

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

Edmund Clarke¹, Muralidhar Talupur², and Helmut Veith³

¹ School of Computer Science, Carnegie Mellon University, USA

² Intel Research, Portland, USA

³ 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 high level compound statements. These two insights allow us to 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 to find many obvious errors, they cannot explore all interleaving behaviors, and may miss subtle errors. Consequently, 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 has been considered by [9, 15, 4, 2, 10]. Another important class of protocol that has been widely studied is mutual exclusion protocols [16, 12, 1, 19].

The approaches by McMillan [15], Chou et al. [4], 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 [12] and Pnueli et al [19, 1] 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 high level 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 story of the Ptolemaic system (in which the sun orbits the earth) 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 often wrong for the 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 [3].

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 $\Delta_2(p)$ holds 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 (*r-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 [6, 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 high level 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 high level communication 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 transition 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^A from the protocol description. In our prototype implementation, this abstract model is an *SMV* model.
4. A *model checker* verifies $P^A \models \varphi(x)$. Note that in P^A , x is interpreted as the

reference process. If $P^A \models \varphi(x)$, 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. [8]. 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 protocol and Szymanski’s algorithm [6]. Although our paper [6] contains several seminal ideas about environment abstraction, it is very different in scope and generality. In particular, the methods in [6] are tailored towards a specific application and a hardwired set of specifications, without a general soundness result.

The method of counter abstraction [19] inspired our approach, and can be seen as a specific, but limited form of environment abstraction. Invisible invariants [18, 1] provide another novel method for verifying parameterized systems. Both these methods are restricted to systems without unbounded integer variables.

The Indexed Predicates method [12, 13] 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 [14–16, 4] 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 [15] which can be non-trivial and require deep understanding of the protocol under verification.

The TVLA method proposed by Reps et al. [27, 20, 26] is an abstract interpretation based approach for shape analysis, and also verification of safety properties of multi-threaded systems. TVLA is widely applicable, and uses first order logical structures as the abstract domain. 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, TVLA uses summarization, which is related to the idea of counting abstraction. While summarization of similar objects is a powerful feature that lets TVLA deal with unbounded systems, it is sometimes necessary to track in detail one object, say o_1 , separately from other objects o_2, \dots, o_n even though they may have the *same properties*. TVLA therefore uses special unary predicates, called *instrumentation* predicates to select some particular object as a special object. At first sight, it might seem that instrumentation predicates can be used to

simulate our notion of a reference process, but this is not the case: The only property distinguishing a reference process from other processes in our system is its id. Thus, if we use instrumentation predicates to simulate the notion of a reference process, the predicates will have to refer to the process id's. This means that once a reference process is chosen by instrumentation predicates, it cannot change along a trace. In environment abstraction, however, the identity of the process that serves as the reference process may change with each abstract transition. Thus, the notion of a reference process, which is crucial to our approach, is inherently different from instrumentation predicates.

Recent work on parameterized verification includes [11, 7]. The former paper [11] proposes a method to find network invariants using finite automata learning algorithms due to Angluin and Beirmann. The latter [7] use a new completion procedure to strengthen split invariants. While parameterized verification is not the primary aim, their method is able to produce parameterized proofs for protocols like the Bakery protocol.

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 $\{1, \dots, K\}$ 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 $\langle s_1, \dots, s_K \rangle$. 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 $\langle s_1, \dots, s_K, t_1, \dots, t_{\text{const}} \rangle$ 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
$\text{pc}[x] = c$	the program counter of process x has value c
$c \in \text{env}(x)$	there is a process $y \neq x$ with program counter value c “ <u>The environment</u> 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

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

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. **Relative Completeness (r-completeness).** For each description $\Delta(x) \in D$ and each label $l(x) \in L$ it holds that either

$$\Delta(x) \rightarrow l(x) \quad \text{or} \quad \Delta(x) \rightarrow \neg l(x).$$

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 r-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 $\Delta(x) \rightarrow l(x)$.

Description of the Abstract System P^A . Given two sets D and L of descriptions and labels which satisfy the two criteria *coverage* and *r-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 $\Delta(x) \rightarrow l(x)$, i.e., $L^A(\Delta(x)) = \{l(x) : \Delta(x) \rightarrow l(x)\}$. 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 r-completeness we have the following general soundness theorem that we will prove in Section A.

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* specifications $\forall x.\varphi(x)$, the following holds:*

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

A generalization to multiple reference process is described in Section A.3.

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. Section A.3.)

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 (and explained in more detail in Section B) 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 Section B of the appendix. This characterization was obtained in previous work [21–23] 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. [8]. 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 [17] – have not been verified parametrically. Figure 1 shows the code for a reader process in the simpler of the two algorithms.

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

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 relatively simple 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 [8] 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

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

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.

Appendix ?? presents the two algorithms from [8] in our input language. For example, the semaphore acquire action at $L1$ in Figure 1 can be modelled as

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

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

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

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

$$\langle \mathbf{pc}, e_1, \dots, e_n, \Delta_{\text{cen}} \rangle$$

to describe an abstract state $\Delta(x)$. Intuitively, \mathbf{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 *r-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

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

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

$$\langle \mathbf{pc}, e_1, \dots, e_n, \Delta_{\text{cen}} \rangle \quad \text{to} \quad \langle \mathbf{pc}', e'_1, \dots, e'_n, \Delta'_{\text{cen}} \rangle$$

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

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

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

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

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

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

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

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

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, \dots, cs_m we first find invariants $I(cs_1), \dots, I(cs_m)$ 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 $P^{\mathcal{A}}$ and the abstract system obtained from the definition of environment abstraction by P^A . Our construction is a natural over-approximation of P^A :

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

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 shown in Appendix C, we verified the safety property

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

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 counterexample. To eliminate this spurious counterexample we make use of the invariant

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

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

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

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: We can on the *original abstract model* and for the reference process that $pc[x] \in [L3..L7] \rightarrow \neg(readcount = 0)$. Then we can conclude by soundness of our abstraction that indeed $\forall x. pc[x] \in [L3..L7] \rightarrow \neg(readcount = 0)$, and thus we can safely add the invariant.

Running SMV on the original model indeed establishes the sought for invariant. Thus, we are justified in adding the invariant as an 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. [15, 4].

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 [6].

