In *23rd Int. Symp. Mathematical Foundations of Computer Science*, MFCS-98, Vol. 1450 of LNCS, pp. 54–71, Springer Verlag, 1998.

Modularization and Abstraction: The Keys to Practical Formal Verification*

Yonit Kesten[1] and Amir Pnueli[2]

[1] Ben Gurion University, ykesten@bgumail.bgu.ac.il,
[2] Weizmann Institute of Science, 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 – 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 is by the extensive use of abstraction and modularization, which breaks the task of verifying a large system into several smaller tasks of verifying simpler systems.
In this methodological 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 modest 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*.

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 methodological 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.

* 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.

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 ([EN95], [SG92]) network invariants, which can be viewed as implicit induction ([KM95], [WL89], [HLR92], [LHR97]), methods that can be viewed as abstraction and approximation of network invariants ([BCG86], [SG89], [CGJ95]), and other methods that can be viewed as based on abstraction ([ID96], [EN96]). The approach described here is based on the idea of *network invariants* as introduced in [WL89], and elaborated in [KM95] into a working method.

There has been extensive study of the use of data abstraction techniques, mostly based on the notions of *abstract interpretation* ([CC77], [CH78]). 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 [CGL94] 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 $\mathcal{K}^\alpha$ and show how to translate (concretize) it into a concrete formula $\psi = \mathcal{C}(\Psi)$. The concretization is such that $M_\forall^\alpha(\psi) = \Psi$.

The survey in [CGL96] 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 [DGG97] 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 [BBLS92] and [GL93].

A linear-time application of abstract interpretation is proposed in [BBM95], applying the abstractions directly to the computational model of *fair transition systems* which is very close to the FKS 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 Kripke Structure

As a computational model for reactive systems, we take the model of *fair kripke structure* (FKS) [KPR98], which is a slight variation on the model of *fair transition system* [MP95]. Such a system $\mathcal{K} : \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ 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.
- $W = \{w_1, \ldots, w_n\} \subseteq V$: 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 whose values (and identities) must be preserved in some of the abstractions we will consider. It is required that $V = W \cup \mathcal{O}$, i.e., for every system variable $u \in V$, u is either *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 $\mathcal{K}$-successor state $s' \in \Sigma$.
- $\mathcal{J} = \{J_1, \ldots, J_k\}$: A set of *justice* requirements (also called *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* requirements (also called *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 $\mathcal{K}$-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 $\mathcal{K}$-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 $\mathcal{K}$ be an FKS for which the above components have been identified. We define a *computation* of $\mathcal{K}$ 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 $\mathcal{K}$-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 FKS $\mathcal{K}$, we denote by $\mathit{Comp}(\mathcal{K})$ the set of all computations of $\mathcal{K}$. An FKS $\mathcal{K}$ is called *feasible* if $\mathit{Comp}(\mathcal{K}) \neq \emptyset$, namely, if $\mathcal{K}$ has at least one computation. The feasibility of an FKS can be checked algorithmically, using symbolic model checking methods, as presented in [KPR98].

An infinite state sequence σ is called a *run* of $\mathcal{K}$ if it satisfies the requirements of initiality and consecution but not, necessarily, any of the fairness requirements.

All our concrete examples are given in SPL (Simple Programming Language), which is used to represent concurrent programs (e.g., [MP95], [MAB+94]). Every SPL program can be compiled into an FKS 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

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

The predicates $\mathit{at\_}\ell_0$ and $\mathit{at\_}\ell_1'$ 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.

3 Operations on FKS's

There are several important operations, one may wish to apply to FKS'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 Kripke structures $\mathcal{K}_1 = \langle V_1, W_1, \mathcal{O}_1, \Theta_1, \rho_1, \mathcal{J}_1, \mathcal{C}_1\rangle$ and $\mathcal{K}_2 = \langle V_2, W_2, \mathcal{O}_2, \Theta_2, \rho_2, \mathcal{J}_2, \mathcal{C}_2\rangle$. There are several ways of forming their parallel composition.

3.1 Asynchronous Parallel Composition

The systems $\mathcal{K}_1$ and $\mathcal{K}_2$ are said to be *compatible* if $W_1 \cap W_2 = \emptyset$ and $V_1 \cap V_2 \subseteq \mathcal{O}_1 \cap \mathcal{O}_2$. 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 compatible systems $\mathcal{K}_1$ and $\mathcal{K}_2$, we define their asynchronous parallel composition, denoted by $\mathcal{K}_1 \| \mathcal{K}_2$, to be the system $\mathcal{K} = \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$, where

$$\begin{array}{lll} V = V_1 \cup V_2 & W = W_1 \cup W_2 & \mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2 \\ \Theta = \Theta_1 \wedge \Theta_2 & \mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2 & \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}$$
$$\rho = (\rho_1 \wedge \mathit{pres}((V_2 - V_1) \cup W_2)) \ \vee\ (\rho_2 \wedge \mathit{pres}((V_1 - V_2) \cup W_1)).$$

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

As seen from the definition, $\mathcal{K}_1$ and $\mathcal{K}_2$ may have disjoint as well as common system variables, and the variables of $\mathcal{K}$ are the union of all of these variables. The initial condition of $\mathcal{K}$ is the conjunction of the initial conditions of $\mathcal{K}_1$ and $\mathcal{K}_2$. The transition relation of $\mathcal{K}$ states that at any step, we may choose to perform a step of $\mathcal{K}_1$ or a step of $\mathcal{K}_2$. 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 $\mathcal{K}_1$, we should preserve all variables in $V_2 - V_1$ and all the variables owned by $\mathcal{K}_2$.

The justice and compassion sets of $\mathcal{K}$ 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 $\mathcal{K}_1$ and $\mathcal{K}_2$, denoted by $\mathcal{K}_1 ||| \mathcal{K}_2$, to be the system $\mathcal{K} = \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where,

$$
\begin{array}{lll}
V = V_1 \cup V_2 & W = W_1 \cup W_2 & \mathcal{O} = \mathcal{O}_1 \cup \mathcal{O}_2 \quad \Theta = \Theta_1 \wedge \Theta_2 \\
\mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2 & \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 & \rho = \rho_1 \wedge \rho_2.
\end{array}
$$

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

As will be shown in the next section, the main use of the 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.

3.3 Modularization of an FKS

Let P be an SPL program and $\mathcal{K}$ its corresponding FKS. The standard compilation of a program into an FKS 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 FKS, which takes into account not only actions performed by the system but also actions (in particular, variable changes) performed by the environment.

Let $\mathcal{K} : \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}_K, \mathcal{C}_K \rangle$ be an FKS, such that $s \notin V$. The *modular version* of $\mathcal{K}$, is given by $\mathcal{K}_M : \langle V_M, W_M, \mathcal{O}_M, \Theta_M, \rho_M, \mathcal{J}_M, \mathcal{C}_M \rangle$, where,

$$
\begin{array}{lll}
V_M = V \cup \{s\} & W_M = W & \mathcal{O}_M = \mathcal{O} \cup \{s\} \\
\Theta_M = \Theta & \mathcal{J}_M = \mathcal{J} & \mathcal{C}_M = \mathcal{C} \\
\rho_M = (\rho \wedge s') \vee (W' = W \wedge \neg s'). & &
\end{array}
$$

That is, $\mathcal{K}_M$ the modular version of $\mathcal{K}$ allows as an additional action a transition which preserves the values of all variables owned by $\mathcal{K}$ 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 a system obtained as the modularization of another FKS as a *Fair Kripke Module* (FKM).

Formally, an FKM is an FKS $\mathcal{K}_M$, with a special boolean variable $s \in V - W$, to which we refer as the *scheduling variable*, satisfying

$$\Theta \rightarrow\ s = 0 \qquad \text{and} \qquad s' = 0\ \rightarrow\ (\rho \iff (W' = W)).$$

The second implication requires that no owned variable is modified by an environment step. It also implies that the module does not constrain the way the environment modifies the variables not owned by the module. That is, any step in which $s' = 0$ (implying that this is an environment step) and the owned variables are preserved must be allowed by ρ.

