Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems*

Edmund Clarke[1], Murali Talupur[2], and Helmut Veith[3,4]

[1] School of Computer Science, Carnegie Mellon University, USA
[2] Intel Strategic CAD Labs, Portland, USA
[3] Fachbereich Informatik, Technische Universität Darmstadt, Germany
[4] Institut für Informatik, Technische Universität München, Germany

Abstract. The parameterized verification of concurrent algorithms and protocols has been addressed by a variety of recent methods. Experience shows that there is a trade-off between techniques which are widely applicable but depend on non-trivial human guidance, and fully automated approaches which are tailored for narrow classes of applications. In this spectrum, we propose a new framework based on environment abstraction which exhibits a large degree of automation and can be easily adjusted to different fields of application. Our approach is based on two insights: First, we argue that natural abstractions for concurrent software are derived from the "Ptolemaic" perspective of a human engineer who focuses on a single reference process. For this class of abstractions, we demonstrate soundness of abstraction under very general assumptions. Second, most protocols in given a class of protocols – for instance, cache coherence protocols and mutual exclusion protocols – can be modeled by small sets of compound statements. These two insights allow to us efficiently build precise abstract models for given protocols which can then be model checked. We demonstrate the power of our method by applying it to various well known classes of protocols.

1 Introduction

In many areas of system engineering, distributed concurrent computation has become an essential design principle. For instance, the controllers on an automobile have to be necessarily distributed. Further, in fundamental areas like chip design, distributed computation often offers the best way to increased performance. Protocols like cache coherence protocols, mutual exclusion protocols, synchronization protocols form the bedrock on which these distributed systems are built. Experience has shown however that designing such protocols correctly is a non-trivial task for human engineers and should be supported by computer-aided verification methods. Although non-rigorous verification techniques such as testing are very effective in finding many obvious errors, they cannot explore all interleaving behaviors, and may miss subtle errors. Consequently,

* This research was sponsored by the Gigascale Systems Research Center (GSRC), Semiconductor Research Corporation (SRC), the National Science Foundation (NSF), the Office of Naval Research (ONR), the Naval Research Laboratory (NRL), the Defense Advanced Research P rojects Agency, the Army Research Office (ARO), and the General Motors Collaborative Research Lab at CMU, and the Deutsche Forschungsgemeinschaft (DFG) under grant FORTAS.

C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963, pp. 33–47, 2008.

rigorous formal verification techniques are indispensable in ensuring the correctness of such protocols.

Important classes of distributed protocols are designed *parametrically*, i.e., for an unlimited number of concurrent processes. For example, cache coherence protocols are designed to be correct independently of the exact number of caches. Verifying a protocol parametrically is however difficult, and is known to be undecidable [24]. Nonetheless, parameterized verification has received considerable attention in the recent years. Parameterized verification of cache coherence protocols is a pressing problem for the hardware industry and has been considered by [11,17,6,4,12]. Another important class of protocols that has been widely studied is mutual exclusion protocols [18,14,2,20].

The approaches by McMillan [17], Chou et al. [6], which have been successfully applied to industrial-strength cache coherence protocols require significant human guidance during verification. On the other hand, researchers have not been able to apply largely automatic methods like the ones by Lahiri et al [14] and Pnueli et al [20,2] to large protocols. Thus, while the ideal is to have a single automatic method to handle the whole class of real life protocols, it has come to be accepted that any practically useful tool will involve some human intervention. The goal then is to minimize the amount of effort and ingenuity required to guide a verification tool successfully. In this paper, we are proposing a framework that addresses this issue.

Our method is built around two insights which we describe in the following subsections: (1) humans tend to reason about distributed systems from the "Ptolemaic" viewpoint of an individual process, and (2) natural classes of protocols can be captured by a small number of compound statements. Combined, these two insights lead to an abstraction framework which accounts for the specifics of distributed systems and can be easily adjusted to different classes of protocols.

Ptolemaic System Analysis. The success of the Ptolemaic system (where earth is the center of the cosmos) over many centuries reveals an innate reasoning principle which the human mind applies to complex systems: we tend to imagine complex systems with the human observer in the center. Although this Ptolemaic intuition is wrong for many systems we encounter in nature, it is naturally built into the systems we *construct.*

Let us look more closely at the case of concurrent systems. During the construction of such a system, the programmer arguably imagines him/herself in the position of one *reference process*, around which the other processes – which constitute *the environment* – evolve. In fact, we usually consider a program to be well written when its correctness can be intuitively understood from the Ptolemaic viewpoint of a single process. Thus, an abstract model that reflects the viewpoint of a reference process is likely to contain sufficient information for asserting system correctness. The goal of environment abstraction is to put this intuition into a formal and practically useful framework.

Our concrete models are concurrent parameterized systems, where the number of processes is the parameter, and all processes execute the same program. We write $P(K)$ to denote a system with $K > 1$ processes. Thus, the formal verification question is

$$\forall K > 1.P(K) \models \forall x.\varphi(x)$$

where $\forall x\varphi(x)$ is an indexed temporal logic specification [5].

Each abstract state in our framework can be described by a formula $\Delta(x)$ where x stands for the process chosen to act as the reference process. The abstract state $\Delta(x)$ will contain (i) a detailed description of the internal state of process x and (ii) a concise abstract description of x's *environment* consisting of other processes. The *abstract transition relation* is defined by a new form of existential abstraction which quantifies over the parameter K and the variable x: If some concrete system $P(K)$ has a process p and a transition from state s_1 to s_2 such that $\Delta_1(p)$ holds in state s_1, and $\Delta_2(p)$ hols in state s_2, then we include a transition from $\Delta_1(x)$ to $\Delta_2(x)$ in the abstract model. Thus, every abstract transition is induced by a concrete transition in some concrete model $P(K)$. Note that $\Delta_1(x)$ and $\Delta_2(x)$ have to satisfy the same process p before and after the transition, i.e., the Ptolemaic reference point does not change during a transition.

