Int J STTT (2000) 2: 328–342

Control and data abstraction: the cornerstones of practical formal verification*

Yonit Kesten[1], **Amir Pnueli**[2]

[1] Dept. of Communication Systems Engineering, Ben Gurion University, Beer-Sheva, Israel; E-mail: ykesten@bgumail.bgu.ac.il
[2] Dept. of Applied Mathematics and Computer Science, the Weizmann Institute of Science, Rehovot, Israel;
E-mail: amir@wisdom.weizmann.ac.il

Abstract. In spite of the impressive progress in the development of the two main methods for formal verification of reactive systems – Symbolic Model Checking and Deductive Verification, they are still limited in their ability to handle large systems. It is generally recognized that the only way these methods can ever scale up is by the extensive use of abstraction and modularization, which break the task of verifying a large system into several smaller tasks of verifying simpler systems.

In this paper, we review the two main tools of compositionality and abstraction in the framework of linear temporal logic. We illustrate the application of these two methods for the reduction of an infinite-state system into a finite-state system that can then be verified using model checking.

The technical contributions contained in this paper are a full formulation of abstraction when applied to a system with both weak and strong fairness requirements and to a general temporal formula, and a presentation of a compositional framework for shared variables and its application for forming *network invariants*.

Key words: Formal verification – Linear temporal logic – Data abstraction – Control abstraction – Network invariant – Model checking – Safety and liveness property – Weak and strong fairness

1 Introduction

In spite of the impressive progress in the development of the two main methods for formal verification of reactive systems – Model Checking (in particular symbolic) and Deductive Verification, they are still limited in their ability to handle large systems. It is generally recognized that the only way these methods can ever scale up to handle industrial-size designs is by the extensive use of abstraction and modularization, which break the task of verifying a large system into several smaller tasks of verifying simpler systems.

In this paper, we review the two main tools of compositionality and abstraction in the framework of linear temporal logic. We illustrate the application of these two methods for the reduction of an infinite-state system into a finite-state system that can then be verified using model checking.

To simplify matters, we have considered two special classes of infinite-state systems for which the combination of compositionality and abstraction can effectively simplify the systems into finite-state ones. The first class is where the unboundedness of the system results from its structure. These are parameterized designs consisting of a parallel composition of finite-state processes, whose number is a varying parameter. For such systems, the source of complexity is the control or the architectural structure. We describe the techniques useful for such systems as *control abstraction*, since it is the control component that we try to simplify. Another source for state complexity is having data variables which range over infinite domains such as the integers. We refer to the techniques appropriate for simplifying such systems as *data abstraction*.

Many methods have been proposed for the uniform verification of parameterized systems, which is the subject of our control abstraction. These include explicit induction [12, 13], network invariants, which can be viewed as implicit induction [15, 20, 23, 35], methods that can be viewed as abstraction and approximation of network invariants [4, 6, 32], and other methods that can be viewed as based on abstraction [13, 16]. The approach described

* This research was supported in part by a gift from Intel, a grant from the U.S.-Israel bi-national science foundation, and an *Infrastructure* grant from the Israeli Ministry of Science and the Arts.

here is based on the idea of *network invariants* as introduced in [35], and elaborated in [20] into a working method.

There has been extensive study of the use of data abstraction techniques, mostly based on the notions of *abstract interpretation* [9, 10]. Most of the previous work was done in a branching context which complicates the problem if one wishes to preserve both existential and universal properties. On the other hand, if we restrict ourselves to a universal fragment of the logic, e.g., ACTL*, then the conclusions reached are similar to our main result for the restricted case that the property ψ contains negations only within assertions.

The paper [7] obtains a similar result for the fragment ACTL*. However, instead of starting with a concrete property ψ and abstracting it into an appropriate ψ^α, they start with an abstract ACTL* formula Ψ evaluated over the abstract system $\mathcal{D}^\alpha$ and show how to translate (concretize) it into a concrete formula $\psi = \mathcal{C}(\Psi)$. The concretization is such that $\alpha^-(\psi) = \Psi$.

The survey in [8] considers an even simpler case in which the abstraction does not concern the variables on which the property ψ depends. Consequently, this is the case in which $\psi^\alpha = \psi$.

A more elaborate study in [11] considers a more complex specification language – L_μ, which is a positive version of the μ-calculus.

None of these three articles considers explicitly the question of fairness requirements and how they are affected by the abstraction process.

Approaches based on simulation and studies of the properties they preserve are considered in [24] and [14].

A linear-time application of abstract interpretation is proposed in [3], applying the abstractions directly to the computational model of *fair transition systems* which is very close to the FDS model considered here. However, the method is only applied for the verification of safety properties. Liveness, and therefore fairness, are not considered.

2 A computational model: fair discrete structure

As a computational model for reactive systems, we take the model of *fair discrete system* (FDS), which is a slight variation on the model of *fair transition system* [28]. The FDS model was first introduced in [19] under the name "Fair Kripke Structure".

An FDS $\mathcal{D} : \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ consists of the following components.

- $V = \{u_1, ..., u_n\}$: A finite set of typed *system variables*, containing data and control variables. The set of *states* (interpretation) over V is denoted by Σ. Note that Σ can be both finite or infinite, depending on the domains of V.

  The variables in V are classified as follows:
  - $W = \{w_1, \ldots, w_n\} \subseteq V$: A finite set of *owned variables*. These are the variables that only the system itself can modify. All other variables can also be modified by the environment. A system is said to be *closed* if $W = V$.
  - $\mathcal{O} = \{o_1, \ldots, o_n\} \subseteq V$: A finite set of *observable variables*. These are the variables which the environment can observe.

  It is required that $V = W \cup \mathcal{O}$, i.e., for every system variable $u \in V$, u is *owned*, *observable*, or both.
- Θ : The *initial condition* – an *assertion* (first-order state formula) characterizing the initial states.
- ρ : A *transition relation* – an assertion $\rho(V, V')$, relating the values V of the variables in state $s \in \Sigma$ to the values V' in a $\mathcal{D}$-successor state $s' \in \Sigma$.
- $\mathcal{J} : \{J_1, \ldots, J_k\}$: A set of *justice* (*weak fairness*) *requirements*. The justice requirement $J \in \mathcal{J}$ is an assertion, intended to guarantee that every computation contains infinitely many J-state (states satisfying J).
- $\mathcal{C} : \{\langle p_1, q_1 \rangle, \ldots \langle p_n, q_n \rangle\}$: A set of *compassion* (*strong fairness*) *requirements*. The compassion requirement $\langle p, q \rangle \in \mathcal{C}$ is a pair of assertions, intended to guarantee that every computation containing infinitely many p-states also contains infinitely many q-states.

We require that every state $s \in \Sigma$ has at least one $\mathcal{D}$-successor. This is often ensured by including in ρ the *idling* disjunct $V = V'$ (also called the *stuttering* step). In such cases, every state s is its own $\mathcal{D}$-successor.

Let $\sigma : s_0, s_1, s_2, ...$, be an infinite sequence of states, φ be an assertion, and let $j \geq 0$ be a natural number. We say that j is a φ-position of σ if s_j is a φ-state.

Let $\mathcal{D}$ be an FDS for which the above components have been identified. We define a *computation* of $\mathcal{D}$ to be an infinite sequence of states $\sigma : s_0, s_1, s_2, ...$, satisfying the following requirements:

- *Initiality:* s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution:* For each $j = 0, 1, ...$, the state s_{j+1} is a $\mathcal{D}$-successor of the state s_j.
- *Justice:* For each $J \in \mathcal{J}$, σ contains infinitely many J-positions.
- *Compassion:* For each $\langle p, q \rangle \in \mathcal{C}$, if σ contains infinitely many p-positions, it must also contain infinitely many q-positions.

For an FDS $\mathcal{D}$, we denote by $\mathcal{C}omp(\mathcal{D})$ the set of all computations of $\mathcal{D}$. An FDS $\mathcal{D}$ is called *feasible* if $\mathcal{C}omp(\mathcal{D}) \neq \emptyset$, namely, if $\mathcal{D}$ has at least one computation.

An infinite state sequence σ is called a *run* of $\mathcal{D}$ if it satisfies the requirements of initiality and consecution but not, necessarily, any of the fairness requirements. System $\mathcal{D}$ is said to be *viable* if every finite run can be extended into a computation. One of the differences between the model of fair transition systems and the FDS model is that every FTS is viable by construction, while it is easy to define an FDS which is not viable, e.g., by having the justice list include the assertion *false*. On the other hand, every FDS which is derived from a program is viable.

All our concrete examples are given in SPL (Simple Programming Language), which is used to represent concurrent programs (e.g., [26, 28]). Every SPL program can be compiled into an FDS in a straightforward manner. In particular, every statement in an SPL program contributes a disjunct to the transition relation. For example, the assignment statement

$$\ell_0 : y := x+1;\ \ell_1 :$$

can be executed when control is at location ℓ_0. When executed, it assigns $x+1$ to y while control moves from ℓ_0 to ℓ_1. This statement contributes to ρ the disjunct

$$\rho_{\ell_0}: \quad at\_\ell_0 \wedge at\_\ell_1' \wedge y' = x+1 \wedge x' = x.$$

The predicates at_ℓ_0 and at_ℓ_1' stand, respectively, for the assertions $\pi_i = 0$ and $\pi_i' = 1$, where π_i is the control variable denoting the current location within the process to which the statement belongs.

Every variable declared in an SPL program is specified as having one of the modes *in*, *out*, *in-out*, or *local*. This specification determines whether the variable is considered to be owned or observable or both according to the following table

Mode	Owned?	Observable?
in	N	Y
out	Y	Y
in-out	N	Y
local	Y	N

3 Operations on fds's

There are several important operations one may wish to apply to FDS's.

The first useful set of operations on programs and systems is forming their parallel composition, implying that the two systems execute concurrently. Consider the two fair discrete systems $\mathcal{D}_1 = \langle V_1, W_1, \mathcal{O}_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1\rangle$, and $\mathcal{D}_2 = \langle V_2, W_2, \mathcal{O}_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2\rangle$. We consider two ways of forming the parallel composition of $\mathcal{D}_1$ and $\mathcal{D}_2$.

3.1 Asynchronous parallel composition

The systems $\mathcal{D}_1$ and $\mathcal{D}_2$ are said to be *composable* if $W_1 \cap W_2 = \emptyset$, $V_1 \cap V_2 = \mathcal{O}_1 \cap \mathcal{O}_2$ and neither system modifies the variables owned by the other, i.e.,

$$\rho_1 \rightarrow \ pres(W_2 \cap V_1) \quad \text{and} \quad \rho_2 \rightarrow \ pres(W_1 \cap V_2).$$

The first condition requires that a variable can only be owned by one of the systems. The second condition requires that variables known to both systems must be observable in both.