Mutual Exclusion Protocols. In [6], we have shown how to verify mutual exclusion protocols such as the Bakery protocol and Szymanski’s algorithm. To model such mutual exclusion protocols we need two compound statements, namely, *guarded transitions* and *update transitions*:

<p>Guarded Transition $pc = L_1 : \mathbf{if} \forall \text{otr} \neq x. \mathcal{G}(x, \text{otr}) \mathbf{then goto} pc = L_2 \mathbf{else goto} pc = L_3$ <i>Semantics:</i> In control location L_1, the process evaluates the guard and changes to control location L_2 or L_3 accordingly.</p>
<p>Update Transition $pc = L_1 : \mathbf{for all} \text{otr} \neq x \quad \mathbf{if} \mathcal{T}(x, \text{otr}) \mathbf{then} u_k := \varphi(\text{otr}) \mathbf{goto} pc = L_2$ <i>Semantics:</i> At location L_1, the process scans over all other processes otr to check if formula $\mathcal{T}(x, \text{otr})$ is true. In this case, the process changes the value of its data variable u_k according to $u_k := \varphi(\text{otr})$. Finally, the process changes to location L_2.</p>

These compound statements are more complex than in Section 4. The abstract domain is also more complex, because each process can have unbounded data variables. To account for these variables, the descriptions $\Delta(x)$ include *inter-predicates* $IP_j(x, y)$, i.e., predicates that span multiple processes [6]. Thus, the $\Delta(x)$ have the form

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

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 [6] 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* $pc = L_1 : \mathbf{goto} pc = L_2$. 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

$$\mathit{guard} : \mathbf{do actions} A_1, A_2, \dots, A_k$$

where A_1, \dots, 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 (whose description is summarized in Appendix D) comprise *goto*, *assign*, *add*, *remove*, *pick* and *remote* actions.

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 we generate can become very large. The reason of course is the level of detail in which the abstract model tracks the concrete executions: The abstract descriptions are constructed by picking *all* control conditions that appear in the protocol code.

There is a promising approach to alleviate this problem: instead of building the best possible abstract model we can build a coarser model which we refine using the circular reasoning scheme of [14, 15, 4]. We have already used a dilute form of this method in dealing with semaphore based algorithms. Such a hybrid approach promises to combine the strengths of our approach and the circular reasoning approach in [14, 4].

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. The abstract domain is obtained syntactically by collecting and combining the predicates from the protocol code. Then we describe the actions of compound statements in terms of this abstract domain. 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.

Correctness of the abstract template invariants ? This question is not much different from asking how we know that a source code model checker is correct. It is easier to convince ourselves of 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.

Scope for abstraction refinement ? While this is not in the scope of the current paper, 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 also a natural part of our future work.

We believe that the different applications of environment abstraction have demonstrated that this approach is working well in practice for a wide range of distributed protocols. We plan to apply our method to real time systems and time triggered systems to further illustrate this point.

References

1. T. Arons, A. Pnueli, S. Ruah, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proc. Intl. Conf. Computer Aided Verification (CAV)*, 2001.
2. K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized verification of cache coherence protocols: Safety and liveness. In *Proc. VMCAI*, 2002.
3. 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.
4. C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *Proc. FMCAD'04*, 2004.
5. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proc. of Principles of Programming Languages*, 1992.
6. E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *Proc. VMCAI*, 2006.
7. A. Cohen and K. Namjoshi. Local proofs for global safety properties. In *CAV*, 2007.
8. P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communication of the ACM*, 14, 1971.
9. G. Delzanno. Automated verification of cache coherence protocols. In *Computer Aided Verification 2000 (CAV 00)*, 2000.
10. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39, 1992.
11. O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *IJCAR*, 2006.
12. S. K. Lahiri and R. Bryant. Constructing quantified invariants. In *Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2004.
13. S. K. Lahiri and R. Bryant. Indexed predicate discovery for unbounded system verification. In *Proc. 16th Intl. Conf. Computer Aided Verification (CAV)*, 2004.
14. K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *Proc. Computer Aided Verification*, 1998.
15. K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Proc. CHARME '01*, volume 2144 of *LNCS*, pages 179–195. Springer, 2001.
16. K. L. McMillan, S. Qadeer, and J. B. Saxe. Induction in compositional model checking. In *Conference on Computer Aided Verification*, pages 312–327, 2000.
17. P. O'Hearn. *Private Communication*. 2006.
18. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proc. TACAS*, 2001.
19. A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$ counter abstraction. In *Proc. CAV'02*, 2002.
20. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *TOPLAS*, 2002.
21. M. Samer and H. Veith. Validity of CTL queries revisited. In *Proceedings of 12th Annual Conference of the European Association for Computer Science (CSL)*, 2003.
22. M. Samer and H. Veith. A syntactic characterization of distributive LTL queries. In *Proc. ICALP*, 2004.
23. M. Samer and H. Veith. Deterministic CTL query solving. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME)*, 2005.
24. I. Suzuki. Proving properties of a ring of finite state machines. *Information Processing Letters*, 28:213–214, 1988.

25. M. Talupur. *Abstraction Techniques for Infinite State verification*. PhD thesis, Carnegie Mellon University, Computer Science Department, 2006.
26. E. Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *In the Proceedings of 18th Symposium on Principles of Programming Languages*, 2001.
27. E. Yahav. *Property Guided Abstractions of Concurrent Heap Manipulating Programs*. PhD thesis, Tel-Aviv University, Israel, 2004.

APPENDIX

A Soundness

In this section, we give a proof of Theorem 1. Before we give the proof of the soundness theorem we introduce a variant of simulation to simplify the presentation.

A.1 Simulation Modulo Renaming

In the classical approach to abstraction in model checking [5], one establishes a simulation relation between the concrete model and its abstraction. By definition, the simulation relation requires that the labels of the concrete model are a subset of the labels of the abstract Kripke structure. (Some authors even require that both structures have the same set of labels.) In our context, however, the definition of Section 2 says that the abstract Kripke structure always has labels of the form $l(x)$, for example, $pc[x] = 5$, while the concrete Kripke structures have labels of the form $l(c)$ where c is a process id, for example $pc[2] = 5$. Thus, we need a more flexible notion of simulation, where we can systematically rename the labels in one Kripke structure to match the labels in the other one. In this way, we obtain a simple variation of the classical abstraction theorem. Below we give the formal definition.

Given a fixed process c , we write $P_c(K)$ to denote the Kripke structure obtained from $P(K)$ where L_K is restricted to *only those labels which refer to process c* . Thus, $P_c(K)$ is labeled only with c -labels.

Fact 2 *Let c be a process in $P(K)$, and $\varphi(x)$ be a temporal formula over atomic labels from L . Then*

$$P(K) \models \varphi(c) \quad \text{if and only if} \quad P_c(K) \models \varphi(c).$$

This follows directly from the fact that the truth of $\varphi(c)$ depends only on c -labels.

Our soundness proofs will require a simple variation of the classical abstraction theorem [5]. Recall that the classical abstraction theorem for ACTL* says that for ACTL* specifications φ and two Kripke structures K_1 and K_2 it holds that $K_1 \succeq K_2$ and $K_1 \models \varphi$ together imply $K_2 \models \varphi$.