The main mathematical contribution of this paper is a soundness result which shows that for a suitably chosen language of descriptions $\Delta(x)$, environment abstraction preserves universally quantified indexed temporal logic specifications, see Section 4. The requirements for choosing the $\Delta(x)$ are quite general: first, each concrete situation has to be covered by at least one $\Delta(x)$ (*coverage*), and second, the $\Delta(x)$ have to be sufficiently expressive to imply truth or falsity of atomic specifications (*completeness*). Our soundness result naturally carries over to the case of multiple reference processes.

While this definition of the abstract model reflects our intuition about distributed system design and ensures soundness of our approach, it is clearly not operational. Since the parameter K is unbounded, it is often not possible to compute the abstract transition relation exactly. It is here that our second insight comes into play.

Abstraction Templates for Compound Statements. The communication and coordination mechanisms between the processes in a distributed system are usually confined to a few basic patterns characteristic for each system type. Thus, when we focus on a particular class of protocols like cache coherence protocols or mutual exclusion protocols, the protocols in that class can be described in terms of a small number of *compound* transactions or statements. For example, to describe cache coherence protocols we need at most six compound statements [25], to describe mutex protocols we need only two statements [8,25], and to describe semaphore based algorithms, we just need a single statement, cf. Sections 4 and 5.

This insight allows us to approximate the abstract transition relation for a given parameterized system in an efficient manner. We know that all transitions of the system fall under a few compound statements. From the general construction principle for Ptolemaic environment abstraction we also know the structure of the abstract domain. Thus, *for each of these compound statements, we can provide an abstraction template*. Technically, we describe this abstraction template in terms of an abstract transition invariant, i.e., a formula expressing the relationship between the variables for the current abstract state and the next abstract state. Note that this invariant is given in a generic fashion, independently of the protocol in which it is used. For each concrete statement, we just have to plug in the specific parameters of that transition into the template invariant. Thus, the template invariants have to be written only once for each statement type. Since there are only a small number of compound statements for each class of protocols, writing the abstract template invariants is usually easy.

Tool Flow of the Environment Abstraction Framework . Once the compound statements for a protocol class are integrated in the framework, the tool flow is as follows:

1. The *user* describes $P(K)$ as a program in terms of the compound statements.
2. The *user* writes an indexed specification $\forall x.\Phi(x)$.
3. The *abstraction tool* computes the abstract model $P^{\mathcal{A}}$ from the protocol description. In our prototype implementation, this abstract model is an *SMV* model.
4. A *model checker* verifies $P^{\mathcal{A}} \models \varphi(x)$. Note that in $P^{\mathcal{A}}$, x is interpreted as the reference process. If $P^{\mathcal{A}} \models \varphi(x)$, then $\forall x.\varphi(x)$ holds for all $P(K), K > 1$. Otherwise, the model checker outputs an abstract counterexample for further analysis.

Structure of the Paper. In Section 3, we describe the environment abstraction framework in a rigorous and general way. In Section 4, we apply environment abstraction to the semaphore based mutual exclusion algorithms by Courtois et al. [10]; these protocols were posed to us as a challenge problem by Peter O'Hearn. In Section 5 we survey our experiences with other classes of protocols.

2 Related Work

In previous work, we used a specific instance of environment abstraction for the verification of the Bakery procotol and Szymanski's algorithm [8]. Although our paper [8] contains several seminal ideas about environment abstraction, it is very different in scope and generality. In particular, the methods in [8] are tailored towards a specific application and a hardwired set of specifications, without a general soundness result.

The method of counter abstraction [20] inspired our approach, and can be seen as a specific, but limited form of environment abstraction. Invisible invariants [19,2] provide another novel method for verifying parameterized systems. Both these methods are restricted to systems without unbounded integer variables.

The Indexed Predicates method [14,15] is similar to predicate abstraction with the crucial difference that predicates can contain free index variables (variables that range over process indices). These indexed predicates are used to construct complex (universally) quantified invariants for parameterized systems. The abstract descriptions used in our abstraction are Indexed Predicates in that they contain free index variables. But the similarity ends there. While we build an abstract transition relation over these descriptions, in the Indexed Predicates method they don't have an abstract transition relation. They only have an abstract reachability relation, which specifies what set of abstract states can one reach starting from another set of abstract states.

The series of papers [16,17,18,6] by McMillan and Chou et al. introduced an important and successful approach for parameterized verification. In this approach, which is based on circular compositional reasoning, a model checker is used as a proof assistant to carry out parameterized verification. The user however has the burden of coming up with *non-interference* lemmas [17] which can be non-trivial and require a deep understanding of the protocol under verification.

The TVLA method by Reps et al. [21] is a widely applicable abstract interpretation based approach for shape analysis, and also for verification of safety properties of multi-threaded systems, cf. Yahav's method [26]. TVLA's canonical abstraction is a

generalization of predicate abstraction similar in spirit to the description formulas in environment abstraction. To make verification of unbounded systems possible, TLVA uses summarization, which is related to the idea of counting abstraction. Special predicates called *instrumentation* predicates are used to focus on particular processes in detail.