For composable systems $\mathcal{D}_1$ and $\mathcal{D}_2$, we define their asynchronous parallel composition, denoted by $\mathcal{D}_1 \| \mathcal{D}_2$, to be the system $\mathcal{D} = \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$, where

$$\begin{array}{ll} V = V_1 \cup V_2 & W = W_1 \cup W_2 \\ \mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2 & \Theta = \Theta_1 \wedge \Theta_2 \\ \mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2 & \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}$$

$$\rho = \begin{pmatrix} \rho_1 \wedge pres(V_2 - V_1) & \vee \\ \rho_2 \wedge pres(V_1 - V_2) & \end{pmatrix}$$

For a set of variables $U \subseteq V$, the predicate $pres(U)$ stands for the assertion $U' = U$, implying that all the variables in U are preserved by the transition.

Obviously, the basic actions of the composed system $\mathcal{D}$ are chosen from the basic actions of its components, i.e., $\mathcal{D}_1$ and $\mathcal{D}_2$. Thus, we can view the execution of $\mathcal{D}$ as the *interleaved execution* of $\mathcal{D}_1$ and $\mathcal{D}_2$.

As seen from the definition, $\mathcal{D}_1$ and $\mathcal{D}_2$ may have disjoint as well as common system variables, and the variables of $\mathcal{D}$ are the union of all of these variables. The initial condition of $\mathcal{D}$ is the conjunction of the initial conditions of $\mathcal{D}_1$ and $\mathcal{D}_2$. The transition relation of $\mathcal{D}$ states that at any step, we may choose to perform a step of $\mathcal{D}_1$ or a step of $\mathcal{D}_2$. However, when we select one of the two systems, we should also take care to preserve the private variables of the other system. For example, choosing to execute a step of $\mathcal{D}_1$, we should preserve all variables in $V_2 - V_1$ and all the variables owned by $\mathcal{D}_2$.

The justice and compassion sets of $\mathcal{D}$ are formed as the respective unions of the justice and compassion sets of the component systems.

Asynchronous parallel composition corresponds to the SPL parallel operator $\|$ constructing a program out of concurrent processes.

3.2 Synchronous parallel composition

We define the *synchronous parallel composition* of $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \| \| \mathcal{D}_2$, to be the system

$$\mathcal{D}: \ \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle,$$

where

$$\begin{array}{ll} V = V_1 \cup V_2 & W = W_1 \cup W_2 \\ \mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2 & \Theta = \Theta_1 \wedge \Theta_2 \\ \mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2 & \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}$$

$$\rho = \rho_1 \wedge \rho_2$$

As implied by the definition, each of the basic actions of system $\mathcal{D}$ consists of the joint execution of an action of $\mathcal{D}_1$ and an action of $\mathcal{D}_2$. Thus, we can view the execution of $\mathcal{D}$ as the *joint execution* of $\mathcal{D}_1$ and $\mathcal{D}_2$.

The main, well-established, use of synchronous parallel composition is for coupling a system with a *tester* which tests for the satisfaction of a temporal formula, and then checking the feasibility of the combined system. In

this work, synchronous composition is also used for coupling the system with a *progress monitor*, used to ensure completeness of the data abstraction methodology presented in Sect. 7.

3.3 Modularization of an FDS

Let P be an SPL program and $\mathcal{D}$ its corresponding FDS. The standard compilation of a program into an FDS views the program as a *closed system* which has no interaction with its environment. In the context of compositional verification, we need an *open system* view of an FDS, which takes into account not only actions performed by the system but also actions (in particular, variable changes) performed by the environment.

Let $\mathcal{D} : \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FDS and $s \notin V$ be a fresh Boolean variable. The *modular version* of $\mathcal{D}$, is given by $\mathcal{D}_M : \langle V_M, W, \mathcal{O}_M, \Theta, \rho_M, \mathcal{J}, \mathcal{C} \rangle$, where,

$$V_M = V \cup \{s\} \qquad \mathcal{O}_M = \mathcal{O} \cup \{s\}$$
$$\rho_M = (\rho \wedge s') \vee (W' = W \wedge \neg s').$$

That is, $\mathcal{D}_M$ the modular version of $\mathcal{D}$ admits as an additional action a transition which preserves the values of all variables owned by $\mathcal{D}$ but allows all other shared variables to change in an arbitrary way. This provides the most general representation of an environment action. The *scheduling variable* s is used to ensure interleaving between the module and its environment. We refer to $\mathcal{D}_M$ as the *modular* or *open* version of system $\mathcal{D}$.

We define a *modular computation* of $\mathcal{D}$ to be any computation of $\mathcal{D}_M$.

3.4 Restricting an open shared variable

When constructing a system out of smaller components, it is often the case that all processes within the system are allowed to access a certain shared variable, but only a subset of the processes is allowed to modify its value. For example, we may have a system

$$\mathcal{D} = \mathcal{D}_1 \parallel \mathcal{D}_2 \parallel \mathcal{D}_3,$$

in which all processes are allowed to access variable x, but only processes $\mathcal{D}_1$ and $\mathcal{D}_2$ are allowed to modify its value.

We provide a special *restriction* (sealing-off) operation, which moves one of the system variables to the category of owned variables, thereby disallowing its modification by the environment.

Let $\mathcal{D} : \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FDS and let $U \subseteq V - W$ be a set of variables which are not owned by $\mathcal{D}$. The result of restricting U in $\mathcal{D}$, denoted by $\mathcal{D}\backslash U$ is the FDS $\mathcal{D}_R : \langle V, W_R, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where $W_R = W \cup U$.

Thus, to represent a system consisting of sub-systems $\mathcal{D}_1, \mathcal{D}_2$, and $\mathcal{D}_3$, in which $\mathcal{D}_3$ is not allowed to modify variable x, we may write

$$\mathcal{D} = ((\mathcal{D}_1 \parallel \mathcal{D}_2)\backslash x) \parallel \mathcal{D}_3.$$

To represent the closing off of an entire system $\mathcal{D}$, we write $\mathcal{D}_R$, which is an abbreviation for $\mathcal{D}\backslash(V - W)$. This restricts the environment from writing on any of the system variables. A system such that $W = V$ is often described as a *closed system*, because it can have no interaction with its environment.

4 Specification language: temporal logic

As a requirement specification language for reactive systems we take *temporal logic* (TL) [27]. For simplicity, we consider only the future fragment of TL. Extending the approach to the full logic is straightforward.

We assume an underlying assertion language $\mathcal{L}$ which contains the predicate calculus and interpreted symbols for expressing the standard operations and relations over some concrete domains. A *temporal formula* is constructed out of state formulas (assertions) to which we apply the Boolean operators $\neg$ and $\vee$ (the other Boolean operators can be defined from these), and the basic temporal operators $\circ$ (*next*) and $\mathcal{U}$ (*until*).

A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, \ldots$, where each state s_j provides an interpretation for the variables mentioned in p.

Given a model σ, we present an inductive definition for the notion of a temporal formula p holding at a position $j \geq 0$ in σ, denoted by $(\sigma, j) \models p$.

- For a state formula p, $\quad (\sigma, j) \models p \iff s_j \models p$
  That is, we evaluate p locally, using the interpretation given by s_j.
- $(\sigma, j) \models \neg p \iff (\sigma, j) \not\models p$
- $(\sigma, j) \models p \vee q \iff (\sigma, j) \models p$ or $(\sigma, j) \models q$
- $(\sigma, j) \models \circ p \iff (\sigma, j+1) \models p$
- $(\sigma, j) \models p \mathcal{U} q \iff$ for some $k \geq j, (\sigma, k) \models q$, and for every i such that $j \leq i < k, (\sigma, i) \models p$

Additional temporal operators can be defined by

$$\Diamond p = \mathit{true}\ \mathcal{U}\ p \ (\mathit{eventually})$$
$$\Box p = \neg \Diamond \neg p \ (\mathit{henceforth})$$

For a temporal formula p and a position $j \geq 0$ such that $(\sigma, j) \models p$, we say that j is a *p-position* (in σ). If $(\sigma, 0) \models p$, we say that p *holds* on σ, and denote it by $\sigma \models p$. A formula p is called *satisfiable* if p holds on some model. A formula p is called *valid*, denoted by $\models p$, if p holds on all models.

Given an FDS $\mathcal{D}$ and a temporal formula p, we say that p is $\mathcal{D}$*-valid*, denoted by $\mathcal{D} \models p$, if p holds on all models which are computations of $\mathcal{D}$. A property φ is said to be *modularly valid* over FDS $\mathcal{D}$, denoted $\mathcal{D} \models_M \varphi$, if φ is $\mathcal{D}_M$-valid, i.e., $\mathcal{D}_M \models \varphi$.

An algorithm for model checking whether a temporal formula p is valid over a finite-state FDS $\mathcal{D}$ is presented in [19]. The paper presents a version of the algorithm using explicit state enumeration methods as well as a symbolic version. Based on the ideas developed in [22] and [5], the approach calls for the construction of a *tester*

for the negation of p. This is an FDS $\mathcal{D}_{\neg p}$ whose computations are all the sequences which satisfy the negated formula $\neg p$. Then, we form the *synchronous parallel composition* $\mathcal{D}_{comb} = \mathcal{D} ||| \mathcal{D}_{\neg p}$ and check for feasibility. If $\mathcal{D}_{comb}$ is found to be feasible, this implies that $\mathcal{D}$ has a computation which violates p and therefore p is not valid over $\mathcal{D}$. If $\mathcal{D}_{comb}$ is found to be infeasible, we can conclude that p is $\mathcal{D}$-valid.

5 Control abstraction

Let $U \subseteq V$ be a subset of the system variables. For a V-state s, we denote by $s\Downarrow_U$ the U-state obtained by *projecting* s onto U. That is, the interpretation s restricted to the domain U.

The state sequence $\widetilde{\sigma} : \widetilde{s_0}, \widetilde{s_1}, \ldots$ is defined to be an *observation* of the FDS $\mathcal{D} : \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ if $\widetilde{\sigma}$ is a stuttering variant of the $\mathcal{O}$-projection

$$\sigma\Downarrow_{\mathcal{O}} = s_0\Downarrow_{\mathcal{O}},\ s_1\Downarrow_{\mathcal{O}},\ \ldots\ ,$$

where $\sigma : s_0, s_1, \ldots$ is a computation of $\mathcal{D}$. Let $Obs(\mathcal{D})$ denote the set of all observations of system $\mathcal{D}$.