Definition 2 (Simulation Modulo Renaming). *Let K be a Kripke structure, c a process id, and d a process id which does not occur in K . Then $K[c/d]$ denotes the Kripke structure obtained from K by replacing each label of the form $l(c)$ by $l(d)$. Simulation modulo renaming $\preceq_{c/d}$ is defined as follows:*

$$K_1 \preceq_{c/d} K_2 \quad \text{iff} \quad K_1[c/d] \preceq K_2.$$

Then $\preceq_{c/d}$ gives rise to a simple variation of the classical abstraction theorem:

Fact 3 (Abstraction Theorem Modulo Renaming) *Let $\varphi(x)$ be a temporal formula over atomic labels from L , and let K_1, K_2 be Kripke structures which are labeled only with c_1 -labels and c_2 -labels respectively. Then the following holds: If $K_2 \preceq_{c_2/c_1} K_1$ and $K_1 \models \varphi(c_1)$, then $K_2 \models \varphi(c_2)$.*

Proof. First note that $K_2 \models \varphi(c_2)$ is equivalent to $K_2[c_2/c_1] \models \varphi(c_1)$: if the labels in the Kripke structure and the atomic propositions in the specification are consistently renamed, then the satisfaction relation does not change.

Thus, given that $K_2 \preceq_{c_2/c_1} K_1$ and $K_1 \models \varphi(c_1)$, it is enough to show that $K_2[c_2/c_1] \models \varphi(c_1)$. By definition of $\preceq_{c/d}$, $K_2 \preceq_{c_2/c_1} K_1$ iff $K_2[c_2/c_1] \preceq K_1$ and by the classical abstraction theorem [5], $K_1 \models \varphi(c_1)$ implies $K_2[c_2/c_1] \models \varphi(c_1)$. This proves the abstraction theorem. \square

A.2 Proof of Soundness

We will show that environment abstraction preserves indexed properties of the form $\forall x.\varphi(x)$ where $\varphi(x)$ is an ACTL* formula over atomic labels from L .

Step 1: Reduction to Simulation. The statement of Theorem 1 says

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

By the semantics of our specification language this is equivalent to saying that for all $K > 1$

$$P^A \models \varphi(x) \quad \text{implies} \quad \forall c \in [1..K].P(K) \models \varphi(c).$$

This in turn is equivalent to showing that, for all $K > 1$ and all processes $c \in [1..K]$,

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

Recall that $P_c(K)$ is the Kripke structure obtained from $P(K)$ which contains only c -labels. By Fact 1 we know that $P(K) \models \varphi(c)$ iff $P_c(K) \models \varphi(c)$. Thus, we need to show that for all $K > 1$ and for all $c \in [1..K]$

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

Note that the last condition (*) can be obtained from the abstraction modulo renaming theorem (Fact 2): We want to carry over the truth of a specification from the abstract Kripke structure P^A to the concrete structure $P_c(K)$, but the labels in the specification change. In other words, if we can show that

$$\boxed{P_c(K) \preceq_{c/x} P^A \quad \text{for all } K \text{ and } c \in [1..K] \quad (**)}$$

then Fact 2 implies (*), and thus, Theorem 1 is proven.

We will now prove these simulations.

Step 2: Proof of Simulation. We will now show how to establish (**), i.e., the simulation relation $P_c(K) \preceq_{c/x} P^A$ between $P_c(K)$ and P^A for all $K > 1$ and $c \in [1..K]$. To this end, we will for each K and c construct an intermediate abstract system $P_{c,K}^A$ such that

$$\begin{aligned} P_c(K) &\preceq_{c/x} P_{c,K}^A && \text{(Simulation 1)} \\ \text{and} &&& \\ P_{c,K}^A &\preceq P^A. && \text{(Simulation 2)} \end{aligned}$$

The required simulation then follows by transitivity of simulation.

Intuitively, the intermediate model $P_{c,K}^A$ is the abstraction of the K -process system $P(K)$ where the reference process c is fixed. In other words, $P_{c,K}^A$ is the abstract model for the specific case of a K -process system *where the Ptolemaic observer is located on process c* . Since K and c are fixed, it follows that $P_{c,K}^A$ is an existential abstraction of $P_c(K)$, as explained in the remark following Definition 1.

We will now formally define the intermediate model $P_{c,K}^A$, and give a rigorous proof.

Construction of $P_{c,K}^A$. The abstract model

$$P_{c,K}^A = \langle D, I_{c,K}^A, R_{c,K}^A, L^A \rangle$$

is defined analogously to P^A for the special case where K and c are fixed. Thus, $P_{c,K}^A$ is the abstract model of the concrete system $P_c(K)$ with a fixed number K of processes and reference process c . More precisely, $P_{c,K}^A$ is defined as follows:

- (a) The state space D is the same as in P^A , i.e., the set of descriptions.
- (b) The set of initial states $I_{c,K}^A$ is the subset of the initial states I^A of P^A for the special case of K and c . Thus, $I_{c,K}^A$ is given by those abstract states $\Delta(x)$ for which there exists a state s in $P_c(K)$ such that $\alpha_c(s) = \Delta(x)$.
- (c) The transition relation $R_{c,K}^A$ is the subset of the transition relation R^A of P^A for the special case of K and c . Thus, there is a transition from $\Delta_1(x)$ to $\Delta_2(x)$ in $R_{c,K}^A$ if and only if there are two states s_1, s_2 in $P_c(K)$ such that $\Delta_1(x) \in \alpha_c(s_1)$, $\Delta_2(x) \in \alpha_c(s_2)$, and $(s_1, s_2) \in R$.
- (d) The labeling function L^A is the same as in P^A .

Proof of Simulation 1. We know from the informal discussion above that $P_{c,K}^A$ is (except for the systematic renaming of the labels) an existential abstraction of $P_c(K)$, and therefore, it should be intuitively clear that we obtain simulation between these two structures. Since we deal with simulation modulo renaming, however, we provide a formal proof. The proof is very similar to the classic proof that existential abstraction gives rise to a simulation relation, but at certain points actually uses the requirements *coverage* and *r-completeness* of Section 2.

Formally, we need to show that $P_c(K) \preceq_{c/x} P_{c,K}^A$ which by definition of simulation modulo renaming is equivalent to

$$P_c(K)[c/x] \preceq P_{c,K}^A.$$

Consider the structure $P_c(K)[c/x]$. This is just the K -process system $P(K)$ restricted to the labels for process c , but because of the renaming, the labels have the form $l(x)$ instead of $l(c)$. Thus, the labels of $P_c(K)[c/x]$ are taken from the set L . Note that the labels of the abstract system P^A are also taken from the set L . The proof idea below is similar to the construction of a simulation relation for existential abstraction.

