Constraint-based Verification of Client-Server Protocols

Giorgio Delzanno[1] and Tevfik Bultan[2]

[1] Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, via Dodecaneso 35, 16146 Italy
e-mail: `giorgio@disi.unige.it`

[2] Department of Computer Science
University of California, Santa Barbara, CA 93106, USA
e-mail: `bultan@cs.ucsb.edu`

Abstract. We show that existing constraint manipulation technology incorporated in the paradigm of *symbolic model checking with rich assertional languages* [KMM$^+$97], can be successfully applied to the verification of client-server protocols with a finite but unbounded number of clients. *Abstract interpretation* is the mathematical bridge between protocol specifications and the constraint-based verification method on heterogeneous data used in the Action Language Verifier, a model checker for CTL [BYK01]. The method we propose is incomplete but fully automatic and sound for *safety* and *liveness* properties. Sufficient conditions for termination of the resulting procedures can be derived by using the theory of [ACJT96]. As a case-study, we apply the method to check safety and liveness properties for a formal model of Steve German's directory-based consistency protocol [PRZ01].

1 Introduction

Formal verification of client-server protocols is an important and challenging problem. Client-server architectures are present at different levels of abstractions in modern computer systems. Consistency protocols for client-server architectures are used, e.g., in multiprocessor systems with shared memory and local caches, distributed file systems, distributed database systems, and web-based applications to ensure the coherency of distributed data. An important class of consistency protocols makes use of central servers to serialize the access to the data. This kind of protocols are often validated on test sets with a fixed number of clients. In many interesting examples, however, it is not possible to fix an a priori bound on the number of clients requesting access to the data. This assumption makes the application of automated (push-button) verification methods like BDD-based *symbolic model checking* [McM93], and state exploration [Hol88] problematic. State explosion limits de facto the applicability of

The work by Tevfik Bultan is supported in part by NSF grant CCR-9970976 and NSF CAREER award CCR-9984822.

finite-state techniques like symbolic model checking to concurrent systems with a relatively small number of components. Thus, although useful for debugging, in general symbolic model checking cannot help us in automatically proving a protocol correct for any possible number of clients.

In the last years many efforts have been spent in order to lift symbolic model checking from finite- to infinite-state applications. Following [KMM$^+$97], this goal can be achieved by employing *rich assertional languages* to reason about potentially *infinite* collections of system states. This idea finds a natural counterpart in the paradigm of *constraint-based model checking*, see e.g. [BGP99,DP99,Fri00]. In this setting, the solutions of existentially quantified *constraint* formulas are used as denotations of an infinite collection of system states. Algorithmic verification procedures for temporal formulas are then defined on top of existing constraint-solvers such as a Presburger arithmetic solver as in [BGP99], and a real constraint solver as in [DP99].

In this paper we show that several verification problems of protocols designed for client-server architectures with a finite but potentially unbounded number of clients can be naturally solved using the *composite-constraint* approach proposed in [BGL00]. In this approach constraints over *heterogeneous* data are used as symbolic representation of states. The methodology we follow consists of the following steps.

We first specify the server and a generic client using finite-state communicating machines in the style of [BCR01,EN98,GS92,Del00]. In our model we allow synchronous and asynchronous communication mechanisms. Furthermore, we allow global variables with Boolean type. As main case-study, we present a formal model for the consistency protocol proposed by Steven German in [Ger00,PRZ01]. Many other examples can be modeled this way as shown, e.g., in [BCR01,Del00,EFM99]. The verification of the safety properties studied in [PRZ01] amounts to the following *parameterized reachability problem*: one has to show that for *any number of clients* unsafe states can never be reached.

Following the methodology proposed in [Del00], we apply a *counting abstraction* to reduce the family of communicating finite-state machines indexed on the number of clients to a transition system with *Boolean* and *integer variables*. Intuitively, the counting abstraction maps a *global state* (whose size depends on the number of clients) into a finite tuple of Boolean and integer values, in which we keep track of the current server state, the value of the global variables, and the number of clients in every possible local state. A formal model based on communicating finite-state machines can be compiled *automatically* into an abstract protocol using a set of rules mapping protocol transitions into guarded commands defined over Boolean, and integer variables. Via this abstraction, verification of safety properties can be reduced to a reachability problem in which initial and target states can be expressed as *composite constraints*, i.e., formulas over Boolean and integer variables. The Action Language Verifier [BYK01], a constraint-based CTL model checker, can then be used to attack this kind of verification problems. Action Language Verifier is built on top of the Composite Symbolic Library [YKTB01] which provides operations to manipulate composite constraints, by integrating a BDD library [CUDD], and a Presburger arithmetic manipulator [Pug92,KMP+95].