The two FDS's $\mathcal{D}_A : \langle V_A, W_A, \mathcal{O}_A, \Theta_A, \rho_A, \mathcal{J}_A, \mathcal{C}_A \rangle$ and $\mathcal{D}_C : \langle V_C, W_C, \mathcal{O}_C, \Theta_C, \rho_C, \mathcal{J}_C, \mathcal{C}_C \rangle$ are defined to be *comparable,* if $\mathcal{O}_A = \mathcal{O}_C$ and $\mathcal{O}_A \cap W_A = \mathcal{O}_C \cap W_C$. The FDS $\mathcal{D}_A$ is an *abstraction* of the comparable $\mathcal{D}_C$, denoted by $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$, if $Obs(\mathcal{D}_C) \subseteq Obs(\mathcal{D}_A)$, i.e., every observation of $\mathcal{D}_C$ is also an observation of $\mathcal{D}_A$. We refer to $\mathcal{D}_C$ and $\mathcal{D}_A$ as the *concrete* and *abstract* systems, respectively. The abstraction relation is obviously reflexive and transitive.

It would have been very useful if the abstraction relation as defined above, had been compositional with respect to (asynchronous) parallel composition. That is, if $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$ had implied $(\mathcal{D}_C \parallel Q) \sqsubseteq (\mathcal{D}_A \parallel Q)$ for every FDS Q. Unfortunately, this is not the case.

Consider, for example, the FDS's corresponding to programs INCX and INCY presented in Fig. 1. Up to stuttering and idling, both of these FDS's have the unique observation

$$\langle x:0\,,\, y:0\rangle,\ \langle x:1\,,\, y:1\rangle,\ \langle x:2\,,\, y:2\rangle,\ \ldots$$

It follows that INCX and INCY have the same set of observations. In particular, this implies that INCX $\sqsubseteq$ INCY.

However, when we consider the program Q given by

in-out x:**integer**
loop forever do
$m_0 :\ x := 0$

we find out that the observation

$$\langle x:0\,,\, y:0\rangle, \langle x:1\,,\, y:1\rangle, \langle x:0\,,\, y:1\rangle, \langle x:1\,,\, y:1\rangle, \ldots$$

belongs to $Obs(\text{INCX} \parallel Q)$ but does not belong to the set $Obs(\text{INCY} \parallel Q)$. Consequently, while INCX $\sqsubseteq$ INCY,

$$(\text{INCX} \parallel Q) \not\sqsubseteq (\text{INCY} \parallel Q),$$

which shows that the abstraction relation $\sqsubseteq$ is not compositional.

Obviously, the problem lies in the fact that the relation $\sqsubseteq$ is based on the set of observations of the closed-system FDS semantics of programs. The difference between programs INCX and INCY can be observed only when we take into account actions of the environment, such as resetting variable x to 0. In the definition of the compuatations (and therefore observations) of the FDS assigned to these programs, such actions are not represented.

Once we diagnose the malady, the remedy is quite straightforward. We say that FDS $\mathcal{D}_A$ is a *modular abstraction* of the comparable $\mathcal{D}_C$, denoted by $\mathcal{D}_C \sqsubseteq_M \mathcal{D}_A$, if $Obs((\mathcal{D}_C)_M) \subseteq Obs((\mathcal{D}_A)_M)$, i.e., every observation of $(\mathcal{D}_C)_M$ the modularized version of $\mathcal{D}_C$ is also an observation of $(\mathcal{D}_A)_M$ the modularized version of $\mathcal{D}_A$.

Note that, while INCY is a plain abstraction of INCX, it is not a modular abstraction of INCX. To see this we point to

$$\langle s:0\,, x:0\,,\, y:0\rangle,\ \langle s:1\,, x:1\,,\, y:1\rangle,\ \langle s:0\,, x:0\,,\, y:1\rangle,$$
$$\langle s:1\,, x:1\,,\, y:1\rangle,\ \ldots\ ,$$

which is an observation of INCX_M but not of INCY_M.

When we upgrade from plain abstraction to modular abstraction, we obtain the desired property of compositionality of the abstraction relation with respect to the operations of parallel composition and restriction, as stated by the following claim:

Claim 1. Let $\mathcal{D}_C$ and $\mathcal{D}_A$ be two comparable FDS's such that $\mathcal{D}_C \sqsubseteq_M \mathcal{D}_A$. Then, for every FDS Q, and temporal formula φ,

1. $(\mathcal{D}_C \parallel Q) \sqsubseteq_M (\mathcal{D}_A \parallel Q)$
2. $(\mathcal{D}_C)_R \sqsubseteq_M (\mathcal{D}_A)_R$
3. $\mathcal{D}_A \models \varphi$ implies $\mathcal{D}_C \models \varphi$

We describe these compositionality properties by saying that the operations of parallel composition and restriction are *monotonic* with respect to modular abstraction, while temporal validity is *anti-monotonic.*

This indicates how we propose to use abstraction in order to simplify the verification task. Namely, given a property p to be verified over a complex system $\mathcal{D}_C$, we use modular abstraction in order to derive a simpler system $\mathcal{D}_A$ and then verify that p is $\mathcal{D}_A$-valid. Note that the implication is still in one direction. Namely, validity over the abstract system implies concrete validity but not, necessarily, vice versa. The most striking applications of this strategy are when $\mathcal{D}_C$ is an infinite-state system, while its abstraction $\mathcal{D}_A$ is finite-state and thus amenable to verification by model checking.

in-out x:**integer where** $x = 0$ **out** y:**integer where** $y = 0$ **loop forever do** $\ell_0 :\ (x, y) := (x+1, x+1)$	**in-out** x:**integer where** $x = 0$ **out** y:**integer where** $y = 0$ **loop forever do** $\ell_0 :\ (x, y) := (y+1, y+1)$
– INCX –	– INCY –

Fig. 1. Programs INCXand INCY

The comparable FDS's P and Q are defined to be *modularly equivalent*, denoted $P \sim_M Q$, if both $P \sqsubseteq_M Q$ and $Q \sqsubseteq_M P$.

6 Verification by abstract network invariants

In this section, we concentrate on cases in which the system is a parallel composition $P(n)\colon\ (P_1 \parallel \cdots \parallel P_n)_R$, where each P_i is a finite-state system. The final restriction of the parallel composition guarantees that no further interference from the environment is possible. The unbounded number of states for system $P(n)$ comes from the fact that we consider an infinite *family* of systems, and yet wish to verify uniformly (i.e., for every value of $n > 1$) that the property p is valid.

The general principles of the method and one of the examples presented in this section are shared with [20]. The main differences between the two presentations are that, while [20] considers processes communicating by synchronous message passing, we focus here on communication by shared variables, and we find the abstraction we use somewhat simpler to comprehend, perhaps due to the different communication mechanisms.

For simplicity, assume that the property p only refers to the observable variables of P_1 and that processes

$$P_2, \ldots, P_n$$

are identical (up to renaming). The strategy we propose can be summarized as follows:

Verification by abstract network invariants

1. Devise a *network invariant* $\mathcal{I}$, which is an FDS intended to provide a modular abstraction for the parallel composition $P_2 \parallel \cdots \parallel P_n$ for any n.
2. Confirm that $\mathcal{I}$ is indeed a network invariant, by model checking that $P_2 \sqsubseteq_M \mathcal{I}$ and that $(\mathcal{I} \parallel \mathcal{I}) \sqsubseteq_M \mathcal{I}$. The technique of model checking a modular abstraction is presented in Sect. 6.3.
3. Model check $\mathcal{D}_R \models p$, where $\mathcal{D}_R$ is the restricted system $(P_1 \parallel \mathcal{I})_R$.

We argue that this strategy is sound. Namely, if $\mathcal{D}_R \models p$ then $P(n) \models p$ for every $n > 1$. Step 2 of the strategy establishes $P_2 \parallel \cdots \parallel P_n \sqsubseteq_M \mathcal{I}$. By monotonicity of the parallel composition (Claim 1), it follows that

$$P_1 \parallel \cdots \parallel P_n \ \sqsubseteq_M\ (P_1 \parallel \mathcal{I}).$$

By monotonicity of the restriction operation, we can conclude that

$$P(n) \ =\ (P_1 \parallel \cdots \parallel P_n)_R \ \sqsubseteq_M\ (P_1 \parallel \mathcal{I})_R \ =\ \mathcal{D}_R.$$

Due to the anti-monotonicity of the validity relation, it follows that $\mathcal{D}_R \models p$ implies $P(n) \models p$, establishing that the proposed strategy is sound.

Step 1 in the strategy is the only one requiring ingenuity and which cannot be fully mechanized. However, while presenting the examples, we will provide some explanations and clues for the choices we made.

6.1 Mutual exclusion by semaphores

As our first running example, we use program MUX-SEM presented in Fig. 2. The program consists of n processes. Each process $P[i]$ cycles through three possible locations: N_i, T_i, and C_i. Location N_i represents the non-critical activity which the process can perform without coordination with the other processes. Location T_i, is the "trying" location, at which a process decides it needs to access its critical location. At the trying location, the process waits for the semaphore variable y to become 1. On entering the critical section C_i, the process sets y to 0. Finally, C_i is the critical location which should be reachable only exclusively by one process at a time. On exit from the critical section, variable y is reset to 1.

The mode **local** specified for variable y identifies y as being owned by the entire system but not by any of the individual processes. This specification ensures that the variable y cannot be modified by an enviornment agent external to the program. By the standard compilation of SPL programs, each process $P[i]$ is associated with a justice requirement $\mathcal{J}_i : \neg C_i$ and a compassion requirement $\mathcal{C}_i : (T_i \wedge y > 0, C_i)$. The justice requirement ensures that process $P[i]$ does not remain stuck forever at location C_i.

local y: **natural where** $y = 1$

$$\Big\|_{i=1}^{n} P[i] :: \left[\begin{array}{l}\textbf{loop forever do}\\ \quad\left[\begin{array}{l} N_i : \textbf{NonCritical}\\ T_i : \textbf{request}\ y\\ C_i : \textbf{Critical};\ \textbf{release}\ y\end{array}\right]\end{array}\right]$$

Fig. 2. Program MUX-SEM

The compassion requirement ensures that $P[i]$ does not remain stuck forever at location T_i while y turns positive infinitely many times. Note that a process may choose to stay forever at N_i or may get stuck at T_i if y turns positive only finitely many times and then remains zero forever. The latter behavior cannot occur in program MUX-SEM but this can be established only by a global analysis of the complete system.

In Fig. 3, we present process INV-CAND, which is our first candidate for the network invariant abstracting $P[2] \parallel \cdots \parallel P[n]$. In this section we choose to represent FDS's by transition diagrams, in which we explicitly list the fairness requirements. Process INV-CAND can be obtained by simpilfying a single copy of the concrete $P[2]$. The simplification consists of merging locations N and T into a single location N and relaxing the fairness requirements associated with this combined location. This simplification is suggested by noting that the main liveness requirement of accessibility is studied only for process $P[1]$. The only liveness properties we require from the environment processes $P[2], \ldots, P[n]$, is that they eventually exit their critical sections and release the semaphore. Thus, while being at location N, process INV-CAND may choose non-deterministically to stay at N or move to C if y equals 1. There is no justice requirement associated with location N, due to the possibility that the process may choose to remain there. On the other hand, with location C, we associate the justice requirement $\neg C$ which excludes behaviors in which INV-CAND get stuck at C. Let us denote by C_a the FDS corresponding to INV-CAND .

