Environment Abstraction for Parameterized Verification[⋆]

Edmund Clarke[1], Muralidhar Talupur[1], Helmut Veith[2]

[1]Carnegie Mellon University, Pittsburgh, PA, USA
[2]Technische Universität München, Munich, Germany

Abstract. Many aspects of computer systems are naturally modeled as parameterized systems which renders their automatic verification difficult. In well-known examples such as cache coherence protocols and mutual exclusion protocols, the unbounded parameter is the number of concurrent processes which run the same distributed algorithm. In this paper, we introduce environment abstraction as a tool for the verification of such concurrent parameterized systems. Environment abstraction enriches predicate abstraction by ideas from counter abstraction; it enables us to reduce concurrent parameterized systems with unbounded variables to precise abstract finite state transition systems which can be verified by a finite state model checker. We demonstrate the feasibility of our approach by verifying the safety and liveness properties of Lamport's bakery algorithm and Szymanski's mutual exclusion algorithm. To the best of our knowledge, this is the first time both safety and liveness properties of the bakery algorithm have been verified at this level of automation.

1 Introduction

We propose a new method for the verification of concurrent parameterized systems which combines predicate abstraction [21] with ideas from counter abstraction [29]. In predicate abstraction, the memory state of a system is approximated by a tuple of Boolean values which indicate whether certain properties ("predicates") of the memory state hold or not. For example, instead of keeping all 64 bits for two integer variables x, y, predicate abstraction may just track the Boolean value of the predicate $x > y$.

Counter abstraction, in contrast, is specifically tailored for concurrent parameterized systems which are composed of finite state processes: for each possible state s of a single finite state process, the abstract state contains a counter C_s which denotes the number of processes currently in state s. Thus, the process identities are abstracted away in counter abstraction. It can be argued that counter abstraction constitutes a very natural abstraction mechanism for protocols. In practice, the counters in counter abstraction are themselves abstracted in that they are cut off at value 2.

[⋆] This research was sponsored by the the National Science Foundation (NSF) under grants no. CCR-9803774 and CCR-0121547. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF. The third author was also supported by the EU GAMES Network.

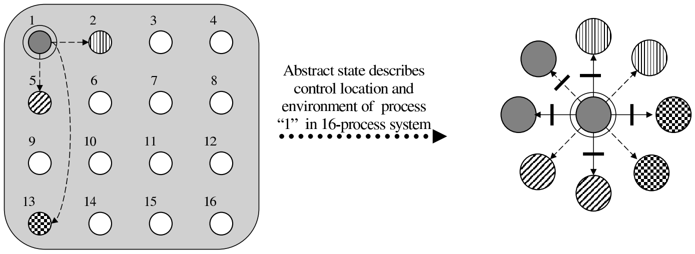
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Abstract state describes control location and environment of process "1" in 16-process system

Fig. 1. Abstraction Mapping.

Counter abstraction however has two main problems: first, it works only for finite state systems, and second, it assumes perfect symmetry, i.e., each process is identical to every other process in every aspect. Well-known algorithms such as Lamport's bakery algorithm are not directly amenable to counter abstraction: the bakery algorithm has an infinite state space due to an unbounded integer variable, and also an inherent asymmetry due to its use of process *id's*.

In this paper, we will address the two disadvantages of counter abstraction by incorporating the idea of counter abstraction into a new form of predicate abstraction: *since the state space is infinite, we do not count the processes in a given state as in traditional counter abstraction, but instead we count the number of processes satisfying a given predicate.* Note that the counters which we actually use in this paper are cut off at the value 1; such degenerated counters are evidently tantamount to existential quantifiers. Counting abstraction, too, usually needs only counters in the range $[0..2]$. Since our abstraction maintains the state of one process explicitly, a range of $[0..1]$ for each counter suffices.

Our new form of abstraction is also different from common predicate abstraction frameworks: Since the number of processes in a concurrent parameterized system is *unbounded*, the system does not have a single infinite-state model, but an infinite sequence of models which increase in complexity. Moreover, since the individual processes can have local data variables with unbounded range (e.g. integers), each of these models is an infinite-state system by itself. Thus, computing the abstract transition relation is a non-trivial task. Note that the predicates need to reflect the properties of a set of concurrent processes whose cardinality we do not know at verification time. To encode the necessary information into the abstract model, we will introduce *environment predicates*.

Environment Predicates. We use an asynchronous model which crucially distinguishes between the finite control variables of a process and the unbounded data variables of a process. The control variables are used to model the finite control of the processes while the data variables can be read by other processes in order to modify their own data variables. The variables can be used in the guards of the other processes, thus facilitating a natural communication among the processes.[1]

[1] We assume that transitions involving global conditions are treated atomically, i.e., while a process is evaluating e.g. a guard, no other process makes any transition. This simplification – which we shall call the *atomicity assumption* further on – is implicit in other works on parameterized verification, see [3, 5, 6, 29].

Figure 1 visualizes the intuition underlying environment abstraction. The grey box on the left hand side represents a concrete state of a system with 16 concurrent processes. The different colors of the disks/processes represent the internal states of the processes, i.e., the positions of the program counter.

The star-shaped graph on the right hand side of Figure 1 represents an abstract state. The abstract state contains one distinguished process – called the *reference process* x – which is at the center of the star. In this example, the reference process x represents process 1 of the concrete state. The disks on the circumference of the star represent the *environment* of the reference process. *Intuitively, the goal of the abstraction is to embed the reference process* x *of the abstract state into an abstract environment as rich as the environment which process* 1 *has in the concrete state.* Thus, the abstract state represents the concrete state "from the point of view of process 1."