In recent work, [13] proposes a method to find network invariants using finite automata learning algorithms due to Angluin and Beirmann. The paper [9] uses a new completion procedure to strengthen split invariants. While parameterized verification is not their primary aim, the method is able to produce parameterized proofs for protocols like the Bakery protocol. Other classical approaches to parameterized verification include regular model checking [1] and the WS1S based method [3]. Early work on parameterized verification was done by Clarke et al [5].

3 A Generic Framework for Environment Abstraction

3.1 System Model

We consider parameterized concurrent systems $P(K)$, where the parameter $K \geq 2$ denotes the number of replicated processes. The processes are distinguished by unique indices in $\{1, \ldots, K\}$ which serve as process id's. Each process executes the same program which has access to its process *id*. We do not make any specific assumptions about the processes, in particular we do not require them to be finite state processes.

Consider a system $P(K)$ with a set S_K of states. Each state $s \in S_K$ contains the entire state information for each of the K concurrent processes, i.e., s is a vector $\langle s_1, \ldots, s_K \rangle$. Technically, $P(K)$ is a Kripke structure (S_K, I_K, R_K, L_K) where I_K is the set of initial states and R_K is the transition relation. We will discuss the labeling L_K for the states in S_K below.

It is easy to extend our framework to parameterized systems which contain one or several *non-replicated processes* in addition. In this case, the states s will be vectors $\langle s_1, \ldots, s_K, t_1, \ldots, t_{\text{const}} \rangle$ where the t_i are the states of the non-replicated processes. In the following exposition, we will for simplicity omit this easy extension.

3.2 Ptolemaic Specifications

The change of focus brought upon by environment abstraction most visibly affects the specification language: We use a variation of indexed ACTL* where the atomic formulas are able to express not only properties of individual processes, but also properties of processes in the environment. In our practical examples, the following two atomic formulas (where c is a constant value) have been most relevant:

Formula	*Meaning*
$\mathsf{pc}[x] = c$	**the program counter of process** x **has value** c
$c \in \mathsf{env(x)}$	**there is a process** $y \neq x$ **with program counter value** c "The environment of x contains a process with program counter value c."

In this language, we can specify mutual exclusion

$$\forall x.\ \mathbf{AG}\ (pc[x] = 5 \rightarrow \neg(5 \in env(x)))$$

and many other important properties with a single quantifier $\forall x$ that ranges over the processes in the system. Intuitively, this is the reason why a single reference process in the abstract model is able to assert correctness of the specification. Below we will discuss what properties are expressible with a single quantifier.

3.3 Environment Abstraction

Our examples of atomic formulas motivate the construction of the abstract model: At each state, we must be able to assert the truth or falsity of the atomic propositions $pc[x] = c$ and $c \in env(x)$. Consequently, the expressions $pc[x] = c$ and $c \in env(x)$ are used as *labels* in the abstract model. We will write L to denote the finite set of atomic formulas, and will call them *labels* further on. Note that L can contain formulas different from the two examples mentioned above.

The *states of the abstract model are formulas* $\Delta(x)$ (called "descriptions") which describe properties of process x and its environment. Similar to the atomic labels, the $\Delta(x)$ also have a free variable referring to the reference process. In contrast to the atomic labels, however, the descriptions will usually be relatively large and intricate formulas which give a quite detailed picture of the global system state from the point of view of reference process x. Intuitively, an abstract state $\Delta(x)$ represents all concrete system states where some process p satisfies $\Delta(p)$. In our running example, the simplest natural choice for the abstract states are descriptions of the form

$$pc[x] = c \ \wedge \ (\bigwedge_{i \in A} \underbrace{\exists y \neq x . pc[y] = i)}_{i \in env(x)} \wedge (\bigwedge_{i \in B} \underbrace{\neg \exists y \neq x . pc[y] = i)}_{\neg (i \in env(x))}$$

where c is a program counter position, and $A \dot{\cup} B$ is a partition of all program counter positions. (Note that this simple base case is a form of counter abstraction; the descriptions we use in applications are often much richer – depending on the complexity of the problem.) Intuitively, this description says that ***"the reference process*** x ***is in program counter location*** c***, and the set of program counter locations of the other processes in the system is*** A***"***. Since these formulas all belong to a simple syntactic class, it is easy to identify $\Delta(x)$ with a tuple, as usually in predicate abstraction, for instance $\langle c, A, B \rangle$. In the logical framework of this section, it is more natural to view descriptions as formulas. In the applications, however, we will usually prefer an appropriate tuple notation.

In the rest of this section, we will assume that we have a fixed *finite* set of descriptions D which constitute the abstract state space.

Soundness Requirements for Labels and Descriptions. Given a label or description $\varphi(x)$, we write $s \models \varphi(c)$ to express that in state s, process c has property φ. We next describe two requirements on the set D of descriptions and the set L of labels to make them useful as building blocks for the abstract model.

1. **Coverage.** For each system $P(K)$, each state s in S_K and each process c there is some description $\Delta(x) \in D$ which describes the properties of c, i.e.,

   $$s \models \Delta(c).$$

   In other words, every concrete situation is reflected by some abstract state.

2. **Completeness.** For each description $\Delta(x) \in D$ and each label $l(x) \in L$ it holds that either
$$\Delta(x) \to l(x) \qquad \text{or} \qquad \Delta(x) \to \neg l(x).$$
In other words, the descriptions in D contain enough information about a process to conclude whether a label holds true for this process or not. The completeness property enables us to give natural labels to each state of the abstract system: An abstract state $\Delta(x)$ has label $l(x)$ if $\Delta(x) \to l(x)$.

