be used to establish the $\mathcal{S}$-validity of formulas of the form $\Box\, p$, where p is a past formula. Here $\{\varphi\}\, \mathcal{T}_\mathcal{S}\, \{\varphi\}$ stands for $\Box(\rho_\tau \wedge \varphi \rightarrow \varphi')$ for all transitions $\tau \in \mathcal{T}_\mathcal{S}$. An invariant φ may not be inductive (that is, it may not be preserved by all transitions), in which case it can be necessary to find a stronger, inductive invariant that implies φ, and prove it first. An alternative approach is to first prove a set of

simpler invariants $p_1, ..., p_k$ and then use them to establish the more complicated invariant φ.

Graphical formalisms can facilitate the task of guiding and understanding a deductive proof. *Verification diagrams* [MP94, BMS95] provide a graphical representation of the verification conditions needed to establish a particular temporal formula over a transition system. In this paper we will use generalized verification diagrams [BMS95, BMS96] to prove response properties.

Generalized Verification Diagrams A verification diagram is a graph, with nodes labeled by assertions and propositions and edges labeled by sets of transitions, that represents a proof that a transition system $\mathcal{S}$ satisfies a temporal property φ. A subset of the nodes is marked as initial. First-order verification conditions associated with the diagram prove that the diagram faithfully represents all computations of $\mathcal{S}$. The edge labeling is used to express the fairness properties of the transitions relevant to the property.

Some of the first-order verification conditions generated by the diagram are as follows (for a full description, see [BMS95]):

- *Initiation*: At least one initial node satisfies the initial condition of $\mathcal{S}$.
- *Consecution*: Any τ-successor of a state that satisfies the assertion of a node n, must satisfy the assertion of some successor node of n.
- *Fairness*: If an edge is labeled with a transition, that transition is guaranteed to be enabled.

To show that the diagram satisfies φ can be checked algorithmically, by viewing the diagram as an automaton (considering its propositional labeling only) and checking that its language is included in the language of the formula. Multiple diagrams can be combined such that the intersection of their (propositional) languages is included in the language of the formula [BMS96].

4.2 Verification of Modular Properties

The goal of modular verification is to reduce the task of verifying a system as a whole to the verification of modules. In this section we will define *modular validity* and we will describe a *proof rule* to establish modular properties.

We use the notion of associated transition systems, introduced in Section 3.3, to define the modular validity of a temporal property:

Definition 8 Modular Validity. We say that a property φ is *modularly valid*, or $\mathbf{M}$-valid for a module $\mathbf{M}$, denoted by

$$\mathbf{M} \models \varphi,$$

if φ is valid over the transition system $\mathcal{S}_{\mathbf{M}}$ associated with $\mathbf{M}$.

The set of computations of the associated transition system includes any computation of a system that contains the module. A modular property is therefore valid over any system that contains the module.

Definition 9 Module descriptions in normal form. A module description is in *normal form* if it is either a direct description or a case distinction where all subexpressions are direct descriptions.

Figure 7 presents a proof rule that reduces modular validity to a set of system validities, based on a case distinction on the guards. The rule requires the module description to be in normal form. Note that any non-recursive description can be transformed into normal form, by first expanding the references and then reducing module expressions to direct descriptions.

For a module $\mathbf{M}$, described in normal form

$$g_1 : M_1 \ldots g_n : M_2$$

and a temporal formula φ,

$$\frac{\mathcal{S}_{[\![M_i]\!]} \models g_i \rightarrow \varphi \qquad \text{for } i = 1 \ldots n}{\mathbf{M} \models \varphi}$$

Fig. 7. Modular validity rule MOD.

4.3 Property Inheritance

Rule MOD of Figure 7 allows us to prove the modular validity of properties. The obvious limitation of the rule lies in the requirement that the module description be in normal form: transforming the description into normal form means that all structural information is lost. The *inheritance* proof rules shown in Figure 8, by contrast, make explicit use of this structure. Property inheritance allows us to use properties that were previously proven to be valid over other modules as lemmas in a modular proof.

Example: In the `Arbiter` example, assume we have shown that `Arbiternode` establishes mutual exclusion between its two clients:

$$\texttt{Arbiternode} \models \Box \neg(\texttt{grL} \wedge \texttt{grR})$$

`Leafnode` is described in terms of `Arbiternode`. It *inherits* the corresponding property

$$\texttt{Leafnode} \models \Box \neg(\texttt{grants[0]} \wedge \texttt{grants[1]})$$

which is in turn inherited by `Arbitertree`(1):

$$\texttt{Arbitertree}(1) \models \Box \neg(\texttt{grants[0]} \wedge \texttt{grants[1]})$$

For module expressions M, N, a mapping β on variables, a mapping t on transition identifiers, and a temporal formula φ,

$$\frac{[\![M]\!] \models \varphi}{[\![M || N]\!] \models \alpha_M(\varphi)} \qquad \frac{[\![N]\!] \models \varphi}{[\![M || N]\!] \models \alpha_N(\varphi)}$$

$$\frac{[\![M]\!] \models \varphi}{[\![\mathbf{Hide}(M, X)]\!] \models \varphi} \qquad \frac{[\![M]\!] \models \varphi}{[\![\mathbf{Hide}(M, T)]\!] \models \varphi}$$

$$\frac{[\![M]\!] \models \varphi}{[\![\mathbf{Rename}(M, \beta)]\!] \models \beta(\alpha(\varphi))} \qquad \frac{[\![M]\!] \models \varphi}{[\![\mathbf{Rename}(M, t)]\!] \models \varphi}$$

$$\frac{[\![M]\!] \models \varphi}{[\![\mathbf{Augment}(M, \beta)]\!] \models \alpha(\varphi)} \qquad \frac{[\![M]\!] \models \beta(\varphi)}{[\![\mathbf{Augment}(M, \beta)]\!] \models \alpha(\varphi)}$$

$$\frac{[\![M]\!] \models \varphi}{[\![\mathbf{Restrict}(M, \beta)]\!] \models \beta(\alpha(\varphi))} \qquad \frac{[\![M]\!]([\![A]\!]) \models \varphi}{[\![M(A)]\!] \models \varphi}$$

$$\frac{[\![M_i]\!] \models g_i \rightarrow \varphi \qquad \text{for } i = 1 \ldots n}{[\![g_1 : M_1 \ldots g_n : M_n]\!] \models \varphi}$$

Fig. 8. Property inheritance for various operators.