Consider the relation

$$\mathcal{I} = \{ \langle s, \Delta(x) \rangle : s \models \Delta(c), s \in S_K, \Delta(x) \in D \}.$$

We claim that \mathcal{I} is a simulation relation between $P_c(K)[c/x]$ and $P_{c,K}^A$:

- (1) Lemma 1 together with the renaming of c into x guarantees that for every tuple $\langle s, \Delta(x) \rangle \in \mathcal{I}$, the states s and $\Delta(x)$ have the same labels.
- (2) Consider a tuple $\langle s, \Delta(x) \rangle \in \mathcal{I}$. Assume that s has a successor state s' , i.e., $(s, s') \in R_K$. We need to show that there exists an abstract state $\Delta'(x)$ such that
 - (i) $(\Delta(x), \Delta'(x)) \in R_{c,K}^A$, and
 - (ii) $\langle s', \Delta'(x) \rangle \in \mathcal{I}$.

To find such a $\Delta'(x)$, consider the abstraction $\alpha_c(s')$ of s' , and choose some description $\Gamma(x) \in \alpha_c(s')$. (Recall that, by definition, $\alpha_c(s')$ contains all descriptions $\Gamma(x)$ for which $s' \models \Gamma(c)$.) By the coverage condition, $\alpha_c(s')$ is non-empty.

We will show by contradiction that $\Gamma(x)$ fulfills the properties (i) and (ii).

Property (i) Assume that $\Gamma(x)$ does not fulfill property (i), i.e., $(\Delta(x), \Gamma(x)) \notin R_{c,K}^A$. Then for all states s_1 and s_2 it must hold that whenever $\Delta(x) \in \alpha_c(s_1)$ and $\Gamma(x) \in \alpha_c(s_2)$ that *there is no transition* between s_1 and s_2 . On the other hand, we assumed above that $\Delta(x) \in \alpha_c(s)$, $\Gamma(x) \in \alpha_c(s')$ and there is a transition from s to s' . Hence we have a contradiction.

Property (ii) Assume now that $\Gamma(x)$ does not fulfill property (ii), i.e., $\langle s', \Gamma(x) \rangle \notin \mathcal{I}$. By the definition of \mathcal{I} this means that $s' \not\models \Gamma(c)$, and thus, $\Gamma(c) \notin \alpha_c(s')$. This gives us the required contradiction.

Thus, $\Delta'(x)$ can be chosen to be $\Gamma(x)$.

- (3) Finally, the coverage property guarantees that for every initial state $s \in I_K$ there exists some $\Delta(x) \in I_{c,K}^A$ such that $\langle s, \Delta(x) \rangle \in \mathcal{I}$.

This concludes the proof of Simulation 1.

Proof of Simulation 2. By construction, $I_{c,K}^A \subseteq I^A$ and $R_{c,K}^A \subseteq R^A$. Therefore, P^A is an over-approximation of $P_{c,K}^A$, and the simulation follows. \square

Remark 2. Note that the requirements coverage and r-completeness for D and L are used in crucial parts of the proof of Simulation 1. R-completeness is used in the proof of Lemma 1 which gives us part (1) of Simulation 1. Part (2) of Simulation 1 requires coverage to make sure that $\alpha_c(s')$ is non-empty. Part (3) of Simulation 1 also requires coverage to ensure the existence of an abstract initial state.

A.3 Abstraction with Multiple Reference Processes

It is easy to extend our framework to two reference processes and specifications with two index variables. Essentially, we replace the free variable x in the labels and descriptions by a pair x, y , and carry this modification through all definitions and proofs. In particular, the coverage and congruence requirements are generalized as follows:

1. **Coverage.** For each system $P(K)$, each state s in $P(K)$ and any two processes c, d there is some description $\Delta(x, y) \in D$ such that $s \models \Delta(c, d)$.
2. **Completeness.** For each description $\Delta(x, y) \in D$ and each label $l(x, y) \in L$ it holds that either $\Delta(x, y) \rightarrow l(x, y)$ or $\Delta(x, y) \rightarrow \neg l(x, y)$.

The construction of the abstract model is analogous to the single index case. We attach labels to each state of P^A such that the abstract state $\Delta(x, y)$ has label $l(x, y)$ iff $\Delta(x, y) \rightarrow l(x, y)$. Then we again obtain a natural preservation theorem:

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

Note that in practice, however, we found such models much harder to model check than in the single index case. The generalization to > 2 indices is straightforward.

B Trade-Off between Labels and Quantifiers

It is natural to ask when a double-indexed specification can be translated into a single-indexed specification as in the example above. Somewhat surprisingly, this question is related to our previous work on temporal logic query languages [21–23].

A temporal logic query is a formula $\underline{\gamma}$ with one occurrence of a distinguished atomic subformula “?” (called a placeholder). Given $\underline{\gamma}$ and a formula ψ , we write $\underline{\gamma}[\psi]$ to denote the formula obtained by replacing ? by ψ .

B.1 Characterizations of Distributive Queries

In [21–23], we have obtained syntactic characterizations for CTL and LTL queries with the distributivity property

$$\underline{\gamma}[\psi_1 \wedge \psi_2] \leftrightarrow \underline{\gamma}[\psi_1] \wedge \underline{\gamma}[\psi_2].$$

Our characterizations are given in terms of *template grammars*, cf. Figures 2 to and 4. In these grammars, a star \star is a wildcard which stands for an arbitrary temporal formula. A formula or query involving a \star is called a template. An instance of a template is a formula or query obtained by instantiating all \star symbols (but leaving the placeholder ? untouched). In the grammar, each non-terminal produces a language of templates and a language of queries in the natural way.

Theorem 2 ([21–23]).

- All LTL queries produced by non-terminals $Q1, Q2, Q7$ in Figure 2 are distributive. Moreover, each template produced by $Q3, Q4, Q5, Q6$ has an instantiation by atomic formulas such that the resulting query is non-distributive. Together, the queries produced by $Q1 \dots Q7$ cover all LTL queries.
- All CTL queries produced by non-terminals different from $Q11$ in Figures 3 and 4 are distributive.

We thus have a fairly tight characterization in case of LTL; an analogous exact characterization for CTL is still open.

B.2 Distribution of Quantifiers into Labels

The prototypical example of a distributive query is $\mathbf{AG}?$, and we have seen above how we can translate double indexed properties involving \mathbf{AG} into single-indexed properties. As argued above, this translation actually amounts to distributing one universal quantifier inside the label of the temporal formula.