Description of the Abstract System P^A. Given two sets D and L of descriptions and labels which satisfy the two criteria *coverage* and *completeness*, the abstract system P^A is a Kripke structure
$$\langle D, I^A, R^A, L^A \rangle$$
where each $\Delta(x) \in D$ has a label $l(x) \in L$ if $\Delta(x) \to l(x)$, i.e., $L^A(\Delta(x)) = \{l(x) : \Delta(x) \to l(c)\}$. Before we describe I^A and R^A, we state the following lemma about preservation of labels which motivates our definition of the abstraction function below:

Lemma 1. *Suppose that $s \models \Delta(c)$. Then the concrete state s has label $l(c)$ iff the abstract state $\Delta(x)$ has label $l(x)$.*

Definition 1. *Given a concrete state s and a process c, the abstraction of s with reference process c is given by the set $\alpha_c(s) = \{\Delta(x) \in D : s \models \Delta(c)\}$.*

Remark 1. (i) The coverage requirement guarantees that $\alpha_c(s)$ is always non-empty. (ii) If the $\Delta(x)$ are mutually exclusive, then $\alpha_c(s)$ always contains exactly one description $\Delta(x)$. (iii) Two processes c, d of the same state s will in general give rise to different abstractions, i.e., $\alpha_c(s) = \alpha_d(s)$ is, in general, not true.
Now we define the *transition relation* of the abstract system by a variation of existential abstraction: R^A contains a transition between $\Delta_1(x)$ and $\Delta_2(x)$ if there exist a concrete system $P(K)$, two states s_1, s_2 and a process r such that

1. $\Delta_1(x) \in \alpha_r(s_1)$ [or, equivalently, $s_1 \models \Delta_1(r)$]
2. $\Delta_2(x) \in \alpha_r(s_2)$ [or, equivalently ,$s_2 \models \Delta_2(r)$]
3. there is a transition from s_1 to s_2 in $P(K)$, i.e., $(s_1, s_2) \in R_K$.

We note three important properties of this definition:

(a) We existentially quantify over K, s_1, s_2, and r. This is different from standard existential abstraction where we only quantify over s_1, s_2. For fixed K and r, our definition is essentially equivalent to existential abstraction. The only difference is the obvious change in the labels: the concrete structure has labels of the form $l(c)$, while the abstract structure has labels of the form $l(x)$.
(b) Since $\Delta_1(x) \in \alpha_r(s_1)$ and $\Delta_2(x) \in \alpha_r(s_2)$, both abstractions Δ_1 and Δ_2 use the same process r. Thus, the Ptolemaic viewpoint of the reference process is not changed in the transition.
(c) The process which is active in the transition from s_1 to s_2 can be any process in $P(K)$, it does not have to be r.

Finally, the set I^A of abstract initial states is the union of the abstractions of concrete states, i.e., $\Delta(x) \in I^A$ if there exists a system $P(K)$ with state $s \in I_K$ and process r such that $\Delta(x) \in \alpha_r(s)$.

For environment abstractions that satisfy coverage and completeness we have the following general soundness theorem.

Theorem 1 (Soundness of Environment Abstraction). *Let $P(K)$ be a parameterized system and P^A be its abstraction as described above. Then for single indexed $ACTL^\star$ specifications $\forall x.\varphi(x)$, the following holds:*

$$P^A \models \varphi(x) \quad \text{implies} \quad \forall K.P(K) \models \forall x.\varphi(x).$$

The reader is referred to the full version of this paper [7] for a proof; the full version also contains the formal generalization of environment abstraction to multiple reference processes.

3.4 Trade-Off between Expressivity of Labels and Number of Index Variables

In this section, we discuss how a well-chosen set of labels L often makes it possible to use a *single* index variable. The Ptolemaic system view explains why we seldom find more than *two* indices in practical specifications: When we specify a system, we tend to track properties *our* process has in relation to other processes in the system, one at a time. Thus, double-indexed specifications of the form $\forall x \neq y.\varphi(x, y)$ often suffice to express the specifications of interest. Properties involving *three* or more processes at a time are rare, as they consider triangles of processes and their relationships. (Note however that our method can, in principle, handle an arbitrary number of index variables, cf. [7].) Let us return to our example specification. Mathematically, we can write this formula in three ways:

(1) $\forall x, y.x \neq y \rightarrow \mathbf{AG}\ (pc[x] = 5) \rightarrow (pc[y] \neq 5)$
(2) $\forall x.\mathbf{AG}\ (pc[x] = 5) \rightarrow \neg(\exists y \neq x.pc[y] = 5)$
(3) $\forall x.\mathbf{AG}\ (pc[x] = 5) \rightarrow \neg(5 \in env(x))$

Going from (1) to (3) we see that the universal quantifier is *distributed* over $\mathbf{AG}$ and *hidden* inside the label $5 \in env(x)$. The Ptolemaic viewpoint again explains why such situations are likely to happen: In many specifications, we consider *our* process along the time axis, but only at each individual time point, we evaluate its relationship to other processes; thus, a *quantification scope inside the temporal operator* suffices.

Formally, it is easy to see that the translation from (1) to (3) depends on the distributivity of conjunction over $\mathbf{AG}(\alpha \rightarrow \beta)$ with respect to β, i.e., $\mathbf{AG}(\alpha \rightarrow (\varphi \wedge \psi))$ is equivalent to $\mathbf{AG}(\alpha \rightarrow \varphi) \wedge \mathbf{AG}(\alpha \rightarrow \psi)$. We give a syntactic characterization of formulas with this property in [7]. Our characterization relies on previous work [22,23] in the context of temporal logic query languages.

