On the Existence of Network Invariants for Verifying Parameterized Systems*

Parosh Aziz Abdulla	Bengt Jonsson
Dept. Computer Systems	Dept. Computer Systems
Uppsala University	Uppsala University
P.O.Box 325	P.O.Box 325,
751 05 Uppsala, Sweden	751 05 Uppsala, Sweden
`parosh@docs.uu.se`	`bengt@docs.uu.se`

Abstract

Over the last decade, finite-state verification methods have been developed to an impressive tool for analysis of complex programs, such as protocols and hardware circuits. Partial-order reduction and BDD-based symbolic model checking have been instrumental in this development. Currently, much effort is devoted to advancing further the power of automated verification to cover also infinite-state systems. A prominent class of infinite-state systems are so-called *parameterized systems*, i.e., systems with many similar processes, in which the number of processes is unbounded and their interconnection pattern may vary within the range of some constraints. In this paper, we partially review the use of induction over the system structure for the verification of parameterized systems. Wolper and Lovinfosse have introduced the term *network invariant* for the induction hypothesis in such a proof by induction. They also observe that well-behaved (e.g., finite-state) network invariants do not always exist, even if the system itself satisfies the property to be verified. The main novel contribution of this paper is to present some sufficient conditions, under

*Supported in part by the Swedish Board for Industrial and Technical Development (NUTEK) and by the Swedish Research Council for Engineering Sciences (TFR).

which the existence of a finite-state network invariant is guaranteed. We also relate the construction of network invariants to the search for standard inductive invariants (as defined e.g., by Floyd and by Manna and Pnueli). Two small examples of network invariants and standard invariants for parameterized systems are included.

1 Introduction

One of the advantages of producing formal models of algorithms and systems is the possibility to analyze and verify them in a rigorous way, in the best case totally automatically by computer. For nondeterministic and parallel programs, relevant verifications include absence of deadlocks, or proving that all executions of a program satisfy a desirable property expressed in temporal logic. Over the last decade, impressive tools have been developed for verification of *finite-state* systems. Partial-order reduction and BDD-based symbolic model checking have been instrumental in this development.

Currently, much effort is devoted to advancing further the power of automated verification to cover also infinite-state systems. Potentially practical tools for automated verification of infinite-state systems have been developed for significant special cases, such as timed automata [ACD90, BGK+96], hybrid automata [Hen95], data-independent systems [JP93, Wol86], Petri nets [JM95], pushdown processes [BS95, Sti96], and systems with unbounded communication channels [Fin94, AJ93, ABJ98].

One particularly interesting class of infinite-state systems are the so-called *parameterized systems*. By a parameterized system, we mean a family of similar systems that depend in a regular way on a parameter. Typically, parameterized systems are built from a (small) finite set of processes, which are replicated and interconnected into networks. Examples of parameterized systems abound: a distributed algorithm can be modeled as a system of many identical processes whose interconnection topology may be arbitrary, a bus protocol consists of a linear array of identical processes where the parameter is the number of processes, etc. Also unbounded data structures, such as queues and trees, could be regarded as parameterized systems by letting each node (cell) be viewed as a process which is connected to its neighbour nodes (cells). A queue would thus be viewed as a linear array of processes of arbitrary length.

In order to make the discussion more concrete, let us consider families of systems of the form

$$P_0 \parallel P \parallel P \parallel \cdots \parallel P$$

composed of one instance of a "global" component P_0 which is common to all systems in the family, and an arbitrary number of instances of some process P. The processes are composed by some associative composition operator $\parallel$, which for the moment will be left unspecified. The verification problem for parameterized systems consists in verifying that

$$P_0 \parallel P \parallel P \parallel \cdots \parallel P \models \phi$$

for some correctness formula ϕ, for any number of copies of P. We assume that the formulation of ϕ is independent of the system size. An example of a correctness property could be "there are never two processes simultaneously in the critical section".

The most straight-forward approach to verifying a parameterized system is to verify the system for a suitable chosen number (say 5) of processes. Finite-state state-space exploration methods can be used to analyze the system efficiently. However, there is no *a priori* guarantee that the system will function correctly with 6 processes. It is therefore of interest to consider methods that verify the correctness of the system with an arbitrary number of components. It may even (as argued, e.g., by Wolper and Boigelot [WB98]) turn out that the parameterized version of the system is easier to verify, since in the best case it may concern the essential reason for correctness, and avoid the particularities of the case with 5 processes.