A useful heuristic that often leads to the generation of network invariants is forming the sequence of FDS's $\mathcal{I}_1 = C_a, \mathcal{I}_2 = C_a \parallel C_a, \mathcal{I}_3 = \mathcal{I}_2 \parallel C_a, \ldots$, and comparing every two successive $\mathcal{I}_i$'s, hoping that the sequence will converge. Convergence means that we identify an index $j \geq 0$ such that $\mathcal{I}_j \sim_M \mathcal{I}_{j+1}$. Trying this approach with the FDS C_a fails. Comparing $\mathcal{I}_2 \colon C_a \parallel C_a$ with $\mathcal{I}_1 \colon C_a$, we find that $(\mathcal{I}_2)_M$ can generate the observation

$$\langle y:1\,,\, s:0\rangle,\ \langle y:0\,,\, s:1\rangle,\ \langle y:1\,,\, s:0\rangle,\ \langle y:0\,,\, s:1\rangle,$$
$$\langle y:1\,,\, s:1\rangle,\ \langle y:0\,,\, s:0\rangle,\ \langle y:1\,,\, s:1\rangle,\ \cdots,$$

which cannot be generated by $(\mathcal{I}_1)_M$. Such a behavior can be explained as a scenario in the behavior of $P[2] \parallel P[3]$ under an unrestricted environment. First, $P[2]$ enters its critical section according to the step $\langle y:1\,,\, s:0\rangle \rightarrow \langle y:0\,,\, s:1\rangle$. Then, while $P[2]$ is still in its critical section, the environment raises y to 1, according to the step $\langle y:0\,,\, s:1\rangle \rightarrow \langle y:1\,,\, s:0\rangle$ (we know that this is an environment step because $s' = 0$). Then $P[3]$ enters its own critical section, as recorded in $\langle y:1\,,\, s:0\rangle \rightarrow \langle y:0\,,\, s:1\rangle$. Following that, $P[2]$ exits its critical section $(\langle y:0\,,\, s:1\rangle, \rightarrow \langle y:1\,,\, s:1\rangle)$, the environment resets y to 1 $(\langle y:1\,,\, s:1\rangle \rightarrow \langle y:0\,,\, s:0\rangle)$, and finally $P[3]$ exits $(\langle y:0\,,\, s:0\rangle,\ \langle y:1\,,\, s:1\rangle)$. What is special about this behavior is that $\mathcal{I}_2$ exits twice in succession without an observable entry between these two exits. In all behaviors of $\mathcal{I}_1 = C_a$, which has only one copy of $P[2]$, every two exits must be separated by an observable entry.

In a similar way, we find that $\mathcal{I}_3$ can exit its critical sections three times in succession, if the environment cooperates, which cannot be done by $\mathcal{I}_2$. This shows that the sequence $\mathcal{I}_1, \mathcal{I}_2, \ldots$ will never converge.

Looking closer at this example, we realize that the factor that differentiates between $\mathcal{I}_1$ and $\mathcal{I}_2$ and between $\mathcal{I}_2$ and $\mathcal{I}_3$ is their response to a behavior of the environment which will never be realized in the closed system, namely raising the semaphore variable to 1 while one of the processes is in its critical section. This leads us to the next (and final) abstraction $\mathcal{I}_{mux}$, presented in Fig. 4.

The system $\mathcal{I}_{mux}$ behaves as C_a as long as the environment behaves properly. However, once it detects that the environment has raised the value of y from 0 to 1 while the system was in the critical section, it goes into a *chaos* control state in which "anything goes". That is, all arbitrary sequences of values for the observable variables will be accepted from this point on. It is obvious that $\mathcal{I}_{mux}$ is an abstraction of C_a because it differs from C_a in all the additional behaviors it is ready to generate once it reached the *chaos* state.

It is not difficult to verify that $\mathcal{I}_{mux}$ is a network invariant. We model checked that $C_a \sqsubseteq_M \mathcal{I}_{mux}$ and that $(\mathcal{I}_{mux} \parallel \mathcal{I}_{mux}) \sqsubseteq_M \mathcal{I}_{mux}$.

It only remains to perform step 3 in the abstraction strategy presented in the beginning of the section. We form the closed system FDS $\mathcal{D} = (P_1 \parallel \mathcal{I})_R$ and use model checking to verify the liveness property $\mathcal{D} \models \Box(N_1 \rightarrow \Diamond C_1)$. This has been done and has established that process $P[1]$ of program MUX-SEM has the property of accessibility for any number of processes.

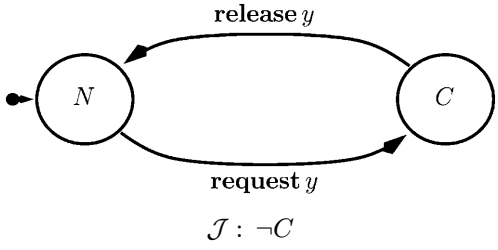

Fig. 3. Process INV-CAND, a candidate for a network invariant

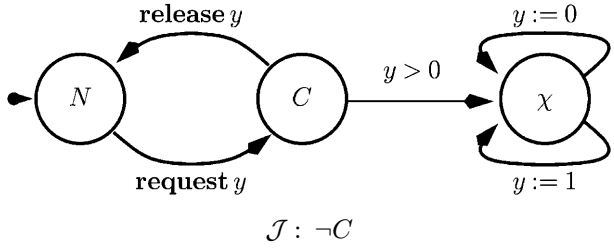

Fig. 4. The FDS $\mathcal{I}_{mux}$, a network invariant for MUX-SEM

6.2 The dining philosophers problem

As a more advanced example, we applied the technique described above to the problem of the dining philosophers. As originally described by Dijkstra, n philosophers are seated at a round table. Each philosopher alternates between a thinking phase and a phase in which he becomes hungry and wishes to eat. There are n chop-sticks placed around the table, one chop-stick between every two philosophers. In order to eat, each philosopher needs to acquire the chop-sticks on both sides. A chop-stick can be possessed by only one philosopher at a time.

A solution to the dining philosophers problem, using semaphores, is presented by program DINE-CONTR of Fig. 5.

In this program, philosophers $P[2], \ldots, P[n]$ reach first for the chop-stick on their left, represented by semaphore variable $c[j]$ for philosopher j, and then for their right chop-stick (semaphore $c[j \oplus_n 1]$). Philosopher $P[1]$ behaves differently, reaching first for his right chop-stick ($c[2]$) and only later for his left chop-stick ($c[1]$). We wish to prove the liveness property of accessibility for each of the philosophers, which can be specified by the temporal formula

$$\psi_{acc}: \quad \Box(at\_\ell_2[j] \rightarrow \Diamond(at\_\ell_4[j])),$$

for every $j = 1, \ldots, n$. This property ensures that every hungry philosopher eventually gets to eat.

Proceeding through a sequence of abstraction steps similar to the previous example, we finally wind up with the FDS $\mathcal{I}_{contr}$ presented in Fig. 6.

The diagram of Fig. 6 consists of two components that operate in parallel, one taking care of the left semaphore L and the other handling the right semaphore R. Whenever an environment fault is detected, i.e., the environment raises a semaphore that has been lowered by the system, both components escape to the *chaos* state after which all behaviors are possible. By the graphical conventions, the transitions to a chaos state have priority over internal transitions such as the one connecting (N_r, R) to $(C_r, \neg R)$.

Since $\mathcal{I}_{contr}$ is intended to abstract behaviors of a string of consecutive philosophers

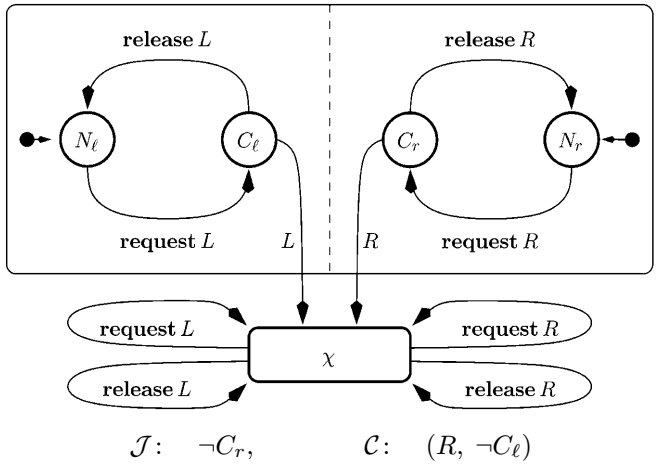

Fig. 6. The FDS $\mathcal{I}_{contr}$, the network invariant for program DINE-CONTR

$$P[i] \parallel P[i+1] \parallel \cdots \parallel P[j],$$

we should not be surprised that the behavior of the left semaphore L is only loosely coupled with that of the right semaphore R. This is because L stands for $c[i]$ (assuming $i > 1$) the left semaphore of process $P[i]$ the leftmost process in the string, while R stands for $c[j \oplus_n 1]$ the right semaphore of $P[j]$, the rightmost philosopher. There is still a weak coupling which is expressed through the fairness requirement. For ordinary philosophers, who take the right chop-stick last, the obligation to release semaphore R once it is taken, can be guaranteed locally, independently of the environment. This is expressed by the justice requirement $\neg C_r$ forbidding the system to remain forever in C_r with the semaphore R occupied. The situation is different with the left semaphore L. No subsystem (modeled by $\mathcal{I}_{contr}$) can unconditionally guarantee release of L once it is taken. Consequently, the fairness requirement guaranteeing the release of L is formulated as the compassion requirement $(R, \neg C_\ell)$ making the release of L (as implied by being at N_ℓ) conditional on the infinite recurrence of an available R. Already at the level of a single philosopher, after acquiring L the system pro-

in n: **integer where** $n \geq 2$
local c: **array** $[1..n]$ **where** $c = 1$

$$\Big\|_{j=2}^{n} P[j] :: \begin{bmatrix} \ell_0 : \textbf{loop forever do} \\ \quad \begin{bmatrix} \ell_1 : \textbf{NonCritical} \\ \ell_2 : \textbf{request } c[j] \\ \ell_3 : \textbf{request } c[j \oplus_n 1] \\ \ell_4 : \textbf{Critical} \\ \ell_5 : \textbf{release } c[j] \\ \ell_6 : \textbf{release } c[j \oplus_n 1] \end{bmatrix} \end{bmatrix} \Big\| P[1] :: \begin{bmatrix} \ell_0 : \textbf{loop forever do} \\ \quad \begin{bmatrix} \ell_1 : \textbf{NonCritical} \\ \ell_2 : \textbf{request } c[2] \\ \ell_3 : \textbf{request } c[1] \\ \ell_4 : \textbf{Critical} \\ \ell_5 : \textbf{release } c[2] \\ \ell_6 : \textbf{release } c[1] \end{bmatrix} \end{bmatrix}$$

Fig. 5. Program DINE-CONTR: solution with one contrary philosopher

ceeds to acquire R. If R is not available with sufficient frequency, the system will fail in obtaining it, and will keep L occupied forever.