4 Verification of the Reader and Writer Algorithms

In this section we apply our framework to two classical semaphore based distributed algorithms by Courtois et al. [10]. The algorithms ensure mutual exclusion of multiple concurrent *readers* and *writers*. To our knowledge, these algorithms – which were posed as challenge problems to us by Peter O'Hearn – have not been verified parametrically. Figure 1 shows the code for a reader process in the simpler of the two algorithms.

L1: P(mutex)
L2: readcount := readcount + 1
L3: if readcount = 1 then P(w)
L4: V(mutex)
L5: * reading is performed *****
L6: P(mutex)
L7: readcount := readcount - 1
L8: if readcount = 0 then V(w)
L9: V(mutex)

Fig. 1. The Reader Algorithm

We first give a single compound statement that suffices to describe the semaphore based algorithms. Then we introduce an appropriate abstract template invariant, and show how to verify the two algorithms in practice.This example should illustrate all the ingredients that go into our method and demonstrate the ease of application.

Compound Statement for Semaphore Based Algorithms. A semaphore is a low-level hardware or OS construct for ensuring mutual exclusion. By design, a semaphore variable can be accessed by only one process at any given time. The basic operations on a semaphore variable w are $P(w)$, which *acquires* the semaphore, and $R(w)$, which *releases* the semaphore.

We model a semaphore w as a boolean variable b_w that can be accessed by all processes. The acquire and release actions $P(w)$ and $R(w)$ are modeled by setting b_w to 1 and 0 respectively. A semaphore based algorithm has N identical local processes corresponding to the readers and writers. Readers and writers do not have the same code but we can create a union of the two syntactically to obtain a single larger process with two possible start states; depending on which state is chosen as the start state the compound process either acts as a reader or as a writer. The state space of each local process is finite. Instead of having multiple local variables, we will assume for simplicity there is only one local variable pc per process.

In addition to the local processes there is one central process C. The central process essentially consists of the shared variables, including the boolean variables used to model the semaphores. As with the local processes, we roll up all the variables of the central process into a single variable st_{cen} for the sake of simplicity. Note that st_{cen} can be an unbounded variable. We will denote the parameterized system by $P(N)$.

The reader and writer algorithms of [10] have three different types of transitions:

1. A simple transition by a local process. For example, the transition at $L5$ in Figure 1.
2. A local transition conditioned on acquiring or releasing a semaphore, e.g. $L1$, $L4$.
3. A transition in which a process modifies a shared variable, e.g., $L2$, $L7$.

All three types of transitions can be guarded by a condition on the central variables. The three types of transitions can be modeled using the compound statement

$$pc = L_1 : \textbf{if } st_{\text{cen}} = C_1 \ \ \textbf{then goto } st_{\text{cen}} = f(st_{\text{cen}}) \wedge pc = L_2 \\ \textbf{else goto } st_{\text{cen}} = g(st_{\text{cen}}) \wedge pc = L_3.$$

The semantics of this statement is intuitive: if the local process is in control location L_1, it checks if the *central process* is in state C_1. In this case, it modifies the central process to a new state $f(st_{\text{cen}})$ (where f is a function, see below) and goes to L_2. Otherwise, the central process is modified to $g(st_{\text{cen}})$ and the local process goes to L_3.

In the semaphore algorithms we consider, the functions f, g are simple linear functions. For instance, in the transition $L2 : readcount := readcount + 1$ of Figure 1 the new value for the central variable $readcount$ is a linear function of the previous value.

In the longer version of this paper [7] we present the two algorithms from [10] in our input language. For example, the semaphore acquire action at $L1$ in Figure 1 can be modelled as

$$pc = L_1 : \textbf{ if } b_{mutex} = 0 \textbf{ then goto } b_{mutex} = 1 \wedge pcl = L_2 \textbf{ else goto } pc = L_1$$

Abstract Domain. Our description formulas $\Delta(x)$ are very similar to the example of Section 3, except for an additional conjunct Δ_{cen}:

$$pc[x] = \textbf{pc} \ \wedge \ (\bigwedge_{i \in A} \exists y \neq x.pc[y] = i) \ \wedge \ (\bigwedge_{i \in B} \neg \exists y \neq x.pc[y] = i) \ \wedge \ \Delta_{\text{cen}}$$

Here, Δ_{cen} is a predicate which describes properties of the central process, analogous to classical predicate abstraction. Since the central process does not depend on the reference process x, the formula Δ_{cen} does not contain the free variable x.

The structure of Δ_{cen} is automatically extracted from the program code. For instance, for the program of Figure 1, Δ_{cen} describes the semaphore variables $w, mutex$ and the two predicates $readcount = 0$ and $readcount = 1$. Thus, Δ_{cen} has the form

$$[\neg]w \ \wedge \ [\neg]mutex \ \wedge \ [\neg](readcount = 0) \ \wedge \ [\neg](readcount = 1).$$

Here, $[\neg]$ stands for a possibly negated subformula. We write D_{cen} to denote the set of all these Δ_{cen} formulas; in our example, D_{cen} has $2^4 = 16$ elements.

As argued above, it is more convenient in the applications to describe an abstract state $\Delta(x)$ as a tuple. Specifically, we will use the tuple

$$\langle \textbf{pc}, e_1, \ldots, e_n, \Delta_{\text{cen}} \rangle$$

to describe an abstract state $\Delta(x)$. Intuitively, **pc** refers to the control location of the reference process, and Δ_{cen} is the predicate abstraction for the central process. The bits e_i describe the presence of an environment process in control location i, i.e., e_i is 1 if $i \in A$. (Equivalently, e_i is 1 if $\Delta(x) \rightarrow i \in env(x)$.)