We define a *modular computation* of $\mathcal{K}$ to be any computation of $\mathcal{K}_M$. A property φ is said to be *modularly valid* over FKS $\mathcal{K}$, denoted $\mathcal{K} \models_M \varphi$, if φ is $\mathcal{K}_M$-valid.

3.4 Modular Composition

We define the *modular composition* of the compatible FKM's $\mathcal{K}_1$ and $\mathcal{K}_2$, denoted by $\mathcal{K}_1 \|_M \mathcal{K}_2$, to be the FKM $\mathcal{K}_M : \langle V_M, W_M, \mathcal{O}_M, \Theta_M, \rho_M, \mathcal{J}_M, \mathcal{C}_M \rangle$, where,

$$\begin{array}{lll} V_M = V_1 \cup V_2 & W_M = W_1 \cup W_2 & \mathcal{O}_M = \mathcal{O}_1 \cup \mathcal{O}_2 \\ \Theta_M = \Theta_1 \wedge \Theta_2 & \mathcal{J}_M = \mathcal{J}_1 \cup \mathcal{J}_2 & \mathcal{C}_M = \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}$$
$$\rho_M = \exists s_1', s_2' : \text{boolean} : (s' = s_1' \vee s_2')\ \wedge\ \neg(s_1' \wedge s_2')\ \wedge\ \rho_1[s \mapsto s_1]\ \wedge\ \rho_2[s \mapsto s_2]$$

A step in the execution of $\mathcal{K}_M$ is either a step of system $\mathcal{K}_1$ where $s' = s_1' = 1$ and $s_2' = 0$, or a step of system $\mathcal{K}_2$ where $s' = s_2' = 1$ and $s_1' = 0$, or an environment step where $s' = s_1' = s_2' = 0$. A step of $\mathcal{K}_1$ (similarly $\mathcal{K}_2$) is governed by the transition relation $\rho_1[s \mapsto s_1]$, which is the assertion ρ_1 in which all references to s and s' are replaced by references to s_1 and s_1', respectively.

The operation of modular composition is commutative and associative.

For closed-system $\mathcal{K}_1$ and $\mathcal{K}_2$ which are composed in parallel, we can first modularize each of the systems individually and then form their modular composition. Alternately, we can form first the asynchronous parallel composition of the two systems and then modularize the combined system. It can be seen that both processes yield the same FKM, which is expressible by the equivalence

$$(\mathcal{K}_1)_M \|_M (\mathcal{K}_2)_M \quad \sim \quad (\mathcal{K}_1 \| \mathcal{K}_2)_M.$$

3.5 Sealing Off an Open System

Assume we have an FKM, consisting of a modular composition of several FKM's:

$$\mathcal{K}_M = \mathcal{K}_1 \|_M \cdots \|_M \mathcal{K}_K.$$

System $\mathcal{K}_{_M}$ is still an open system, admitting arbitrary interference by the environment. Once we know that all the processes in the system have been included (possibly including a process that represents the environment) and no further interaction with the external world is expected, way may *seal off* the system, formally excluding any further external communication. This is done by declaring all variables to be owned by the system, and eliminating the scheduling variable s.

Let $\mathcal{K}_{_M} : \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ be an FKM representing an open system. The result of sealing off $\mathcal{K}_{_M}$ is an FKS $\mathcal{K}_{_C} : \langle V_{_C}, W_{_C}, \mathcal{O}_{_C}, \Theta_{_C}, \rho_{_C}, \mathcal{J}_{_C}, \mathcal{C}_{_C} \rangle$, where

$$
\begin{array}{lllllllll}
V_{_C} & = & V - \{s\} & \quad W_{_C} & = & V & \quad \mathcal{O}_{_C} & = & \mathcal{O} - \{s\} \\
\Theta_{_C} & = & \Theta & \quad \mathcal{J}_{_C} & = & \mathcal{J} & \quad \mathcal{C}_{_C} & = & \mathcal{C} \\
\rho_{_C} & = & \rho[s \mapsto \mathit{true}] & & & & & &
\end{array}
$$

Note that sealing off an FKM is the inverse operation to modularizing an FKS, and therefore taking the asynchronous composition of two closed systems is equivalent to the FKS obtained by sealing off the modular composition of their modular versions, as stated by the following equivalences:

$$
\begin{array}{rcl}
([\mathcal{K}]_{_M})_{_C} & \sim & \mathcal{K} \qquad (1) \\
([\mathcal{K}_1]_{_M} \parallel_{_M} [\mathcal{K}_2]_{_M})_{_C} & \sim & \mathcal{K}_1 \| \mathcal{K}_2. \qquad (2)
\end{array}
$$

4 Requirement Specification Language: Temporal Logic

As a requirement specification language for reactive systems we take *temporal logic* (TL) [MP91]. 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 $\bigcirc$ (*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 $(\sigma, j) \models q$
- $(\sigma, j) \models \bigcirc p \iff (\sigma, j+1) \models p$
- $(\sigma, j) \models p \mathcal{U} q \iff$ 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 $\diamondsuit\, p = \mathit{true} \mathcal{U} p$ (*eventually*) and $\square\, p = \neg \diamondsuit \neg p$ (*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 it holds on some model. A formula p is called *valid*, denoted by $\models p$, if it holds on all models.

Given an FKS $\mathcal{K}$ and a temporal formula p, we say that p is *$\mathcal{K}$-valid*, denoted by $\mathcal{K} \models p$, if p holds on all models which are computations of $\mathcal{K}$.

An algorithm for model checking whether a temporal formula p is valid over a finite-state FKS $\mathcal{K}$ is presented in [KPR98]. 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 [LPS81] and [CGH94], the approach calls for the construction of a *tester* for the negation of p. This is an FKS $\mathcal{K}_{\neg p}$ whose computations are all the sequences which satisfy the negated formula p. Then we form the *synchronous parallel composition* $\mathcal{K}_{comb} = \mathcal{K} ||| \mathcal{K}_{\neg p}$ and check for feasibility. If $\mathcal{K}_{comb}$ is found to be feasible, this implies that $\mathcal{K}$ has a computation which violates p and therefore p is not valid over $\mathcal{K}$. If $\mathcal{K}_{comb}$ is found to be infeasible, we can conclude that p is $\mathcal{K}$-valid.

5 Control Abstraction

Let $\sigma : s_0, s_1, \ldots$ be an infinite sequence of V-states, and let $U \subseteq V$ be a subset of V. We say that the infinite state sequence $\widetilde{\sigma} : \widetilde{s}_0, \widetilde{s}_1, \ldots$ is a *U-preserving variant* of σ if s_i and $\widetilde{s}_i$ agree on the interpretation of the variables in U, for every $i = 0, 1, \ldots$. We define $\sigma : s_0, s_1, \ldots$ to be an *observation* of the FKS $\mathcal{K} = \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ if σ is a $\mathcal{O}$-preserving variant of a computation of $\mathcal{K}$. Let $Obs(\mathcal{K})$ denote the set of all observations of system $\mathcal{K}$.

The FKS $\mathcal{K}_A = \langle V_A, W_A, \mathcal{O}_A, \Theta_A, \rho_A, \mathcal{J}_A, \mathcal{C}_A \rangle$ is defined to be *observation compatible* with the FKS $\mathcal{K} = \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ if $\mathcal{O}_A = \mathcal{O}$. The FKS $\mathcal{K}$ is an *abstraction* of the observation compatible $\mathcal{K}$, denoted by $\mathcal{K} \sqsubseteq \mathcal{K}_A$, if $Obs(\mathcal{K}) \subseteq Obs(\mathcal{K}_A)$. We refer to $\mathcal{K}$ and $\mathcal{K}_A$ as the *concrete* and *abstract* systems, respectively. The abstraction relation is obviously reflexive and transitive.