It is straightforward to verify (using model checking) that $\mathcal{I}_{contr}$ modularly abstracts any of the processes $P[2], \ldots, P[n]$ and that $(\mathcal{I}_{contr} \parallel \mathcal{I}_{contr}) \sqsubseteq_M \mathcal{I}_{contr}$. It follows that $\mathcal{I}_{contr}$ is a network invariant for any sequence of regular philosophers. We can combine $\mathcal{I}_{contr}$ with $P[1]$ to establish the accessibility properties of the contrary philosopher $P[1]$.

We can also verify the accessibility property for all ordinary philosophers. To do so, we consider the combination $P[1] \parallel \mathcal{I}_{contr} \parallel P[ordinary] \parallel \mathcal{I}_{contr}$, in which we use the network invariant $\mathcal{I}_{contr}$ as an abstraction for the sequence of philosophers separating $P[1]$ from $P[ordinary]$ and then again as an abstraction for the sequence of philosophers separating $P[ordinary]$ from $P[1]$ in the other direction.

In all of these combinations, we should remember to close the ring by identifying the leftmost semaphore of the combination with the rightmost semaphore.

6.3 Model checking modular abstraction

When carrying out the abstraction process as described in this section, we are repeatedly required to verify that one FDS modularly abstracts another. Most of the available *computer aided verification* (CAV) tools for LTL (e.g., STEP [2] and TLV-BASIC [31]) are designed to support verification tasks. That is, they accept as inputs a system description, equivalent to an FDS $\mathcal{D}$, and a temporal formula φ and attempt to establish (or refute) that $\mathcal{D} \models \varphi$.

In this section, we show how the modular abstraction problem $\mathcal{D}_C \sqsubseteq_M \mathcal{D}_A$ can be *reduced* into a verification problem. This reduction can be used in order to establish the modular abstraction relation between systems while using the available LTL verification tools.

The idea of proving abstraction (equivalently refinement) by forming a superposition of the abstract and concrete systems, as we do here, has been proposed in [17]. The underlying theory of proving abstraction by simulation relations is thoroughly discussed in [1] and applied in [21, 25, 29, 34].

In theory, for the case that both $\mathcal{D}_C$ and $\mathcal{D}_A$ are finite-state, the modular abstraction problem is algorithmically solvable. All that is required is to convert the two systems into ω-automata, compute the complement of the $\overline{\mathcal{D}_A}$-automaton, and check that the languages of $\mathcal{D}_C$ and $\overline{\mathcal{D}_A}$ have an empty intersection. However, very few symbolic model checkers provide that capability of complementing a system and, even when they do, this operation could be exponentially expensive.

Instead, we base our approach on the simple observation that when $\mathcal{D}_A$ is deterministic, it is possible to construct a combined system which will try to emulate the joint computation of the two systems. For the case that $\mathcal{D}_A$ is non-deterministic, we rely on the user to provide an additional restriction on the possible actions of $\mathcal{D}_A$, reducing them to a single possible action. Thus, we trade computational complexity for full automation, and our approach may require user interaction.

Consider two comparable FDS's:

$$\mathcal{D}_C : \langle V_C, W_C, \mathcal{O}_C, \Theta_C, \rho_C, \mathcal{J}_C, \mathcal{C}_C \rangle$$

and it's proposed abstraction

$$\mathcal{D}_A : \langle V_A, W_A, \mathcal{O}_A, \Theta_A, \rho_A, \mathcal{J}_A, \mathcal{C}_A \rangle,$$

and assume we wish to establish that $\mathcal{D}_C \sqsubseteq_M \mathcal{D}_A$. Without loss of generality, we can assume that $V_C \cap V_A = \emptyset$, but that there exists a 1-1 correspondence between the variables of $\mathcal{O}_C$ and those of $\mathcal{O}_A$.

We say that the FDS $\mathcal{D}_S : \langle V_S, W_S, \mathcal{O}_S, \Theta_S, \rho_S, \mathcal{J}_S, \mathcal{C}_S \rangle$ is a *superposition* of $\mathcal{D}_C$ and $\mathcal{D}_A$ if it has the following form:

$$\begin{aligned}
V_S &= V_C \cup V_A \\
W_S &= W_C \cup W_A \\
\mathcal{O}_S &= \mathcal{O}_C \cup \mathcal{O}_A \\
\Theta_S &= \Theta_C \wedge \Theta_A \wedge \Theta_d \\
&\quad \wedge ((\exists V_A : \Theta_A \wedge \Theta_d \wedge \mathcal{O}_A = \mathcal{O}_C) \rightarrow \mathcal{O}_A = \mathcal{O}_C) \\
\rho_S &= (\rho_P \wedge ((\exists V'_A : \rho_P \wedge \mathcal{O}'_A = \mathcal{O}'_C) \rightarrow \mathcal{O}'_A = \mathcal{O}'_C)) \\
&\quad \vee (\rho_E \wedge ((\exists V'_A : \rho_E \wedge \mathcal{O}'_A = \mathcal{O}'_C) \rightarrow \mathcal{O}'_A = \mathcal{O}'_C))
\end{aligned}$$

where

$$\begin{aligned}
\rho_P &= \rho_C \wedge \rho_A \wedge \rho_d \\
\rho_E &= pres(W_C) \wedge pres(W_A) \\
\mathcal{J}_S &= \mathcal{J}_C \qquad \text{and} \quad \mathcal{C}_S = \mathcal{C}_C
\end{aligned}$$

The general idea in the construction of the superposition system $\mathcal{D}_S$ is that every computation of $\mathcal{D}_S$ induces a computation of $\mathcal{D}_C$ (when projected on V_C) and a run of $\mathcal{D}_A$ (when projected V_A). Thus, a computation of $\mathcal{D}_S$ can be viewed as a joint computation of the two systems $\mathcal{D}_C$ and $\mathcal{D}_A$. There are two desired features a successful superposition of $\mathcal{D}_C$ and $\mathcal{D}_A$ should satisfy.

1. Every computation of $\mathcal{D}_C$ is induced by some computation of $\mathcal{D}_S$. Thus, the additional conjuncts in Θ_S and ρ_S should not restrict the behavior of $\mathcal{D}_C$.
2. To the best of its ability, $\mathcal{D}_S$ should attempt to maintain the correspondence $\mathcal{O}_C = \mathcal{O}_A$. This explains the role of the implications conjuncted into Θ_S and ρ_S. These implications require that, if it is possible to choose abstract variables which are consistent with the constraints of $\mathcal{D}_A$ and maintain $\mathcal{O}_C = \mathcal{O}_A$, then such a choice should be made.

Note also that the system $\mathcal{D}_S$ has already been modularized by defining ρ_S as the choice between a system step ρ_P which is compatible with $\rho_C \wedge \rho_A$ and an environment step ρ_E which only guarantees the preservation of $W_S = W_C \cup W_A$.

The system $\mathcal{D}_S$ has Θ_d and ρ_d as open parameters, which should be provided by the user. Once they are specified, $\mathcal{D}_S$ can be automatically constructed from $\mathcal{D}_C$ and

$\mathcal{D}_A$, and this is what has been implemented in the current TLV-BASIC implementation of the modular abstraction checker within TLV.

A very simple choice is to take $\Theta_d = \rho_d = 1$, namely, take them both as being identically true. This choice is adequate in all cases that the abstract system $\mathcal{D}_A$ is deterministic. Determinism in the abstraction context means that, for every $\mathcal{D}_A$-state s and a set of specified values U for the observable variables $\mathcal{O}_A$, there exists at most one s', a ρ_A-successor of s, such that $s'[\mathcal{O}_A] = U$. All the examples presented in this section, such as the network invariants presented in Fig. 4 and Fig. 6 are deterministic. In fact, one of the reasons for eliminating the trying location T in process INV-CAND and the other network invariants was to make them deterministic. In view of this, all the modular abstractions mentioned in this section were resolved by superposition systems in which we have taken $\Theta_d = \rho_d = 1$.

The following claim makes precise the relation between computations of $\mathcal{D}_S$, computations of $\mathcal{D}_C$ and runs of $\mathcal{D}_A$.

Claim 2. If σ is a computation of $\mathcal{D}_S$, then $\sigma\Downarrow_{V_C}$ is a computation of $\mathcal{D}_C$ and $\sigma\Downarrow_{V_A}$ is a run of $\mathcal{D}_A$.

In the preceding discussion, we listed two features which are desirable in a good superposition. However, these features are not automatically guaranteed. In Fig. 7, we present a proof rule whose premises guarantee that the system $\mathcal{D}_S$ has the desired features.

A1. $\Theta_C \rightarrow \exists V_A : \Theta_A \wedge \Theta_d$

A2. $\rho_C \rightarrow \exists V'_A : \rho_A \wedge \rho_d$

A3. $\mathcal{D}_S \models \begin{pmatrix} \Box(\mathcal{O}_C = \mathcal{O}_A) \\ \wedge \bigwedge_{J \in \mathcal{J}_A} \Box\Diamond J \\ \wedge \bigwedge_{(p,q) \in \mathcal{C}_A} (\Box\Diamond p \rightarrow \Box\Diamond q) \end{pmatrix}$

$$\mathcal{D}_C \sqsubseteq_M \mathcal{D}_A$$

Fig. 7. Rule MOD-ABST

Premise A1 guarantees that for every value assignment to the concrete variables V_C satisfying Θ_C, there exists a value assignment to the abstract variables V_A satisfying $\Theta_A \wedge \Theta_d$. Thus, $\Theta_A \wedge \Theta_d$ does not restrict the choice of values for V_C.

Premise A2 stipulates a similar non-restriction requirement for ρ_C. It requires that, for every value assignment to V_C, V'_C, and V_A, which make $\rho_C(V_C, V'_C)$ true, there exists a value assignment to V'_A which satisfies $\rho_A(V_A, V'_A) \wedge \rho_d(V_C, V'_C, V_A, V'_A)$. Thus, $\rho_A \wedge \rho_d$ does not restrict the choice of values for V_C, V'_C, and V_A.

Finally, premise A3 requires that every computation σ of $\mathcal{D}_S$ maintains the invariant $\Box(\mathcal{O}_C = \mathcal{O}_A)$, and $\sigma\Downarrow_{V_A}$, the projection of σ on the abstract variables V_A, yields a computation of $\mathcal{D}_A$. According to Claim 2, $\sigma\Downarrow_{V_A}$ is a run of $\mathcal{D}_A$. Adding to it the fact that σ satisfies all the fairness requirements of $\mathcal{D}_A$, as established by A3, we can conclude that $\sigma\Downarrow_{V_A}$ is also a computation of $\mathcal{D}_A$.