We note that the abstract descriptions $\Delta(x)$ and the corresponding tuples can be constructed automatically and syntactically from the protocol code. Since our labels of interest are of the form $pc[x] = c$ and $c \in env(x)$, it is easy to see that the *coverage* and *completeness* properties are satisfied by construction.

Abstraction Template Invariants. To describe the abstract template invariant, we will consider two cases: (i) the executing process is the reference process and (ii) the executing process is an environment process. For both cases, we will describe a suitable abstract invariant, and take their disjunction. Recall the general form

$$\begin{aligned} pc = L_1 : \textbf{if } st_{\text{cen}} = C_1 \ & \textbf{ then goto } st_{\text{cen}} = f(st_{\text{cen}}) \wedge pc = L_2 \\ & \textbf{ else goto } st_{\text{cen}} = g(st_{\text{cen}}) \wedge pc = L_3. \end{aligned}$$

of the compound statement. We will give an invariant for the abstract transition

$$\langle \textbf{pc}, e_1, \ldots, e_n, \Delta_{\text{cen}} \rangle \qquad \text{to} \qquad \langle \textbf{pc}', e'_1, \ldots, e'_n, \Delta'_{\text{cen}} \rangle$$

Case 1: Reference Process Executing. The invariant I_{ref} in this case is

$$\mathbf{pc} = L_1 \wedge \tag{1}$$

$$\big[(C_1 \models \Delta_{\text{cen}} \wedge \Delta'_{\text{cen}} \in \mathbf{f}(\Delta_{\text{cen}}) \wedge \mathbf{pc}' = L_2) \bigvee \tag{2}$$

$$(C_1 \not\models \Delta_{\text{cen}}) \wedge \Delta'_{\text{cen}} \in \mathbf{g}(\Delta_{\text{cen}}) \wedge \mathbf{pc}' = L_3)\big] \tag{3}$$

Condition (1) says that the reference process is at control location L_1. Condition (2) corresponds to the **then** branch: it says that the central process is approximated to be in state C_1; in the next state, the reference process is in control location L_2 and the new approximation of the central process is non-deterministically picked from the set $\mathbf{f}(\Delta_{\text{cen}})$. As usually in predicate abstraction, $\mathbf{f}$ is an over-approximation of function f:

$$\mathbf{f}(\Delta_{\text{cen}}) = \{\Delta'_{\text{cen}} \in D_{\text{cen}} \mid \exists st_{\text{cen}}.\ st_{\text{cen}} \models \Delta_{\text{cen}} \text{ and } f(st_{\text{cen}}) \models \Delta'_{\text{cen}}\}$$

Usually the operations on variables in a protocol are not more complicated than simple linear operations; consequently, the predicates involved in our environment abstraction are simple, too. Therefore, computing $\mathbf{f}$ is trivial with standard decision procedures.

Condition (3), which corresponds to the **else** branch, is similar to condition (2).

Case 2: Environment Process Executing. The invariant I_{env} in this case is

$$e_{L_1} = 1 \wedge \tag{4}$$

$$\big[(C_1 \models \Delta_{\text{cen}} \wedge \Delta'_{\text{cen}} \in \mathbf{f}(\Delta_{\text{cen}}) \wedge e'_{L_2} = 1) \bigvee \tag{5}$$

$$(C_1 \not\models \Delta_{\text{cen}}) \wedge \Delta'_{\text{cen}} \in \mathbf{g}(\Delta_{\text{cen}}) \wedge e'_{L_3} = 1)\big] \tag{6}$$

Condition (4) says that some environment process is in control location L_1). Condition (5) is similar to Condition (2) of *Case 1* above, with the exception that $e'_{L_2} = 1$ forces a process in the environment to go to location L_2. Condition (6) is analogous to (5).

The invariant I for the compound statement is the disjunction $I = I_{\text{ref}} \vee I_{\text{env}}$ of the invariants in the two cases. Given a protocol with compound statements $cs_1, \ldots, cs_m$ we first find invariants $I(cs_1), \ldots, I(cs_m)$ by plugging in the concrete parameters of each statement into the template invariant I. The disjunction of these individual invariants gives us the abstract transition relation.

We denote the abstract system obtained from the template invariant as $P^{\mathscr{A}}$ and the abstract system obtained from the definition of environment abstraction by $P^{\mathcal{A}}$. Our construction is a natural over-approximation of $P^{\mathcal{A}}$:

Fact 1. *Every state transition from $\Delta(x)$ to $\Delta'(x)$ in $P^{\mathcal{A}}$ also occurs in $P^{\mathscr{A}}$.*

Practical Application. For our experiments, we already had a prototype implementation of environment abstraction to deal with cache coherence and mutual exclusion protocols. We added new procedures to allow our tool to read in protocols using the new compound statement and to perform automatic abstraction of the protocol, as described in the previous section.

The procedure to compute next values for Δ_{cen}, i.e., $\mathbf{f}(\Delta_{\text{cen}})$, was handled by an internal decision procedure. (This is a carry over from our previous work with environment abstraction. In hindsight, calling an external decision procedure is a better option).

Our tool, written in Java, takes less than a second to find the abstract models given the concrete protocol descriptions. We use Cadence SMV to verify the abstract model.

For both algorithms in [10], we verified the safety property

$$\forall x \neq y. \mathbf{AG}\big(pc[x] \in \{LR, LW\} \rightarrow \neg LW \in env(x)\big)$$

where LR and LW are the program locations for reading and writing respectively.