The inheritance rules for the different module operators in Figure 8 can be justified by showing that a refinement mapping [AL88] exists between the transition systems associated with the modules given in the premise and those in the conclusion. We consider refinement mappings that are induced by a substitution relation:

Definition 10 Refinement. Let S^A and S^C be two transition systems, called the *abstract* and *concrete* transition system, respectively, and $\alpha : V^A \mapsto E(V^C)$ a substitution relation mapping variables from S^A to expressions over variables in S^C. The transition system S^C is an α-refinement of S^A, denoted $S^C \sqsubseteq_\alpha S^A$, if for every computation σ^C of S^C there exists a computation σ^A of S^A such that $\sigma^C = \alpha(\sigma^A)$, where α is extended to a mapping on computations in the obvious way.

The proof rule in Figure 9 states that if S^C is an α-refinement of S^A, properties of S^A are inherited by S^C.
Justification: Assume $S^A \models \varphi$, and $S^C \sqsubseteq_\alpha S^A$. Let σ^C be a computation of S^C. By the definition of refinement, there exists a computation σ^A of S^A

For two transition systems $\mathcal{S}_1,\ \mathcal{S}_2$
and a temporal formula φ,

$$\frac{\begin{array}{l}\text{IH1. } \mathcal{S}^C \sqsubseteq_\alpha S^A \\ \text{IH2. } \mathcal{S}^A \models \varphi\end{array}}{\mathcal{S}^C \models \alpha(\varphi)}$$

Fig. 9. General inheritance rule G-INH.

such that $\sigma^C = \alpha(\sigma^A)$. Because $S^A \models \varphi$ we have in particular $\sigma^A \models \varphi$, and thus $\alpha(\sigma^A) \models \alpha(\varphi)$ as required.

Figure 10 presents a proof rule to establish a refinement relation between two transition systems. The rule assumes the existence of a surjective transition mapping $\gamma : \mathcal{T}^C \mapsto \mathcal{T}^A$ that maps each concrete transition to a corresponding abstract transition with the same or weaker fairness condition.

For two transition systems S^C, S^A and a surjective function $\gamma : \mathcal{T}^C \mapsto \mathcal{T}^A$,
such that $\tau^c \in \mathcal{J}^C$ implies $\gamma(\tau^c) \notin \mathcal{C}^A$,
and $\tau^c \in \mathcal{T}^C - (\mathcal{J}^C \cup \mathcal{C}^C)$ implies $\gamma(\tau^c) \notin \mathcal{J}^A \cup \mathcal{C}^A$,

$$\frac{\begin{array}{lll}\text{R1. } \Theta^C \rightarrow \alpha(\Theta^A) & & \\ \text{R2. } \rho_{\tau^C} \rightarrow \alpha(\rho_{\gamma(\tau^C)}) & \text{for every } \tau^C \in \mathcal{T}^C & \\ \text{R3. } \alpha(En_{\gamma(\tau^C)}) \rightarrow En_{\tau^C} & \text{for every } \tau^C \in \mathcal{T}^C & \text{with } \gamma(\tau^C) \in \mathcal{J}^A \cup \mathcal{C}^A\end{array}}{S^C \sqsubseteq_\alpha S^A}$$

Fig. 10. Basic refinement rule B-REF.

Justification: Assume that $\sigma^C = s_o, s_1, \ldots$ is a computation of S^C. We have to show that the premises ensure there exists a corresponding computation $\sigma^A = \alpha(s_0), \alpha(s_1), \ldots$ of the abstract system S^A. By R1 and $s_0 \models \Theta^C$ we have $\alpha(s_0) \models \Theta^A$. For every two consecutive states s_i, s_{i+1} in σ^C, the transition relation of some concrete transition τ_c must be satisfied; by R2 the transition relation of the corresponding abstract transition $\gamma(\tau^C)$ is satisfied for states $\alpha(s_i), \alpha(s_{i+1})$. It remains to show that σ^A is fair. Since γ is onto, there exists for every fair transition τ^A a transition τ^C with equal or stronger fairness, and thus by R2 and R3 τ^A's fairness conditions can only be violated if τ^C's fairness conditions are violated.

Clearly, refinement under a transition mapping γ, denoted by $\sqsubseteq^\gamma$, is a stronger property than refinement alone. However, it suffices for our purposes, and results in simpler verification conditions than, for example, the more gen-

eral proof rule presented in [KMP94], where proving refinement is reduced to the proof of a number of temporal properties.

4.4 Module Abstraction

It often is the case that some components of a module are irrelevant to the validity of a property to be proven. Module abstraction allows us to ignore some or all of the details of those components in the proof, thus simplifying the proof.

The idea is that in a module expression, subexpressions can be replaced by other expressions denoting simpler modules, provided those modules *modularly simulate* the original module, that is, the new module can simulate the original module in any expression.

Definition 11 Modular simulation. A module $[\![M^A]\!]$ *simulates* a module $[\![M^C]\!]$, denoted by $[\![M^C]\!] \sqsubseteq [\![M^A]\!]$, if for all modular expressions $\mathcal{E}(M)$,

$$S_{[\![\mathcal{E}(M^C)]\!]} \sqsubseteq S_{[\![\mathcal{E}(M^A)]\!]}$$

A proof rule to establish modular simulation between two modules is shown in Figure 11.