If we look at the problem of actually verifying that any system in the family satisfied ϕ, we notice that the problem has an unbounded structure along two dimensions. One dimension is "time", since we must check a property of potentially unbounded execution sequences. Another dimension is "space", since the property should hold for an arbitrary system size. To verify that the system is correct, in spite of any of these unboundednesses, we may use induction. In the following two paragraphs, we describe the use of induction, first in the time dimension and then in the space dimension.

The standard way to handle the time dimension in verification is to find an (inductive) invariant of the system, which is preserved by all computation steps of the system, and which implies ϕ (e.g., [MP95]). For instance, if ϕ

is the above correctness property "there are never two processes simultaneously in the critical section", then an inductive invariant of the system could be obtained by conjoining "at most one process has the token" (assuming that possessing the token is necessary for entering the critical section). This method of verification can be applied if we can express sets of system states of arbitrary size. Finding an invariant may involve some ingenuity. However, once it is found, the rest of the verification is relatively simple and possible to automate.

A method for handling the space dimension in parameterized system verification is to find a uniform abstraction of all system sizes. Such an abstraction is a single system N whose behavior "contains" (in a sense to be made more precise later in the paper) the behavior of any system in the family. In particular, if $N \models \phi$, then $P_0 \parallel P \parallel \cdots \parallel P \models \phi$ for any system size. One sufficient criterion for checking that N is indeed an abstraction is to check that it is inductive. Letting $\leq$ denote the relation "contained in", this means that it should be checked that $P_0 \leq N$ and that $N \parallel P \leq N$ besides checking that $N \models \phi$. The term "network invariant" has been introduced by Wolper and Lovinfosse [WL89] (similar independent work was also reported by Kurshan and McMillan [KM89]) for an abstraction which is inductive in this sense. Again, finding a network invariant N may involve some ingenuity. However, the point is that once N is found, the correctness checks should be relatively simple and possible to automate.

For verifying a parameterized system, there are thus two basically different ways to employ induction: either using standard (computational) induction, or using induction over the structure of the parameterized system. To avoid confusion, we will use the term *computational invariant* to mean an inductive invariant in the "time-dimension". Computational invariants expressed in some formalism for represented sets of states of unbounded system sizes have been used e.g., in [JK95, KMM+97, EFM99, AČJYK96, AJ98, ABJN99]. Network invariants have been employed in [KM89, WL89, EN95, LHR97]. Abstraction has also been used in [LS97, CGJ95].

In this paper, we will consider the question "when does there exist a computational or network invariant of a certain simple form?" If we consider the case that each component is finite-state, it is natural to try to look for "finite" computational and network invariants. In the case of computational invariants, we will take this to mean an invariant which considers an arbitrary "finite" part of the system structure. In the case of network invariants, we will take this to mean an invariant which can be expressed by a finite-state transition system. As observed by Wolper and Lovinfosse [WL89], it follows from a basic undecidability result by Apt and Kozen [AK86], that such invariants do not exist in general. In this paper, we will give some sufficient criteria for the existence of these invariants. We hope that this gives a better understanding in particular of the network invariant method. Wolper and Lovinfosse stated some undecidability results concerning their existence, but gave no positive results, other than a few examples where the method worked. Mechanized search for network invariants have been considered by Lesens et al [LHR97], who employ maximal fixpoint iteration, enhanced by approximation techniques, in the verification of some examples.

The paper is organized as follows. In the next section, we introduce the basic definitions of parameterized systems. The use of invariants in verification is described in Section 3. Section 4 contains criteria for the existence of computational and network invariants. In Section 5, we illustrate the verification of a parameterized system in which the components are non-finite state (timed automata).

2 Basic Definitions

Parameterized Systems By a parameterized system, we mean a family of systems, each of which is a composition of form

$$M_n \equiv P_0 \parallel P[1] \parallel P[2] \parallel \cdots \parallel P[n]$$

where P_0 is an optional "global" component, common to all instances in the family, and where $P[1], \ldots, P[n]$ are identical processes. Each component $P[i]$ and P_0 is modeled as a labeled transition system. For the moment, we do not require that this transition system be finite-state. The labels on transitions are taken from a set of visible actions, extended by an invisible action (denoted τ). We assume that any component can always perform an idling component transition, labeled by τ, which does not change its state. We use M to denote the infinite-state system which is the "union" of the family $\{M_n\}_{n=0}^{\infty}$ of system instances. One could think of M as a system, which initially decides on a value of the parameter n, and then becomes M_n.