Using the theory proposed in [ACJT96], it is possible to prove the decidability of the resulting verification method for safety properties expressed via a special class of composite constraints in which the arithmetic part denotes *upward closed* sets of abstract states. Interestingly, the safety properties for the German's protocol considered in [PRZ01] can be expressed using this class of composite constraints.

As a practical result, we were able to automatically verify interesting safety properties like mutual exclusion for readers and writers for our case-study. Being a full-fledged model checker for temporal properties expressed in CTL, the Action Language Verifier also allowed us to automatically verify *liveness* properties. To our knowledge, this is the first time that constraint technology based on *composite symbolic representations* are used to verify formal models of client-server protocols for arbitrary number of clients.

Plan of the paper. In Section 2, we will informally describe our case-study. In Section 3, we will show how to formally specify it. In Section 4, we will introduce the *counting abstraction*. In Sections 5, 6, and 7 we will describe the tools we used to analyze the abstract protocol and the results we obtained. Finally, in Section 8, we will draw some conclusions and discuss related works.

2 A Consistency Protocol for Multi-client Systems

In this section we informally describe a *directory-based* consistency protocol for multi-client systems with sharing data (cache lines, memory pages, etc.) inspired by the protocol proposed by Steven German [Ger00] presented in [PRZ01]. The protocol is designed for a system consisting of a single *home node* and an arbitrary number of clients. The home node serializes requests for the data. A transaction begins when a client with *null* access rights sends a request either for *shared* or *exclusive* access to the home node. If the home node is not serving another request (it is *idle*), it can pick up a new request from one of the clients. The home node maintains the set of sharers identifiers, and the list of sharers that have to be invalidated before serving a given request. Furthermore, it uses an internal Boolean flag, we will call *ex*, to indicate whether or not *home granted exclusive access* to the data. When the home node is granting exclusive access, or granting shared access and there is a client with exclusive access right, the home node must invalidate all clients. The home node sends out invalidate messages to one client at a time. When a client in state *shared* or *exclusive* receives an invalidate message, it downgrades its access rights, and sends an acknowledgment back to the home node. The home node removes the client from the list of sharers when it gets the invalidate acknowledgment. When all necessary invalidations have been done, the home node sends a reply message to the client who made the request. A reply is either a grant of shared access or a grant of exclusive access.

The client updates its access when it receives a grant message from the home node. The protocol should ensure the following two *safety* properties (the first one is considered also in [PRZ01]): ($P1$) *at most* one process per time can obtain the *exclusive* access right; ($P2$) *exclusive* and *shared* access rights are mutually exclusive. The challenge here is to prove $P1$ and $P2$ for *any* number of clients. For this purpose, we will first turn the informal specification into a formal one.

3 Communicating Finite-state Machines

The specification language we propose is obtained by merging the asynchronous CCS-like model of [GS92] (one monitor, and many clients with asynchronous communication), the broadcast protocols of [EN98] (synchronous communication), the model used to specify cache coherence protocols of [Del00] (synchronous communication, conditions over the global state), and the global/local machines proposed in [BCR01] (asynchronous communication with global and local variables).

Global machines. A global machine is a tuple $\langle \mathcal{B}, Q_S, Q, \Sigma, \delta \rangle$, where: $\mathcal{B}$ is the tuple of *global Boolean variables*; Q_S is the finite set of states of the server; Q is the finite set of states of the local machines; and Σ is the set of synchronization labels used to build the set of possible actions $\mathcal{A}_\Sigma$ of a process. Specifically, let φ be a Boolean formula over $\mathcal{B}$ and $\mathcal{B}'$ (the primed version of the variables in $\mathcal{B}$). Then, an action has one of the following form:

- *Internal action:* $\ell : \varphi$ for $\ell \in \Sigma$.
- *Rendez-vous:* $!\ell : \varphi$ (send) and $?\ell$ (receive).
- *Broadcast:* $!!\ell : \varphi$ (send) and $??\ell$ (receive).