Justification Consider two modules $[\![M^C]\!]$ and $[\![M^A]\!]$ with identical interface I. Assume a surjective transition mapping $\gamma : \mathcal{T}^C \to \mathcal{T}^A$ between the sets of transitions that fulfills the condition S1 - S3 (which are identical to the premises R1-R3 in rule B-REF, with α being the identity) and that is consistent with the transition labeling, expressed by premises S4 and S5 respectively: each exported label of a concrete transition τ is also a label of $\gamma(\tau)$, and if a concrete transition τ has an internal label, then so does $\gamma(\tau)$.

For two modules $\mathbf{M}^C, \mathbf{M}^A$ with a common interface,
and a surjective function $\gamma : \mathcal{T}^C \mapsto \mathcal{T}^A$,
such that $\tau^c \in \mathcal{J}^C$ implies $\gamma(\tau^c) \notin \mathcal{C}^A$,
and $\tau^c \in \mathcal{T}^C - (\mathcal{J}^C \cup \mathcal{C}^C)$ implies $\gamma(\tau^c) \notin \mathcal{J}^A \cup \mathcal{C}^A$,

S1. $\Theta^C \to \Theta^A$
S2. $\rho_{\tau^C} \to \rho_{\gamma(\tau^C)}$ for every $\tau^C \in \mathcal{T}^C$
S3. $En_{\gamma(\tau^C)} \to En_{\tau^C}$ for every $\tau^C \in \mathcal{T}^C$ with $\gamma(\tau^C) \in \mathcal{J}^A \cup \mathcal{C}^A$
S4. $\forall l \in T \,.\, \lambda^C(\tau^C, l) \to \lambda^A(\gamma(\tau^C), l)$ for every $\tau^C \in \mathcal{T}^C$
S5. $\begin{pmatrix} \exists l^C \in T_{id} - T \,.\, \lambda^C(\tau^A, l^C) \\ \to \\ \exists l^A \in T_{id} - T \,.\, \lambda^A(\gamma(\tau^C), l^A) \end{pmatrix}$ for every $\tau^C \in \mathcal{T}^C$

$\mathbf{M}^C \sqsubseteq \mathbf{M}^A$

Fig. 11. Modular simulation rule M-SIM.

We show by induction on expressions that $S_{[\![\mathcal{E}(M^C)]\!]} \sqsubseteq S_{[\![\mathcal{E}(M^A)]\!]}$. For the base case we have to show $S_{[\![M^C]\!]} \sqsubseteq S_{[\![M^A]\!]}$. As the modules $[\![M_A]\!]$ and $[\![M_C]\!]$ have identical interfaces, we can extend γ to a transition mapping γ' on the associated transition systems as follows:

$$\gamma'(\tau) = \begin{cases} \tau \ \text{ if } \tau = \tau_{env} \\ \tau' \text{ if } \rho_\tau = \rho_{\tau_0} \wedge \bigwedge_{v \in V_i} v = v' \text{ and } \rho_{\tau'} = \rho_{\gamma(\tau_0)} \wedge \bigwedge_{v \in V_i} v = v' \\ \quad \text{for some transition } \tau_0 \in \mathcal{T}_{[\![M^C]\!]}. \\ \tau' \text{ if } \rho_\tau = \rho_{\tau_0} \text{ and } \rho_{\tau'} = \rho_{\gamma(\tau_0)} \\ \quad \text{for some transition } \tau_0 \in \mathcal{T}_{[\![M^C]\!]}. \end{cases}$$

For the inductive step, for each of the operations we can show that given $[\![M^C]\!] \sqsubseteq^\gamma [\![M^A]\!]$, there is a transition mapping γ', such that $[\![\mathcal{E}(M^C)]\!] \sqsubseteq^{\gamma'} [\![\mathcal{E}(M^A)]\!]$. Hence, $[\![M^C]\!] \sqsubseteq^\gamma [\![M^A]\!]$ implies that there is a transition mapping γ'', such that $S_{[\![\mathcal{E}(M^C)]\!]} \sqsubseteq^{\gamma''} S_{[\![\mathcal{E}(M^A)]\!]}$.

The proof rule M-INH in Figure 12, a specialization of the general proof inheritance rule shown in Figure 9, allows us to replace modules in an expression by simpler modules that simulate them.

For modular expressions M, N, and $\mathcal{E}(M)$,

M1. $[\![\mathcal{E}(N)]\!] \models \varphi$
M2. $[\![M]\!] \sqsubseteq [\![N]\!]$
———————————
$[\![\mathcal{E}(M)]\!] \models \varphi$

Fig. 12. Modular inheritance proof rule M-INH.

Interface Abstraction For each class of modules with the same interface there is a largest element with respect to the simulation preorder, called the *interface abstraction*, which can be generated automatically.

Definition 12 Interface Abstraction. Let $\mathbf{M} = \langle I, B\rangle$ be a module with interface $I = \langle V, T\rangle$. The *interface abstraction* of $\mathbf{M}$ is the module $\mathbf{A_M} = \langle I, B^*\rangle$ where $B^* = \langle V_p^* = \emptyset, \Theta^* = true, \mathcal{T}^* = \{\tau_a\}, \lambda^* = \{\tau_a\} \times (T \cup \{l_{proxy}\}), \mathcal{J}^* = \emptyset, \mathcal{C}^* = \emptyset\rangle$

The interface abstraction relies solely on information given by the interface. Using the transition mapping

$$\gamma(\tau) = \begin{cases} \tau_a & \text{if } \exists l \in T_M \ . \ \lambda_M(\tau, l) \\ \tau^* \text{ with } \rho_{\tau^*} = \rho_{\tau_a} \wedge \bigwedge_{v \in V_i} v = v' & \text{otherwise} \end{cases}$$

it is easy to show that $M \sqsubseteq A_M$. The transition τ_a can simulate any transition in B. The labeling function covers all exported transitions, and the *proxy* label, which is not exported, ensures that in any composition there is a non-synchronizing transition.