Each transition of the system may involve one or more components. The possible synchronizations between components are constrained by a *synchronization predicate* Φ. In the remainder of this paper, we will consider a totally unordered system topology, in which any component can communicate with any other, regardless of their positions in the syntactic description of M_n. The synchronization predicate Φ is then a set of multisets of actions which describes the possible combinations of labels on component transitions that make up a transition of the parameterized system. For instance, if the set $\{\text{send}, \text{receive}\}$ is in Φ, but neither of send nor receive is in any other set of the synchronization predicate, then each occurrence of a send must be synchronized with a matching receive. In order to be able to model broadcast and global synchronizations, we allow one of the actions in a synchronization to have the special superscript ω, meaning "all other processes". For instance the set $\{\text{send}, \text{receive}^{\omega}\}$ denotes a broadcast by one process to all other processes.

We could also have considered parameterized systems with a linearly ordered topology, in which components are arranged in an order, as shown in the syntactic description of M_n. The synchronization predicate will then be a set of *strings* of labels.

Correctness Properties We will only consider linear-time safety properties of parameterized systems, which can be stated as a requirement on the executions of the controller, and a *bounded* arbitrary set of distinct components. Such properties can be transformed into invariants of the form

$$\forall i_1, \ldots, i_k . \mathit{alldiff}(i_1, \ldots, i_k) \implies \phi(i_1, \ldots, i_k)$$

where ϕ is a predicate on the local states of the controller and the components with indices $i_1, \ldots, i_k$, and where $\mathit{alldiff}(i_1, \ldots, i_k)$ states that all indices $i_1, \ldots, i_k$ are distinct. We shall call such a formula a *universal assertion* As an example, we can specify mutual exclusion by the invariant

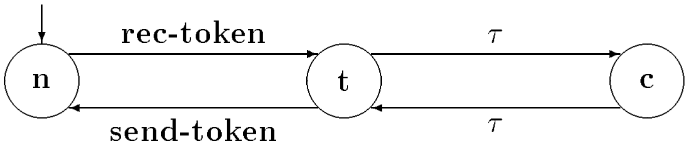
$$\forall i \neq j . \neg(\mathit{critical}[i] \ \wedge \ \mathit{critical}[j])$$

Examples of correctness properties that can be expressed in this form are: mutual exclusion, clock synchronization, security properties, etc.

The checking of an invariant can be equivalently reformulated as a reachability property. For instance, the above invariant means that it is not possible to reach a state which satisfies the negation $\exists i \neq j.(critical[i] \ \wedge \ critical[j])$ of the above invariant. We shall call such a formula an *existential assertion*

Example A simple example of a parameterized system is the following simple token-passing mutual exclusion algorithm. An arbitrary set of processes compete for a critical resource. Access to the resource is modeled by a process being in its critical section. In order to enter the critical section, a process must first acquire a token, which is passed in some arbitrary manner between components.

We model each process as a labeled transition system with three states: **n** (not possessing the token), **t** (possessing the token), and **c** (in the critical section), and two visible actions: **send-token** and **rec-token**. A state-transition diagram of a process is given below:



The synchronization predicate requires **send-token** and **rec-token** to occur in synchronized pairs. In a network, all process are initialized in the state **n**, except for one process, which starts in state **t**. There is no controller process. For this particular example, we could imagine both an unordered and a linearly ordered system topology (of course, any reasonable topology could be considered, but that is not in the scope of this paper).

A correctness property for this system is mutual exclusion, formulated as the invariant

$$\forall i \neq j. \ \neg(\mathbf{c}[i] \ \wedge \ \mathbf{c}[j])$$

3 Methods for Verification of Invariants of Parameterized Systems

In this section, we present the use of computational and of network invariants in the verification of a parameterized system.

3.1 Computational Invariants

A parametrized system can be viewed as an (infinite-state) program (which initially decides a value of n), to which standard verification methods can be applied. A safety property in the form of an invariant can be proved in the standard way by finding a computational (inductive) invariant for the system M. In order to automate this method, we must find a representation of (possibly infinite) sets of states of M. For instance, if each component $P[i]$ is finite-state, then in the case of unstructured topology one might use constraints on the number of occurrences of each control state of $P[i]$. In the case of a linear network, regular expressions could serve this purpose [CGJ95, KMM+97, ABJN99].

The standard requirements on a computational invariant I are:

1. any initial state of any M_n satisfies I,
2. each transition from a system state satisfying I leads to a system state which also satisfies I,
3. I implies ϕ.

It is a standard result that the set of invariants I that satisfy these requirements form a lattice with a smallest and a greatest element. The lattice is non-empty if and only if $M_n \models \phi$ for all n. There is a least N which satisfies the first two conditions: this is the set of reachable states of M_n. There is also a largest N which satisfies the two last conditions.

It is well-known that a computational invariant always exists if ϕ holds, but there is no guarantee that it can in general be expressed in the particular chosen representation (this follows from the undecidability of parameterized system verification [AK86]). Methods for automatically searching for a computational invariant include forward and backward reachability analysis, sometimes augmented by approximation techniques, using a symbolic representation of sets of states.