The Boolean formula φ is used to express pre-and post-conditions (using primed variables) on the global variables $\mathcal{B}$. In the rest of the paper, we will use ℓ to indicate the action $\ell : true$. We will clarify the semantics of actions in the next paragraphs. The behavior of the server and of the clients is described via the transition relation $\delta : (Q_S \times \mathcal{A}_\Sigma \times Q_S) \cup (Q \times \mathcal{A}_\Sigma \times Q)$. In the following, we will use $s \xrightarrow{\alpha} s'$ to indicate that $\langle s, \alpha, s' \rangle \in \delta$, and we will restrict δ to be deterministic. In order to define an operational semantics we must fix the number of clients, say k, as shown next. A *global state* for k clients is a tuple $G = \langle s_0, \rho, \mathbf{s} \rangle$, where $s_0 \in Q_S$ (server state), ρ is an evaluation for the variables in $\mathcal{B}$, and $\mathbf{s} = \langle s_1, \ldots, s_k \rangle$ (local states) is such that $s_i \in Q$ for $i : 1, \ldots, k$. The execution of a protocol is formalized through the relation $\Rightarrow$ defined next. Let $G = \langle s_0, \rho, \mathbf{s} \rangle$, $\mathbf{s} = \langle s_1, \ldots, s_k \rangle$, $G' = \langle s'_0, \rho', \mathbf{s}' \rangle$, and $\mathbf{s}' = \langle s'_1, \ldots, s'_k \rangle$. Define $\gamma = \rho \cup \rho'$. Then, $G \stackrel{\ell}{\Longrightarrow} G'$ provided one of the following conditions holds:

- $\exists i$ s.t. $s_i \xrightarrow{\ell:\varphi} s'_i$, $\gamma(\varphi) = true$.
- $\exists i, j$ s.t. $s_i \xrightarrow{!\ell:\varphi} s'_i$, $\gamma(\varphi) = true$, and $s_j \xrightarrow{?\ell} s'_j$.
- $\exists i$ s.t. $s_i \xrightarrow{!!\ell:\varphi} s'_i$, $\gamma(\varphi) = true$, and $\forall j$ s.t. δ is defined on $??\ell$, $s_j \xrightarrow{??\ell} s'_j$.

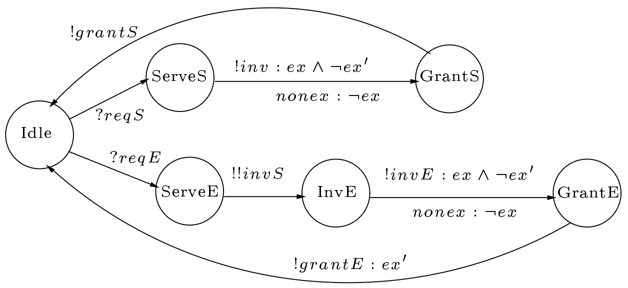

Fig. 1. Specification of the Home node.

In all the previous cases we assume that: $\rho'(b') = \rho(b)$ for any variable $b \in \mathcal{B}$ such that b' does not occur in the guard; and $s_i = s'_i$ if the i-th client is *not involved* in the action. A *run* of a global machine is a sequence of global states $G_0 G_1 \ldots$ such that $G_i \stackrel{\ell}{\Longrightarrow} G_{i+1}$ for $i \geq 0$. G_0 is the *initial* global state of the run. A global state G is reachable from G', written $G \stackrel{*}{\Rightarrow} G'$, if and only if there exists a run from G to G'.

3.1 A Formal Model for the Consistency Protocol

To specify our protocol, we use a Boolean variable *ex* (representing the flag *home granted exclusive right*), the machine for the home node described in Fig. 1, and the one for a generic client described in Fig. 2. Recall that the home node is supposed to serialize the requests and serve one client at a time. As in [Ger00,PRZ01], we consider message buffers of capacity at most one. Using synchronous communication, via the labels *reqE* and *reqS* we model the capability of the home node of storing the identifier of the client to be served: on reception of a request, home moves from *Idle* to one of the 'busy' states *ServeE*, *ServeS*. Differently from [Ger00,PRZ01], instead of handling invalidation via two global variables storing the identifiers of clients to be invalidated we use *broadcast* communication as explained below. Let us assume that the home node has to serve a request for *exclusive* access. Since all sharers must be invalidated, the server sends an *invalidation* broadcast to all clients in state *shared*. All sharers react to the broadcast downgrading their access rights. After having invalidated all sharers, home checks the flag *ex* to see if it still needs to invalidate clients with exclusive access. If the flag is on, instead of using broadcast, home assumes that only one process can be in *exclusive* state, and sends him the invalidation message *invE* using synchronous communication. The same situation repeats when the home node has to serve a request for *shared* access and the flag *ex* is on. On reception of the invalidation message, the client with exclusive access downgrades it to *null*. The flag *ex* is set to false (using the post-condition $\neg ex'$) after the invalidation process in state *ServeS* and *InvE*. The flag *ex* is set to true (using the post-condition ex') after granting exclusive access. When the home node is in state *ServeS* and *ex* is off, the server immediately grants shared access