4.5 Induction for Recursive Descriptions

A natural way to prove properties over a recursively defined module is by induction on the parameter value. Let $\prec$ be a well-founded order over the domain $\mathcal{D}$ of the parameters of a module M. The principle of well-founded induction is formulated as follows:

> To show that $M(X) \models \varphi(X)$ for all $X \in \mathcal{D}$, it suffices to show that for an arbitrary value $X \in \mathcal{D}$
>
> $$M(A) \models \varphi(A) \text{ for all } A \prec X \text{ (IH)}$$
> $$\text{implies}$$
> $$M(X) \models \varphi(X)$$
>
> The antecedent is called the inductive hypothesis, and the consequent is called the conclusion of the *inductive step.*

For unknown parameter values the transition system associated with a recursively described module cannot be computed directly. Module abstraction, e.g., interface abstraction, can be used to derive an abstraction with a non-recursive description.

5 Verification of the Arbiter

The two properties we want to prove about the arbiter system are *mutual exclusion:* no two clients can hold the grant simultaneously, expressed by

$$mux(h): \ \Box(\forall i,j : [0..2^h-1] \,.\, (\texttt{grants}[i] \wedge \texttt{grants}[j]) \rightarrow i=j)$$

and *eventual access:* any client who requests a grant will eventually get a grant, expressed by

$$acc(h): \ \Box(\forall i : [0..2^h-1] \,.\, \texttt{requests}[i] \rightarrow \Diamond\, \texttt{grant}[i])$$

5.1 Mutual exclusion

The `Arbiter` system was formally specified in Section 3.7, as a composition of `Arbitertree` and `Top`. By the property inheritance rules, to prove

$$\texttt{Arbiter}(h) \models mux(h)$$

it is sufficient to prove

$$\texttt{Arbitertree}(h) \models mux(h)$$

The recursive description of Arbitertree suggests a proof by induction. For the base case[2] we have to show

$$\texttt{Arbitertree}(1) \models mux(1)$$

which, using the definition of Arbitertree for $h = 1$, the definition of Leafnode and the property inheritance rules can be reduced to the proof of

$$\texttt{Arbiternode} \models \Box\neg(\texttt{grL} \wedge \texttt{grR})$$

This property is easily established by applying the invariance rule to the associated transition system of Arbiternode.

By the induction principle, to prove for $h > 1$

$$\texttt{Arbitertree}(h) \models mux(h)$$

we may make use of the inductive hypothesis, for any $h^* \leq h$, in particular $h^* = h - 1$:

$$\texttt{Arbitertree}(h-1) \models mux(h-1)$$

and thus, by the property inheritance rules, we inherit

$$\texttt{Arbitertree}(h) \models \Box(\forall i,j : [0..2^{(h-1)} - 1] \,.\, (\texttt{grants}[i] \wedge \texttt{grants}[j]) \rightarrow i = j)$$

$$\texttt{Arbitertree}(h) \models \Box(\forall i,j : [2^{(h-1)}..2^{h} - 1] \,.\, (\texttt{grants}[i] \wedge \texttt{grants}[j]) \rightarrow i = j)$$

for the left and right subtree respectively. The two properties express that each subtree establishes mutual exclusion among its set of clients. Unfortunately, they do not establish mutual exclusion for the tree itself: they do not prohibit the case in which both subtrees simultaneously have given out a grant. To rule out this case, we establish two additional properties. The first property states that no client holds a grant unless the tree holds a grant. This property only holds of the Arbitertree if we assume that its environment does not retract a grant before the Arbitertree releases the grant. Thus we formulate this as an assumption-guarantee property:

$$\texttt{Arbitertree}(h) \models \begin{array}{l} \text{Assumption:} \\ \Box(\ominus(\texttt{gr} \wedge \texttt{req}) \rightarrow \texttt{gr}) \\ \text{Guarantee:} \\ \Box\neg\texttt{gr} \rightarrow (\forall i : [0..2^{h} - 1] \,.\, \neg\texttt{grants}[i]) \end{array} \tag{1}$$

The second property states that only one of the two subtrees can hold the grant, expressed by

$$\texttt{Arbitertree}(h) \models \Box\neg(\texttt{grL} \wedge \texttt{grR})$$

[2] Strictly speaking, there is no base case. However, due to the definition of the system and the fact that our proofs use mostly stepwise induction, distinguishing a base case and inductive step seemed more natural

The latter property is inherited directly from the same property proven earlier for the `Arbiternode`. From the first property, by the property inheritance rules, we inherit

$$\texttt{Arbitertree}(h) \models \left(\begin{array}{c} \Box(\ominus(\texttt{grL} \wedge \texttt{reqL}) \rightarrow \texttt{grL}) \\ \rightarrow \\ \Box\,\neg\texttt{grL} \rightarrow (\forall i : [0..2^{h-1}-1]\,.\,\neg\texttt{grants}[i]) \end{array}\right)$$

$$\texttt{Arbitertree}(h) \models \left(\begin{array}{c} \Box(\ominus(\texttt{grR} \wedge \texttt{reqR}) \rightarrow \texttt{grR}) \\ \rightarrow \\ \Box\,\neg\texttt{grR} \rightarrow (\forall i : [2^{h-1}..2^{h}-1]\,.\,\neg\texttt{grants}[i]) \end{array}\right)$$

The assumptions are discharged by proving

$$\texttt{Arbiternode} \models \Box(\ominus(\texttt{grL} \wedge \texttt{reqL})) \rightarrow \texttt{grL} \tag{2}$$

and

$$\texttt{Arbiternode} \models \Box(\ominus(\texttt{grR} \wedge \texttt{reqR})) \rightarrow \texttt{grR} \tag{3}$$

using the invariance rule. These properties are inherited directly by `Arbitertree`(h).

It remains to prove (1). This property is again established by induction. The case $h = 1$ is proved using the invariance rule. For the case $h > 1$ we need, in addition to (2) and (3), the auxiliary property that a client of a node can have the grant only if the node owns the grant and has not released it yet, expressed by

$$\texttt{Arbiternode} \models \begin{array}{l} \text{Assumption:} \\ \Box(\ominus(\texttt{gr} \wedge \texttt{req}) \rightarrow \texttt{gr}) \\ \text{Guarantee:} \\ \Box((\texttt{grL} \vee \texttt{grR}) \rightarrow (\texttt{gr} \wedge \texttt{req})) \end{array}$$

This property is readily established using the invariance rule, and the assumptions are again discharged using (2) and (3).