The following claim states that rule MOD-ABST is sound.

Claim 3. If the premises of rule MOD-ABST are valid for some choice of Θ_d and ρ_d, then $\mathcal{D}_A$ is a modular abstraction of $\mathcal{D}_C$.

7 Data abstraction

In this section, we present a general methodology for *data abstraction*, strongly inspired by the notion of abstract interpretation [9]. Since in this case we do not deal with compositionality and modularization, we use a slightly simpler FDS model, in which system variables are not classified into W and $\mathcal{O}$.

Let $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FDS, and Σ denote the set of states of $\mathcal{D}$, the *concrete states*. Let $\alpha : \Sigma \mapsto \Sigma_A$ be a mapping of concrete states into *abstract states*. We say that α is a finitary abstraction mapping, if Σ_A is a finite set. The strategy of verification by data abstraction can be summarized as follows:

Verification by data abstraction

1. Define a finitary abstraction mapping α to abstract the (possibly infinite) *concrete* FDS $\mathcal{D}$ into a *finite*, *abstract* FDS $\mathcal{D}^\alpha$.
2. Abstract the *concrete* temporal property ψ into a *finitary abstract* temporal property ψ^α.
3. Verify $\mathcal{D}^\alpha \models \psi^\alpha$.
4. Infer $\mathcal{D} \models \psi$.

An implementation of this general strategy which specifies a recipe for defining the abstractions $\mathcal{D}^\alpha$ and ψ^α for a given α is called a *data abstraction method*.

A data abstraction method is said to be *safe* (equivalently, *sound*) if, for every FDS $\mathcal{D}$, temporal formula ψ, and a state abstraction mapping α (not necessarily finitary), $\models \psi^\alpha$ implies $\models \psi$, and $\mathcal{D}^\alpha \models \psi^\alpha$ implies $\mathcal{D} \models \psi$.

7.1 Safe abstraction of temporal formulas

To provide a syntactic representation of the abstraction mapping, we assume a set of *abstract variables* V_A and a set of expressions $\mathcal{E}^\alpha$, such that the equality $V_A = \mathcal{E}^\alpha(V)$ syntactically represents the semantic mapping α.

Let $p(V)$ be an assertion. We wish to define the abstraction $p^\alpha(V_A)$ such that $\models p^\alpha(V_A)$ implies $\models p(V)$. We introduce the operator α^-, defined by

$$\alpha^-(p(V)): \quad \forall V \left(V_A = \mathcal{E}^\alpha(V) \quad \rightarrow \quad p(V)\right) \wedge map(V_A),$$

where $map(V_A) : \exists V (V_A = \mathcal{E}^\alpha(V))$. The assertion $\alpha^-(p)$ holds for an abstract state $S \in \Sigma_A$ iff S is mappable and the assertion p holds for *all* concrete states $s \in \Sigma$ such that $s \in \alpha^{-1}(S)$, i.e., all states s such that $S = \alpha(s)$. Alternatively, $\alpha^-(p)$ is the largest set of mappable states $X \subseteq \Sigma_A$

such that $\alpha^{-1}(X) \subseteq \|p\|$, where $\|p\|$ represents the set of states which satisfy the assertion p. If $\alpha^-(p)$ is valid, then $\|\alpha^-(p)\| = \Sigma_A$ implying $\alpha^{-1}(\|\alpha^-(p)\|) = \Sigma$ which, by the above inclusion, leads to $\|p\| = \Sigma$ establishing the validity of p.

For complex formulas, we have to consider assertions which are nested within an odd number of negations. To abstract an assertion under such a context, we define the operator α^+, dual to α^-, as follows

$$\alpha^+(p(V)): \quad \exists V \left(V_A = \mathcal{E}^\alpha(V) \quad \wedge \quad p(V) \right).$$

The assertion $\alpha^+(p)$ holds for an abstract state $S \in \Sigma_A$ iff the assertion p holds for *some* concrete state $s \in \Sigma$ such that $s \in \alpha^{-1}(S)$, i.e., some state s such that $S = \alpha(s)$. Alternatively, $\alpha^+(p)$ is the smallest set $X \subseteq \Sigma_A$ such that $\|p\| \subseteq \alpha^{-1}(X)$.

An abstraction α is said to be *precise with respect to an assertion* p if $\alpha^+(p) \sim \alpha^-(p)$. A sufficient condition for α to be precise with respect to p is that the abstract variables include a Boolean variable B_p with the definition $B_p = p$.

Having defined the abstractions α^- and α^+ which operate on assertions, we lift them to the abstractions α_τ^- and α_τ^+ which can be applied to temporal formulas. These temporal abstractions are defined inductively, as presented in Fig. 8.

We respectively refer to $\alpha_\tau^-(p)$ and $\alpha_\tau^+(p)$ as the *universal* (or *contracting*) and *existential* (or *expanding*) abstraction of the formula p.

Note that equivalent temporal formulas may have different abstractions. For example, the contracting abstractions of the equivalent formulas

$$p \vee (q \vee \Diamond r) \quad \text{and} \quad (p \vee q) \vee \Diamond r,$$

where p, q, and r are assertions (state formulas) are respectively given by the formulas

$$\alpha^-(p) \vee \alpha^-(q) \vee \Diamond \alpha^-(r) \quad \text{and} \quad \alpha^-(p \vee q) \vee \Diamond \alpha^-(r),$$

which may be inequivalent. Similarly, the respective abstractions of

$$p \wedge (q \wedge \Box \mathrm{T}) \quad \text{and} \quad p \wedge q$$

are

$$\alpha^+(p) \wedge \alpha^+(q) \quad \text{and} \quad \alpha^+(p \wedge q).$$

Claim 4. Let ψ be a temporal formula and α be an abstraction mapping. Then

$$\models \alpha_\tau^-(\psi) \text{ implies } \models \psi \quad \text{and} \quad \models \psi \text{ implies } \models \alpha_\tau^+(\psi)$$

The proof of this claim appears in [18].

In the following sections, we denote by ψ^α the contracting abstraction $\alpha_\tau^-(\psi)$ of the temporal formula ψ.

7.2 Safe abstraction of FDS's

In the previous subsection, we established that the abstraction of the temporal formula ψ into $\psi^\alpha = \alpha_\tau^-(\psi)$ is *safe* (equivalently *sound*) in the sense that if ψ^α is valid, then so is ψ.

Here we will establish sufficient conditions for the joint abstraction of the FDS $\mathcal{D}$ and the temporal formula ψ to be safe (sound) in the sense that $\mathcal{D}^\alpha \models \psi^\alpha$ implies $\mathcal{D} \models \psi$. To do so, we reduce the problem of the safe joint abstraction of an FDS and a temporal property into the problem of safe abstraction of a single temporal property, a problem that has been solved in the preceding subsection.

Given an FDS $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, there exists a temporal formula $Sem(\mathcal{D})$, called the *temporal semantics* of $\mathcal{D}$ [18], such that, for every infinite state sequence σ, it holds that $\sigma \models Sem(\mathcal{D})$ *iff* $\sigma \in \mathcal{C}omp(\mathcal{D})$. The temporal semantics of an FDS $\mathcal{D}$ is given by

$$Sem(\mathcal{D}): \quad \begin{pmatrix} \Theta(V) \wedge \Box \rho(V, {}^\circ V) \wedge \\ \bigwedge_{J \in \mathcal{J}} \quad \Box\Diamond J(V) \quad \wedge \\ \bigwedge_{(p,q) \in \mathcal{C}} \quad (\Box\Diamond p(V) \rightarrow \Box\Diamond q(V)) \end{pmatrix},$$

where we use the temporal expression ${}^\circ V$ to denote the *next values* of the system variables V. Given a verification problem $\mathcal{D} \stackrel{?}{\models} \psi$, we construct the temporal formula

$$Ver(\mathcal{D}, \psi): \quad Sem(\mathcal{D}) \rightarrow \psi.$$

It is not difficult to establish that $\mathcal{D} \models \psi$ iff $Ver(\mathcal{D}, \psi)$ is valid.

For a state formula p,

$\alpha_\tau^-(p) = \alpha^-(p)$	$\alpha_\tau^+(p) = \alpha^+(p)$

For a formula $\varphi \in \{\neg p, p \vee q, \circ p, p \mathcal{U} q\}$, which is not a state formula,

$\alpha_\tau^-(\neg p) = \neg \alpha_\tau^+(p)$	$\alpha_\tau^+(\neg p) = \neg \alpha_\tau^-(p)$
$\alpha_\tau^-(p \vee q) = \alpha_\tau^-(p) \vee \alpha_\tau^-(q)$	$\alpha_\tau^+(p \vee q) = \alpha_\tau^+(p) \vee \alpha_\tau^+(q)$
$\alpha_\tau^-(\circ p) = \circ \alpha_\tau^-(p)$	$\alpha_\tau^+(\circ p) = \circ \alpha_\tau^+(p)$
$\alpha_\tau^-(p \mathcal{U} q) = (\alpha_\tau^-(p)) \mathcal{U} (\alpha_\tau^-(q))$	$\alpha_\tau^+(p \mathcal{U} q) = (\alpha_\tau^+(p)) \mathcal{U} (\alpha_\tau^+(q))$

Fig. 8. Abstractions of temporal formulas

Applying a safe α-abstraction to $\mathit{Ver}(\mathcal{D},\psi)$, we obtain

$$\alpha_\tau^-(\mathit{Ver}(\mathcal{D},\psi)) = \begin{pmatrix} \alpha^+(\Theta) \wedge \Box\alpha^{++}(\rho) & \wedge \\ \bigwedge_{J\in\mathcal{J}} \Box\Diamond(\alpha^+(J) & \wedge \\ \bigwedge_{(p,q)\in\mathcal{C}} (\Box\Diamond\alpha^-(p) \rightarrow \Box\Diamond(\alpha^+(q)) & \end{pmatrix}$$

where

$$\alpha^{++}(\rho)\colon \quad \exists V, {}^\circ V\colon \quad \begin{pmatrix} V_A = \mathcal{E}^\alpha(V) & \wedge \\ {}^\circ V_A = \mathcal{E}^\alpha({}^\circ V) & \wedge \\ \rho(V, {}^\circ V) & \end{pmatrix}$$

Based on the way $\alpha_\tau^-(\mathit{Ver}(\mathcal{D},\psi))$ abstracts the different components of $\mathcal{D}$, we define the *α-abstracted version* of $\mathcal{D}$ to be the FDS $\mathcal{D}^\alpha = \langle V_A, \Theta^\alpha, \rho^\alpha, \mathcal{J}^\alpha, \mathcal{C}^\alpha\rangle$, where

$$\Theta^\alpha = \alpha^+(\Theta) \qquad \rho^\alpha = \alpha^{++}(\rho)$$
$$\mathcal{J}^\alpha = \{\alpha^+(J) \mid J\in\mathcal{J}\}$$
$$\mathcal{C}^\alpha = \{(\alpha^-(p), \alpha^+(q)) \mid (p,q)\in\mathcal{C}\}$$

The following claim defines our recipe for verification by data abstraction and states its soundness (safety).