It can be shown that some of the FKS operations defined in Section 3 are monotonic with respect to the abstraction relation. In particular, if $\mathcal{K} \sqsubseteq \mathcal{K}_A$ then $(\mathcal{K} \|_M \mathcal{K}_2) \sqsubseteq (\mathcal{K}_A \|_M \mathcal{K}_2)$ and $\mathcal{K}_C \sqsubseteq (\mathcal{K}_A)_C$. That is, the operations of modular composition and closure (sealing off) of an FKM are monotonic with respect to abstraction. Furthermore, if p is a temporal formula whose free variables belong to $\mathcal{O} = \mathcal{O}_A$, then

$$\mathcal{K}_A \models p \qquad \text{implies} \qquad \mathcal{K} \models p,$$

to which we refer as the *anti-monotonicity* of the validity relation.

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 $\mathcal{K}$, we use abstraction in order to derive a simpler system $\mathcal{K}_A$ and then verify that p is $\mathcal{K}_A$-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 $\mathcal{K}$ is an infinite-state system, while its abstraction $\mathcal{K}_{_A}$ is finite-state and thus amenable to verification by model checking.

In this section, we concentrate on cases in which the system is a parallel composition $P(n)$: $P_1\|\cdots\|P_n$, where each P_i is a finite-state system. 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) that the property p is valid.

The method and one of the examples presented in this section are taken from [KM95]. The main differences between the two presentations are that, while [KM95] considers processes communicating by synchronous message passing we have reformulated the framework to communication by shared variables, and we find the abstraction we use somewhat simpler to comprehend, perhaps due to the switch in the 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:

Strategy 1 (Verification by Abstract Networks Invariants)

1. *Generate* FKM $\mathcal{M}_i$*, representing the modular behavior of process* P_i*, for* $i = 1, \ldots, k$.
2. *Derive a* network invariant $\mathcal{I}$*, which is an* FKM *intended to provide an abstraction for the modular composition* $\mathcal{M}_2 \parallel_{_M} \cdots, \parallel_{_M} \mathcal{M}_n$ *for any* n.
3. *Confirm that* $\mathcal{I}$ *is indeed a network invariant, by model checking that* $\mathcal{M}_2 \sqsubseteq \mathcal{I}$ *and that* $(\mathcal{I} \parallel_{_M} \mathcal{I}) \sqsubseteq \mathcal{I}$.
4. *Model check* $\mathcal{K} \models p$*, where* $\mathcal{K}$ *is the closed system* $(\mathcal{M}_1 \|_{_M} \mathcal{I})_{_C}$.

We argue that this strategy is sound. Namely, if $\mathcal{K} \models p$ then $P(n) \models p$ for every $n > 0$. Step 2 of the strategy establishes $\mathcal{M}_2 \parallel_{_M} \cdots, \parallel_{_M} \mathcal{M}_n \sqsubseteq \mathcal{I}$. By monotonicity of the modular composition, $\mathcal{M}_1 \parallel_{_M} \cdots, \parallel_{_M} \mathcal{M}_n \sqsubseteq (\mathcal{M}_1 \parallel_{_M} \mathcal{I})$. Using Equation 2 of Section 3 and the monotonicity of the closure operation, we can conclude

$$P(n) \;=\; (P_1\|\cdots\|P_n) \;=\; (\mathcal{M}_1 \parallel_{_M} \cdots, \parallel_{_M} \mathcal{M}_n)_{_C} \;\sqsubseteq\; (\mathcal{M}_1 \parallel_{_M} \mathcal{I})_{_C} \;=\; \mathcal{K}.$$

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

Step 2 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 for the choices we made.

5.1 Mutual Exclusion by Semaphores

As our first running example, we use a program that manages mutual exclusion by semaphores. 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

y: **natural where** $y = 0$

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

Fig. 1. Program MUX-SEM.

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.

In Fig. 2, we present the FKM corresponding to one of the (identical) processes of program MUX-SEM. In this diagrammatic representation, nodes correspond to sets of states. For example, the node labeled by $(N, 1)$ corresponds to the two states $\langle N, y : 1, s : 1\rangle$ and $\langle N, y : 1, s : 0\rangle$. To simplify the presentation, we used a single node to represent these two states which only differ in the interpretation of s. All edges connecting a node to itself have been eliminated. A solid edge in the diagram represents a step of the module itself, while a dotted edge represents a step of the environment. Thus, while only the process can decide to move from control state N to control state C, only the environment can change y from 1 to 0 (and vice versa) while control is still at N. The fairness requirements associated with this FKM are the justice requirement $\neg C$ ensuring that the system will not stay forever at control location C, and the compassion requirement $(T \wedge y = 1,\ C)$ guaranteeing that if the process is waiting at T and y equals 1 infinitely many times, then control will eventually proceed to C.

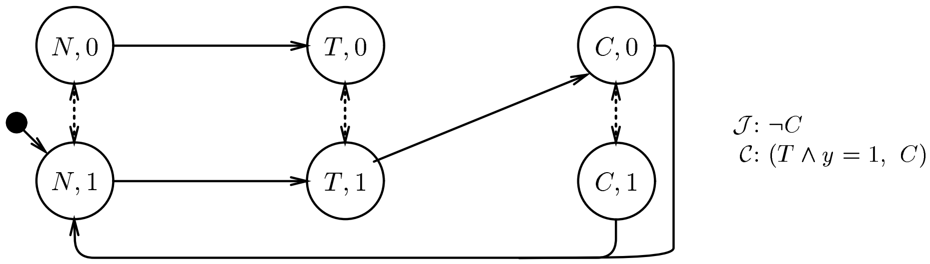


Fig. 2. The FKM $\mathcal{K}_1$ corresponding to process $P[1]$ of program MUX-SEM.

A first abstraction we can apply to $\mathcal{K}_1$ is to observe that as far as the sequences of values of the observable variables are concerned, there is no need to distinguish between the control locations N and T. This leads to the FKM $\mathcal{K}_2$ presented in Fig. 3.

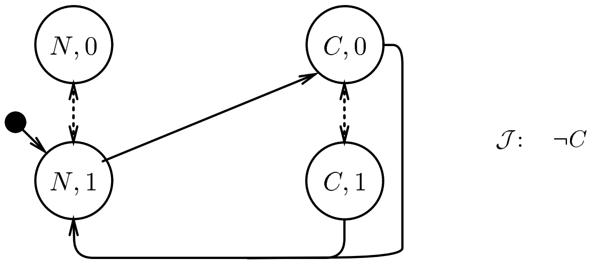


Fig. 3. The FKM $\mathcal{K}_2$ abstracting FKM $\mathcal{K}_1$.

Note that the compassion requirement has been eliminated. This implies that the system can tolerate a behavior in which it never sets y to 0 beyond a certain point. However, such a behavior was allowed also in FKM $\mathcal{K}_1$ by remaining forever at control location N beyond a certain point.