In what follows, we shall be particularly interested in finding computational invariants that are expressed as *universal assertions*. It turns out that backward reachability analysis suits the problem of finding universal assertions.

Example To continue the example in the previous section, we see that a simple computational invariant of any M_n is

$$\forall i \neq j.\ \neg[(\mathbf{t}[i] \ \vee \ \mathbf{c}[i]) \ \wedge \ (\mathbf{t}[j] \ \vee \ \mathbf{c}[j])]$$

which is indeed a universal assertion.

Below, we describe the relation to backward reachability in more detail.

Backward Reachability Analysis The basic idea of backward reachability analysis is to start from the negation of the invariant to be checked. If the invariant is a universal assertion, then the negation is an existential assertion. We assume that we can represent sets of states of an arbitrary but bounded set of components. The idea in backward reachability analysis is to compute the set of states from which a state satisfying the negated invariant can be reached.

If we further assume that each action of the parameterized system involves a bounded number of components (i.e., the synchronization predicate does not contain any occurrence of the superscript ω), then the set of predecessors of an existential assertion can again be expressed as an existential assertion.

Having a procedure for calculating $pre(\phi)$, a formula that represents the set of predecessors of states that satisfy ϕ, we can now perform symbolic verification as follows. Assume that $final$ is the existential constraint that represent the set of undesirable states, and let *initial* be a characterization of the initial states of a parameterized system. Typically *initial* will contain a universal quantification over the set of all processes in the network, e.g., saying that they are in their initial states.

A simple description of an algorithm for checking whether $final$ is reachable from *initial* is as follows [KMM+97].

Let $\phi_0 := final$
For $i = 0, 1, 2, \ldots$ **repeat**
   If $(\phi_i \ \wedge \ initial) \neq false$ **then return reachable**
   Let $\phi_{i+1} := \phi_i \ \vee \ pre(\alpha, \phi_i)$
until $\phi_{i+1} = \phi_i$
return unreachable

3.2 Network Invariants

A method, which has been proposed as a general solution to the verification of infinite-state systems, is to find a finite-state abstraction of the system, which preserves the correctness properties of interest, and thereafter model-checking this finite-state abstraction [LS97, LHR97, ID99]. Applied to our framework, this would entail the construction of a process N, whose behavior "contains" the behavior of any instance M_n of the family. Since we are considering to check only safety properties, we could for the containment relation (which we will denote by $\sqsubseteq$) choose either inclusion between finite traces of observable actions, or simulation. In both cases, we want that

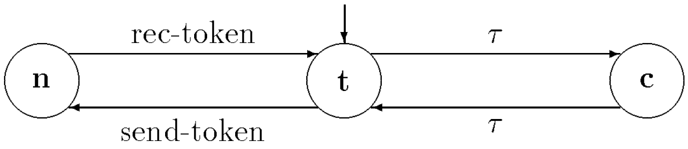
$$P_0 \parallel P[1] \parallel \cdots \parallel P[n] \sqsubseteq N,$$

A methodology, which has been proposed [WL89, KM89], is to find a (preferably finite-state) process N, for which the above relation can be checked by induction, checking

- $P_0 \sqsubseteq N$,
- $N \parallel P \sqsubseteq N$, and
- $N \models \phi$.

By standard fixpoint results, it follows [LHR97] that the set of labeled transition systems N that satisfy the above criteria form a lattice with a greatest and smallest element. The lattice is non-empty if and only if $M_n \models \phi$ for all n. There is a least N which satisfies the first two conditions: this is the "union" of all M_n. There is also a largest N which satisfies the two last conditions: this is the limit of the so-called quotient construction [And95, KLL$^+$97]. However, there is no guarantee that any of these be expressible by a finite-state program even if P_0 and P are finite-state. The discovery of a suitable N is therefore often performed manually (as e.g., in [WL89, KM89]), or using approximation techniques [LHR97].

Example A network invariant of the system in the preceding section is the finite-state process



where in the two left states, no process is in the critical section, and in the right state, exactly one process is in the critical section.

4 Generating Simple Invariants

In this section, we shall consider the question of finding a computational or network invariant of a particularly simple form. Recall that we only consider the verification of invariants expressed by universal assertions.

4.1 Universal Computational Invariants

We shall be concerned with finding computational invariants that are universal. For instance, in the small mutual exclusion example, it was possible to find a universal invariant. From results by Abdulla et al. [AČJYK96], the following theorem follows directly.