Claim 5. (Soundness) The abstraction method which, for a given α, abstracts ψ into $\alpha_\tau^-(\psi)$ and abstracts $\mathcal{D}$ into $\mathcal{D}^\alpha = \langle V_A, \Theta^\alpha, \rho^\alpha, \mathcal{J}^\alpha, \mathcal{C}^\alpha\rangle$, is safe. That is,

$\mathcal{D}^\alpha \models \psi^\alpha$ implies $\mathcal{D} \models \psi$.

Proof. An immediate consequence of claim 4 and the definitions of $\mathcal{D}^\alpha$ and ψ^α. □

As an example, we consider program BAKERY-2, presented in Fig. 9.

Program BAKERY-2 is obviously an infinite-state system, since the variables y_1 and y_2 can assume arbitrarily large values.

The temporal properties we wish to establish are given by

$$\psi_{exc}\colon \Box\neg(at\_\ell_4 \wedge at\_m_4)$$
$$\psi_{acc}\colon \Box(at\_\ell_2 \rightarrow \Diamond at\_\ell_4),$$

The safety property ψ_{exc} requires *mutual exclusion*, guaranteeing that the two processes never co-reside in their respective critical section at the same time. The liveness property ψ_{acc} requires *accessibility* for process P_1, guaranteeing that, whenever P_1 reaches location ℓ_2 it will eventually reach location ℓ_4.

Following [3], we define abstract Boolean variables $B_{p_1}, B_{p_2}, \ldots, B_{p_k}$, one for each atomic data formula, where the atomic data formulas for BAKERY-2 are $y_1 = 0$, $y_2 = 0$, and $y_1 < y_2$. Note that the formula $y_2 \le y_1$ is equivalent to the negation of $y_1 < y_2$ and needs not be included as an independent atomic formula.

The abstract system variables consist of the concrete control variables, which are left unchanged, and a set of abstract Boolean variables $B_{p_1}, B_{p_2}, \ldots, B_{p_k}$. The abstraction mapping α is defined by

$$\alpha\colon \quad \{B_{p_1} = p_1, B_{p_2} = p_2, \ldots, B_{p_k} = p_k\}$$

That is, the Boolean variable B_{p_i} has the value *true* in the abstract state iff the assertion p_i holds at the corresponding concrete state.

local y_1, y_2 : **natural where** $y_1 = y_2 = 0$

P_1	P_2
ℓ_0 : **loop forever do**	m_0 : **loop forever do**
ℓ_1 : **NonCritical**	m_1 : **NonCritical**
ℓ_2 : $y_1 := y_2 + 1$	m_2 : $y_2 := y_1 + 1$
ℓ_3 : **await** $y_2 = 0 \vee y_1 < y_2$	m_3 : **await** $y_1 = 0 \vee y_2 \le y_1$
ℓ_4 : **Critical**	m_4 : **Critical**
ℓ_5 : $y_1 := 0$	m_5 : $y_2 := 0$

Fig. 9. Program BAKERY-2: the Bakery algorithm for two processes

local $B_{y_1=0}, B_{y_2=0}, B_{y_1<y_2}$: **boolean initially** $B_{y_1=0} = B_{y_2=0} = 1, B_{y_1<y_2} = 0$

P_1	P_2
ℓ_0 : **loop forever do**	m_0 : **loop forever do**
ℓ_1 : **NonCritical**	m_1 : **NonCritical**
ℓ_2 : $(B_{y_1=0}, B_{y_1<y_2}) := (0,0)$	m_2 : $(B_{y_2=0}, B_{y_1<y_2}) := (0,1)$
ℓ_3 : **await** $B_{y_2=0} \vee B_{y_1<y_2}$	m_3 : **await** $B_{y_1=0} \vee \neg B_{y_1<y_2}$
ℓ_4 : **Critical**	m_4 : **Critical**
ℓ_5 : $(B_{y_1=0}, B_{y_1<y_2}) := (1, \neg B_{y_2=0})$	m_5 : $(B_{y_2=0}, B_{y_1<y_2}) := (1,0)$

Fig. 10. Program BAKERY-2: the Bakery algorithm for two processes

It is straightforward to compute the α-induced abstractions of the initial condition Θ^α and the transition relation ρ^α. In Fig. 10, we present program BAKERY-2 (with a capital B), the α-induced abstraction of program BAKERY-2.

Since the properties we wish to verify refer only to the control variables (through the at_ℓ and at_m expressions), they are not affected by the abstraction. Program BAKERY-2 is a finite-state program, and we can apply model checking to verify that it satisfies the two properties of mutual exclusion and accessibility. By Claim 5, we can infer that the original program BAKERY-2 also satisfies these two temporal properties.

7.3 Augmentation by progress monitors

Program BAKERY-2 is an example of successful data abstraction. However, there are cases when abstraction alone is inadequate for transforming an infinite-state system satisfying a property into a finite-state abstraction which maintain the property. In the following we illustrate the problem and the proposed solution on a simple example. For the treatment of the general case, see [18]. In Fig. 11, we present a simple looping program. The assignment at statement ℓ_2 assigns to y non deterministically the values $y+1$ or y. The property we wish to verify is that program SUB-ADD always terminates, independently of the initial value of the natural variable y.

A natural abstraction for the variable y is defined by

$$Y = \begin{pmatrix} \textbf{if } y = 0 \textbf{ then } \textit{zero} \\ \textbf{else if } y = 1 \textbf{ then } \textit{one} \\ \textbf{else } \textit{large}, \end{pmatrix}$$

where y is abstracted into the three-valued domain

$$\{\textit{zero}, \textit{one}, \textit{large}\}.$$

However, applying this abstraction yields the abstract program SUB-ADD-ABS-1, presented in Fig. 12, where the abstract functions *sub2* and *add1* are defined by

$$\textit{sub2}(Y) = \begin{pmatrix} \textbf{if } Y = \{\textit{zero}, \textit{one}\} \textbf{ then } \textit{zero} \\ \textbf{else } \{\textit{zero}, \textit{one}, \textit{large}\}, \end{pmatrix}$$

$$\textit{add1}(Y) = \begin{pmatrix} \textbf{if } Y = \textit{zero} \textbf{ then } \textit{one} \\ \textbf{else } \textit{large}. \end{pmatrix}$$

Unfortunately, program SUB-ADD-ABS-1 need not terminate, because the function *sub2* can always choose to yield *large* as a result.

Termination of programs like program SUB-ADD can always be established by identification of a *progress measure* that never increases and sometimes is guaranteed to decrease. In this case, for example, we can use the progress measure $\delta : y + at\_\ell_2$ which never increases and always decreases on the execution of statement ℓ_1. To obtain a working abstraction, we first compose program SUB-ADD with an additional module, to which we refer as the *progress monitor* for the progress measure δ, as shown in Fig. 13.

The construct **always do** appearing in MONITOR M_δ means that the assignment which is the body of this construct is executed at *every* step. The comparison function $\textit{comp}(\delta, \delta')$ is defined by

$$\textit{comp}(\delta, \delta') = \begin{pmatrix} \textbf{if } \delta < \delta' \textbf{ then } 1 \\ \textbf{else if } \delta = \delta' \textbf{ then } 0 \\ \textbf{else } -1. \end{pmatrix}$$

Note that the expressions on the right-hand-side of the assignments in the monitor allow references to the *new* values of δ as computed in the same step by the monitored program.

The presentation of the monitor module M_δ in Fig. 13 is only for illustration purposes. The precise definition of this module is given by the following FDS:

$V = V_{\mathcal{D}} \cup \{inc : \{-1, 0, 1\}\}$
$\Theta : \mathrm{T}$
$\rho : inc' = \textit{comp}(\delta, \delta')$
$\mathcal{J} : \emptyset, \qquad \mathcal{C} : \{(inc < 0 inc > 0)\}$

$$\begin{array}{c} y\colon \textbf{ natural} \\ \ell_0 : \textbf{while } y > 1 \textbf{ do} \\ \quad \begin{bmatrix} \ell_1 : y := y - 2 \\ \ell_2 : y := \{y+1, y\} \\ \ell_3 : \textbf{skip} \end{bmatrix} \\ \ell_4 : \end{array}$$

Fig. 11. Program SUB-ADD

$$\begin{array}{c} Y\colon \{\textit{zero}, \textit{one}, \textit{large}\} \\ \ell_0 : \textbf{while } Y = \textit{large} \textbf{ do} \\ \quad \begin{bmatrix} \ell_1 : \quad Y := \textit{sub2}(Y) \\ \ell_2 : \quad Y := \{\textit{add1}(Y), Y\} \\ \ell_3 : \textbf{skip} \end{bmatrix} \\ \ell_4 : \end{array}$$

Fig. 12. Program SUB-ADD-ABS-1 abstracting program SUB-ADD

$$\begin{array}{c} y\colon \textbf{ natural} \\ \begin{bmatrix} \ell_0 : \textbf{while } y > 1 \textbf{ do} \\ \quad \begin{bmatrix} \ell_1 : y := y - 2 \\ \ell_2 : y := \{y+1, y\} \\ \ell_3 : \textbf{skip} \end{bmatrix} \\ \ell_4 : \end{bmatrix} \;|||\; \begin{bmatrix} \textbf{define } \delta = y + at\_\ell_2 \\ inc \quad : \{-1, 0, 1\} \\ m_0 : \textbf{always do} \\ \quad inc := \textit{comp}(\delta, \delta') \end{bmatrix} \\ \text{– SUB-ADD –} \qquad\qquad \text{– MONITOR } M_\delta \text{ –} \end{array}$$

Fig. 13. Program SUB-ADD composed with a monitor

where $V_{\mathcal{D}}$ are the system variables of the monitored FDS $\mathcal{D}$. Thus, at every step of the computation, module M_δ compares the new value of δ (δ') with the current value, and sets variable *inc* to -1, 0, or 1, according to whether the value of δ has decreased, stayed the same, or increased, respectively. This FDS has no justice requirements but has the single compassion requirement $(inc < 0, inc > 0)$ stating that δ cannot decrease infinitely many times without also increasing infinitely many times. This requirement is a direct consequence of the fact that δ ranges over the well-founded domain of the natural numbers, which does not allow an infinitely decreasing sequence.

It is possible to represent this composition as (almost) equivalent to the sequential program presented in Fig. 14, where we have conjoined the repeated assignment of module M_δ with every assignment of process SUB-ADD. The "almost" qualification admits that we did not conjoin this assignment with the transition associated with location ℓ_0 which tests the value of y and decides when to terminate. In a fully formal treatment of this example, the assignment will also be conjoined to this testing transition.