A useful heuristic that often leads to the generation of network invariants is forming the sequence of FKM's $\mathcal{I}_1 = \mathcal{K}_2$, $\mathcal{I}_2 = \mathcal{K}_2 \|_M \mathcal{K}_2$, $\mathcal{I}_3 = \mathcal{I}_2 \|_M \mathcal{K}_2, \ldots$, and comparing every successive $\mathcal{I}_i$'s, hoping that the sequence will converge. Trying this approach with the FKM $\mathcal{K}_2$ fails. Comparing $\mathcal{I}_2 : \mathcal{K}_2 \|_M \mathcal{K}_2$ with $\mathcal{I}_1 : \mathcal{K}_2$, we find that $\mathcal{I}_2$ can generate the observation (displaying the values of y and s)

$$\begin{array}{lllllllll} \langle y:1\,,\, s:0 \rangle & \longrightarrow & \langle y:0\,,\, s:1 \rangle & \dashrightarrow & \langle y:1\,,\, s:0 \rangle & \longrightarrow & \langle y:0\,,\, s:1 \rangle & \longrightarrow \\ \langle y:1\,,\, s:1 \rangle & \dashrightarrow & \langle y:0\,,\, s:0 \rangle & \longrightarrow & \langle y:1\,,\, s:1 \rangle & \cdots, & & \end{array}$$

which cannot be generated by $\mathcal{K}_1$. A step $\langle y : 1\,,\, s : - \rangle \longrightarrow \langle y : 0\,,\, s : 1 \rangle$ in this behavior corresponds to the module entering a critical section. A transition $\langle y : 0\,,\, s : - \rangle \dashrightarrow \langle y : 1\,,\, s : 0 \rangle$ corresponds to a step in which the environment changes y from 0 to 1. Thus, the first three steps in the displayed behavior describe a scenario in which the module entered twice a critical section without exiting even once.

A transition $\langle y : 0\,,\, s : - \rangle \longrightarrow \langle y : 1\,,\, s : 1 \rangle$ corresponds to the module changing y from 0 to 1, which describes an exit from the critical section. The transition $\langle y : 1\,,\, s : - \rangle \dashrightarrow \langle y : 0\,,\, s : 0 \rangle$ corresponds to a step in which the environment changes y from 1 to 0. Thus, the last three steps display a behavior in which $\mathcal{I}_2$ exits twice the critical section without re-entering after the first exit, provided the environment lowers the value of y after the first exit. In a similar way, we find that $\mathcal{I}_3$ can enter 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 abstraction $\mathcal{K}_3$, presented in Fig. 4.

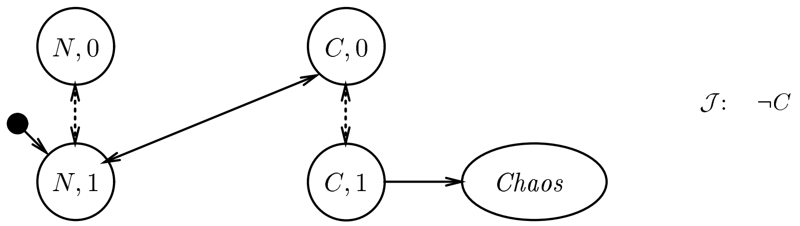


Fig. 4. The FKM $\mathcal{I} = \mathcal{K}_3$ with chaos.

The system $\mathcal{K}_3$ behaves as $\mathcal{K}_2$ as long as the environment behaves properly. However, once it detects that the environment 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 $\mathcal{K}_3$ is an abstraction of $\mathcal{K}_2$ because it differs from $\mathcal{K}_2$ in all the additional behaviors it is ready to generate once it reached the *chaos* state.

It is not difficult to verify that $\mathcal{I} = \mathcal{K}_3$ is a network invariant. We model checked that $\mathcal{K}_2 \sqsubseteq \mathcal{K}_3$ and that $(\mathcal{K}_3 \|_M \mathcal{K}_3) \sqsubseteq \mathcal{K}_3$.

It only remains to perform step 4 in the abstraction strategy presented in the beginning of the section. We form the closed-system FKS $\mathcal{K} = (\mathcal{K}_1 \|_M \mathcal{I})_M$ and use model checking to verify the liveness property $\mathcal{K} \models \Box(N_1 \rightarrow \Diamond C_1)$. This has been done and established that process $P[1]$ of program MUX-SEM has the property of accessibility for any number of processes.

5.2 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      n : integer where n ≥ 2
local   c : array [1..n] where c = 1

  n
 ||   P[j] :: [ ℓ0 : loop forever do
j=2              [ ℓ1 : NonCritical
                   ℓ2 : request c[j]
                   ℓ3 : request c[j ⊕n 1]
                   ℓ4 : Critical
                   ℓ5 : release c[j]
                   ℓ6 : release c[j ⊕n 1] ] ]