It is not hard to see that such a translation is possible for all specifications which are distributive with respect to one index variable: Consider a double-indexed specification $\forall x, y. x \neq y \rightarrow \varphi(x, y)$ where all occurrences of y in φ are located in a subformula $\theta(x, y)$ of φ . Then we can write φ as a query $\underline{\gamma}[\theta]$. Now suppose that $\underline{\gamma}$ is distributive. On

each concrete finite system $P(K)$, the universal quantification reduces to a conjunction, i.e.,

$$P(K) \models \forall x, y. x \neq y \rightarrow \underline{\gamma}[\theta(x, y)] \quad \text{iff} \quad P(K) \models \forall x. \bigwedge_{1 \leq c \leq K, c \neq x} \underline{\gamma}[\theta(x, c)]$$

which by distributivity of $\underline{\gamma}$ is equivalent to

$$P(K) \models \forall x. \underline{\gamma} \left[\bigwedge_{1 \leq c \leq K, c \neq x} \theta(x, c) \right]$$

and thus to

$$P(K) \models \forall x. \underline{\gamma} [\forall y. x \neq y \rightarrow \theta(x, c)].$$

For a suitable label $l(x) := \forall y. x \neq y \rightarrow \theta(x, y)$ this can be written as

$$P(K) \models \forall x. \underline{\gamma}[l(x)].$$

Thus, we have shown the following theorem:

Theorem 3. *Let $\Psi = \forall x, y. x \neq y \rightarrow \underline{\gamma}[\theta]$ be a double indexed specification such that (i) all occurrences of y in $\underline{\gamma}[\theta]$ occur in the subformula θ , and (ii) $\underline{\gamma}[?]$ is distributive. Then Ψ is equivalent to the single indexed specification*

$$\forall x. \underline{\gamma}[l(x)].$$

where $l(x) := \forall y. x \neq y \rightarrow \theta(x, y)$ is a label.

Here we see the *trade-off* between the expressive power of the labels and the number of indices: by increasing the complexity of the labels, we reduce the number of index variables.

Corollary 1. *Under the assumptions of the above theorem, consider the special case where $\theta(x, y)$ has the form $pc(y) = L$. Then Ψ is equivalent to*

$$P(K) \models \forall x. \underline{\gamma}[\neg L \in env(x)].$$

While the characterization of distributive queries together with our trade-off results are giving us a good understanding about the scope of single-indexed specifications, it is clear that not all two-indexed specifications can be rewritten with a single index. Consider for example the formula

$$\forall x, y. x \neq y \rightarrow \mathbf{AF}pc[x] = 5 \wedge pc[y] = 5.$$

Here, it is evidently not possible to move the quantifier inside, and this in fact can be derived from the characterization in [21–23]. Consequently, this specification cannot be expressed with a single index.

B.3 Modalities for Environment Processes

Above we have described how to introduce a label for the expression $l(x) = \forall y. x \neq y \rightarrow \theta(x, y)$ and treating it as a compound formula. Thus, these labels are hiding a quantifier. This procedure reminds very strongly of the semantics of modal logic. In fact, the classical modalities \Box and \Diamond are well-known examples of hidden quantifiers which quantify over possible worlds.

In our setting, we can also view the processes in the system as different worlds in which the local variables have different values. Thus, we can introduce a modality $\boxtimes\varphi$ which says that φ holds true for all other processes. In this way, we can express single-indexed properties over $pc[x] = L$ and $L \in env(x)$ in a very different syntax which does not refer to $pc[x] = L$ and $L \in env(x)$ explicitly:

$$\mathbf{AG} \ pc = 5 \rightarrow \boxtimes(pc \neq 5)$$

Thus, instead of introducing index variables, we can extend temporal logic by a new non-temporal modality \boxtimes which enables us to talk about different worlds / processes. We will continue these investigations in future work.

$\langle Q1 \rangle ::=$	$?$	$\star \wedge \langle Q2 \rangle$	$\langle Q2 \rangle \mathring{U} \star$	$\star \mathring{U} \langle Q2 \rangle$	
	$\star \bar{U} \langle Q2 \rangle$	$\langle Q2 \rangle \mathring{W} \star$	$\star \mathring{W} \langle Q2 \rangle$	$\star \bar{W} \langle Q2 \rangle$	
	$\star \wedge \langle Q1 \rangle$	$\star \vee \langle Q1 \rangle$	$\mathbf{X} \langle Q1 \rangle$	$\langle Q1 \rangle \mathring{U} \star$	
	$\star \bar{U} \langle Q1 \rangle$	$\langle Q1 \rangle \mathring{W} \star$	$\star \bar{W} \langle Q1 \rangle$;	
$\langle Q2 \rangle ::=$	$\langle Q1 \rangle \mathbf{U} \star$	$\langle Q1 \rangle \mathbf{W} \star$	$\star \vee \langle Q2 \rangle$	$\mathbf{X} \langle Q2 \rangle$	
	$\langle Q2 \rangle \mathbf{U} \star$	$\star \mathbf{U} \langle Q2 \rangle$	$\langle Q2 \rangle \mathbf{W} \star$	$\star \mathbf{W} \langle Q2 \rangle$;
$\langle Q3 \rangle ::=$	$\mathbf{F} \langle Q1 \rangle$	$\mathbf{F} \langle Q5 \rangle$	$\mathbf{F} \langle Q6 \rangle$	$\mathbf{G} \langle Q4 \rangle$	
	$\mathbf{G} \langle Q6 \rangle$	$\langle Q4 \rangle \mathring{U} \star$	$\star \mathbf{U} \langle Q1 \rangle$	$\star \mathbf{U} \langle Q5 \rangle$	
	$\star \mathbf{U} \langle Q6 \rangle$	$\star \mathring{U} \langle Q1 \rangle$	$\star \mathring{U} \langle Q5 \rangle$	$\star \mathring{U} \langle Q6 \rangle$	
	$\star \bar{U} \langle Q4 \rangle$	$\star \bar{U} \langle Q5 \rangle$	$\star \bar{U} \langle Q6 \rangle$	$\langle Q4 \rangle \mathbf{W} \star$	
	$\langle Q6 \rangle \mathbf{W} \star$	$\langle Q4 \rangle \mathring{W} \star$	$\star \mathbf{W} \langle Q5 \rangle$	$\star \mathbf{W} \langle Q6 \rangle$	
	$\star \bar{W} \langle Q4 \rangle$	$\star \bar{W} \langle Q5 \rangle$	$\star \bar{W} \langle Q6 \rangle$	$\star \wedge \langle Q3 \rangle$	
	$\star \vee \langle Q3 \rangle$	$\mathbf{X} \langle Q3 \rangle$	$\mathbf{F} \langle Q3 \rangle$	$\mathbf{G} \langle Q3 \rangle$	
	$\langle Q3 \rangle \mathring{U} \star$	$\star \mathbf{U} \langle Q3 \rangle$	$\star \mathring{U} \langle Q3 \rangle$	$\star \bar{U} \langle Q3 \rangle$	
	$\langle Q3 \rangle \mathbf{W} \star$	$\langle Q3 \rangle \mathring{W} \star$	$\star \mathbf{W} \langle Q3 \rangle$	$\star \bar{W} \langle Q3 \rangle$;
$\langle Q4 \rangle ::=$	$\langle Q3 \rangle \mathbf{U} \star$	$\langle Q5 \rangle \mathbf{U} \star$	$\langle Q6 \rangle \mathbf{U} \star$	$\langle Q5 \rangle \mathbf{W} \star$	
	$\star \vee \langle Q4 \rangle$	$\mathbf{X} \langle Q4 \rangle$	$\langle Q4 \rangle \mathbf{U} \star$	$\star \mathbf{U} \langle Q4 \rangle$	
	$\star \mathbf{W} \langle Q4 \rangle$;			
$\langle Q5 \rangle ::=$	$\star \mathring{U} \langle Q4 \rangle$	$\star \mathbf{W} \langle Q1 \rangle$	$\star \mathring{W} \langle Q1 \rangle$	$\star \mathring{W} \langle Q3 \rangle$	
	$\star \mathring{W} \langle Q4 \rangle$	$\star \mathring{W} \langle Q6 \rangle$	$\star \wedge \langle Q5 \rangle$	$\mathbf{X} \langle Q5 \rangle$	
	$\langle Q5 \rangle \mathring{U} \star$	$\langle Q5 \rangle \mathring{W} \star$	$\star \mathring{W} \langle Q5 \rangle$;	
$\langle Q6 \rangle ::=$	$\star \wedge \langle Q4 \rangle$	$\star \vee \langle Q5 \rangle$	$\star \wedge \langle Q6 \rangle$	$\star \vee \langle Q6 \rangle$	
	$\mathbf{X} \langle Q6 \rangle$	$\langle Q6 \rangle \mathring{U} \star$	$\langle Q6 \rangle \mathring{W} \star$;	
$\langle Q7 \rangle ::=$	$\mathbf{F} \langle Q2 \rangle$	$\mathbf{F} \langle Q4 \rangle$	$\mathbf{G} \langle Q1 \rangle$	$\mathbf{G} \langle Q2 \rangle$	
	$\mathbf{G} \langle Q5 \rangle$	$\star \wedge \langle Q7 \rangle$	$\star \vee \langle Q7 \rangle$	$\mathbf{X} \langle Q7 \rangle$	
	$\mathbf{F} \langle Q7 \rangle$	$\mathbf{G} \langle Q7 \rangle$	$\langle Q7 \rangle \mathbf{U} \star$	$\langle Q7 \rangle \mathring{U} \star$	
	$\star \mathbf{U} \langle Q7 \rangle$	$\star \mathring{U} \langle Q7 \rangle$	$\star \bar{U} \langle Q7 \rangle$	$\langle Q7 \rangle \mathbf{W} \star$	
	$\langle Q7 \rangle \mathring{W} \star$	$\star \mathbf{W} \langle Q7 \rangle$	$\star \mathring{W} \langle Q7 \rangle$	$\star \bar{W} \langle Q7 \rangle$;