Theorem 4.1 *If P_0 and P are finite-state, and all synchronizations occur between a bounded number of components, then the largest invariant is universal. Furthermore, it (or rather its negation) can be found after a finite number of iterations of the backward reachability algorithm in the previous section.*

The main reason why the algorithm terminates is that it is impossible to generate infinite sequences $\phi_0, \phi_1, \phi_2, \phi_3, \ldots$ which are strictly increasing. Any such sequence will converge to a largest set after a finite number of iterations.

These results have been extended to cover a class of finite-state systems that allow also broadcast [EFM99], and to timed automata with one clock in [AJ98].

4.2 Finite-State Network Invariants

Let us thereafter consider network invariants. A natural desideratum is that such an invariant should not be more difficult to analyze than a single process P. That is, if the processes are finite-state, then the network invariant should be finite-state. In the example, it was easy to find a finite-state abstraction. However, Wolper and Lovinfosse [WL89] observe that by basic undecidability results [AK86], there must be cases where a network invariant does not exist (assuming that linearly structured networks are also allowed). In the literature, we have not found any results that guarantee the existence of finite-state network invariants, and we will therefore present conditions under which their existence can be guaranteed.

To this end, let us employ a standard framework for describing abstractions of concurrent systems. The abstraction N will be generated by collecting states of M into equivalence classes. This can be described by an abstraction mapping h from states of the family M to states of N [Lam89]. The condition for N to be an abstraction is then that

1. $h(s)$ is an initial state of N whenever s is an initial state of M, and that
2. if $s \xrightarrow{a} s'$ is a transition of M then if $h(s) \xrightarrow{a} h(s')$ is a transition of N.

In order to be able to verify that the abstraction proves the invariant, we shall consider the negation

$$\exists i_1, \ldots, i_k . \mathit{alldiff}(i_1, \ldots, i_k) \ \wedge \ \neg\phi(i_1, \ldots, i_k)$$

of the invariant to be proven. Each state n of N should be labeled by the set of *maximal subsets* of $\neg\phi(i_1, \ldots, i_k)$ that can occur in any state s with $n = h(s)$, in order that it can be checked whether the composition of N with other components may violate the invariant in some state (Clarke et al. [CGJ95] describe how to perform this for linearly ordered systems).

We will now investigate conditions for when the process N is inductive. Note that not all abstractions N are inductive: for instance the trivial N which has only one state with self-loops for all actions, is most often not inductive.

Definition 4.2 An abstraction mapping is *inductive* if whenever $h(s_1) = h(s_2)$, then for any state t of P we have $h(s_1 \parallel t) = h(s_2 \parallel t)$.

Inductive abstraction mappings generate inductive abstractions, as shown by the following:

Theorem 4.3 *An abstraction N of M which is obtained from M by an inductive abstraction function, is an inductive network invariant.*

Proof: Consider the parallel composition $N \parallel P$ of N and P. We shall prove that there exists an abstraction mapping g from states of $N \parallel P$ to N, from which the theorem follows directly. We first observe that an inductive abstraction mapping h from states of M to states of N induces a mapping g from states of $N \parallel P$ to states of N by letting $g(h(s) \parallel t) = h(s \parallel t)$, where s is a state of M and t is a state of P, and where $s \parallel t$ denotes the state s of M extended with one extra component in state t. The mapping g is well-defined since h is inductive. It can also be checked that g is indeed an abstraction mapping.

1. If n and t are initial, then $n = h(s)$ for some initial s, and so $g(n \parallel t) = h(s \parallel t)$ is initial.

2. If $n \parallel t \longrightarrow n' \parallel t'$ is a transition of $N \parallel P$, then there is a transition $s \longrightarrow s'$ of M with $h(s) = n$ and $h(s') = n'$. Thus $s \parallel t \longrightarrow s' \parallel t'$ is a transition of M, and hence $h(s \parallel t) \longrightarrow h(s' \parallel t') = g(n \parallel t) \longrightarrow g(n' \parallel t')$ is a transition of N.

$\square$

We will apply this theorem to obtain a result concerning the existence of finite-state abstraction mappings.

Theorem 4.4 *Assume that P_0 and P are finite-state, and that there is a universal computational invariant involving k quantified variables which proves that M satisfies the invariant. Then there is a finite-state network invariant satisfying the required properties.*

Proof: We construct N by partitioning the states that satisfy the computational invariant into a finite number of equivalence classes. Thereafter, we make N into an abstraction by inserting transitions and initial states, as required by the conditions of the abstraction function. (we let h be "is contained in this equivalence class"). We must only take care that the partitioning of states that satisfy the invariant be inductive. We therefore observe that the set of multisets of component states that satisfy a universal assertion is *downward closed* in the sense that if one state s satisfies the invariant, then any "substate" (consisting of the states of a subset of the components in s) also satisfy the invariant. If the invariant has k quantified variables, we can always find such a partitioning by letting two states be equivalent if each component state occurs either the same number of times in both states or more than k number of times in both states. □

We have thus shown that if there is a universal computational invariant, then there also exists a finite-state network invariant.