 ||   P[1] :: [ ℓ0 : loop forever do
                 [ ℓ1 : NonCritical
                   ℓ2 : request c[2]
                   ℓ3 : request c[1]
                   ℓ4 : Critical
                   ℓ5 : release c[2]
                   ℓ6 : release c[1] ] ]
```

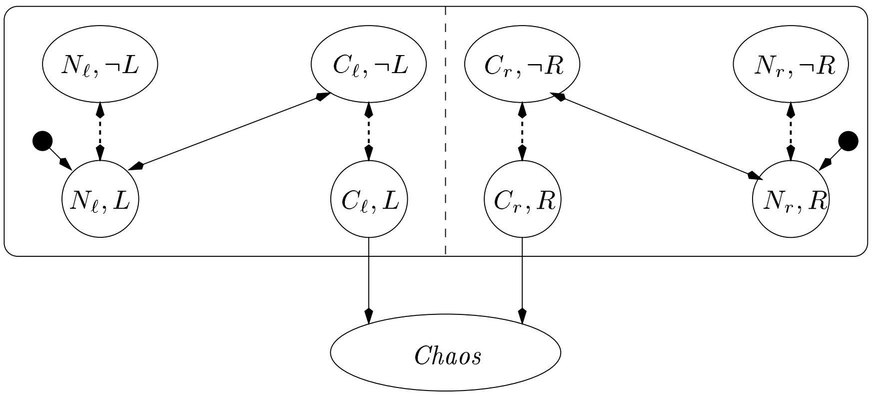
Fig. 5. Program DINE-CONTR: solution with one contrary philosopher.

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

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

for every $j = 1, \ldots, n$. This property ensure 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 FKM $\mathcal{I}_{contr}$ presented in Fig. 6.



$\mathcal{J}$: $\neg C_r$, $\qquad$ $\mathcal{C}$: $(R, \neg C_\ell)$

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

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 philosophers $P[i] \parallel_{M} P[i+1] \parallel_{M} \cdots \parallel_{M} P[j]$, 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 the left semaphore of process $P[i]$ the leftmost process in the string, while R stands for 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 N_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, N_ℓ) 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 proceeds 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 $\mathcal{I}_{contr}$ abstracts any of the processes $P[2], \ldots, P[n]$ and that $(\mathcal{I}_{contr} \parallel_{M} \mathcal{I}_{contr}) \sqsubseteq \mathcal{I}_{contr}$. 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 $P[1] \parallel_{M} \mathcal{I}_{contr} \parallel_{M} P[\mathit{ordinary}] \parallel_{M} \mathcal{I}_{contr}$, in which we use the network invariant $\mathcal{I}_{contr}$ as an abstraction for the sequence of philosophers separating $P[1]$ from $P[\mathit{ordinary}]$ and then again as an abstraction for the sequence of philosophers separating $P[\mathit{ordinary}]$ 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.

5.3 Reducing Abstraction to Verification

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

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

Consider two observation compatible FKS's: $\mathcal{K} = \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ and it's proposed abstraction $\mathcal{K}_A = \langle V_A, W_A, \mathcal{O}_A, \Theta_A, \rho_A, \mathcal{J}_A, \mathcal{C}_A \rangle$, and assume we wish to establish that $\mathcal{K} \sqsubseteq \mathcal{K}_A$. Without loss of generality, we can assume that $V \cap V_A = \emptyset$ but that there exists a 1-1 correspondence between the variables of $\mathcal{O}$ and those of $\mathcal{O}_A$.

We say that the FKS $\mathcal{K}_I = \langle V_I, W_I, \mathcal{O}_I, \Theta_I, \rho_I, \mathcal{J}_I, \mathcal{C}_I \rangle$ is an *interpolation* of $\mathcal{K}$ and $\mathcal{K}_A$ if it satisfies the following conditions:

$$
V_I = V \cup V_A \qquad W_I = W \cup W_A \qquad \mathcal{O}_I = \mathcal{O} \cup \mathcal{O}_A \qquad \mathcal{J}_I = \mathcal{J} \qquad \mathcal{C}_I = \mathcal{C}
$$

$$
\begin{array}{ccc}
\Theta \rightarrow \exists V_A : \Theta_I & \text{and} & \Theta_I \rightarrow \Theta_A \\
\rho \rightarrow \exists V'_A : \rho_I & \text{and} & \rho_I \rightarrow \rho_A
\end{array}
$$

A partial explanation of the term "interpolating" is that, from a logical viewpoint, $\mathcal{K}_I$ (at least Θ_I and ρ_I) is weaker than $\mathcal{K}$ and stronger than $\mathcal{K}_A$. The intention of the interpolating system $\mathcal{K}_I$ is that it tries to emulate the joint behavior of $\mathcal{K}$ and $\mathcal{K}_A$ in a way that would allow any previously admissible step of $\mathcal{K}$ (as implied by $\rho \rightarrow \exists V'_A : \rho_I$ and immediately match it by a $\mathcal{K}_A$ step, as implied by $\rho_I \rightarrow \rho_A$. Thus ρ should not exclude any possible $\mathcal{K}$ but is allowed to select among the possible $\mathcal{K}_A$ steps a single one that matches the corresponding $\mathcal{K}$-step.

Example 1. To illustrate the concepts we show how to construct an interpolating FKS for the FKM's $\mathcal{K}_1$ (Fig. 2) and $\mathcal{I}$ (Fig. 4) in preparation for proving the abstraction $\mathcal{K}_1 \sqsubseteq \mathcal{I}$.

The abstracted FKM corresponding to $\mathcal{K}_1$ is given by:

$$
\begin{array}{l}
V = \{\pi : \{N, T, C\}, y : \{0,1\}, s : \{0,1\}\} \qquad W = \{\pi\} \qquad \mathcal{O} = \{y, s\} \\
\Theta \;:\; \pi = N \;\wedge\; y = 1 \;\wedge\; s = 0 \\
\rho \;:\; \left(\begin{array}{ll}
 & s' = 0 \;\wedge\; \pi' = \pi \\
\vee & s' = 1 \;\wedge\; \left(\begin{array}{ll}
 & \pi' = \pi \;\wedge\; y' = y \\
\vee & \pi = N \;\wedge\; \pi' = T \;\wedge\; y' = y \\
\vee & \pi = T \;\wedge\; y = 1 \;\wedge\; \pi' = C \;\wedge\; y' = 0 \\
\vee & \pi = C \;\wedge\; \pi' = N \;\wedge\; y' = 1
\end{array}\right)
\end{array}\right) \\
\mathcal{J} \;:\; \pi \neq C \\
\mathcal{C} \;:\; (\pi = T \;\wedge\; y = 1,\; \pi = C)
\end{array}
$$

The abstracting FKM corresponding to $\mathcal{I}$ is given by:

$$
\begin{array}{l}
V_A = \{\Pi : \{N, C, \chi\}, Y : \{0,1\}, S : \{0,1\}\} \qquad W_A = \{\Pi\} \qquad \mathcal{O}_A = \{Y, S\} \\
\Theta_A \;:\; \Pi = N \;\wedge\; Y = 1 \;\wedge\; S = 0 \\
\rho_A \;:\; \left(\begin{array}{ll}
 & S' = 0 \;\wedge\; \Pi' = \Pi \\
\vee & S' = 1 \;\wedge\; \left(\begin{array}{ll}
 & \Pi' = \Pi \;\wedge\; Y' = Y \\
\vee & \Pi = N \;\wedge\; Y = 1 \;\wedge\; \Pi' = C \;\wedge\; Y' = 0 \\
\vee & \Pi = C \;\wedge\; Y = 0 \;\wedge\; \Pi' = N \;\wedge\; Y' = 1 \\
\vee & \Pi = C \;\wedge\; Y = 1 \;\wedge\; \Pi' = \chi \\
\vee & \Pi = \chi \;\wedge\; \Pi' = \chi
\end{array}\right)
\end{array}\right) \\
\mathcal{J}_A \;:\; \Pi \neq C \qquad \mathcal{C}_A \;:\; \emptyset
\end{array}
$$

In this presentation we denote *Chaos* by χ.

The interpolating FKM $\mathcal{K}_I$ is given by:

$$
\begin{array}{l}
V_I = \{\pi, y, s, \Pi, Y, S\} \qquad W_I = \{\pi, \Pi\} \qquad \mathcal{O}_I = \{y, s, Y, S\} \\
\Theta_I \;:\; \Theta \;\wedge\; \Theta_A \\
\rho_I \;:\; \rho \;\wedge\; \left(\begin{array}{ll}
 & S' = s' \\
\wedge & \left(\begin{array}{lll}
\textbf{if} & S' = 0 & \textbf{then}\; Y' = y' \;\wedge\; \Pi' = \Pi \\
\textbf{else-if} & \Pi = \chi & \textbf{then}\; \Pi' = \chi \;\wedge\; Y' = y' \\
\textbf{else-if} & \Pi = N \;\wedge\; Y = 1 \;\wedge\; y' = 0 & \textbf{then}\; \Pi' = C \;\wedge\; Y' = 0 \\
\textbf{else-if} & \Pi = C \;\wedge\; Y = 0 \;\wedge\; y' = 1 & \textbf{then}\; \Pi' = N \;\wedge\; Y' = 1 \\
\textbf{else-if} & \Pi = C \;\wedge\; Y = 1 & \textbf{then}\; \Pi' = \chi \;\wedge\; Y' = y' \\
\textbf{else} & \Pi' = \Pi \;\wedge\; Y' = Y &
\end{array}\right)
\end{array}\right) \\
\mathcal{J} \;:\; \pi \neq C \\
\mathcal{C} \;:\; (\pi = T \;\wedge\; y = 1,\; \pi = C)
\end{array}
$$

It is not difficult to check that $\mathcal{K}_I$ is an interpolating FKM for $\mathcal{K}_1$ and $\mathcal{I}$. ▟

The above example illustrates a standard technique for constructing an interpolating FKM for two given FKM's $\mathcal{K}$ and $\mathcal{K}_A$. The standard construction takes Θ_I to be $\Theta \wedge \Theta_A$. For ρ_I we take $\rho \wedge \rho^*_A$, where ρ^*_A is obtained by adding extra conjuncts to the disjuncts of ρ_A and assigning them different priorities. For example, ρ_I assigns the lowest priority to the idling disjunct $\Pi' = \Pi \wedge Y' = Y$ of ρ_A. That is, only if we fail to activate any of the preceding disjuncts, we choose the idling step.

The following claim relates computations of $\mathcal{K}$ to the computations of $\mathcal{K}_I$ and runs of $\mathcal{K}_I$ to runs of $\mathcal{K}_A$.