To describe the environment of a process, we need to consider the relationships which can hold between the data variables of two processes. We can graphically indicate a specific relationship between any two processes by a corresponding arrow between the processes; the form of the arrow (full, dashed, etc.) determines *which* relationship the two processes have. In the figure, we assume that we have only two relationships R_1, R_2. For example, $R_1(x, y)$ might say that the local variable t of process x has the same value as local variable t in process y, while $R_2(x, y)$ might say that t has different values in processes x and y. Relationship R_1 is indicated by a full arrow, and R_2 is indicated by a dashed arrow. For better readability, not all relationships between the 16 processes are drawn.

More precisely, the environment of the reference process is described as follows: we enumerate all cases how the data variables in the reference process can relate to the data variables in a different process, as well as all possible program counter values which the other process can take. In our example, we have 2 relationships R_1, R_2 and 4 program counter positions, giving 8 different *environment conditions*. Therefore, the abstract state contains 8 environment processes on the circumference. For each of these 8 environment conditions, we indicate by the absence or presence of a *bar*, if this environment condition is actually satisfied by some process in the concrete state. For example, the dashed arrow from process 1 to the vertically striped process 2 in the concrete state necessitates a dashed arrow from x to a vertically striped process in the abstract state. Similarly, since there is no full arrow starting at process 1 in the concrete state, all full arrows in the abstract state have a bar. An *environment predicate* is a quantified formula which indicates the presence or absence of an environment condition for the reference process. We will give a formal definition of these notions in Section 4.

Note that a single abstract state in general represents an infinite number of concrete states. Moreover, a given concrete state gives rise to several abstract states, each of which is induced by choosing a different possible reference process. For example, the concrete state in Figure 1 may induce up to 16 abstract states, one for each process.

Existential Abstraction for Parameterized Systems. We construct an abstract system by a variant of existential abstraction. We include an abstract transition if *in some concrete instance of the parameterized system we can find a concrete transition between concrete states which match the abstract states with respect to the same reference pro-*

cess. The abstract model obtained by environment abstraction is a sound abstraction which preserves both safety and liveness properties.

In this paper we use a simple input language which is general enough to describe most practically relevant symmetric protocols, and to demonstrate the underlying principles of our abstraction. We believe that our abstraction method can be naturally generalized for additional constructs as well.

To handle liveness we augment the abstract model using an approach suggested by [29]. Note that in contrast to the indexed predicates method [24,25], our approach constructs an abstract transition system, instead of computing the set of reachable abstract states. This feature of our approach is crucial for verifying liveness properties.

Tool Chain and Experiments. Our approach provides an automated tool chain in the tradition of model checking.

1. The user feeds the protocol described in our language to the verification tool.
2. The environment abstraction tool extracts a finite state model from the process description, and puts the model in NuSMV format.
3. NuSMV verifies the specified properties.

Using the abstraction method described here, we have been able to verify automatically the safety and liveness properties of two well known mutual exclusion algorithms, namely Lamport's Bakery algorithm [26] and Szymanski's algorithm [31]. While safety and liveness properties of Szymanski's algorithm have been automatically verified with atomicity assumption by Baukus et al [5], this is the first time both safety and liveness of Lamport's bakery algorithm have been verified (with the atomicity assumption) at this level of automation.

2 Discussion of Related Work

Verification of parameterized systems is well known to be undecidable [2,30]. Many interesting approaches to this problem have been developed over the years, including the use of symbolic automata-based techniques [1,23,8,7], invariant based techniques [3,28], predicate abstraction [24], or exploiting symmetry [11,14,17,15,16]. Some of the earliest work on verifying parameterized systems includes works by Browne et al [9], German and Sistla [20], Emerson and Sistla [16]. In the rest of this section, we will concentrate on the work which is closest to our approach.

Counter Abstraction [4,12,13,29,20] is an intuitive method to use on parameterized systems. Pnueli et al [29] who coined the term counter abstraction show how systems composed of symmetric and finite state processes can be handled automatically. Protocols which either break symmetry by exploiting knowledge of process id's or which have infinite state spaces however require manual intervention. Thus, the verification of Szymanski's and the Bakery protocol in [29] requires manual introduction of new variables. The method also makes assumptions on the atomicity of guards.

The *Invisible Invariants* method was introduced in a series [28,3,18,19] of papers. The idea behind this technique is to find an invariant for the parameterized system by examining concrete systems for low valuations of the parameter(s). The considered system model is powerful enough to model various mutual exclusion and cache coherence protocols which do not need unbounded integer variables. In [3], a modified version of the bakery algorithm is verified: the original bakery algorithm is modified to eliminate unbounded integer variables. In contrast, the method proposed in the current paper can handle the original bakery protocol without such modifications. The authors of [3] implicitly make the assumption that guards are evaluated atomically.

The *Indexed Predicates* method [24, 25] is a new form of predicate abstraction for infinite state systems. This method relies on the observation that complex invariants are built of simple *indexed predicates*, i.e., predicates which have free index variables. By choosing a set of indexed predicates appropriately one can use a modified form of predicate abstraction to find a system invariant. In comparison to the above mentioned work, this method makes weaker atomicity assumptions.

Our method is also based on predicate abstraction; in fact, the notion of a reference process can be viewed as an "index" in the indexed predicates framework. However, the contribution we make is very different: (i) We focus on concurrent parameterized systems which enables us to use the specific and precise technique of environment abstraction. Our abstraction method exploits and reflects the structure of communicating protocols. (ii) In the indexed predicates approach there is no notion of an abstract transition relation. Thus, their approach, which is tailored for computing reachable states, works only for safety properties. In our framework, the abstract model does have a transition relation, and we can verify liveness properties as well as safety properties. (iii) The indexed predicate technique requires manual intervention or heuristics for choosing appropriate predicates. In contrast, our technique is automatic.

A method pioneered by Baukus et al [5] models an infinite class of systems by a single *WS1S system* which is then abstracted into a finite state system. While this is an automatic technique it cannot handle protocols such as the Bakery algorithm which have unbounded integer variables. The global conditions are assumed to be atomic.