Example In the example of the mutual exclusion algorithm, the three states of the network invariant correspond to the following partitioning of the states.

n : 0 occurrences of both **t** and **c**
t : 1 occurrence of **t** and 0 of **c**
c : 0 occurrences of **t** and 1 of **c**

5 Application: Fischer's Protocol

In this section, we will illustrate the results of the preceding section by verifying a parameterized version of Fischer's protocol.

The protocol has been used as a measure of the performance of tools for verification of timed automata. The example was suggested by Fred Schneider and has been verified manually (e.g., [AL92]) and using theorem provers (e.g., [Sha93]). Several tools for verifying automata with a fixed number of clocks have been used to verify it for an increasing number of processes (e.g., [ACHH92]). Kristoffersen et al. [KLL+97] describes an experiment where the number of processes is 50.

The purpose of the protocol is to guarantee mutual exclusion in a concurrent

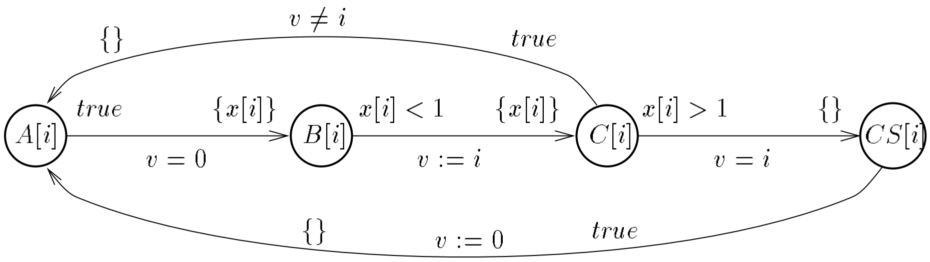


Figure 1: Fischer's Protocol for Mutual Exclusion.

system consisting of an arbitrary number of processes, using clocks and a shared variable. Each process has a local clock, and runs a protocol before entering the critical section. Each process has a local control state, which in our model assumes values in the set $\{A, B, C, CS\}$ where A is the initial state and CS represents the critical section. The processes also read from and write to a shared variable whose value is either $\perp$ or the index of one of the processes. A description in a graphical pseudo-code (taken from [KLL$^+$97]) of the behavior of a process with index i is given in Figure 1.

Intuitively, the protocol behaves as follows: A process wishing to enter the critical section starts in state A. If the value of the shared variable is $\perp$, the process can proceed to state B and reset its local clock. From state B, the process can proceed to state C if the clock value is still less than 1. In other words, the clock implements a timeout which guarantees that the process either stays in state B at most one time unit, or gets stuck in B forever. When moving from B to C, the process sets the value of the shared variable to its own index i and again resets its clock. From state C, the process can proceed to the critical section if the clock is strictly more than 1 and if the value of the shared variable is still i, the index of the process. Thus, in state C the clock enforces a delay which is longer than the length of the timeout in state B. Finally, when exiting the critical section, the process resets the shared variable to $\perp$. Processes that get stuck in state C can reenter the protocol by returning to state A. Since we do not intend to model liveness properties, such as e.g., absence of starvation, we do not impose requirements that force processes to change their state[1].

A rough argument for the correctness of the protocol goes as follows. The

[1] In fact, our formalism cannot express such requirements, although they can be added in terms of e.g., fairness constraints.

conditions on the shared variable ensure that a process cannot reach B if any other process is in C or CS. The timing conditions on the clocks ensure that a process cannot move from C to CS if some other process is still in B. Thus, if a set of processes start the mutual exclusion protocol and all arrive in C, then the process which was the last to enter C will read its own identity in the shared variable and enter the critical section.