Claim 1. *Let σ be an infinite state sequence. If σ is a computation of $\mathcal{K}$, then there exists $\widetilde{\sigma}$, a V-preserving variant of σ which is a computation of both $\mathcal{K}$ and $\mathcal{K}_I$. If σ is a run of $\mathcal{K}_I$ it is also a run of $\mathcal{K}_A$.*

Combining the two parts of Claim 1, we conclude that for every σ a computation of $\mathcal{K}$ there exists $\widetilde{\sigma}$ a V-preserving variant of σ which is both a computation of $\mathcal{K}_I$ and a run of $\mathcal{K}_A$.

Next, we identify a temporal formula $\psi_{_A}$ which guarantees that $\widetilde{\sigma}$ is actually a computation of $\mathcal{K}_{_A}$ and one that agrees with σ on the values of all observable variables. This formula is given by

$$\psi_{_A}: \quad \Box(\mathcal{O} = \mathcal{O}_{_A}) \;\wedge\; \bigwedge_{J \in \mathcal{J}_{_A}} \Box \Diamond J \;\wedge\; \bigwedge_{(p,q) \in \mathcal{C}_{_A}} (\Box \Diamond p \rightarrow \Box \Diamond q).$$

The first conjunct of $\psi_{_A}$ requires that the variables $\mathcal{O}$ and $\mathcal{O}_{_A}$ agree on their values in all states. The middle conjunct requires that all justice requirements of $\mathcal{K}_{_A}$ are satisfied, while the last conjunct guarantees the fulfillment of the compassion requirements of $\mathcal{K}_{_A}$.

The following claim summarizes our method for reducing an abstraction proof into a verification task.

Claim 2. *If $\mathcal{K}_{_I}$ is an* FKS *interpolating between $\mathcal{K}$ and $\mathcal{K}_{_A}$ and $\mathcal{K}_{_I} \models \psi_{_A}$ then $\mathcal{K}_{_A}$ is an abstraction of $\mathcal{K}$.*

It is possible to view the construction of the interpolating $\mathcal{K}_{_I}$ as a task similar to the formulation of (the inverse of) the *refinement mapping* in the sense of [AL91]. In both cases, the construction selects the appropriate abstract transition to match any step of the concrete system.

6 Data Abstraction

In this section, we present a general methodology for *data abstraction*, strongly inspired by the notion of *abstract interpretation* [CC77]. Consider an FKS $\mathcal{K} = \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, and let Σ denote the set of states of $\mathcal{K}$, the *concrete states*. Let $\alpha : \Sigma \mapsto \Sigma_{_A}$ be a mapping of concrete states into *abstract states*. The strategy of *verification by data abstraction* can be summarized as follows:

Strategy 2 (Verification by Data Abstraction)

- *Define an abstraction mapping α to abstract the concrete* FKS *$\mathcal{K}$ into an* abstract FKS $\mathcal{K}^{\alpha}_{_A}$.
- *Abstract the temporal property ψ into an abstract property ψ^{α}.*
- *Verify $\mathcal{K}^{\alpha}_{_A} \models \psi^{\alpha}$.*
- *Infer $\mathcal{K} \models \psi$.*

The main question is how to define the abstractions $\mathcal{K}^{\alpha}_{_A}$ and ψ^{α} such that $\mathcal{K}^{\alpha}_{_A} \models \psi^{\alpha}$ implies $\mathcal{K} \models \psi$.

Example 2. Consider program ANY-Y of Fig. 7, for which we wish to establish the invariance property $\psi\colon \Box(y \geq 0)$.

Program ANY-Y is an infinite-state system since the integer variable y can assume arbitrarily high values. To reduce the complexity of this system, we may consider an abstract variable Y, ranging over the finite (abstract) domain

$$
\begin{array}{c}
x,\ y\text{: }\mathbf{integer\ where}\ x = y = 0 \\[1em]
P_1 :: \left[\begin{array}{l} \ell_0 : \mathbf{while}\ x = 0\ \mathbf{do} \\ \quad \left[\ell_1 :\ y := y + 1\right] \\ \ell_2 : \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0 : x := 1 \\ m_1 : \end{array}\right]
\end{array}
$$

Fig. 7. Program ANY-Y: A simple concurrent program.

$\{\mathit{neg}, \mathit{zero}, \mathit{pos}\}$. The abstraction function α maps the domain of y into the domain of Y as follows:

$$\alpha(y)\text{:}\quad \mathbf{if}\ y < 0\ \mathbf{then}\ \boldsymbol{neg}\ \mathbf{else\text{-}if}\ y = 0\ \mathbf{then}\ \boldsymbol{zero}\ \mathbf{else}\ \boldsymbol{pos}$$

With this mapping, we can obtain the abstract version of ANY-Y, called ANY-Y$^\alpha$, and presented in Fig. 8.

A corresponding α-abstraction of the property to be verified is given by ψ^α: $\Box(Y \in \{\mathit{zero}, \mathit{pos}\})$. Since program ANY-Y$^\alpha$ is a finite-state program, we can use model checking in order to verify ANY-Y$^\alpha \models \psi^\alpha$. Following the verification-by-data-abstraction strategy, we can infer ANY-Y $\models \Box(y \geq 0)$. ▟

$$
\begin{array}{c}
\begin{array}{ll} x\text{: }\mathbf{integer} & \mathbf{where}\ x = 0 \\ Y\text{: }\{\mathit{neg}, \mathit{zero}, \mathit{pos}\} & \mathbf{where}\ Y = \mathit{zero} \end{array} \\[1em]
P_1 :: \left[\begin{array}{l} \ell_0 : \mathbf{while}\ x = 0\ \mathbf{do} \\ \quad \left[\ell_1 :\ Y := \left(\begin{array}{ll} \mathbf{if} & Y = \mathit{neg} \\ \mathbf{then} & \{\mathit{neg}, \mathit{zero}\} \\ \mathbf{else} & \mathit{pos} \end{array}\right)\right] \\ \ell_2 : \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} m_0 : x := 1 \\ m_1 : \end{array}\right]
\end{array}
$$

Fig. 8. Program ANY-Y$^\alpha$: Abstracted version of ANY-Y.

6.1 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. Thus, for Example 2, the expressions $\mathcal{E}^\alpha_{\pi_1}$, $\mathcal{E}^\alpha_{\pi_2}$, $\mathcal{E}^\alpha_{x}$, and $\mathcal{E}^\alpha_{Y}$ are given by π_1, π_2, x, and **if** $y < 0$ **then** ***neg*** **else-if** $y = 0$ **then** ***zero*** **else** ***pos***, respectively.

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

$$M_\forall^\alpha(p): \quad \forall V \left(V_A = \mathcal{E}^\alpha(V) \quad \rightarrow \quad p(V) \right).$$

The assertion $M_\forall^\alpha(p)$ holds for an abstract state $S \in \Sigma_A$ iff 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 $S = \alpha(s)$. This can also be expressed by the inclusion $\alpha^{-1}(\|M_\forall^\alpha(p)\|) \subseteq \|p\|$, where $\|p\|$ and $\|M_\forall^\alpha(p)\|$ represent the sets of states which satisfy the assertions, respectively. If $M_\forall^\alpha(p)$ is valid, then $\|M_\forall^\alpha(p)\| = \Sigma_A$ implying $\alpha^{-1}(\|M_\forall^\alpha(p)\|) = \Sigma$ 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 $M_\exists^\alpha$, dual to $M_\forall^\alpha$, as follows

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

The assertion $M_\exists^\alpha(p)$ 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 $S = \alpha(s)$. This can also be expressed by the inclusion $\|p\| \subseteq \alpha^{-1}(\|M_\exists^\alpha(p)\|)$.

Having defined the effects of $M_\forall^\alpha$ and $M_\exists^\alpha$ when applied to assertions, we proceed to define the result of their application to an arbitrary temporal formula.

$$\begin{array}{ll}
M_\forall^\alpha(\neg(p)) = \neg M_\exists^\alpha(p) & M_\exists^\alpha(\neg(p)) = \neg M_\forall^\alpha(p) \\
M_\forall^\alpha(p \vee q) = M_\forall^\alpha(p) \vee M_\forall^\alpha(q) & M_\exists^\alpha(p \vee q) = M_\exists^\alpha(p) \vee M_\exists^\alpha(q) \\
M_\forall^\alpha(\bigcirc\, p) = \bigcirc\, M_\forall^\alpha(p) & M_\exists^\alpha(\bigcirc\, p) = \bigcirc\, M_\exists^\alpha(p) \\
M_\forall^\alpha(p \mathcal{U} q) = (M_\forall^\alpha(p)) \mathcal{U} (M_\forall^\alpha(q)) & M_\exists^\alpha(p \mathcal{U} q) = (M_\exists^\alpha(p)) \mathcal{U} (M_\exists^\alpha(q))
\end{array}$$