The *inductive method* of [27] based on model checking is applied to verify both safety and liveness of the Bakery algorithm, notably without assuming atomicity. This approach however is not automatic: the user is required to provide lemmas and theorems to prove the properties under consideration. Our approach in contrast is fully automatic.

Regular model checking [8] is an interesting verification technique very different from ours. It is based on modeling systems using regular languages. This technique is applicable to a wide variety of systems but it requires the user to express systems in terms of regular languages which is a non-trivial process and requires user ingenuity.

Henzinger et. al. [22] also consider the problem of unbounded number of threads but the system model they consider is different. The communication between threads occurs through shared variables, whereas in our case, each process can look at the state of the other processes.

In summary, automatic methods such as the WS1S method and counter abstraction are restricted in the systems they can handle and make use of the atomicity assumption. In contrast, the methods which make no or weaker assumptions about atomicity tend to require user intervention, either in the form of providing appropriate predicates or in the form of lemmas and theorems which lead to the final result. In this paper, we assume atomicity of guards and describe a method which can handle well known mutual exclusion protocols such as the Bakery and Szymanski's protocols automatically. Importantly, our method is able to abstract and handle unbounded integer variables. To the best of our knowledge, this is the first time that the Bakery algorithm (under atomicity assumption) has been verified automatically.

The method of environment abstraction described here has a natural extension which eliminates the atomicity assumption. This extension of our method, which will be described in future work, has been used to verify the Bakery algorithm and Szymanski's protocol without any restrictions.

3 System Model

Parameterized Systems. We consider asynchronous systems composed of an unbounded number of processes which communicate via shared variables. Each process can modify its own variables, but has only read access to the variables of the other processes. Each process has two sets of variables: the control variables $\mathbf{F} = \{f_1, \ldots, f_c\}$, where each f_i has a finite, constant range and the data variables $\mathbf{U} = \{u_1, \ldots u_d\}$, where each u_i is an unbounded integer. Intuitively, the two sets of variables serve different purposes: (i) The control variables in $\mathbf{F}$ determine the internal control state of the process. As they have a finite domain, the variables in $\mathbf{F}$ amount to the finite control of the process. (ii) The data variables in $\mathbf{U}$ contain actual data which can be read by other processes to calculate their own data variables.

All processes run the same protocol P. For a given protocol P, a system consisting of K processes running P will be denoted by $\mathcal{P}(K)$. Thus, the number K of processes is the system parameter. We will write $\mathcal{P}(\mathbf{N})$ to denote the infinite collection $\mathcal{P}(2), \mathcal{P}(3), \ldots$ of systems. To be able to refer to the state of individual processes in a system $\mathcal{P}(K)$ we will assume that each process has a distinct and fixed *process id* from the range $[1..K]$. We will usually refer to processes and their variables via their process id's. In particular, $f_a[i]$ and $u_b[i]$ denote the variables f_a and u_b of the process with id i. The set of *local states* of a process i is then naturally given by the different valuations of the tuple $\langle f_1[i], \ldots, f_c[i], u_1[i], \ldots, u_d[i]\rangle$. The global state of system $\mathcal{P}$(K) is given by a tuple $\langle \mathcal{L}_1, \ldots, \mathcal{L}_K\rangle$, where each $\mathcal{L}_i$ is the local state of process i. The initial state of each process is given by a fixed valuation of the local state variables. Note that all processes in a system $\mathcal{P}(K)$ are identical except for their $id's$. Thus, the process id's are the only means to break the symmetry between the processes. A process can use the reserved expression slf to refer to its own process id. When a protocol text contains the variables f_a or u_b without explicit reference to a process id, then this stands for $f_a[\mathsf{slf}]$ and $u_b[\mathsf{slf}]$ respectively.

A concrete valuation of the variables in $\mathbf{F}$ determines the control state of a process. *Without loss of generality, we can assume for simplicity that* $\mathbf{F}$ *has only one variable* pc *which determines the control state of a process.* Thus, in the rest of the paper $\mathbf{F} = \{\mathsf{pc}\}$, although in program texts we may take the freedom to use more than one finite range control variable. A formula of the form $\mathsf{pc} = \mathrm{const}$ is called a *control assignment*. The range of pc is called the set of control locations.

Guarded Transitions and Update Transitions. We will describe the transition relation of the processes in terms of two basic constructs, *guarded transitions* for the finite control, and the more complicated *update transitions* for modifying data variables. A guarded transition has the form

$$\mathsf{pc} = L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } \mathsf{pc} = L_2 \textbf{ else goto } \mathsf{pc} = L_3$$

or shorter

$$L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2 \textbf{ else goto } L_3$$

where L_1, L_2, L_3 are control locations. In the guard $\forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ the variable otr ranges over the process id's of all other processes. The condition $\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ is any formula involving the data variables of processes slf, otr and the pc variable of otr. The semantics of a guarded transition is straightforward: in control location L_1, the process evaluates the guard and changes to control location L_2 or L_3 accordingly.

Update transitions are needed to describe protocols such as the Bakery algorithm where a process *computes* a data value depending on all values which it can read from other processes. For example, the Bakery algorithm has to compute the maximum of a certain data variable (the "ticket variable") in all other processes. Thus, we define an update transition to have the general form

$$\begin{array}{ll} L_1 : & \textbf{for all } \mathsf{otr} \neq \mathsf{slf} \quad \textbf{if } \mathcal{T}(\mathsf{slf}, \mathsf{otr}) \textbf{ then } u_k := \phi(\mathsf{otr}) \\ & \textbf{goto } L_2 \end{array}$$

where L_1 and L_2 are control assignments, and $\mathcal{T}(\mathsf{slf}, \mathsf{otr})$ is a condition involving data variables of processes $\mathsf{slf}, \mathsf{otr}$. The semantics of the update transition is best understood in an operational manner: In control location L_1, the process scans over all the other processes (in nondeterministically chosen order), and for each process otr checks if the formula $\mathcal{T}(\mathsf{slf}, \mathsf{otr})$ is true. In this case, the process changes the value of its data variable u_k according to $u_k := \phi(\mathsf{otr})$, where $\phi(\mathsf{otr})$ is an expression involving variables of process otr. Thus, the variable u_k can be reassigned multiple times within a transition. Finally, the process changes to control location L_2. *We assume that both guarded and update transitions are atomic, i.e., during their execution no other process makes a move.*