Fig. 2. The template grammar for distributive LTL queries.

$\langle Q1 \rangle ::=$?	$\star \wedge \langle Q3 \rangle$	$\star \wedge \langle Q4 \rangle$
	$\star \vee \langle Q2 \rangle$	AX $\langle Q3 \rangle$	AX $\langle Q4 \rangle$
	AX $\langle Q6 \rangle$	AX $\langle Q7 \rangle$	A ($\langle Q3 \rangle \mathring{U} \star$)
	A ($\langle Q4 \rangle \mathring{U} \star$)	A ($\star \mathring{U} \langle Q4 \rangle$)	A ($\star \mathring{U} \langle Q5 \rangle$)
	A ($\star \bar{U} \langle Q2 \rangle$)	A ($\star \bar{U} \langle Q3 \rangle$)	A ($\star \bar{U} \langle Q4 \rangle$)
	A ($\star \bar{U} \langle Q5 \rangle$)	A ($\langle Q3 \rangle \mathring{W} \star$)	A ($\langle Q4 \rangle \mathring{W} \star$)
	A ($\star \bar{W} \langle Q2 \rangle$)	A ($\star \bar{W} \langle Q3 \rangle$)	A ($\star \bar{W} \langle Q4 \rangle$)
	A ($\star \bar{W} \langle Q5 \rangle$)	$\star \wedge \langle Q1 \rangle$	$\star \vee \langle Q1 \rangle$
	AX $\langle Q1 \rangle$	A ($\langle Q1 \rangle \mathring{U} \star$)	A ($\star \bar{U} \langle Q1 \rangle$)
	A ($\langle Q1 \rangle \mathring{W} \star$)	A ($\star \bar{W} \langle Q1 \rangle$) ;	
$\langle Q2 \rangle ::=$	$\star \wedge \langle Q5 \rangle$	AX $\langle Q5 \rangle$	A ($\langle Q5 \rangle \mathring{U} \star$)
	A ($\star \mathring{U} \langle Q3 \rangle$)	A ($\langle Q5 \rangle \mathring{W} \star$)	A ($\star \mathring{W} \langle Q3 \rangle$)
	A ($\star \mathring{W} \langle Q4 \rangle$)	A ($\star \mathring{W} \langle Q5 \rangle$)	$\star \wedge \langle Q2 \rangle$
	AX $\langle Q2 \rangle$	A ($\langle Q2 \rangle \mathring{U} \star$)	A ($\langle Q2 \rangle \mathring{W} \star$) ;
$\langle Q3 \rangle ::=$	AF $\langle Q6 \rangle$	A ($\langle Q1 \rangle \mathbf{U} \star$)	A ($\langle Q2 \rangle \mathbf{U} \star$)
	A ($\langle Q4 \rangle \mathbf{U} \star$)	A ($\langle Q5 \rangle \mathbf{U} \star$)	A ($\langle Q6 \rangle \mathbf{U} \star$)
	A ($\langle Q7 \rangle \mathbf{U} \star$)	A ($\star \mathbf{U} \langle Q6 \rangle$)	$\star \vee \langle Q3 \rangle$
	AF $\langle Q3 \rangle$	A ($\langle Q3 \rangle \mathbf{U} \star$)	A ($\star \mathbf{U} \langle Q3 \rangle$) ;
$\langle Q4 \rangle ::=$	$\star \vee \langle Q5 \rangle$	AF $\langle Q5 \rangle$	AF $\langle Q7 \rangle$
	A ($\langle Q6 \rangle \mathring{U} \star$)	A ($\langle Q7 \rangle \mathring{U} \star$)	A ($\star \mathbf{U} \langle Q5 \rangle$)
	A ($\star \mathbf{U} \langle Q7 \rangle$)	A ($\star \mathring{U} \langle Q7 \rangle$)	A ($\star \bar{U} \langle Q6 \rangle$)
	A ($\star \bar{U} \langle Q7 \rangle$)	A ($\langle Q1 \rangle \mathbf{W} \star$)	A ($\langle Q2 \rangle \mathbf{W} \star$)
	A ($\langle Q3 \rangle \mathbf{W} \star$)	A ($\langle Q5 \rangle \mathbf{W} \star$)	A ($\langle Q6 \rangle \mathbf{W} \star$)
	A ($\langle Q7 \rangle \mathbf{W} \star$)	A ($\langle Q6 \rangle \mathring{W} \star$)	A ($\langle Q7 \rangle \mathring{W} \star$)
	A ($\star \mathbf{W} \langle Q3 \rangle$)	A ($\star \mathbf{W} \langle Q5 \rangle$)	A ($\star \mathbf{W} \langle Q6 \rangle$)
	A ($\star \mathbf{W} \langle Q7 \rangle$)	A ($\star \bar{W} \langle Q6 \rangle$)	A ($\star \bar{W} \langle Q7 \rangle$)
	$\star \vee \langle Q4 \rangle$	AF $\langle Q4 \rangle$	A ($\star \mathbf{U} \langle Q4 \rangle$)
	A ($\langle Q4 \rangle \mathbf{W} \star$)	A ($\star \mathbf{W} \langle Q4 \rangle$) ;	
$\langle Q5 \rangle ::=$	A ($\star \mathring{U} \langle Q6 \rangle$)	A ($\star \mathring{W} \langle Q6 \rangle$)	A ($\star \mathring{W} \langle Q7 \rangle$) ;