We respectively refer to $M_\forall^\alpha(p)$ and $M_\exists^\alpha(p)$ as the *universal* and *existential abstraction* of the formula p. The following claim identifies the abstractions that can be used within the verification-by-data-abstraction strategy, i.e., the *safe abstractions*:

Claim 3. *Let ψ be a temporal formula and α an abstraction mapping. If the abstraction $\psi^\alpha = M_\forall^\alpha(\psi)$ is valid, then so is ψ.*

The claim can be proven by structural induction on the sub-formulas of ψ.

Example 3. Consider, for example, the temporal formula $\psi : (\square(y \geq 2) \rightarrow \square(y \geq -1))$. Applying the abstraction α presented in Example 2 yields the formula

$$\begin{array}{rcl}
\psi^\alpha = M_\forall^\alpha(\psi) & = & (\square(M_\exists^\alpha(y \geq 2)) \rightarrow \square(M_\forall^\alpha(y \geq -1))) = \\
& & (\square(Y = pos) \rightarrow \square(Y \in \{zero, pos\})).
\end{array}$$

Since ψ^α is valid, we can safely conclude that so is ψ. ■

6.2 Safe Abstraction of FKS's

Next, we consider the abstraction of an FKS $\mathcal{K}$ into $\mathcal{K}^\alpha$ such that $\mathcal{K}^\alpha \models \psi^\alpha$ implies $\mathcal{K} \models \psi$. We reduce the problem of the safe joint abstraction of an FKS 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 FKS $\mathcal{K} = \langle V, W, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$, there exists a temporal formula $Sem(\mathcal{K})$, called the *temporal semantics* of $\mathcal{K}$ [Pnu81], such that, for every infinite state sequence σ, $\sigma \models Sem(\mathcal{K})$ *iff* $\sigma \in Comp(\mathcal{K})$. The temporal semantics of an FKS $\mathcal{K}$ is given by

$$Sem(\mathcal{K}): \quad \Theta \;\wedge\; \Box\,\rho(V, \bigcirc V) \;\wedge\; \bigwedge_{J \in \mathcal{J}} \Box \Diamond J \;\wedge\; \bigwedge_{(p,q)\in\mathcal{C}} (\Box \Diamond p \rightarrow \Box \Diamond q),$$

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

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

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

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

$$M^\alpha_\forall(Ver(\mathcal{K}, \psi)) =$$

$$\left(M^\alpha_\exists(\Theta) \;\wedge\; \Box\, M^\alpha_\exists(\rho(V, \bigcirc V)) \;\wedge\; \left[\begin{array}{l} \bigwedge\limits_{J\in\mathcal{J}} \Box \Diamond (M^\alpha_\exists(J) \quad \wedge \\ \bigwedge\limits_{(p,q)\in\mathcal{C}} (\Box \Diamond M^\alpha_\forall(p) \rightarrow \Box \Diamond (M^\alpha_\exists(q)) \end{array}\right] \right) \rightarrow M^\alpha_\forall(\psi).$$

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

$$\begin{array}{ll} \Theta^\alpha = M^\alpha_\exists(\Theta) & \rho^\alpha = M^\alpha_\exists(\rho) \\ \mathcal{J}^\alpha = \{M^\alpha_\exists(J) \mid J \in \mathcal{J}\} & \mathcal{C}^\alpha = \{(M^\alpha_\forall(p), M^\alpha_\exists(q)) \mid (p,q) \in \mathcal{C}\} \end{array}$$

Example 4. Let us show how the definition of $\mathcal{K}^\alpha$ leads to the construction of program ANY-Y$^\alpha$, the abstracted version of program ANY-Y, as presented in Example 2. Since the mapping α only affected the variable y, abstracting it into the three-valued variable Y, we will only consider the parts of Θ and ρ which refer to y. Note that none of the fairness requirements for program ANY-Y refers to y.

The initial condition Θ refers to y only through the conjunct $y = 0$. The corresponding abstraction is given by

$$\begin{array}{rl} M^\alpha_\exists(y = 0) & = \\ & \exists y.(Y = \textbf{if } y < 0 \textbf{ then } \boldsymbol{neg} \textbf{ else-if } y = 0 \textbf{ then } \boldsymbol{zero} \textbf{ else } \boldsymbol{pos}) \wedge (y = 0) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \sim \quad (Y = zero) \end{array}$$

The transition relation ρ refers to y only through the part of a disjunct given by $y' = y + 1$. The corresponding abstraction is

$$
\begin{aligned}
M^{\alpha}_{\exists}(y' = y + 1) \quad &= \\
\exists y, y' &\left(\begin{array}{l} \quad Y \ = \textbf{if } y \ < 0 \textbf{ then } \boldsymbol{neg} \textbf{ else-if } y \ = 0 \textbf{ then } \boldsymbol{zero} \textbf{ else } \boldsymbol{pos} \\ \wedge\, Y' = \textbf{if } y' < 0 \textbf{ then } \boldsymbol{neg} \textbf{ else-if } y' = 0 \textbf{ then } \boldsymbol{zero} \textbf{ else } \boldsymbol{pos} \\ \wedge\, y' = y + 1 \end{array}\right) \\
&\sim (Y = \mathit{neg} \ \wedge \ Y' \in \{\mathit{neg}, \mathit{zero}\}) \ \vee \ (Y \in \{\mathit{zero}, \mathit{pos}\} \ \wedge \ Y' = \mathit{pos}) \\
&\sim Y' = (\textbf{if } Y = \boldsymbol{neg} \textbf{ then } \{\boldsymbol{neg}, \boldsymbol{zero}\} \textbf{ else } \boldsymbol{pos}.)
\end{aligned}
$$

Reconstructing a program from the abstracted components Θ^{α} and ρ^{α}, we obtain program ANY-Y$^{\alpha}$ presented in Fig. 8. ■

The following claim guarantees the safety of the abstractions jointly applied to the system and the property we wish to verify.

Claim 4. *If the abstracted formula $M^{\alpha}_{\forall}(\psi)$ is valid over the abstracted* FKS $\mathcal{K}^{\alpha}$*, then ψ is valid over $\mathcal{K}$. That is,*

$$\mathcal{K}^{\alpha} \models M^{\alpha}_{\forall}(\psi) \qquad \textit{implies} \qquad \mathcal{K} \models \psi.$$

6.3 Determination of the Abstract Domain

The theory presented above assumed that the abstract domain, represented by the abstract variables V_A and their types, and the abstraction mapping α are already given. In this subsection, we consider some recommendations for the choice of an appropriate mapping, given an FKS $\mathcal{K}$ and a temporal property ψ.

Assuming that the FKS $\mathcal{K}$ is derived from a program P, let $p_1, p_2, \ldots, p_k$ be the set of all atomic formulas referring to the data (non-control) variables appearing within conditions in the program P and within the temporal formula ψ.

As a running example, we use program BAKERY-2, presented in Fig. 9.

$$\textbf{local } y_1, y_2 : \textbf{natural where } y_1 = y_2 = 0$$

$$
\begin{array}{ccc}
\left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} \ell_1 : \textbf{NonCritical} \\ \ell_2 : y_1 := y_2 + 1 \\ \ell_3 : \textbf{await } y_2 = 0 \ \vee \ y_1 < y_2 \\ \ell_4 : \textbf{Critical} \\ \ell_5 : y_1 := 0 \end{array}\right] \end{array}\right]
& \| &
\left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \left[\begin{array}{l} m_1 : \textbf{NonCritical} \\ m_2 : y_2 := y_1 + 1 \\ m_3 : \textbf{await } y_1 = 0 \ \vee \ y_2 \leq y_1 \\ m_4 : \textbf{Critical} \\ m_5 : y_2 := 0 \end{array}\right] \end{array}\right] \\
- P_1 - & & - P_2 -
\end{array}
$$

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

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 for program BAKERY-2 are given by

$$\begin{aligned}\psi_{exc} &: \Box\neg(at\_\ell_4 \ \wedge \ at\_m_4)\\ \psi_{acc} &: \Box(at\_\ell_2 \quad \rightarrow \quad \Diamond\, at\_\ell_4),\end{aligned}$$

The property ψ_{exc} requires *mutual exclusion*, guaranteeing that the two processes never co-reside in their respective critical section at the same time. The 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.

For program BAKERY-2, the atomic data formulas are $y_1 = 0$, $y_2 = 0$, and $y_1 < y_2$. Note that the formula $y_2 \leq y_1$ is equivalent to the negation of $y_1 < y_2$ and needs not be included as an independent atomic formula.

Proceeding with the general case, the abstract system variables consist of the concrete control variables, which are left unchanged, and a set of abstract boolean variables $B_{p_1}, B_{p_2}, \ldots, B_{p_k}$, one for each atomic data formula. The abstraction mapping α is defined by