Example 1. As an example of a protocol written in this language, consider a parameterized system $\mathcal{P}(\mathbf{N})$ where each process P has one finite variable $\mathsf{pc} : \{1, 2, 3\}$ representing a program counter, one unbounded/integer variable $t : \mathbf{Int}$, and executes the following program:

```
1 : goto 2
2 : if ∀otr ≠ slf.t[slf] ≠ t[otr] then goto 3
3 : t := t[otr] + 1;   goto 1
```

The statement $1 : \ \textbf{goto}\ 2$ is syntactic sugar for

$$\mathsf{pc} = 1 : \ \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\textbf{true then goto } \mathsf{pc} = 2 \textbf{ else goto } 1$$

Similarly, $3: \ t := t[\mathsf{otr}] + 1; \quad \mathbf{goto} \ = 1$ is syntactic sugar for

$$\mathsf{pc} = 3: \ \textbf{if} \ \forall \mathsf{otr} \neq \mathsf{slf}.\textbf{true then} \ t := t[\mathsf{otr}] + 1 \ \textbf{goto} \ \mathsf{pc} = 1.$$

This example also illustrates that most commonly occurring transition statements in protocols can be written in our input language. □

Note that we have not specified the operations and predicates which are used in the conditions and assignments. Essentially, this choice depends on the protocols and the power of the decision procedures used. For the protocols considered in this paper, we need linear order and equality on data variables as well as incrementation, i.e., addition by 1. The full version of the paper [10] contains the descriptions of the Bakery algorithm and Szymanski's algorithm in terms of our language.

4 Environment Abstraction

In this section, we describe the principal framework of environment abstraction. In Section 5 we will discuss how to actually compute abstract models for the class of parameterized systems introduced in the previous section. Both tasks are non-trivial, as we need to construct a finite abstract model which reflects the properties of $\mathcal{P}(K)$ for all $K \geq 1$. We shall write $\mathcal{P}(\mathbf{N}) \models \Phi$ to say that $\mathcal{P}(K) \models \Phi$ for all parameters $K > 1$. Given a specification Φ and a system $\mathcal{P}(\mathbf{N})$, we will construct an abstract model $\mathcal{P}^{\mathcal{A}}$ and an abstract specification $\Phi^{\mathcal{A}}$ such that $\mathcal{P}^{\mathcal{A}} \models \Phi^{\mathcal{A}}$ implies $\mathcal{P}(\mathbf{N}) \models \Phi$. The converse does not have to hold, i.e., the abstraction is sound but not complete.

We will first describe how to construct the abstract model. We have already informally visualized and discussed the abstraction concept using Figure 1. More formally, our approach is best understood by viewing the abstract state as a *description* $\Delta(x)$ of the computing environment of a reference process x. Since x is a variable, we can then meaningfully say that the description $\Delta(x)$ holds true or false for a concrete process. We write $g \models \Delta(p)$ to express that in a global state g, $\Delta(x)$ holds true for the process p.

An abstract state (i.e., a description $\Delta(x)$) contains (i) detailed information about the current internal state of x and (ii) information about the internal states of other processes and their relationship to x. Since the number of other processes is not fixed, we can either *count* the number of processes which are in a given relationship to x, or, as in the current paper, keep track of the *existence* of such processes.

Technically, our descriptions reuse the predicates which occur in the control statements of the protocol description. Let S be the number of control locations in the program P. The internal state of a process x can be described by a predicate of the form

$$pc[x] = L$$

where $L \in \{1..S\}$ is a control location.

In order to describe the relations between the data variables of different processes we collect *all* predicates $\mathcal{EP}_1(x, y), \ldots, \mathcal{EP}_r(x, y)$ which occur in the guards of the program. From now on we will refer to these predicates as the *inter-predicates* of the program. Since in most practical protocols, synchronization between processes involves only one or two data variables, the number of inter-predicates is usually quite small. The relationship between a process x and a process y is now described by a formula of the form

$$R_i(x,y) \doteq \pm\mathcal{EP}_1(x,y) \wedge \ldots \wedge \pm\mathcal{EP}_r(x,y)$$

where $\pm\mathcal{EP}_i$ stands for $\mathcal{EP}_i$ or its negation $\neg\mathcal{EP}_i$. It is easy to see that there are 2^r possible relationships $R_1(x,y),\ldots,R_{2^r}(x,y)$ between x and y. In the example of Figure 1, the two relationship predicates R_1, R_2 are visualized by full and dashed arrows.

Fact 1 *The relationship conditions $R_1(x,y),\ldots,R_{2^r}(x,y)$ are mutually exclusive.*

Before we explain the descriptions $\Delta(x)$ in detail, let us first describe the most important building blocks for the descriptions which we call *environment predicates*. An environment predicate expresses that for process x we can find another process y which has a given relationship to process x and a certain internal state. The environment predicates thus have the form

$$\exists y.y \neq x \wedge R_i(x,y) \wedge \mathsf{pc}[y] = j.$$

An environment predicate says the following: *there exists a process y different from x whose relationship to x is described by the $\mathcal{EP}$ predicates in R_i, and whose internal state is j.* There are $T := 2^r \times S$ different environment predicates; we name them $\mathcal{E}_1(x),\ldots,\mathcal{E}_T(x)$, and their quantifier-free matrices $E_1(x,y),\ldots,E_T(x,y)$. Note that each $E_k(x,y)$ has the form $y \neq x \wedge R_i(x,y) \wedge \mathsf{pc}[y] = j$.