Fig. 3. The template grammar for distributive CTL queries (Part I).

$\langle Q6 \rangle ::=$	$\mathbf{A}(\langle Q8 \rangle \mathbf{U} \star)$	$\mathbf{A}(\langle Q9 \rangle \mathbf{U} \star)$	$\star \vee \langle Q6 \rangle$;
$\langle Q7 \rangle ::=$	$\star \wedge \langle Q6 \rangle$ $\mathbf{A}(\langle Q8 \rangle \mathbf{W} \star)$ $\star \vee \langle Q7 \rangle$	$\star \vee \langle Q8 \rangle$ $\mathbf{A}(\langle Q9 \rangle \mathbf{W} \star)$	$\star \vee \langle Q9 \rangle$ $\star \wedge \langle Q7 \rangle$;
$\langle Q8 \rangle ::=$	$\mathbf{AF} \langle Q9 \rangle$ $\mathbf{AG} \langle Q4 \rangle$ $\mathbf{A}(\star \mathbf{U} \langle Q9 \rangle)$ $\mathbf{A}(\star \mathbf{W} \langle Q9 \rangle)$ $\mathbf{AX} \langle Q8 \rangle$ $\mathbf{A}(\langle Q8 \rangle \mathring{\mathbf{U}} \star)$ $\mathbf{A}(\star \bar{\mathbf{U}} \langle Q8 \rangle)$ $\mathbf{A}(\star \bar{\mathbf{W}} \langle Q8 \rangle)$	$\mathbf{AG} \langle Q1 \rangle$ $\mathbf{AG} \langle Q6 \rangle$ $\mathbf{A}(\star \mathring{\mathbf{U}} \langle Q9 \rangle)$ $\mathbf{A}(\star \bar{\mathbf{W}} \langle Q9 \rangle)$ $\mathbf{AF} \langle Q8 \rangle$ $\mathbf{A}(\star \mathbf{U} \langle Q8 \rangle)$ $\mathbf{A}(\langle Q8 \rangle \mathring{\mathbf{W}} \star)$	$\mathbf{AG} \langle Q3 \rangle$ $\mathbf{AG} \langle Q7 \rangle$ $\mathbf{A}(\star \bar{\mathbf{U}} \langle Q9 \rangle)$ $\star \wedge \langle Q8 \rangle$ $\mathbf{AG} \langle Q8 \rangle$ $\mathbf{A}(\star \mathring{\mathbf{U}} \langle Q8 \rangle)$ $\mathbf{A}(\star \mathbf{W} \langle Q8 \rangle)$;
$\langle Q9 \rangle ::=$	$\mathbf{A}(\star \mathring{\mathbf{W}} \langle Q8 \rangle)$ $\mathbf{A}(\langle Q9 \rangle \mathring{\mathbf{U}} \star)$	$\star \wedge \langle Q9 \rangle$ $\mathbf{A}(\langle Q9 \rangle \mathring{\mathbf{W}} \star)$	$\mathbf{AX} \langle Q9 \rangle$ $\mathbf{A}(\star \mathring{\mathbf{W}} \langle Q9 \rangle)$;
$\langle Q10 \rangle ::=$	$\mathbf{AG} \langle Q2 \rangle$ $\star \wedge \langle Q10 \rangle$ $\mathbf{AF} \langle Q10 \rangle$ $\mathbf{A}(\langle Q10 \rangle \mathring{\mathbf{U}} \star)$ $\mathbf{A}(\star \bar{\mathbf{U}} \langle Q10 \rangle)$ $\mathbf{A}(\star \mathbf{W} \langle Q10 \rangle)$	$\mathbf{AG} \langle Q5 \rangle$ $\star \vee \langle Q10 \rangle$ $\mathbf{AG} \langle Q10 \rangle$ $\mathbf{A}(\star \mathbf{U} \langle Q10 \rangle)$ $\mathbf{A}(\langle Q10 \rangle \mathbf{W} \star)$ $\mathbf{A}(\star \mathring{\mathbf{W}} \langle Q10 \rangle)$	$\mathbf{AG} \langle Q9 \rangle$ $\mathbf{AX} \langle Q10 \rangle$ $\mathbf{A}(\langle Q10 \rangle \mathbf{U} \star)$ $\mathbf{A}(\star \mathring{\mathbf{U}} \langle Q10 \rangle)$ $\mathbf{A}(\langle Q10 \rangle \mathring{\mathbf{W}} \star)$ $\mathbf{A}(\star \bar{\mathbf{W}} \langle Q10 \rangle)$;
$\langle Q11 \rangle ::=$	$\mathbf{AF} \langle Q1 \rangle$ $\mathbf{A}(\star \mathbf{U} \langle Q2 \rangle)$ $\mathbf{A}(\star \mathbf{W} \langle Q1 \rangle)$ $\mathbf{A}(\star \mathring{\mathbf{W}} \langle Q2 \rangle)$ $\mathbf{AX} \langle Q11 \rangle$ $\mathbf{A}(\langle Q11 \rangle \mathbf{U} \star)$ $\mathbf{A}(\star \mathring{\mathbf{U}} \langle Q11 \rangle)$ $\mathbf{A}(\langle Q11 \rangle \mathring{\mathbf{W}} \star)$ $\mathbf{A}(\star \bar{\mathbf{W}} \langle Q11 \rangle)$	$\mathbf{AF} \langle Q2 \rangle$ $\mathbf{A}(\star \mathring{\mathbf{U}} \langle Q1 \rangle)$ $\mathbf{A}(\star \mathbf{W} \langle Q2 \rangle)$ $\star \wedge \langle Q11 \rangle$ $\mathbf{AF} \langle Q11 \rangle$ $\mathbf{A}(\langle Q11 \rangle \mathring{\mathbf{U}} \star)$ $\mathbf{A}(\star \bar{\mathbf{U}} \langle Q11 \rangle)$ $\mathbf{A}(\star \mathbf{W} \langle Q11 \rangle)$	$\mathbf{A}(\star \mathbf{U} \langle Q1 \rangle)$ $\mathbf{A}(\star \mathring{\mathbf{U}} \langle Q2 \rangle)$ $\mathbf{A}(\star \mathring{\mathbf{W}} \langle Q1 \rangle)$ $\star \vee \langle Q11 \rangle$ $\mathbf{AG} \langle Q11 \rangle$ $\mathbf{A}(\star \mathbf{U} \langle Q11 \rangle)$ $\mathbf{A}(\langle Q11 \rangle \mathbf{W} \star)$ $\mathbf{A}(\star \mathring{\mathbf{W}} \langle Q11 \rangle)$;