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

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.

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), which is obtained by the abstraction as described in the preceding subsections.

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

$$\left[\begin{array}{l}\ell_0 : \textbf{loop forever do}\\ \left[\begin{array}{l}\ell_1 : \textbf{NonCritical}\\ \ell_2 : (B_{y_1=0}, B_{y_1<y_2}) := (0,0)\\ \ell_3 : \textbf{await } B_{y_2=0} \ \vee \ B_{y_1<y_2}\\ \ell_4 : \textbf{Critical}\\ \ell_5 : (B_{y_1=0}, B_{y_1<y_2}) := (1, \neg B_{y_2=0})\end{array}\right]\end{array}\right] \;\Big|\Big|\; \left[\begin{array}{l}m_0 : \textbf{loop forever do}\\ \left[\begin{array}{l}m_1 : \textbf{NonCritical}\\ m_2 : (B_{y_2=0}, B_{y_1<y_2}) := (0,1)\\ m_3 : \textbf{await } B_{y_1=0} \ \vee \ \neg B_{y_1<y_2}\\ m_4 : \textbf{Critical}\\ m_5 : (B_{y_2=0}, B_{y_1<y_2}) := (1,0)\end{array}\right]\end{array}\right]$$

$- P_1 -$ $\qquad\qquad$ $- P_2 -$

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

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 4, we can infer that the original program BAKERY-2 also satisfies these two temporal properties.

6.4 Proving Power of the Verification-by-Data-Abstraction Strategy

The strategy of verification-by-data-abstraction (VBDA) is most effective when used to abstract an infinite-state system into a finite-state system. A potential user of this methodology may wonder whether this is always possible, or are there cases in which this strategy cannot be applied.

Here we establish that, for the case that the requirement ψ is a *safety property* [Lam77], then the VBDA strategy is as powerful as the deductive rule INV [MP95] and is, in fact complete.

To simplify the presentation, we consider the case that the requirement ψ is an *invariance property* of the form $\square\, p$, where p is an assertion. To verify that an FKS $\mathcal{K}$ satisfies the invariance property $\square\, p$, we may use rule INV, which is shown in [MP95] to be complete.

Rule INV For assertions p and φ,

$$\begin{array}{l} \text{I1. } \varphi \rightarrow p \\ \text{I2. } \Theta \rightarrow \varphi \\ \text{I3. } \rho \wedge \varphi \;\rightarrow\; \varphi' \\ \hline \quad \mathcal{K} \models \square\, p \end{array}$$

The following claim establishes that the VBDA strategy is as powerful as rule INV, and is, therefore complete.

Claim 5. *Let $\mathcal{K}$ be a feasible* FKS. *If $\mathcal{K} \models \square\, p$ is provable by rule* INV, *it can also be established by the* VBDA *strategy.*

Proof. Assume that $\mathcal{K}$ is a feasible FKS and $\mathcal{K} \models \square\, p$ is provable by rule INV. As the set of abstract variables we take the single boolean variable B_φ, and adopt the abstraction function $\alpha : B_\varphi = \varphi$. It is not difficult to see that, in the abstracted system, $\Theta^\alpha : B_\varphi$, while $\rho^\alpha : B_\varphi \rightarrow B'_\varphi$. A careful analysis of the abstracted fairness requirements yields that, due to the assumption of feasibility, all justice requirements are existentially abstracted into an assertion of the form B_φ or *true*. Similarly, every compassion requirement is abstracted into an assertion pair of one of the forms

$$(\mathit{false}, -),\ (\neg B_\varphi, -),\ (-, B_\varphi),\ (-, \mathit{true}).$$

It follows that the fairness requirements impose no restriction on computations of the abstracted system $\mathcal{K}^\alpha$, all of whose states satisfy $B_\varphi = \mathit{true}$.

The abstracted property is given by

$$(\square\, p)^\alpha \;=\; M^\alpha_\forall(\square\, p) \;=\; \square\, B_\varphi.$$

It follows (and can be trivially model checked) that $\mathcal{K}^\alpha \models (\square\, p)^\alpha$, from which, by the VBDA strategy, we can infer that $\mathcal{K} \models \square\, p$. ∎

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Properties preserving simulations. In G. Bochman and D. Probst, editors, *4th Computer Aided Verification Workshop*, volume 663 of *Lect. Notes in Comp. Sci.*, pages 251–263. Springer-Verlag, 1992.

[BBM95] N. Bjørner, I.A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In 1^{st} *Intl. Conf. on Principles and Practice of Constraint Programming*, volume 976 of *Lect. Notes in Comp. Sci.*, pages 589–623. Springer-Verlag, 1995.

[BCG86] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many finite state processes. In *Proc. 5th ACM Symp. Princ. of Dist. Comp.*, pages 240–248, 1986.

[CC77] P. Cousot and R. Cousot. 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.

[CGH94] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In D. L. Dill, editor, *Proc. 6th Conference on Computer Aided Verification*, volume 818 of *Lect. Notes in Comp. Sci.*, pages 415–427. Springer-Verlag, 1994.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *6th International Conference on Concurrency Theory (CONCUR'95)*, pages 395–407, Philadelphia, PA, August 1995.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Trans. Prog. Lang. Sys.*, 16(5):1512–1542, 1994.

[CGL96] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking. In *Model Checking, Abstraction and Composition*, volume 152 of *Nato ASI Series F*, pages 477–498. Springer-Verlag, 1996.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM Symp. Princ. of Prog. Lang.*, pages 84–96, 1978.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Prog. Lang. Sys.*, 19(2), 1997.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22th ACM Conf. on Principles of Programming Languages, POPL'95*, San Francisco, 1995.

[EN96] E.A. Emerson and K.S. Namjoshi. Automatic verification of parameterized synchronous systems. In R. Alur and T. Henzinger, editors, *Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1996.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In C. Courcoubetis, editor, *Proc. 5^{th} Intl. Conference on Computer Aided Verification (CAV'94)*, volume 697 of *Lect. Notes in Comp. Sci.*, pages 71–84. Springer-Verlag, 1993.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7):523–543, 1992.

[ID96] C.N. Ip and D. Dill. Verifying systems with replicated components in Murφ. In R. Alur and T. Henzinger, editors, *Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1996.

[KM95] R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.

[KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, Lect. Notes in Comp. Sci. Springer-Verlag, 1998.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Engin.*, 3:125–143, 1977.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*, Paris, 1997.

[LPS81] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proc. 8th Int. Colloq. Aut. Lang. Prog.*, volume 115 of *Lect. Notes in Comp. Sci.*, pages 264–277. Springer-Verlag, 1981.

[MAB$^+$94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Comp. Sci., Stanford University, Stanford, California, 1994.

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

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:1–20, 81.

[SG89] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lect. Notes in Comp. Sci.*, pages 151–165. Springer-Verlag, 1989.

[SG92] A.P. Sistla and S.M. German. Reasoning about systems with many processes. *J. ACM*, 39:675–735, 1992.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lect. Notes in Comp. Sci.*, pages 68–80. Springer-Verlag, 1989.