Fact 2 *If an environment process y satisfies an environment condition $E_i(x,y)$, then it cannot simultaneously satisfy any other environment condition $E_j(x,y)$, $i \neq j$.*

Fact 3 *Let $E_i(x,y)$ be an environment condition and $\mathcal{G}(x,y)$ be a boolean formula over the inter-predicates $\mathcal{EP}_1(x,y),\ldots,\mathcal{EP}_r(x,y)$ and predicates of the form $\mathsf{pc}[y] = L$. Then either $E_i(x,y) \Rightarrow \mathcal{G}(x,y)$ or $E_i(x,y) \Rightarrow \neg\mathcal{G}(x,y)$.*

We are ready to return to the descriptions $\Delta(x)$. A description $\Delta(x)$ has the format

$$\mathsf{pc}[x] = i \quad \wedge \quad \pm\mathcal{E}_1(x) \wedge \pm\mathcal{E}_2(x) \wedge \cdots \wedge \pm\mathcal{E}_T(x), \qquad \text{where } i \in [1..S]. \qquad (*)$$

Intuitively, a description $\Delta(x)$ therefore gives detailed information on the internal state of process x, and how the other processes are related to process x. Note the correspondence of $\Delta(x)$ to the abstract state in Figure 1: the control location i determines the color of the central circle, and the $\mathcal{E}_j$ determine the processes surrounding the central one.

We will now represent descriptions $\Delta(x)$ by tuples of values, as usual in predicate abstraction. The possible descriptions $(*)$ only differ in the value of the program counter $\mathsf{pc}[x]$ and in where they have negations in front of the $\mathcal{E}$ predicates. Denoting negation by 0 and absence of negation by 1, every description $\Delta(x)$ can be identified with a tuple $\langle \mathbf{pc}, e_1, \ldots e_T\rangle$ where $\mathbf{pc}$ is a control location, and each e_i is a boolean variable. From this point of view, we have two ways to speak about abstract states: as descriptions $\Delta(x)$, and as tuples $\langle \mathbf{pc}, e_1, \ldots, e_T\rangle$. Thinking of abstract states as descriptions is more intuitive in the conceptual phase of this work, while the latter approach is more in line with traditional predicate abstraction, and closer to the algorithms we use.

Example 2. Consider again the protocol shown in Example 1. There is only one inter-predicate $\mathcal{EP}_1(x,y) \doteq t[x] \neq t[y]$. Thus we have two possible relationship conditions $R_1(x,y) \doteq t[x] = t[y]$ and $R_2(x,y) \doteq t[x] \neq t[y]$. Consequently, we have 6 different environment predicates:

$$\begin{array}{ll} \mathcal{E}_1(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 1 \wedge R_1(x,y) & \mathcal{E}_4(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 1 \wedge R_2(x,y) \\ \mathcal{E}_2(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 2 \wedge R_1(x,y) & \mathcal{E}_5(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 2 \wedge R_2(x,y) \\ \mathcal{E}_3(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 3 \wedge R_1(x,y) & \mathcal{E}_6(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 3 \wedge R_2(x,y) \end{array}$$

The abstract state then is a 7-tuple $\langle \mathbf{pc}, e_1, \ldots, e_6 \rangle$ where $\mathbf{pc}$ refers to the internal state of the reference process x. For each $i \in [1..6]$, the bit e_i tells whether there is an environment process $y \neq x$ such that the environment predicate $\mathcal{E}_i(x)$ becomes true. □

Definition 1 (Abstract States). *Given a parameterized system $\mathcal{P}(\mathbf{N})$ with control locations $\{1,..,S\}$ and environment predicates $\mathcal{E}_1(x), \ldots, \mathcal{E}_T(x)$, the abstract state space contains tuples $\langle \mathbf{pc}, e_1, \ldots e_T \rangle$, where*

- $\mathbf{pc} \in \{1,..,S\}$ *denotes the control location of the reference process.*
- *each e_j is a Boolean variable corresponding to the predicate $\mathcal{E}_j(x)$.*

Since the concrete system $\mathcal{P}(K)$ contains K processes, a state $s \in \mathcal{P}(K)$ can give rise to up to K different abstract states, one for every different choice of the reference process.

Definition 2 (Abstraction Mapping). *Let $P(K)$, $K > 1$, be a concrete system and $p \in [1..K]$ be a process. The abstraction mapping α_p induced by p* maps *a global state g of $\mathcal{P}(K)$ to an abstract state $\langle \mathbf{pc}, e_1, \ldots, e_T \rangle$ where*

$$\mathbf{pc} = \textit{ the value of } \mathsf{pc}[p] \textit{ in state } g \quad \textit{and for all } e_j \textit{ we have } e_j = 1 \Leftrightarrow g \models \mathcal{E}_j(p).$$

Definition 3 (Abstract Model). *The abstract model $\mathcal{P}^{\mathcal{A}}$ is given by the transition system $(S^{\mathcal{A}}, \Theta^{\mathcal{A}}, \rho^{\mathcal{A}})$ where*

- $S^{\mathcal{A}} = \{1,..,S\} \times \{0,1\}^T$, *the set of abstract states, contains all valuations of the tuple $\langle \mathbf{pc}, e_1, \ldots, e_T \rangle$.*
- $\Theta^{\mathcal{A}}$, *the set of initial abstract states, is the set of abstract states $\hat{s}$ such that there exists a concrete initial state s of a concrete system $\mathcal{P}(K)$, $K > 1$, such that there exists a concrete process p with $\alpha_p(s) = \hat{s}$.*
- $\rho^{\mathcal{A}} \subseteq S^{\mathcal{A}} \times S^{\mathcal{A}}$ *is a transition relation on the abstract states defined as follows: There is a transition from abstract state $\hat{s}_1$ to abstract state $\hat{s}_2$ if* there exist
  - *(i) a concrete system $\mathcal{P}(K), K > 1$ with a process p*
  - *(ii) a concrete transition from concrete state s_1 to s_2 in $\mathcal{P}(K)$*

  such that $\alpha_p(s_1) = \hat{s}_1$ *and* $\alpha_p(s_2) = \hat{s}_2$.