Verification by Backwards Reachability Analysis If we perform a backwards reachability analysis for Fischer's protocol, starting from negation of the invariant being equivalent to

$$\exists i, j. CS[i] \ \wedge \ CS[j]$$

then we end up with convergence and the following fixedpoint

$$\begin{array}{ll} & \exists i,j.\ \ CS[i] \ \wedge \ CS[j] \\ \vee & \exists i,j.\ \ C[i] \ \wedge \ v = i \ \wedge \ CS[j] \\ \vee & \exists i,j.\ \ B[i] \ \wedge \ x[i] < 1 \ \wedge \ CS[j] \\ \vee & \exists i,j.\ \ B[i] \ \wedge \ x[i] < 1 \ \wedge \ C[j] \ \wedge \ v = j \ \wedge \ x[i] < x[j] \\ \vee & \exists i,j.\ \ A[i] \ \wedge \ v = 0 \ \wedge \ CS[j] \\ \vee & \exists i,j.\ \ C[i] \ \wedge \ v = 0 \ \wedge \ CS[j] \end{array}$$

The negation of this assertion is a universal invariant, which can be written as follows.

$$\begin{array}{lrl} & \forall i,j. & \neg(CS[i] \ \wedge \ CS[j]) \\ \wedge & \forall i,j. & (C[i] \ \wedge \ CS[j]) \implies v \neq i \\ \wedge & \forall i,j. & (B[i] \ \wedge \ CS[j]) \implies x[i] \geq 1 \\ \wedge & \forall i,j. & (B[i] \ \wedge \ C[j]) \implies (x[i] \geq 1 \ \vee \ v \neq j \ \vee \ x[i] \geq x[j]) \\ \wedge & \forall i. & (A[i] \ \wedge \ CS[i] \implies v \neq 0 \\ \wedge & \forall i. & (C[i] \ \wedge \ CS[i] \implies v \neq 0 \end{array}$$

However, a simpler to write computational invariant is the following:

$$\begin{array}{lrl} & \forall i. & CS[i] \implies v = i \\ \wedge & \forall i,j. & (B[i] \ \wedge \ CS[j]) \implies x[i] \geq 1 \\ \wedge & \forall i,j. & (B[i] \ \wedge \ C[j]) \implies (x[i] \geq 1 \ \vee \ v \neq j \ \vee \ x[i] \geq x[j]) \end{array}$$

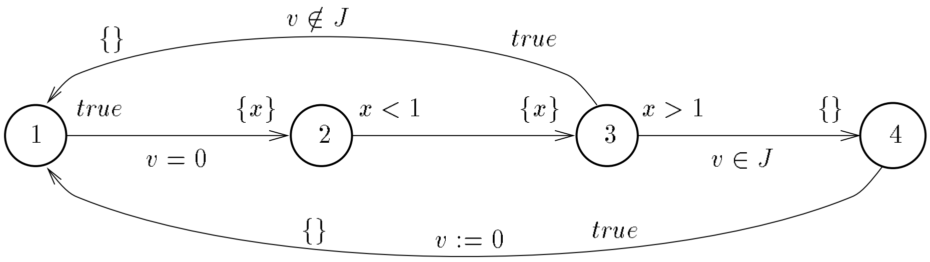


Figure 2: Network Invariant for Fischer's Protocol. Not shown in the graph is a self-loop at state 2, which sets v to an arbitrary index in J

Finding a Network Invariant The interface between components concerns the global variable v, the timing, and whether or not a component is in the critical section. We can therefore find a network invariant as a system with four states, shown in Figure 2. The figure shows an abstraction of a network containing indices in a set J of indices. Here, the four states are abstractions of network states with the following properties. We use $idle[i]$ to denote the property $A[i] \lor (B[i] \land x[i] \geq 1) \lor (C[i] \land v \neq i)$ denoting processes that are not active in competing for the critical section.

$$
\begin{array}{rl}
1: & \forall i.\ \ idle[i] \\
2: & (\forall i.\ \ idle[i] \lor B[i] \lor (C[i] \land x[i] < 1)) \land (\forall i,j.(B[i] \land C[j]) \implies x[i] \geq x[j]) \\
3: & \forall i.\ \ idle[i] \lor C[i] \\
4: & \forall i.\ \ idle[i] \lor C[i] \lor (CS[i]) \land v = i)
\end{array}
$$

Not shown in the graph is a self-loop at state 2, which sets v to an index in J

6 Conclusion

In this paper, we have considered methods for verifying parameterized systems using induction. We have considered the use of computational induction, and of induction over the network structure. The main contribution has been to give conditions under which there are guarantees for the existence of inductive invariants of a simple form. For the case of computational induction, we have merely repeated earlier results. For network invariants, we have presented a result that guarantees the existence of a network invariant, provided that there is a simple computational invariant.

We should point out that our results are only partial, and we believe that they can be made stronger. As they are, they indicate that simple network invariants exists whenever simple computational invariants exist. The results also show that the construction of both simple network and computational invariants can be automated for finite-state systems with only bounded synchronization.

References

[ABJ98] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *Proc.* 10^{th} *Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 305–318, 1998.

[ABJN99] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *Proc.* 11^{th} *Int. Conf. on Computer Aided Verification*, 1999.