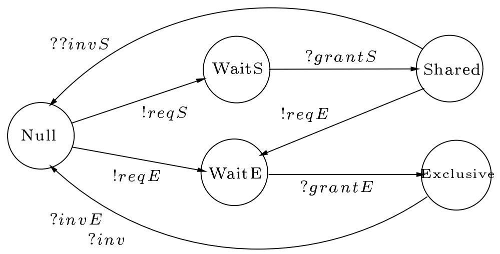

Fig. 2. Specification of a Client.

to the requesting client. In addition to the rule specified in [Ger00,PRZ01], we add the possibility for *sharers* to request an upgrade of their rights. This is accomplished via the transition $Shared \rightarrow ReqE$ labeled with $reqE$ in Fig. 2. The *initial global state* of the protocol with k clients is defined as $\langle Idle, \langle false \rangle, \mathbf{s} \rangle$, where $\mathbf{s}$ is the vector $\langle s_1, \ldots, s_k \rangle$ and $s_1 = \ldots = s_k = Null$.

Verification of Safety Properties. Let $G(k)$ denote a global state with k clients, and $\mathbb{B}_F(k)$ be the set of *unsafe* global states with k clients w.r.t. a given safety property F. Then, we say that the abstract protocol is k-safe if and only if there are no runs $G_o(k) \xrightarrow{*} G(k)$ such that $G(k) \in \mathbb{B}_F(k)$. In order to prove that a protocol is *safe* for all possible system configurations, it is necessary then to prove that it is k-safe for any $k \geq 1$. According to [EN98], we will call the reachability problem for arbitrary values of k a *parameterized reachability problem.* In our example $\mathbb{B}_{P1}$ and $\mathbb{B}_{P2}$ can be characterized as the sets of global states G *containing* the following *minimal violations*: $(P1)$ G contains one occurrence of *shared* and one of *exclusive*; $(P2)$ G contains two occurrences of *exclusive*. In other words, as often happens with safety properties, the set of unsafe states is *upward closed* (if a global state with k processes contains a violation generated by $k' < k$ processes, then it is unsafe). Furthermore, note that the description of unsafe states is independent from the *identifiers* of individual clients. In fact, we are not interested in proving that process 2 and process 6 are not violating mutual exclusion, we want to prove it for any pair of clients!

4 An Abstract Model

When trying to check safety properties that can be expressed independently from individual identifiers, it is often very useful to apply the following *counting abstraction.* The idea is to define an abstract state consisting of: (1) a *control* part obtained by merging the Boolean variables and the server control location; (2) a collection of counters to keep track of the number of clients in each local state $q \in Q$. Formally, let $G = \langle s, \rho, \mathbf{s} \rangle$ be a global state. The abstract state $G^{\#}$ is defined as: $G^{\#} = \langle s, \rho, \mathbf{c} \rangle$, where $\mathbf{c} = \langle c_1, \ldots, c_n \rangle$, and $c_i =$ *number of occurrences of* q_i in $\mathbf{s}$ for $i : 1, \ldots, n$, and $n = |Q|$. When applied to the transition relation δ, the counting abstraction returns the abstract protocol $M^{\#}$ that can be formally described as a transition system with Boolean and integer data paths. Formally, the abstract protocol consists of the control locations Q_S, the Boolean variables $\mathcal{B}$, and the *non-negative* integer variables $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$; x_i represents the *counter* of the number of clients in state $q_i \in Q$. In the rest of the paper we will often use x_q to denote the counter associated to state $q \in Q$. Abstract transitions are *guarded commands* $s \rightarrow s' : C$, where $s, s' \in Q_S$, and C is a formula defined over the variables in $\mathcal{B} \cup \mathcal{B}' \cup \mathbf{x} \cup \mathbf{x}'$ as follows.