4.1 Specifications

We will now focus on the properties that we want to verify. By a *one process* control condition we mean a boolean formula over expressions of the form $\mathsf{pc}[x] = L, L \in \{1, .., S\}$. By a two process control condition we mean a boolean formula over expressions of the form $\mathsf{pc}[x] = L_1, \mathsf{pc}[y] = L_2,$ where $L_1, L_2 \in \{1, .., S\}$.

Definition 4 (Two-Indexed Safety Properties). *A two-indexed safety property is a specification $\forall x, y.\ \mathbf{AG}\phi(x, y)$, where x, y are variables which refer to* distinct *processes, and $\phi(x, y)$ is a two process control condition.*

Definition 5 (Liveness Properties). *A liveness property is a specification of the form $\forall x.\ \mathbf{AG}(\phi(x) \rightarrow \mathbf{F}\psi(x))$, where $\phi(x)$ and $\psi(x)$ are one process control conditions.*

A standard example of a two-indexed safety property is the mutual exclusion property $\forall x, y.\ \mathbf{AG}\neg(\mathsf{pc}[x] = \mathrm{crit} \wedge \mathsf{pc}[y] = \mathrm{crit})$, where crit is the control location of the critical section. An example of a liveness property is the formula $\forall x.\ \mathbf{AG}\ (\mathsf{pc}[x] = \mathrm{try} \rightarrow\ \mathbf{F}\ \mathsf{pc}[x] = \mathrm{crit})$ which expresses that a process gets to enter the critical section if it wants to.

We first show how to abstract a formula $\phi(x, y)$ without any temporal operators. The abstraction $\phi^{\mathcal{A}}$ of $\phi(x, y)$ is a predicate over the abstract states that is satisfied by those and only those abstract states $\hat{s}$ for which there exists a system $\mathcal{P}(K)$, $K > 1$ with a process p, and a global state s of $\mathcal{P}(K)$ such that

$$\alpha_p(s) = \hat{s} \text{ and } \forall q \neq p.\ (s \models \phi(p, q)).$$

Intuitively, we treat x as the reference process and y as an environment process and find which abstract states correspond to the concrete formula $\phi(x, y)$. Similarly, for a single index property $\phi(x)$, its abstraction $\phi^{\mathcal{A}}$ is the predicate that is satisfied by those and only those abstract states $\hat{s}$ for which there exists a system $\mathcal{P}(K)$, $K > 1$, with a process p and a global state s of $\mathcal{P}(K)$ such that $\alpha_p(s) = \hat{s}$ and $s \models \phi(p)$.

Now we can define the abstract specifications: The abstraction of a two-indexed safety property $\forall x, y.\ \mathbf{AG}\phi(x, y)$ is the formula $\mathbf{AG}\phi^{\mathcal{A}}$. The abstraction of a single-indexed liveness property $\forall x.\ \mathbf{AG}\ (\phi(x) \rightarrow\ \mathbf{F}\psi(x))$ is the formula $\mathbf{AG}\ (\phi^{\mathcal{A}} \rightarrow\ \mathbf{F}\psi^{\mathcal{A}})$.

Theorem 1 (Soundness of Abstraction). *Let $\mathcal{P}(\mathbf{N})$ be a parameterized system and $\mathcal{P}^{\mathscr{A}}$ be an* over-approximation *of its abstraction $\mathcal{P}^{\mathcal{A}}$. Given any two-indexed safety or single-indexed liveness property Φ and its abstraction $\Phi^{\mathcal{A}}$ we have $\mathcal{P}^{\mathscr{A}} \models \Phi^{\mathcal{A}}$ implies $\mathcal{P}(\mathbf{N}) \models \Phi$.*

4.2 Extensions for Fairness and Liveness

The abstract model that we have described, while sound, might be too coarse in practice to be able to verify liveness properties. The reason is two fold:

(i) Spurious Infinite Paths. Our abstract model may have infinite paths which cannot occur in any concrete system. This happens when two concrete states s_1 and s_2, where s_1 transitions to s_2, both map to the same abstract state $\hat{s}$, leading to a self-loop involving $\hat{s}$. Such a self-loop can lead to a spurious infinite path which hinders the verification of liveness properties.

(ii) Fairness Conditions. Liveness properties are usually expected to hold under some fairness conditions. A typical example of a fairness condition is that every process x must leave the critical section a finite time after entering it. This is expressed formally by the fairness condition $\mathsf{pc}[x] \neq \mathrm{crit}$. In this paper we will consider fairness conditions $\mathsf{pc}[x] \neq L$, where L is a control location. Liveness properties are then expected to hold on *fair paths*: an infinite path in a concrete system $\mathcal{P}(K)$, $K \geq 1$ is *fair* only if the fairness condition $\mathsf{pc}[i] \neq L$ holds for each process i infinitely often.

To handle these situations, we adapt a method developed by Pnueli et al. [29] in the context of counter abstraction to our environment abstraction. To this end, we augment our abstract model by adding new *Boolean* variables $\textbf{from}_i, \textbf{to}_i$ for every $i \in [1..T]$. Thus our new abstract states are tuples $\langle \textbf{pc}, e_1, \ldots, e_T, \textbf{from}_1, \ldots, \textbf{from}_T, \textbf{to}_1, \ldots, \textbf{to}_T \rangle$. We will now briefly describe this extension.