Fig. 4. The template grammar for distributive CTL queries (Part II).

C Reader and Writer Algorithms

This section contains the two semaphore-based reader and writer algorithms given in the paper [8]. Recall that the compound statements assume that all the variables of a local process are rolled up into a single variable and, similarly, all the variables of the central process are rolled up into a single variable. In the description below, however, we will make use of the real variable names for better readability.

```

pc = 1 : if m = 0
then goto pc = 2  $\wedge$  m = 1
else goto pc = 1

pc = 2 : if true
then goto pc = 3  $\wedge$  readcount = readcount + 1

pc = 3 : if readcount = 1
then goto pc = 4
else goto pc = 3

pc = 4 : if w = 0
then goto pc = 5  $\wedge$  w = 1
else goto pc = 4

pc = 5 : if true
then goto pc = 6  $\wedge$  m = 0

pc = 6 : if true
then goto pc = 7

pc = 7 : if m = 0
then goto pc = 8  $\wedge$  m = 1
else goto pc = 8

pc = 8 : if true
then goto pc = 9  $\wedge$  readcount = readcount - 1

pc = 9 : if readcount = 0
then goto pc = 10
else goto pc = 9

pc = 9 : if true
then goto pc = 10  $\wedge$  w = 0

pc = 10 : if true
then goto pc = 1  $\wedge$  m = 0

```

Fig. 5. Algorithm 1: Reader

```

pc = 1 : if w = 0
then goto pc = 2  $\wedge$  w = 1
else goto pc = 1

pc = 2 : if true
then goto pc = 3

pc = 3 : if true
then goto pc = 1  $\wedge$  w = 0

```

Fig. 6. Algorithm 1: Writer

D Description of Basic Actions for Cache Coherence

goto $st_{cen} = C$
The directory control variable st_{cen} is set to C
assign $ptr_i = ptr_j$ and assign $set_i = set_j$
The next value of the pointer variable ptr_i is set to the current value of ptr_j .
add ptr_i to set_j and remove ptr_i from set_j .
Add or remove the cache pointed to by ptr_i from set set_j .
pick ptr_i from \mathcal{S}^L [where \mathcal{S}^L is a list of (constant) cache control locations]
ptr_i is <i>non-deterministically</i> made to point to one of the caches with control location is in \mathcal{S}^L .
remote \mathcal{V} : goto $pc = L$ [where L is a cache control location and \mathcal{V} is a pointer variable]
Enforce the new control location L on the cache pointed to by \mathcal{V} .

```

pc = 1 : if m3 = 0
then goto pc = 2  $\wedge$  m3 = 1
else goto pc = 1

pc = 2 : if r = 0
then goto pc = 3  $\wedge$  r = 1
else goto pc = 2

pc = 3 : if m1 = 0
then goto pc = 4  $\wedge$  m1 = 1
else goto pc = 3

pc = 4 : if true
then goto pc = 5  $\wedge$  readcount = readcount + 1

pc = 5 : if readcount = 1
then goto pc = 6
else goto pc = 5

pc = 6 : if w = 0
then goto pc = 7  $\wedge$  w = 1
else goto pc = 6

pc = 7 : if true
then goto pc = 8  $\wedge$  m1 = 0

pc = 8 : if true
then goto pc = 9  $\wedge$  r = 0

pc = 9 : if true
then goto pc = 10  $\wedge$  m3 = 0

pc = 10 : if true
then goto pc = 11

pc = 11 : if m1 = 0
then goto pc = 12  $\wedge$  m1 = 1
else goto pc = 11

pc = 12 : if true
then goto pc = 13  $\wedge$  readcount = readcount - 1

pc = 13 : if readcount = 0
then goto pc = 14
else goto pc = 13

pc = 14 : if true
then goto pc = 15  $\wedge$  w = 0

pc = 15 : if true
then goto pc = 1  $\wedge$  m1 = 0

```

Fig. 7. Algorithm 2: Reader

```

pc = 1 : if m2 = 0
then goto pc = 2  $\wedge$  m2 = 1
else goto pc = 1

pc = 2 : if true
then goto pc = 3  $\wedge$  writecount = writecount + 1

pc = 3 : if writecount = 1
then goto pc = 4
else goto pc = 3

pc = 4 : if r = 0
then goto pc = 5  $\wedge$  r = 1
else goto pc = 4

pc = 5 : if true
then goto pc = 6  $\wedge$  m1 = 0

pc = 6 : if w = 0
then goto pc = 7  $\wedge$  w = 1
else goto pc = 6

pc = 7 : if true
then goto pc = 8

pc = 8 : if true
then goto pc = 9  $\wedge$  w = 0

pc = 9 : if m2 = 0
then goto pc = 10  $\wedge$  m2 = 1
else goto pc = 9

pc = 10 : if true
then goto pc = 11  $\wedge$  writecount = writecount - 1

pc = 11 : if writecount = 0
then goto pc = 12
else goto pc = 11

pc = 12 : if true
then goto pc = 13  $\wedge$  r = 0

pc = 13 : if true
then goto pc = 1  $\wedge$  m2 = 0

```

Fig. 8. Algorithm 2: Writer