5.2 Eventual Access

To show accessibility for all clients of the arbiter system, we expect we have to make some assumptions on the environment, for example, that clients will eventually release a grant. However, rather than trying to identify these assumptions up front, we choose to discover them in the course of the proof, and we will add them to the property as appropriate. As it is more convenient to do the proof at the level of the `Arbitertree` rather than for the arbiter system, assumptions about the parent of the `Arbitertree` (called the *server*) are added as well. These will be discharged at the end by the `Top` module.

To show

$$\texttt{Arbiter}(h) \models acc(h)$$

it suffices to show

$$\texttt{Arbitertree}(h) \models acc(h) \tag{4}$$

A proof by induction yields as the base case

$$\texttt{Arbitertree}(1) \models acc(1)$$

which, by the property inheritance rules can be reduced to

$$\texttt{Arbiternode} \models \Box(\texttt{reqL} \rightarrow \Diamond\, \texttt{grL}) \tag{5}$$

and

$$\texttt{Arbiternode} \models \Box(\texttt{reqR} \rightarrow \Diamond\, \texttt{grR}) \tag{6}$$

Figure 13 shows a generalized verification diagram to prove (5). It represents the desired flow of the module that establishes the property. The initiation conditions (all initial states are covered by the diagram) and the fairness conditions (the assertions on nodes with outgoing labeled edges imply the enabling condition of the corresponding transition) are readily established. However, consecution (every τ-successor of an assertion is covered) does not hold, for example it fails for node n_1 for the environment transition. Since `gr` and `reqL` are input variables of `Arbiternode`, the environment transition may modify them arbitrarily. However, if we assume that the parent (server) does not retract a grant before it is released, expressed by

$$\Box(\ominus(\texttt{gr} \wedge \texttt{req}) \rightarrow \texttt{gr}) \quad (\textit{server}) \tag{7}$$

and that the (right) client does not retract a request before it receives a grant,

$$\Box(\ominus(\neg\texttt{grR} \wedge \texttt{reqR}) \rightarrow \texttt{reqR}) \quad (\textit{client}) \tag{8}$$

the consecution condition holds for n_1. For the consecution condition of n_2 we need to assume that the right client does not request a grant before the previous one is retracted,

$$\Box(\ominus(\texttt{grR} \wedge \neg\texttt{reqR}) \rightarrow \neg\texttt{reqR}) \quad (\textit{client}) \tag{9}$$

and for the consecution of node n_4 we assume that the parent does not give a grant unless it is requested.

$$\Box(\ominus(\neg\texttt{gr} \wedge \neg\texttt{req}) \rightarrow \neg\texttt{gr}) \quad (\textit{server}) \tag{10}$$

Additional assumptions are necessary to ensure progress. The fairness of `ReleaseRight`, `RequestGrant` and `GrantLeft` ensure progress from nodes n_2, n_4 and n_6, however no progress is guaranteed from nodes n_1, n_3 and n_5 (note that no progress is required from n_0 or n_7, because in n_0 the antecedent of our property is false, while in n_7 the goal is true). Progress from n_1 requires the (right) client to eventually release the grant:

$$\Box(\texttt{grL} \rightarrow \Diamond(\neg\texttt{reqL} \vee \neg\texttt{grL})) \quad (\textit{client}) \tag{11}$$

To guarantee progress from n_3 and n_5 we have to assume that the server will eventually retract the grant when it is released,

$$\Box(\texttt{req} \rightarrow \Diamond(\texttt{gr} \vee \neg\texttt{req})) \quad (\textit{server}) \tag{12}$$

and that the server will eventually give a grant when one is requested,

$$\Box(\neg\texttt{req} \rightarrow \Diamond(\neg\texttt{gr} \vee \texttt{req})) \quad (\textit{server}) \tag{13}$$

The proof of (6) is similar; it generates the same assumptions for the server and the symmetrical asssumptions for the clients.

Generalizing these assumptions about the `Arbiternode` clients to the clients of an `Arbitertree` of height h we get,

$$\begin{array}{l} \textit{clients}(h) : \\ \quad \forall i : [0..2^h - 1] \;.\; \Box(\ominus(\neg\texttt{grants}[i] \wedge \texttt{requests}[i]) \rightarrow \texttt{requests}[i]); \\ \quad \forall i : [0..2^h - 1] \;.\; \Box(\ominus(\texttt{grants}[i] \wedge \neg\texttt{requests}[i]) \rightarrow \neg\texttt{requests}[i]); \\ \quad \forall i : [0..2^h - 1] \;.\; \Box(\texttt{grants}[i] \rightarrow \Diamond(\neg\texttt{requests}[i] \vee \neg\texttt{grants}[i])) \end{array}$$