The abstraction of the program of Fig. 14 will abstract y into a variable Y ranging over $\{zero, one, large\}$. The variable *inc*, ranging over the finite domain $\{-1, 0, 1\}$, is not abstracted. The resulting abstraction is presented in Fig. 15.

The program SUB-ADD-ABS-2 (Fig. 15) differs from program SUB-ADD-ABS-1 (Fig. 12) by the additional compassion requirement $(inc < 0, inc > 0)$. However, it is this additional requirement which forces program SUB-ADD-ABS-2 to terminate. This is because a run in which *sub1* always yields *large* as a result is a run in which *inc* is negative infinitely many times (on every visit to ℓ_1) and is never positive beyond the first state. The fact that SUB-ADD-ABS-2 always terminates can now be successfully model-checked.

The extension to the case that the progress measure ranges not over the naturals but over lexicographic tuples of naturals is straightforward.

$$
\begin{array}{l}
\qquad\quad y \;:\; \textbf{natural} \\
\qquad\quad inc\colon\ \{-1,0,1\} \\
\ell_0 : \textbf{while } y > 0 \textbf{ do} \\
\quad \left[\begin{array}{ll}
\ell_1 : (y, inc) := & (y-2,\ comp(\delta, \delta')) \\
\ell_2 : (y, inc) := & (\{y+1, y\},\ comp(\delta, \delta')) \\
\ell_3 : \quad inc := & comp(\delta, \delta')
\end{array}\right] \\
\ell_4 :
\end{array}
$$

Fig. 14. A sequential equivalent of the monitored program

$$
\begin{array}{l}
\qquad Y \quad\quad : \{zero, one, large\} \\
\qquad inc \quad\quad : \{-1, 0, 1\} \\
\qquad \textbf{compassion}\ (inc < 0, inc > 0) \\
\ell_0 : \textbf{while } Y = large \textbf{ do} \\
\quad \left[\begin{array}{l}
\ell_1 : (Y, inc) := (sub2(Y),\ -1) \\
\ell_2 : (Y, inc) := (\{add1(Y), Y\},\ \{0, -1\}) \\
\ell_3 : \quad inc := 0
\end{array}\right] \\
\ell_4 :
\end{array}
$$

Fig. 15. Abstracted version of the monitored- Program SUB-ADD-ABS-2

7.4 *The data abstraction method is complete*

In a separate work [18], concentrating on the data abstraction method, we have established that this method is relatively *complete*. Completeness in this context means that for every (possibly infinite) system $\mathcal{D}$ and a temporal property ψ, such that $\mathcal{D} \models \psi$, there exists a (progress) monitor M_δ whose composition with $\mathcal{D}$ does not constrain the computations of $\mathcal{D}$, and a finitary state abstraction mapping α, such that $(\mathcal{D} \| M_\delta)^\alpha \models \psi^\alpha$. This implies that whenever ψ is a property valid for $\mathcal{D}$, we can apply the method of data abstraction described in this section in order to formally verify that ψ is $\mathcal{D}$-valid.

8 Conclusions

The paper presented two central techniques for reducing a big verification task into several smaller ones. These techniques are especially impressive when they reduce an infinite-state system into a finite-state one.

The first technique is based on *control abstraction* and reduces an unbounded environment for a single module into an abstract environment model which represents the relevant features of the environment. Often, the unbounded environment represents a set of brother processes and the derived abstract model represents a network-invariant which is independent of the size of this set.

The second technique is that of *data abstraction* in which variables ranging over infinite domains are abstracted into variables ranging over finite domains. The method presents a general recipe for computing such an abstraction for every user-provided state mapping, such that the abstraction preserves all counter-examples to any temporal property. This means that if the property has been verified to be valid on the abstract level, its concrete version is guaranteed to be valid (no false positives).

An important feature of our formulations of these two methods is that they are not restricted to the verification of safety properties as are many previous formulations of similar approaches, but deal quite effectively with liveness properties, in fact with all temporally expressible properties. On the system side, they take full account of both weak (justice) and strong (compassion) fairness requirements.

Acknowledgements. We gratefully acknowledge the valuable contribution of Elad Shahar who implemented rule MOD-ABST in his TLV system. We thank Monica Marcus for her help in debugging various versions of the rule.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 82(2): 253–284, May 1991
2. Bjørner, N., Browne, I.A., Chang, E., Colón, M., Kapur, A., Manna, Z., Sipma, H.B., Uribe, T.E.: STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995
3. Bjørner, N., Browne, I.A., Manna, Z.: Automatic generation of invariants and intermediate assertions. In: 1^{st} Intl. Conf. on Principles and Practice of Constraint Programming. LNCS 976. Berlin, Heidelberg, New York: Springer-Verlag, 1995, pp. 589–623
4. Browne, M.C., Clarke, E.M., Grumberg, O.: Reasoning about networks with many finite state processes. In: Proc. 5th ACM Symp. Princ. of Dist. Comp., pp. 240–248, 1986
5. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. In: Dill, D.L. (ed.): Proc. 6th Conference on Computer Aided Verification. LNCS 818. Berlin, Heidelberg, New York: Springer-Verlag, 1994, pp. 415–427
6. Clarke, E.M., Grumberg, O., Jha, S.: Verifying parametrized networks using abstraction and regular languages. In: 6th International Conference on Concurrency Theory (CONCUR '95), pp. 395–407, Philadelphia, PA, August 1995
7. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Prog. Lang. Sys. 16(5): 1512–1542, 1994
8. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking. In: Model Checking, Abstraction and Composition, Nato ASI Series Fvolume 152. Berlin, Heidelberg, New York: Springer-Verlag, 1996, pp. 477–498
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th Annual Symposium on Principles of Programming Languages. ACM Press, 1977
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. 5th ACM Symp. Princ. of Prog. Lang., pp. 84–96, 1978
11. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Trans. Prog. Lang. Sys. 19(2), 1997
12. Emerson, E.A., Namjoshi, K.S.: Automatic verification of parameterized synchronous systems. In: Alur, R., Henzinger, T. (eds.): Proc. 8^th Intl. Conference on Computer Aided Verification (CAV '96). LNCS. Berlin, Heidelberg, New York: Springer-Verlag, 1996
13. Emerson, E.A., Namjoshi, K.S.: Automatic verification of parameterized synchronous systems. In: Alur, R., Henzinger, T. (eds.): Proc. 8^th Intl. Conference on Computer Aided Verification (CAV '96). LNCS. Berlin, Heidelberg, New York: Springer-Verlag, 1996
14. Graf, S., Loiseaux, C.: A tool for symbolic program verification and abstraction. In: Courcoubetis, C. (ed.): Proc. 5^th Intl. Conference on Computer Aided Verification (CAV '94). LNCS 697. Berlin, Heidelberg, New York: Springer-Verlag, 1993, pp. 71–84
15. Halbwachs, N., Lagnier, F., Ratel, C.: An experience in proving regular networks of processes by modular model checking. Acta Informatica 29(6/7): 523–543, 1992
16. Ip, C.N., Dill, D.: Verifying systems with replicated components in Murφ. In: Alur, R., Henzinger, T. (eds.): Proc. 8^th Intl. Conference on Computer Aided Verification (CAV '96). LNCS. Berlin, Heidelberg, New York: Springer-Verlag, 1996
17. Johnsson, B.: Compositional specification and verification of distributed systems. ACM Trans. Prog. Lang. Sys. 16(2): 259–303, 1994
18. Kesten, Y., Pnueli, A.: Verification by augmented finitary abstraction. Information and Computation, a special issue. To appear in 2000
19. Kesten, Y., Pnueli, A., Raviv, L.: Algorithmic verification of linear temporal logic specifications. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.): Proc. 25th Int. Colloq. Aut. Lang. Prog. LNCS 1443. Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 1–16
20. Kurshan, R.P., McMillan, K.L.: A structural induction theorem for processes. Information and Computation 117: 1–11, 1995
21. Lam, S.S., Shankar, A.U.: Refinement and projection of relational specifications. In: Stepwise refinement of distributed systems. Models, formalismus, correctness. LNCS 430. Berlin, Heidelberg, New York: Springer-Verlag, 1990, pp. 454–486
22. Lehmann, D., Pnueli, A., Stavi, J.: Impartiality, justice and fairness: The ethics of concurrent termination. In: Proc. 8th Int. Colloq. Aut. Lang. Prog. LNCS 115. Berlin, Heidelberg, New York: Springer-Verlag, 1981, pp. 264–277
23. Lesens, D., Halbwachs, N., Raymond, P.: Automatic verification of parameterized linear networks of processes. In: 24th ACM Symposium on Principles of Programming Languages, POPL '97, Paris, 1997
24. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. Formal methods in system design 6(1): 11–44, 1995
25. Lynch, N., Tuttle, M.: Hierarchical correctness proofs for distributed algorithms. In: Proc. 6th ACM Symp. Princ. of Dist. Comp., 1987, pp. 137-151
26. Manna, Z., Anuchitanukul, A., Bjørner, N., Browne, A., Chang, E., Colón, M., De Alfaro, L., Devarajan, H., Sipma, H., Uribe, T.E.: STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994
27. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems: specification. Springer-Verlag, New York, 1991
28. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Berlin, Heidelberg, New York: Springer-Verlag, New York, 1995
29. Orava, F.: Verifying safety and deadlock properties of networks of asynchronously communicating processes. In: Protocol specifications, testing and verification IX. North-Holland, 1989, pp. 352–372
30. Pnueli, A.: The temporal semantics of concurrent programs. Theoretical Computer Science 13:1–20, 1981
31. Pnueli, A., Shahar, E.: A platform for combining deductive with algorithmic verification. In: Alur, R., Henzinger, T. (eds.): Proc. 8^th Intl. Conference on Computer Aided Verification (CAV '96). LNCS. Berlin, Heidelberg, New York: Springer-Verlag, 1996, pp. 184–195
32. Shtadler, Z., Grumberg, O.: Network grammars, communication behaviors and automatic verification. In: Sifakis, J. (ed.): Automatic Verification Methods for Finite State Systems, pp. 151–165. LNCS 407. Berlin, Heidelberg, New York: Springer-Verlag, 1989
33. Sistla, A.P., German, S.M.: Reasoning about systems with many processes. J. ACM 39: 675–735, 1992
34. Stark, E.W.: Proving entailments between conceptual state specifications. Theory. Comp. Sci. 56: 135–154, 1988
35. Wolper, P., Lovinfosse, V.: Verifying properties of large sets of processes with network invariants. In: Sifakis, J. (ed.): Automatic Verification Methods for Finite State Systems. LNCS 407. Berlin, Heidelberg, New York: Springer-Verlag, 1989, pp. 68–80