- The *internal* action $s \xrightarrow{\ell:\varphi} t$ is compiled into the formula $\varphi \wedge x_s \geq 1 \wedge x'_s = x_s - 1 \ \wedge \ x'_t = x_t + 1$.
- The *rendez-vous* $p \xrightarrow{!\ell:\varphi} q$, $r \xrightarrow{?\ell} s$ (all states distinct each other) is compiled into the formula $\varphi \wedge x_p \geq 1 \wedge x_r \geq 1 \wedge x'_p = x_p - 1 \ \wedge \ x'_q = x_q + 1 \ \wedge \ x'_r = x_r - 1 \ \wedge \ x'_s = x_s + 1$.
- Finally, consider the *broadcast* $p \xrightarrow{!!\ell:\varphi} q$, $s_i \xrightarrow{??\ell} s$ for $i : 1 \ldots m$ (all states distinct each other). Then, δ is compiled into the formula $\varphi \wedge x_p \geq 1 \wedge x'_p = x_p - 1 \wedge x'_q = x_q + 1 \ \wedge \ x'_s = x_s + x_{s_1} + \ldots + x_{s_m} \ \wedge \ x'_{s_1} = 0 \wedge \ldots \wedge \ x'_{s_m} = 0$.

In all above cases additional constraints of the form $x'_s = x_s$ and $b' = b$ are implicitly assumed for all integer and Boolean variables that are not involved in any action. The transition system resulting from the application of the counting abstraction is a Vector Addition System with state (server location and Boolean variables), a model underlying the usual operational semantics of Petri Nets, extended with special *transfer* arcs associated to broadcast operations [EN98,EFM99,Del00]. Given two abstract states $G^\#_1 = \langle s, \rho, \mathbf{c} \rangle$ and $G^\#_2 = \langle s', \rho', \mathbf{c}' \rangle$, we say that $G^\#_1 \Rightarrow G^\#_2$ if and only if there exists a transition $s \rightarrow s' : C$ in $M^\#$ such that $C[\rho/\mathcal{B}, \rho'/\mathcal{B}', \mathbf{c}/\mathbf{x}, \mathbf{c}'/\mathbf{x}'] = \mathit{true}$. Given an abstract protocol $M^\#$ and an initial state $G^\#_0$, an abstract run is a sequence $G^\#_0, G^\#_1, \ldots$ of abstract states such that $G^\#_i \Rightarrow G^\#_{i+1}$ for $i \geq 0$. Then, we have the following proposition.

Proposition 1. *Let M be a global machine, and $M^\#$ be the corresponding abstract protocol. Then, $G_0 \stackrel{*}{\Rightarrow} G_1$ if and only if $G^\#_0 \stackrel{*}{\Rightarrow} G^\#_1$ for any G_0, G_1.*

The abstract protocol for the example of Section 2 is described by the transitions of Fig. 3 defined over the control locations $Idle$, $ServeS$, $ServeE$, $GrantE$, $InvE$ and $GrantS$, the Boolean variables ex, and the integer variables x_N for the client state $Null$, x_{WE} for $WaitE$, x_{WS} for $WaitS$, x_S for $Shared$, and x_E for $Exclusive$. It is important to note that the representation of abstract global states is *independent* from the number of clients, and that it fully exploits the symmetries in their behavior. To check properties $P1$ and $P2$, it remains now to describe our approach to attack the reachability problem coming out from Proposition 1.

5 Composite Symbolic Representation

In order to analyze the behavior of a protocol for any possible number of *clients*, we need a finite representation for *infinite* collections of abstract states. One

$(reqS)$	$Idle \rightarrow ServeS:$	$x_N \geq 1 \wedge x'_N = x_N - 1 \wedge x'_{WS} = x_{WS} + 1$
$(reqE)$	$Idle \rightarrow ServeE:$	$x_N \geq 1 \wedge x'_N = x_N - 1 \wedge x'_{WE} = x_{WE} + 1$
$(reqE)$	$Idle \rightarrow ServeE:$	$x_S \geq 1 \wedge x'_S = x_S - 1 \wedge x'_{WE} = x_{WE} + 1$
(inv)	$ServeS \rightarrow GrantS:$	$ex \wedge \neg ex' \wedge x_E \geq 1 \wedge x'_E = x_E - 1 \wedge x'_N = x_N + 1$
$(nonex)$	$ServeS \rightarrow GrantS:$	$\neg ex$
$(invS)$	$ServeE \rightarrow InvE:$	$x'_N = x_N + x_S \;\wedge\; x'_S = 0$
$(invE)$	$InvE \rightarrow GrantE:$	$ex \;\wedge\; \neg ex' \;\wedge x_E \geq 1 \wedge x'_N = x_N + 1 \wedge x'_E = x_E - 1$
$(nonex)$	$InvE \rightarrow GrantE:$	$\neg ex$
$(grantS)$	$GrantS \rightarrow Idle:$	$x_{WS} \geq 1 \wedge x'_{WS} = x_{WS} - 1 \wedge x'_S = x_S + 1$
$(grantE)$	$GrantE \rightarrow Idle:$	$x_{WE} \geq 1 \wedge x'_{WE} = x_{WE} - 1 \wedge x'_E = x_E + 1 \wedge ex'$