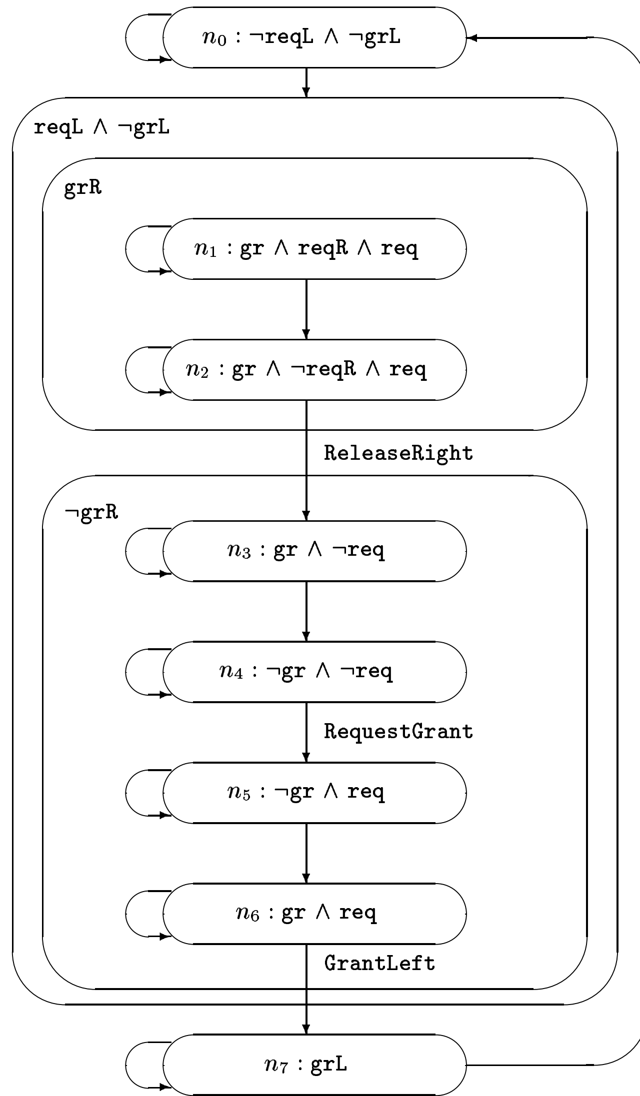

Fig. 13. Verification diagram for property (5).

We now weaken our accessibility property to include the assumptions:

$$\texttt{Arbitertree}(h) \models \mathit{server} \wedge \mathit{clients}(h) \rightarrow \mathit{acc}(h) \tag{14}$$

where *server* stands for the conjunction of assumptions made about the parent:

$$\mathit{server} : \begin{array}{l} \Box(\ominus(\texttt{gr} \wedge \texttt{req}) \rightarrow \texttt{gr}) \wedge \\ \Box(\ominus(\neg\texttt{gr} \wedge \neg\texttt{req}) \rightarrow \neg\texttt{gr}) \wedge \\ \Box(\neg\texttt{req} \rightarrow \Diamond(\neg\texttt{gr} \vee \texttt{req})) \wedge \\ \Box(\texttt{req} \rightarrow \Diamond(\texttt{gr} \vee \neg\texttt{req})) \end{array}$$

To prove the induction step, we make use of the inductive hypothesis for $h^* = h - 1$,

$$\texttt{Arbitertree}(h-1) \models \mathit{server} \wedge \mathit{clients}(h-1) \rightarrow \mathit{acc}(h-1)$$

which is inherited by $\texttt{Arbitertree}(h)$ as

$$\texttt{Arbitertree}(h) \models \mathit{server}_L \wedge \mathit{clients}_L(h) \rightarrow \mathit{acc}_L(h)$$

and

$$\texttt{Arbitertree}(h) \models \mathit{server}_R \wedge \mathit{clients}_R(h) \rightarrow \mathit{acc}_R(h)$$

where acc_L and acc_R stand for accessibility for the clients $0 \dots 2^{(h-1)} - 1$ and $2^{h-1} \dots 2^h - 1$, respectively,

$$\mathit{acc}_L(h) : \forall i : [0..2^{h-1} - 1] \,.\, \Box(\texttt{requests}[i] \rightarrow \Diamond\, \texttt{grants}[i])$$

$$\mathit{acc}_R(h) : \forall i : [2^{h-1}..2^h - 1] \,.\, \Box(\texttt{requests}[i] \rightarrow \Diamond\, \texttt{grants}[i])$$

Similarly, $\mathit{clients}_L(h)$ and $\mathit{clients}_R$ refer to the assumptions made about the clients $0 \dots 2^{(h-1)} - 1$ and $2^{h-1} \dots 2^h - 1$, respectively; server_L stands for the server assumptions made by the left subtree:

$$\mathit{server}_L : \texttt{Arbitertree}(h) \models \begin{array}{l} \Box(\ominus(\texttt{grL} \wedge \texttt{reqL}) \rightarrow \texttt{grL}) \wedge \\ \Box(\ominus(\neg\texttt{grL} \wedge \neg\texttt{reqL}) \rightarrow \neg\texttt{grL}) \wedge \\ \Box(\neg\texttt{reqL} \rightarrow \Diamond(\neg\texttt{grL} \vee \texttt{reqL})) \wedge \\ \Box(\texttt{reqL} \rightarrow \Diamond(\texttt{grL} \vee \neg\texttt{reqL})) \end{array}$$

that is `req` and `gr` are replaced by `reqL` and `grL`. Similarly server_R stands for the server assumptions made by the right subtree. It is easy to see that

$$\texttt{Arbitertree}(h) \models \mathit{clients}(h) \rightarrow \mathit{clients}_L(h) \wedge \mathit{clients}_R(h)$$

and

$$\texttt{Arbitertree}(h) \models \mathit{acc}_L(h) \wedge \mathit{acc}_R(h) \rightarrow \mathit{acc}(h)$$

Thus it remains to discharge server_L and server_R. For server_L we show

$$\texttt{Arbitertree}(h) \models \Box(\ominus(\texttt{grL} \wedge \texttt{reqL}) \rightarrow \texttt{grL}) \tag{15}$$

$$\texttt{Arbitertree}(h) \models \Box(\ominus(\neg\texttt{grL} \wedge \neg\texttt{reqL}) \rightarrow \neg\texttt{grL}) \tag{16}$$

$$\texttt{Arbitertree}(h) \models \mathit{server} \rightarrow \Box(\neg\texttt{reqL} \rightarrow \Diamond(\neg\texttt{grL} \vee \texttt{reqL})) \tag{17}$$

$$\texttt{Arbitertree}(h) \models \mathit{server} \land \mathit{clients}(h) \rightarrow \Box(\texttt{reqL} \rightarrow \Diamond\,\texttt{grL}) \qquad (18)$$

Property (15) is identical to (2), which was established before. We show (16) by applying INV to the corresponding `Arbiternode` property. Property (17) can be reduced to

$$\texttt{Arbiternode} \models \Box(\ominus(\texttt{gr} \land \texttt{req} \rightarrow \texttt{gr})) \rightarrow \Box(\neg\texttt{reqL} \rightarrow \Diamond(\neg\texttt{grL} \lor \texttt{reqL})) \quad (19)$$

which is proven by the generalized verification diagram shown in Figure 14. The (server) assumption $\Box(\ominus(\texttt{gr}\land\texttt{req})) \rightarrow \texttt{gr}$ is necessary to ensure that in node n_0 `gr` is preserved by the environment transition.