Our first attempt to verify the protocol produced a spurious counterexample. To understand the reason for this counterexample, consider the protocol shown in Figure 1. Each time a reader enters the section between lines $L3$ and $L7$, $readcount$ is incremented. When a reader exits the section, $readcount$ is decremented. The semaphore w, which controls a writer's access to the critical section, is released only when $readcount = 0$ and this happens only when no reader is between lines $L3$ and $L7$. Our abstract model tracks only the predicate $readcount = 0$. The decrement operation on readcount in line $L7$ is abstracted to a non-deterministic choice over $\{0, 1\}$ for the value of the predicate $(readcount = 0)$. Thus, the predicate can become true (i.e., take value 1) even when there are readers between lines $L3$ and $L7$ and this leads to the spurious counter example. To eliminate this spurious counterexample we make use of the invariant

$$pc[x] \in [L3..L7] \rightarrow readcount \neq 0$$

This invariants essentially says that for a process between lines $L3$ and $L7$, $readcount$ has to be non-zero. We abstract this invariant into two invariants

$$\mathbf{pc} \in [L3..L7] \rightarrow \neg(readcount = 0) \quad \text{and} \quad \Big(\bigvee_{L \in [L3..L7]} .e_L\Big) \rightarrow \neg(readcount = 0).$$

for the reference process and the environment respectively. Constraining the abstract model with these two invariants, we are able to prove the safety property. The model checking time is less than a minute for both semaphore algorithms.

There still remains an important question: *How do we know that the invariant added to the abstract model is true?* First, we note that the invariant is a local invariant in that it refers only to one process and it is quite easy to convince ourselves that it holds. To prove formally that the invariant holds, we proceed as follows: Running the model checker on the *original abstract model* establishes $\mathbf{pc} \in [L3..L7]\neg(\rightarrow readcount = 0)$. From Theorem 1 we can conclude that $\forall x.pc[x] \in [L3..L7] \rightarrow readcount \neq 0$, and thus we are justified in using this invariant as assumption in proving the safety property. Note that this approach is close in spirit to adding *non-interference* lemmas, as described by McMillan and Chou et al. [17,6].

5 Survey of Other Environment Abstraction Applications

In this section, we survey other, more involved applications of the environment abstraction principle. For a more detailed discussion of these applications, we refer the reader to Talupur's thesis [25], and our predecessor paper [8].

Mutual Exclusion Protocols. In [8], we have shown how to verify mutual exclusion protocols such as the Bakery protocol and Szymanski's algorithm. We need two compound statements which are more complex than in Section 4:

Guarded Transition
$\mathsf{pc} = L_1$: **if** $\forall \mathsf{otr} \neq x.\mathcal{G}(x, \mathsf{otr})$ **then goto** $\mathsf{pc} = L_2$ **else goto** $\mathsf{pc} = L_3$
Semantics: In control location L_1, the process evaluates the guard and changes to control location L_2 or L_3 accordingly.

Update Transition
$\mathsf{pc} = L_1$: **for all** $\mathsf{otr} \neq x$ **if** $\mathcal{T}(x, \mathsf{otr})$ **then** $u_k := \varphi(\mathsf{otr})$ **goto** $\mathsf{pc} = L_2$
Semantics: At location L_1, the process scans over all other processes otr to check if formula $\mathcal{T}(x, \mathsf{otr})$ is true. In this case, the process changes the value of its data variable u_k according to $u_k := \varphi(\mathsf{otr})$. Finally, the process changes to location L_2.

The abstract domain is also more complex, because each process can have unbounded data variables. To account for these variables, the $\Delta(x)$ include *inter-predicates* $IP_j(x, y)$, i.e., predicates that span multiple processes. Thus, the $\Delta(x)$ have the form

$$pc[x] = c \quad \wedge \bigwedge_{(i,j)\in A} \exists y \neq x.pc[y] = i \wedge IP_j(x, y) \wedge \bigwedge_{(i,j)\in B} \neg\exists y \neq x.pc[y] = i \wedge IP_j(x, y)$$

for suitable A and B. The inter-predicates are automatically picked from the program code. For example, a typical inter-predicate for Bakery is $t[x] > t[y]$, which says that the ticket variable of process x is greater than the ticket variable of process y.

The abstraction templates for this language are quite involved, providing a quite precise abstract semantics which is necessary for this protocol class. While [8] assumed that the compound statements are atomic, we later improved the abstraction to verify the mutex property of Bakery without this assumption. We defer a full discussion of these results to a future publication, and refer the reader to [25].

Cache Coherence Protocols. For cache coherence protocols we require six compound statements. Like semaphore based protocols, cache coherence systems also have a central process. The replicated processes, i.e., the caches, have very simple behaviors, and essentially move from one control location to another. This is modeled by the trivial *local transition* $pc = L_1$: **goto** $pc = L_2$. Unlike semaphore based protocols, the directory (central process) can exhibit complex behaviors, as it has pointer variables and set variables referring to caches. The *compound statement for the directory* has the form

$$guard: \ \textbf{do actions}\ A_1, A_2, .., A_k$$

where $A_1, \ldots, A_k$ are *basic actions* and *guard* is a condition on the directory's control location and its pointer and set variables. The basic actions comprise *goto*, *assign*, *add*, *remove*, *pick* and *remote* actions, cf. [7].

The descriptions $\Delta(x)$ used for cache coherence are similar to those of Section 4, but owing to the complexity of the directory process, Δ_{cen} is more elaborate than in the semaphore case. We have used this framework to verify the coherence property of several versions of German's protocol and a simplified version of the Flash protocol [25].