Fig. 3. Abstract client-server protocol.

such representation could be obtained using linear constraints to encode sets tuples of integer and Boolean values. However, since manipulation of arithmetic constraints is expensive this strategy is not likely to scale. To solve this problem, we will use the *composite constraints* of [BGL00] as symbolic representation of infinite collections of global states. To explain this idea, let us first introduce a new set $\mathcal{L}$ of Boolean variables, which will be used to encode the control locations Q_S of the server; if $|Q_S| = m$, then we need $log_2 m$ variables. In our setting, a composite constraint is a formula $\varphi_{bool} \wedge \varphi_{int}$, where φ_{bool} is a Boolean formula over the Boolean variables $\mathcal{B} \cup \mathcal{L}$, and φ_{int} is a *disjunction* of linear arithmetic constraints over the variables $\mathbf{x}$ of the abstract protocol. The denotation of a composite constraint is defined as follows:

$$[\![\varphi_{bool} \wedge \varphi_{int}]\!] = \{\langle s, \rho, \mathbf{c}\rangle \mid \varphi_{bool} \text{ is true in } \rho_s \cup \rho, \text{ and } \varphi_{int} \text{ is satisfied in } \mathbf{c}\},$$

where ρ_s is the evaluation of variables $\mathcal{L}$ encoding location $s \in Q$. Composite constraints allow us to finitely and compactly represent *initial* and *unsafe states* for *parameterized* verification problems that can be formulated independently from client identifiers. As an example, the initial configuration of our protocol is described as the composite constraint Φ_0 defined as $\varphi_{Idle} \wedge \neg ex \wedge x_N \geq 1 \wedge x_S = 0 \wedge x_E = 0 \wedge x_{WE} = 0 \wedge x_{WS} = 0$, where φ_{Idle} is the Boolean formula over $\mathcal{L}$ representing location $Idle$. Furthermore, the set of potential violations of the *mutual exclusion* properties $P1$ and $P2$ can be represented as $\Phi_1 \vee \Phi_2$ where $\Phi_1 = x_S \geq 1 \wedge x_E \geq 1$, and $\Phi_2 = x_E \geq 2$. Based on this observation, it follows that we can reduce the verification problem for M and properties $P1$ and $P2$ to the following *reachability* problem for $M^{\#}$: *For any $G_0^{\#} \in [\![\Phi_0]\!]$, there are no runs $G_0^{\#} \stackrel{*}{\Rightarrow} G^{\#}$ of $M^{\#}$, such that $G^{\#} \in [\![\Phi_1 \vee \Phi_2]\!]$.*

Based on this idea, we encode collections of abstract states of the protocol using *composite symbolic representations* which are disjunctions of composite constraints [BGL00]. Formally, a composite symbolic representation Φ is in the form:

$$\Phi = \bigvee_i \Phi_i = \bigvee_i \varphi_{bool_i} \wedge \varphi_{int_i}$$

where each φ_{bool_i} is a Boolean formula, and each φ_{int_i} is a disjunction (set) of linear arithmetic constraints as mentioned above. Each φ_{int_i} can be represented in a disjunctive form as $\varphi_{int_i} = \bigvee_j \; \varphi_{int_{ij}}$ where $\varphi_{int_{ij}} = \bigwedge_k c_{ijk}$ and each c_{ijk} is an atomic linear constraint. Operations on arithmetic and Boolean constraints can be used to implement a symbolic *predecessor* operator **Pre** that computes the effect of firing the transition of an abstract protocol backwards on a composite symbolic representation. We first note that we can represent a guarded command t of $M^{\#}$ via the composite constraint φ_t defined over the variables $\mathcal{B}, \mathcal{L}, \mathbf{x}$ and their primed versions $\mathcal{B}', \mathcal{L}', \mathbf{x}'$ ($\mathcal{L}$ and $\mathcal{L}'$ are used to represent the old and the new control locations, respectively). Based on this observation, $Pre_t(\Phi)$ is defined as the existentially quantified formula (with variables in $\mathcal{L}, \mathcal{B}, \mathbf{x}$) defined as follows:

$$Pre_t(\Phi) \;\equiv\; \exists \mathcal{L}'.\exists \mathcal{B}'.\exists \mathbf{x}'.\; \varphi_t(\mathcal{L}, \mathcal{B}, \mathbf{x}, \mathcal{L}', \mathcal{B}', \mathbf{x}') \wedge (\bigvee_i \varphi_{bool_i}(\mathcal{L}', \mathcal{B}') \wedge \varphi_{int_i}(\mathbf{x}'))$$