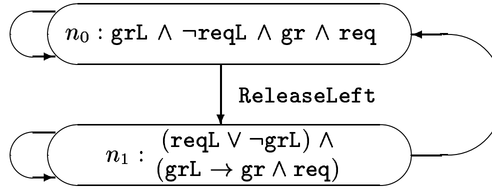

Fig. 14. Verification diagram for property (19).

To show property (18) we make use of property (5), which was proven under the assumptions (7)-(13). The server assumptions, (7), (12), and (13) are discharged immediately by *server* in the antecedent. Thus it remains to show

$$\texttt{Arbitertree}(h) \models \mathit{server} \land \mathit{clients}(h) \rightarrow \Box(\ominus(\neg\texttt{grR} \land \texttt{reqR}) \rightarrow \texttt{reqR}) \quad (20)$$

$$\texttt{Arbitertree}(h) \models \mathit{server} \land \mathit{clients}(h) \rightarrow \Box(\ominus(\texttt{grR} \land \neg\texttt{reqR}) \rightarrow \neg\texttt{reqR}) \quad (21)$$

$$\texttt{Arbitertree}(h) \models \mathit{server} \land \mathit{clients}(h) \rightarrow \Box(\texttt{grL} \rightarrow \Diamond(\neg\texttt{reqL} \lor \neg\texttt{grL})) \quad (22)$$

Property (20) is shown by case analysis. For $h = 1$ the consequent is directly implied by $\mathit{clients}(1)$. For the case $h > 1$ we prove

$$\texttt{Arbiternode} \models \Box(\ominus(\neg\texttt{gr} \land \texttt{req}) \rightarrow \texttt{req}) \qquad (23)$$

using the invariance rule; the desired property is then inherited from the right child. Property (21) is proven in the same way, by proving

$$\texttt{Arbiternode} \models \Box(\ominus(\texttt{grR} \land \neg\texttt{reqR}) \rightarrow \neg\texttt{reqR}) \qquad (24)$$

The proof of (22) proceeds in a similar fashion except here we need to use induction, where both the base case ($h = 1$) and the inductive step ($h > 1$) rely

on the following `Arbiternode` property

$$\texttt{Arbiternode} \models \begin{array}{l} \text{Assumption:} \\ \Box(\ominus(\neg\texttt{grL} \wedge \texttt{reqL}) \rightarrow \texttt{reqL}); \\ \Box(\ominus(\neg\texttt{grR} \wedge \texttt{reqR}) \rightarrow \texttt{reqR}); \\ \Box(\ominus(\texttt{grL} \wedge \neg\texttt{reqL}) \rightarrow \neg\texttt{reqL}); \\ \Box(\ominus(\texttt{grR} \wedge \neg\texttt{reqR}) \rightarrow \neg\texttt{reqR}); \\ \Box(\texttt{grL} \rightarrow \Diamond(\neg\texttt{reqL} \vee \neg\texttt{grL})); \\ \Box(\texttt{grR} \rightarrow \Diamond(\neg\texttt{reqR} \vee \neg\texttt{grR})) \\ \text{Guarantee:} \\ \Box(\texttt{gr} \rightarrow \Diamond\neg\texttt{req} \vee \neg\texttt{gr}) \end{array} \quad (25)$$

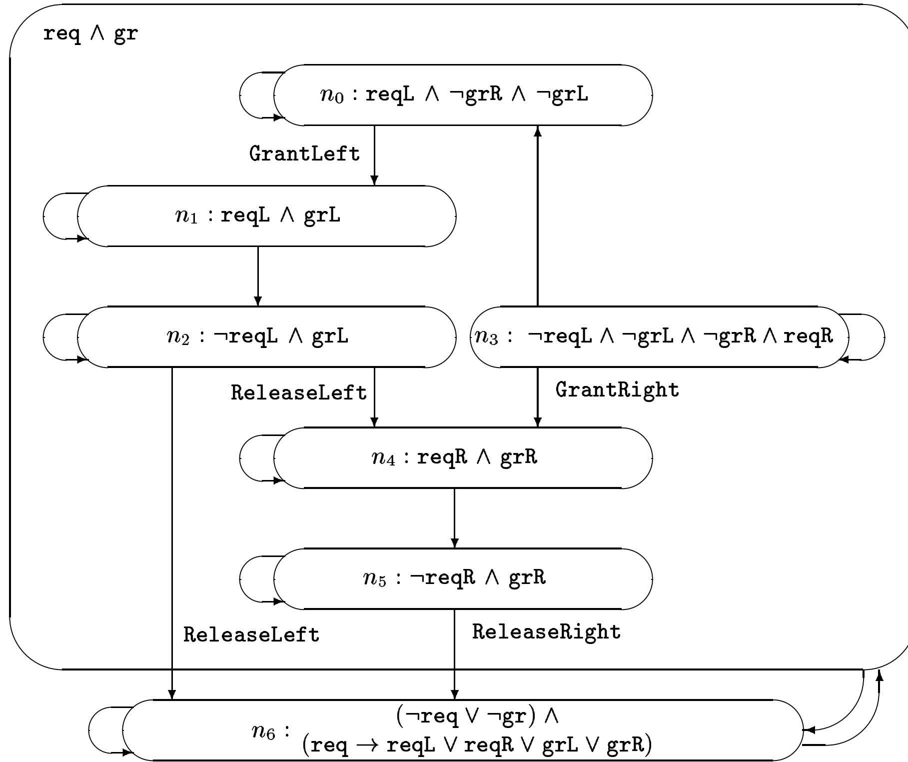

Fig. 15. Verification diagram for property (25).

which is proven by the generalized verification diagram shown in Figure 15. The first four assumptions are necessary to ensure the consecution requirement for nodes n_0, n_3, n_2, and n_5 respectively. The last two assumptions are used to ensure progress from node n_1 and n_4.