Intuitively, the new **from**, **to** variables keep track of the *immediate* history of an abstract state, that is, the last step by which the abstract state was reached. The variable $\textbf{from}_i$ is **true** if a process y having satisfied $E_i(x, y)$ in the previous state does not satisfy $E_i(x, y)$ in the new state. Similarly, the variable $\textbf{to}_i$ is **true** if the active process having satisfied $E_j(x, y), j \neq i$ in the previous state satisfies $E_i(x, y)$ in the new state. To eliminate the spurious infinite paths arising from loops described in item **(i)** above, we add for each $i \in [1..T]$ a *compassion condition* [29] $\langle \textbf{from}_i, \textbf{to}_i \rangle$ which says *If* $\textbf{from}_i = \mathbf{true}$ *holds infinitely often in a path, then* $\textbf{to}_i = \mathbf{true}$ *must hold infinitely often as well.*

Let us now turn to item **(ii)**. Given a concrete fairness condition of the form $\mathsf{pc}[x] \neq L$, the corresponding abstract fairness condition for the *reference process* is given by $\textbf{pc} \neq L$. Moreover, we introduce fairness conditions $\neg(\textbf{from}_i = \textbf{false} \wedge e_i = 1)$ for all those environments $E_i(x, y)$ *which require process y to be in control location L*, i.e., those $E_i(x, y)$ which contain the subformula $\mathsf{pc}[y] = L$. For such an environment condition E_i, the fairness condition $\neg(\textbf{from}_i = \textbf{false} \wedge e_i = 1)$ excludes the case that there are environment processes satisfying $E_i(x, y)$ which never move. For a more detailed explanation and proofs please consult the full version.

5 Computing the Abstract Model

In our implementation, we consider protocols in which all inter-predicates $\mathcal{EP}_i(x, y)$ have the form $t[x] \prec t[y]$ where $\prec \in \{<, >, =\}$ and t is a data variable.[2] Thus, each local process compares its own variables only with their counterparts in other processes. Most real protocols satisfy this condition. Our results however do not depend on this particular choice of inter-predicates.

Computing the abstract transition relation is evidently complicated by the fact that there is an infinite number of concrete systems. To get around this problem, we consider each concrete transition statement of the program separately and *over-approximate* the

[2] The incrementation operation occurs only on the right hand side of assignments in update transitions.

set of abstract transitions it can lead to. Their union will be our abstract transition relation.

A concrete transition can either be a guarded transition or an update transition. Each transition can be executed by the reference process or one of the environment processes. Thus there are four cases to consider:

Active process is . . .	guarded transition	update transition
. . . reference process	Case 1	Case 2
. . . environment process	Case 3	Case 4

In this section we will consider Case 1, that is, the reference process executing the guarded transition. Computing the abstract transition in other cases is similar in spirit but quite lengthy. We refer the reader to the full version [10] of this paper for a more detailed description of how we compute the abstract initial condition and the abstract transition relation.

Let us now turn to Case 1 in detail, and consider the guarded transition

$$L_1 : \quad \textbf{if}\ \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})\ \textbf{then goto}\ L_2\ \textbf{else goto}\ L_3. \qquad (*)$$

Suppose the reference process is executing this guarded transition statement. If at least one of the environment processes contradicts the guard $\mathcal{G}$ then the reference process transitions to control location L_3, i.e., the *else branch*. Otherwise, the reference process goes to L_2. We will now formalize the conditions under which the *if* and *else* branches are taken.

Definition 6 (Blocking Set for Reference Process). *Let $\mathscr{G} \doteq \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ be a guard. We say that an environment condition $E_i(x, y)$ blocks the guard $\mathscr{G}$ if $E_i(x, y) \Rightarrow \neg\mathcal{G}(x, y)$. The set $\mathcal{B}^x(\mathscr{G})$ of all indices i such that $E_i(x, y)$ blocks $\mathscr{G}$ is called the blocking set of the reference process for guard $\mathscr{G}$.*

Note that by Fact 3, either $E_i(x, y) \Rightarrow \neg\mathcal{G}(x, y)$ or $E_i(x, y) \Rightarrow \mathcal{G}(x, y)$ for every environment $E_i(x, y)$. The intuitive idea behind the definition is that $\mathcal{B}^x(\mathscr{G})$ contains the indices of all environment conditions which enforce the *else branch*. We will now explain how to represent the guarded transition $(*)$ in the abstract model: we introduce an abstract transition from $\hat{s_1} = \langle \mathbf{pc}, e_1, .., e_T, \mathbf{from}_1, .., \mathbf{from}_T, \mathbf{to}_1, .., \mathbf{to}_T \rangle$ to $\hat{s_2} = \langle \mathbf{pc}', e_1, .., e_T, \mathbf{from}'_1, .., \mathbf{from}'_T, \mathbf{to}'_1, .., \mathbf{to}'_T \rangle$ if

1. $\mathbf{pc} = L_1$, i.e., the reference process is in location L_1,
2. one of the following two conditions holds:
   - *If Branch:* $\forall i \in \mathcal{B}^x(\mathscr{G}).\ (e_i = 0)$ and $\mathbf{pc}' = L_2$, i.e., the guard is true and the reference process moves to control state L_2.
   - *Else Branch:* $\neg\forall i \in \mathcal{B}^x(\mathscr{G}).\ (e_i = 0)$ and $\mathbf{pc}' = L_3$, i.e., the guard is false and the reference process moves to control state L_3.
3. all the variables $\mathbf{from}'_1, .., \mathbf{from}'_T$ and $\mathbf{to}'_1, .., \mathbf{to}'_T$ are false, expressing that none of the environment processes changes its state.

Thus, in order to compute the abstract transition we just need to find the blocking set $\mathcal{B}^x(\mathscr{G})$. This task is easy for predicates involving only linear order.