Since existential quantification distributes over the disjunction we get

$$Pre_t(\Phi) \;\equiv\; \bigvee_i Pre_t(\Phi_i).$$

By hypothesis, Boolean and arithmetic constraints have no variables in common. Thus, the existential quantification also distributes over the conjunction to obtain

$$Pre_t(\Phi_i) \;\equiv\; (\exists \mathcal{L}'.\exists \mathcal{B}'.\; \varphi'_{bool_i}) \wedge (\exists \mathbf{x}'.\; \varphi'_{int_i}),$$

where φ'_{bool_i} and φ'_{int_i} are obtained collecting together, respectively, the Boolean and arithmetic constraints of $\varphi_t, \varphi_{bool_i}$, and φ_{int_i}. Furthermore, we can distribute the existential quantification over the set of linear arithmetic constraints in $\varphi'_{int_i} = \bigvee_j \; \varphi'_{int_{ij}}$ such that:

$$Pre_t(\Phi_i) \;\equiv\; (\exists \mathcal{L}'.\exists \mathcal{B}'.\; \varphi'_{bool_i}) \wedge (\bigvee_j (\exists \mathbf{x}'.\varphi'_{int_{ij}}))$$

Eliminating $\mathbf{x}'$ amounts to replacing every primed variable with its definition in φ_t. Hence, if $\varphi_{int_{ij}}$ is a set of linear constraints so is $\exists \mathbf{x}'.\varphi_{int_{ij}}$. The symbolic predecessor operator associated with $M^{\#}$ is defined then as:

$$\mathbf{Pre}(\Phi) = \bigvee_{t \in M^{\#}} Pre_t(\Phi).$$

The operator preserves our composite symbolic representation. Furthermore, it is easy to check that

$$[\![\mathbf{Pre}(\Phi)]\!] = \{G_1^{\#} \mid G_1^{\#} \Longrightarrow G_2^{\#},\; G_2^{\#} \in [\![\Phi]\!]\}.$$

Symmetrically, it is possible to define a symbolic *successor* operator **Post** such that **Post**(Φ) returns the set of abstract states reachable from Φ (we omit its definition for brevity).

Symbolic forward and backward exploration procedures can be implemented then using the **Pre** and **Post** operators. The symbolic forward exploration procedure works on a composite symbolic representation $Current$. Given an initial set of composite constraints Φ_0, we first set $Current := \Phi_0$. Then, we apply **Post** to all the constraints in $Current$ to compute a new set of constraints New. If each composite constraint in New entails $Current$, we stop. Otherwise we add the composite constraints in New to $Current$ and continue. Symbolic backward exploration can be implemented similarly by starting from the composite constraints representing *unsafe* states, and using the **Pre** operator at each step.

In order to keep the *number of disjuncts* generated during a fixpoint computation small, it is possible to use simplification rules as the ones used in the Composite Symbolic Library described in the next section: for each composite constraint $\varphi_{bool_i} \wedge \varphi_{int_i}$ it checks if φ_{bool_i} is satisfiable, and removes the constraint if it is not; it looks for composite constraints $\varphi_{bool_i} \wedge \varphi_{int_i}$ and $\varphi_{bool_j} \wedge \varphi_{int_j}$ such that $[\![\varphi_{bool_i}]\!] = [\![\varphi_{bool_j}]\!]$, and merges them to form one composite constraint $\varphi_{bool_i} \wedge (\varphi_{int_i} \vee \varphi_{int_j})$. Since the boolean part of the composite constraint allows efficient equivalence and satisfiability checks (as is the case for BDDs) these simplification operations can be implemented efficiently and applied after each step in the symbolic forward and backward exploration procedures.

Interestingly, symbolic backward and forward exploration are not equivalent. As shown in [EFM99], symbolic forward exploration (enriched with acceleration operators *à la* Karp and Miller [EN98]) may not terminate for transition systems associated to *broadcast* protocols (a subclass of global machines). On the other hand, symbolic backward exploration is always guaranteed to terminate when the seed of the exploration is a constraint representing *upward-closed* set of abstract states. We will discussed this point in the next section.