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proc.* 5^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 414–425, Philadelphia, 1990.

[ACHH92] R. Alur, C. Courcoubetis, T. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verificationof hybrid systems. In Grossman, Nerode, Ravn, and Rischel, editors, *Hybrid Systems*, number 736 in Lecture Notes in Computer Science, pages 209–229, 1992.

[AČJYK96] Parosh Aziz Abdulla, Karlis Čerāns, Bengt Jonsson, and Tsay Yih-Kuen. General decidability theorems for infinite-state systems. In *Proc.* 11^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.

[AJ93] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *Proc.* 8^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 160–170, 1993.

[AJ98] Parosh Aziz Abdulla and Bengt Jonsson. Verifying networks of timed processes. In Bernhard Steffen, editor, *Proc. TACAS '98, 7^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 298–312, 1998.

[AK86] K. Apt and D.C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22:307–309, 1986.

[AL92] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In de Bakker, Huizing, de Roever, and Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, 1992.

[And95] H.R. Andersen. Partial model checking (extended abstract). In *Proc. 10^{th} IEEE Int. Symp. on Logic in Computer Science*, pages 398–407. IEEE Computer Society Press, 1995.

[BGK+96] J. Bengtsson, W. O. D. Griffioen, K.J. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In R. Alur and T. Henzinger, editors, *Proc. 8^{th} Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 244–256, New Brunswick, USA, 1996. Springer Verlag.

[BS95] O. Burkart and B. Steffen. Composition, decomposition, and model checking of pushdown processes. *Nordic Journal of Computing*, 2(2):89–125, 1995.

[CGJ95] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In Lee and Smolka, editors, *Proc. CONCUR '95, 6^{th} Int. Conf. on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 395–407. Springer Verlag, 1995.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. 14^{th} IEEE Int. Symp. on Logic in Computer Science*, 1999.

[EN95] E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *Proc. 22^{th} ACM Symp. on Principles of Programming Languages*, 1995.

[Fin94] A. Finkel. Decidability of the termination problem for completely specified protocols. *Distributed Computing*, 7(3), 1994.

[Hen95] T.A. Henzinger. Hybrid automata with finite bisimulations. In *Proc. ICALP '95*, 1995.

[ID99] C. Norris Ip and David L. Dill. Verifying systems with replicated components in Murφ. *Formal Methods in System Design*, 14(3), May 1999.

[Jan90] P. Jančar. Decidability of a temporal logic problem for Petri nets. *Theoretical Computer Science*, 74:71–93, 1990.

[JK95] Bengt Jonsson and Lars Kempe. Verifying safety properties of a class of infinite-state distributed algorithms. In *Proc. 7^{th} Int. Conf. on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 42–53. Springer Verlag, 1995.

[JM95] P. Jančar and F. Moller. Checking regular properties of Petri nets. In *Proc. CONCUR '95, 6^{th} Int. Conf. on Concurrency Theory*, pages 348–362, 1995.

[JP93] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 107(2):272–302, Dec. 1993.

[KLL$^+$97] K.J. Kristoffersen, F. Larroussinie, K. G. Larsen, P. Pettersson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. In *TAPSOFT '97 7th International Joint Conference on the Theory and Practice of Software Development*, Lecture Notes in Computer Science, Lille, France, April 1997. Springer Verlag.

[KM89] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proc. 8^{th} ACM Symp. on Principles of Distributed Computing, Canada*, pages 239–247, Edmonton, Alberta, 1989.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proc. 9^{th} Int. Conf. on Computer Aided Verification*, volume 1254, pages 424–435, Haifa, Israel, 1997. Springer Verlag.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems,. *Communications of the ACM*, 32(1):32–45, Jan. 1989.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *Proc. 24^{th} ACM Symp. on Principles of Programming Languages*, 1997.

[LS97] D. Lesens and H. Saidi. Abstraction of parameterized networks. *Electronic Notes in Theoretical Computer Science*, 9, 1997.

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

[Sha93] N. Shankar. Verification of real-time systems using PVS. In Courcoubetis, editor, *Proc. 5^{th} Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 280–291, 1993.

[Sti96] C. Stirling. Decidability of bisimulation equivalence for normed pushdown processes. In *Proc. CONCUR '96, 7^{th} Int. Conf. on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 217–232. Springer Verlag, 1996.

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, Vancouver, July 1998. Springer Verlag.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants (extended abstract). In Sifakis, editor, *Proc. Workshop on Computer Aided Verification*, number 407 in Lecture Notes in Computer Science, pages 68–80, 1989.

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc. 13^{th} ACM Symp. on Principles of Programming Languages*, pages 184–193, Jan. 1986.