Our experiments with the original Flash protocol showed that the abstract model can become very large. The reason is the high precision of the abstract domain based on *all* control conditions from the protocol code. There is a promising approach to alleviate this problem: instead of building the best possible abstract model we build a coarser model which we refine using the circular reasoning scheme of [16,17,6]. Such a hybrid approach combines the strengths of our approach and the circular reasoning approach.

6 Conclusion

Environment abstraction provides a uniform platform for different types of parameterized systems. To adjust our tool to a new class of protocols, we have to identify the compound statements for that class, and specify the actions of compound statements in terms of abstraction templates. This task requires ingenuity, but is a one time task. Once a 'library' for a class of protocols is built, it can be used to verify any protocol in the class automatically or with minimum user guidance.

Let us address some common questions we have encountered.

Human involvement present in too many places ? The end user who applies our tool to a specific protocol can be different from the verification engineer who builds the library. To verify a protocol, the user has to know only the compound statements; providing the abstract template invariants is the task of the tool builder.

Compound statements too complex ? The compound statements try to pack as many basic patterns as possible in a single statement and thus can be complex. But it is easy to create familiar looking syntactic sugar for often used instances of the compound statements.

Correctness of the abstraction templates ? This question is not much different from asking if a source code model checker is correct. It is easier to convince ourselves about the correctness of a small number of declarative transition invariants than to reason about a huge piece of software. In future work, we plan to investigate formal methods to ensure correctness of the abstraction.

Abstraction refinement ? There are many ways of refining our abstract model. In particular, we can (i) enrich the environment predicates to count the number of processes in a certain environment, (ii) increase the number of reference processes, and (iii) enrich the $\Delta(x)$ descriptions by additional predicates. This is a natural part of our future work.

In conclusion, the environment abstraction framework works well for a variety of protocols by striking what we believe is the right balance between automation and class specific reasoning. As part of future work, we plan to apply this framework to real time and time triggered systems to further illustrate this point.

References

1. Abdullah, P., Buojjani, A., Jonsson, B., Nilsson, M.: Handling Global Conditions in Parameterized System Verification. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 134–145. Springer, Heidelberg (1999)
2. Arons, T., Pnueli, A., Ruah, S., Zuck, L.: Parameterized Verification with Automatically Computed Inductive Assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, Springer, Heidelberg (2001)
3. Baukus, K., Bensalem, S., Lakhnech, Y., Stahl, K.: Abstracting WS1S Systems to Verify Parameterized Networks. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, Springer, Heidelberg (2000)
4. Stahl, K., Baukus, K., Lakhnech, Y.: Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, Springer, Heidelberg (2002)
5. Browne, M.C., Clarke, E.M., Grumberg, O.: Reasoning about Networks with Many Identical Finite State Processes. Information and Computation 81, 13–31 (1989)

6. Chou, C.-T., Mannava, P.K., Park, S.: A Simple Method for Parameterized Verification of Cache Coherence Protocols. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, Springer, Heidelberg (2004)
7. Clarke, E., Talupur, M., Veith, H.: Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems, www.cs.cmu.edu/~tmurali/tacas08.pdf
8. Clarke, E., Talupur, M., Veith, H.: Environment Abstraction for Parameterized Verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2005)
9. Cohen, A., Namjoshi, K.: Local Proofs for Global Safety Properties. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 55–67. Springer, Heidelberg (2007)
10. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent Control with "Readers" and "Writers". Communication of the ACM 14 (1971)
11. Delzanno, G.: Automated Verification of Cache Coherence Protocols. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, Springer, Heidelberg (2000)
12. German, S.M., Sistla, A.P.: Reasoning about Systems with Many Processes. Journal of the ACM 39 (1992)
13. Grinchtein, O., Leucker, M., Piterman, N.: Inferring Network Invariants Automatically. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 483–497. Springer, Heidelberg (2006)
14. Lahiri, S.K., Bryant, R.: Constructing Quantified Invariants. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, Springer, Heidelberg (2004)
15. Lahiri, S.K., Bryant, R.: Indexed Predicate Discovery for Unbounded System Verification. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 135–147. Springer, Heidelberg (2004)
16. McMillan, K.L.: Verification of an implementation of tomasulo's algorithm by compositional model checking. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 110–121. Springer, Heidelberg (1998)
17. McMillan, K.L.: Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, Springer, Heidelberg (2001)
18. McMillan, K.L., Qadeer, S., Saxe, J.B.: Induction in Compositional Model Checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 312–327. Springer, Heidelberg (2000)
19. Pnueli, A., Ruah, S., Zuck, L.: Automatic Deductive Verification with Invisible Invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, Springer, Heidelberg (2001)
20. Pnueli, A., Xu, J., Zuck, L.: Liveness with $(0, 1, \infty)$-Counter Abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, Springer, Heidelberg (2002)
21. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. In: TOPLAS (2002)
22. Samer, M., Veith, H.: A Syntactic Characterization of Distributive LTL Queries. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 1099–1110. Springer, Heidelberg (2004)
23. Samer, M., Veith, H.: Deterministic CTL Query Solving. In: Proc. of the 12th International Symposium on Temporal Representation and Reasoning (TIME) (2005)
24. Suzuki, I.: Proving Properties of a Ring of Finite State Machines. Information Processing Letters 28, 213–214 (1988)
25. Talupur, M.: Abstraction Techniques for Infinite State Verification. PhD thesis, Carnegie Mellon University, Computer Science Department (2006)
26. Yahav, E.: Verifying safety properties of concurrent Java programs using 3-valued logic. In: The Proceedings of 18th Symposium on Principles of Programming Languages (2001)