	Inter-preds	Intra-preds	Reachable states	Safety	Liveness
Szymanski	1	8	$O(2^{14})$	0.1s	1.82s
Bakery	3	5	$O(2^{146})$	68.55s	755.0s

Fig. 2. Running Times

6 Experimental Results

We have implemented a prototype of our abstraction method in JAVA. As argued above, our implementation handles protocols in which all the predicates appearing in the guards involve only $\{<, >, =\}$. Thus, in this preliminary implementation, the decision problems that arise during the abstraction are simple and are handled by our abstraction program internally. We verified the safety and liveness properties of Szymanski's mutual exclusion protocol and Lamport's bakery algorithm. These two protocols have an intricate combinatorial structure and have been used widely as benchmarks for parameterized verification. For safety properties, we verified that no two processes can be present in the critical section at the same time. For liveness, we verified the property that if a process wishes to enter the critical section then it eventually will.

We used the NuSMV model checker to verify the finite abstract model. The model checking times are shown in Figure 2. The abstraction time is negligible, less than 0.1s. Figure 2 also shows the number of predicates and the size of the reachable state space as reported by NuSMV. All experiments were run on a 2.4 GHz Pentium machine with 512 MB main memory.

7 Conclusion

We have enriched predicate abstraction by ideas from counter abstraction to develop a new framework for verifying parameterized systems. We have applied this method to verify, under the atomicity assumption, the safety and liveness properties of two well known mutual exclusion protocols.

The main focus of this paper was the verification of small but very intricate systems. In these systems, the challenge is to handle the tightly inter-twined execution of an unbounded number of processes and to maintain predicates which are spanning multiple processes.

At the heart of our approach lies a notion of abstraction – *environment abstraction* – which describes the status of a concurrent system from the point of view of a single process. In addition to safety properties, environment abstraction naturally allows to verify fairness properties as well. The framework presented in this paper is a specific instance of environment abstraction tailored for distributed mutual exclusion algorithms. The general approach can be naturally extended in several ways:

- In this paper, the internal state of a process is described by a control location $\mathsf{pc} = L$. In a more general framework, the state of a process can be described using additional predicates which relate the different data variables of one process. This extension is quite straightforward but omitted from the current paper for the sake of simplicity.

– We have also extended the method to deal with systems in which there is a central process in addition to the K local processes. This extension allows us to handle *directory based cache coherence protocols* and will be reported in future work.
– The most important improvement of our results concerns the *elimination of the atomicity assumption* as to achieve automated protocol verification in a non-simplified setting for the first time. We recently have reached this goal by an extension of environment abstraction. We will report these results in future work.

To conclude, we want to emphasize that viewing a concurrent system from the point of view of a single process closely matches the reasoning involved in designing a distributed algorithm. We therefore believe that environment abstraction naturally yields powerful system abstractions.

Acknowledgments

The authors are grateful to the anonymous referees and Ken McMillan, and Lenore Zuck for discussions and comments which helped to improve the presentation of this paper.

References

1. P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model-checking made simple and efficient. In *Proc. 13th International Conference on Concurrency Theory (CONCUR)*, 2002.
2. K. Apt and D. Kozen. Limits for automatic verification of finite state concurrent systems. *Information Processing Letters*, 15:307–309, 1986.
3. T. Arons, A. Pnueli, S. Ruah, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proc. 13th Intl. Conf. Computer Aided Verification (CAV)*, 2001.
4. T. Ball, S. Chaki, and S. Rajamani. Verification of multi-threaded software libraries. In *ICSE*, 2001.
5. K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S systems to verify parameterized networks. In *Proc. TACAS*, 2000.
6. K. Baukus, Y. Lakhnech, and K. Stahl. Verification of parameterized protocols. In *Journal of Universal of Computer Science*, 2001.
7. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *15th Intern. Conf. on Computer Aided Verification (CAV'03)*. LNCS, Springer-Verlag, 2003.
8. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *12th Intern. Conf. on Computer Aided Verification (CAV'00)*. LNCS, Springer-Verlag, 2000.
9. M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81:13–31, 1989.
10. E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *www.cs.cmu.edu/∼tmurali/vmcai06.ps*.
11. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal model checking. In *Proc. 5th Intl. Conf. Computer Aided Verification (CAV)*, 1993.
12. G. Delzanno. Automated verification of cache coherence protocols. In *Computer Aided Verification 2000 (CAV 00)*, 2000.

13. A. E. Emerson and V. Kahlon. Model checking guarded protocols. In *Eighteenth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 361–370, 2003.
14. E. A. Emerson, J. Havlicek, and R. Trefler. Virtual symmetry. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.
15. E. A. Emerson and A. Sistla. Utilizing symmetry when model-checking under fairness assumptions: An automata theoretic approach. *TOPLAS*, 4, 1997.
16. E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proc. 5th Intl. Conf. Computer Aided Verification (CAV)*, 1993.
17. E. A. Emerson and R. Trefler. From asymmetry to full symmetry. In *CHARME*, 1999.
18. Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with incomprehensible ranking. In *Proc. VMCAI*, 2004.
19. Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. In *Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2004.
20. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39, 1992.
21. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. CAV 97)*, volume 1254, pages 72–83. Springer Verlag, 1997.
22. T. Henzinger, R. Jhala, and R. Majumdar. Race checking with context inference. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 2004.
23. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Proc. CAV'97*, volume 1254 of *LNCS*, pages 424–435. Springer, June 1997.
24. S. K. Lahiri and R. Bryant. Constructing quantified invariants. In *Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2004.
25. S. K. Lahiri and R. Bryant. Indexed predicate discovery for unbounded system verification. In *Proc. 16th Intl. Conf. Computer Aided Verification (CAV)*, 2004.
26. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
27. K. L. McMillan, S. Qadeer, and J. B. Saxe. Induction in compositional model checking. In *Conference on Computer Aided Verfication*, pages 312–327, 2000.
28. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2001.
29. A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$ counter abstraction. In *Computer Aided Verification 2002 (CAV 02)*, 2002.
30. I. Suzuki. Proving properties of a ring of finite state machines. *Information Processing Letters*, 28:213–214, 1988.
31. B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proc International Conference on Supercomputing Systems*, 1988.