In the base case, when `Arbitertree`(1) inherits this property, all assumptions are discharged by $clients(1)$. For the case $h > 1$, the first four assumptions are discharged by (23) and (24) and the corresponding properties for the left side, and the last two properties are discharged by the inductive hypothesis inherited from the left and right subtree. This concludes the proof of (14).

We now finish the proof of accessibility for the `Arbiter` system. It is easy to show

$$\texttt{Top} \models server$$

and therefore we can discharge the *server* assumption for the `Arbiter` system, and we have

$$\texttt{Arbiter}(h) \models clients(h) \rightarrow acc(h)$$

Thus, as expected, accessibility for the arbiter system relies on the cooperation of the clients.

6 Conclusions

We have presented a formal framework for the modular description and verification of fair transition systems, and demonstrated its use on an example. We proposed several deductive proof techniques to establish and re-use modular properties.

In the verfication of the arbiter system we showed how assumptions are generated naturally in the course of the proof. Diagrams were constructed representing the intended flow of the module, and verification conditions involving input variables were added as assumptions to the property we set out to prove. These assumptions were then carried along until they either could be discharged by properties proven over other modules, or they could be proven directly over the larger inheriting module.

The main contribution of the paper is the proposal for a concrete framework for mechanized modular deductive verification. As observed by Shankar, many have studied assumption-guarantee reasoning, but few have used it in practice. A notable exception are the proofs carried out using I/O automata and their extensions, timed and hybrid I/O automata. However, since most of these proofs were carried out by hand, there was no need for a formal proof system.

- The *modular verification rule* allows us to prove properties over modules with non-recursive descriptions.
- Property *inheritance* provides an incremental proof method: properties of a module A can be reused in any module B whose description refers to A.
- *Modular abstraction* allows us to focus the proof on relevant components. Interface abstraction is a generic abstraction, based on the interface information.
- The *induction rule* makes the methodology applicable to recursive designs.

Although not considered in this paper, the verification methodology can be adapted to other verification techniques, such as *deductive model checking* [SUM96]. It is also straightforward to extend the framework to real-time and hybrid systems, modeled by clocked and hybrid transition systems [MP95a]. In these systems fair, clocked and hybrid parameterized transition modules can be freely combined into one module system. Extra care has to be taken to ensure that time steps synchronize for all parallel modules.

Acknowledgements: We thank Nikolaj Bjørner, Mark Pichora and Tomás Uribe for their careful reading and many helpful suggestions.

References

[AH96] R. Alur and T.A. Henzinger, editors. *Proc. 8th Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*. Springer-Verlag, July 1996.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proc. 3rd IEEE Symp. Logic in Comp. Sci.*, pages 165–175. IEEE Computer Society Press, 1988.

[AL93] M. Abadi and L. Lamport. Conjoining specifications. Technical Report SRC-118, DEC-SRC, December 1993.

[BBC+95] N.S. Bjørner, A. Browne, E.S. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[BBC+96] N.S. Bjørner, A. Browne, E.S. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Alur and Henzinger [AH96], pages 415–418.

[BK84] H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. In *Seminar on Concurrency*, vol. 197 of *LNCS*, pages 35–61. Springer-Verlag, 1984.

[BMS95] A. Browne, Z. Manna, and H.B. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, vol. 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.

[BMS96] A. Browne, Z. Manna, and H.B. Sipma. Hierarchical verification using verification diagrams. In 2nd *Asian Computing Science Conf.*, vol. 1179 of *LNCS*, pages 276–286. Springer-Verlag, December 1996.

[BMSU97] N.S. Bjørner, Z. Manna, H.B. Sipma, and T.E. Uribe. Deductive verification of real-time systems using STeP. In *4th Intl. AMAST Workshop on Real-Time Systems*, vol. 1231 of *LNCS*, pages 22–43. Springer-Verlag, May 1997.

[Cha93] E.S. Chang. *Compositional Verification of Reactive and Real-Time Systems.* PhD thesis, Computer Science Department, Stanford University, Stanford, California, 1993. Tech. Report STAN-CS-TR-94-1522.

[Dil88] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* PhD thesis, Carnegie-Mellon Univ., 1988. Available as Technical Report CMU-CS-88-119.

[GGS88] S. Garland, J. Guttag, and J. Staunstrup. Verification of vlsi circuits using lp. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 329–345. Elsevier Science Publishers B.V. (North Holland), 1988.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. Prog. Lang. Sys.*, 16(3):843–871, May 1994.

[JT95] B. Jonsson and Y.K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. In *TAPSOFT '95*, pages 262–276, 1995.

[KMP94] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In J.W. de Bakker, W.P. de Roever, and G. Rosenberg, editors, *A Decade of Concurrency*, vol. 803 of *LNCS*, pages 273–346. Springer-Verlag, 1994.

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

[LT89] N.A. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In M. Hagiya and J.C. Mitchell, editors, *Proc. International Symposium on Theoretical Aspects of Computer Software*, vol. 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.

[MP95a] Z. Manna and A. Pnueli. Clocked transition systems. In *Proc. of the Intl. Logic and Software Engineering Workshop*, August 1995. Beijing, China.

[MP95b] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, sub-series F: Computer and System Science, pages 123–144. Springer-Verlag, 1985.

[Sei80] C.L. Seitz. Ideas about arbiters. *Lambda*, pages 10–14, 1980.

[Sha93] N. Shankar. A lazy approach to compositional verification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, December 1993.

[Sha98] N. Shankar. Lazy compositional verification. In *this volume*, 1998.

[Sta94] J. Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.

[SUM96] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In Alur and Henzinger [AH96], pages 208–219.