Conditions for Termination. One interesting class of linear constraints that can be used to represent set of unsafe states with the special property of being *upward-closed* is that of *additive constraints* considered in [DEP99]. An additive constraint consists of a conjunction of atomic formulas of the form $a_1 \cdot y_1 + \ldots a_n \cdot y_n \geq c$, where a_i is a nonnegative integer constant, and y_i is a variable ranging over non-negative integers. As shown in [DEP99], the class of additive constraints equipped with the usual notion of entailment between linear constraints form a *well-quasi ordering*. This implies that there cannot be infinite chains of additive constraints whose elements are not comparable to each other with respect to the entailment relation. *Composite additive constraints* are obtained by restricting the linear arithmetic part of a composite constraint to be additive. Composite additive constraints are closed under application of the symbolic operator **Pre** associated to an abstract protocol. As a consequence, we have the following result.

Proposition 2. *Let Φ a* composite constraint representation. *Then, the symbolic backward exploration algorithm taking Φ as seed of the computation terminates and computes a symbolic representation of $Pre^*([\![\Phi]\!])$.*

Note that the composite constraints representing the unsafe states associated to property $P1$ and $P2$ of Section 3.1 are composite additive constraints. As a consequence, we have the following corollary.

Corollary 1. *The verification of properties $P1$ and $P2$ for the protocol of Section 3.1 is decidable.*

In the next section we will discuss the practical issues related to our methodology.

6 Tool Support for the Composite Constraint Method

The *Composite Symbolic Library* uses an object-oriented design to combine different assertional languages [YKTB01]. An abstract interface defines the operations used in symbolic verification: Boolean operations, equivalence and entailment tests, and image computations (for the **Pre** and **Post** operators). To define a new assertional language one simply has to implement the abstract interface with specialized operations. Currently, the Composite Symbolic Library provides two basic symbolic representations: BDDs via the Colorado University Decision Diagram Package (CUDD) [CUDD], and linear integer arithmetic constraints via the Omega Library [KMP+95]. Operations on composite symbolic representation are implemented using corresponding operations on these basic symbolic representations [BGL00]. The object-oriented design of the Composite Symbolic Library makes it possible to write polymorphic verification procedures, i.e. verification procedures that dynamically select symbolic representations based on the input specification. The input language of the Composite Symbolic Library is called the *Action Language* [Bul00]. Action Language is a specification language for reactive software systems which supports both synchronous and asynchronous compositions and hierarchical specifications; currently, it supports Boolean, enumerated, and integer types. In this setting, a specification consists of a set of modules and atomic actions. Modules can be defined by composing other modules or actions using synchronous or asynchronous compositions. Atomic actions are defined using formulas on primed and unprimed variables as in our abstract protocol example. In action formulas only Boolean logic and linear arithmetic operators are allowed. Given an input specification, the Action Language Verifier [BYK01] translates the input specification to a composite constraint representation and checks the verification conditions by computing forward or backward fixpoints using the Composite Symbolic Library. Verification conditions are specified in temporal logic CTL.

In general, for the class of systems that can be specified in the Action Language CTL model checking is undecidable. To achieve convergence, one can use conservative approximation techniques. Such operations have been successfully used for the verification of infinite-state systems using linear-arithmetic constraints (see e.g. [BGP99]). The Action Language Verifier extends these results to composite symbolic representations. Specifically, it implements a generalization of the widening operation on convex polyhedra to compute upper-bounds for fixpoints which do not converge. It also uses truncated fixpoint computations to compute lower-bounds. Using both these techniques, it is possible to compute both *lower* and *upper approximations* for any CTL property.

The Composite Symbolic Library and the Action Language Verifier are available at the URL `http://www.cs.ucsb.edu/~bultan/composite/`.

7 Experimental Results

In our experiments we focused on two kinds of CTL formulas: safety properties expressing *mutual exclusion* and liveness properties expressing *freedom from starvation*. In general, a *safety property*, expressed via the CTL formula $AG(\Phi)$, holds whenever all reachable states belong to the set of *safe* states Φ. Clearly, it can be proved by contraposition, by showing that there are no reachable states that belong the set of unsafe states $\neg\Phi$, in CTL this corresponds to the following equivalence $AG(\Phi) = \neg EF(\neg\Phi)$. Furthermore, one can show that $EF(\Psi) = Pre^*(\Psi)$ for any Ψ. This implies that AG-properties can be verified by first using symbolic backward reachability with seed $\neg\Phi$ to compute EF, and checking that the initial states are not in the resulting set of states. In CTL it is possible to express more complicated formulas like the liveness property $AG(\Phi_1 \rightarrow AF(\Phi_2))$. This formula can be read as follows: if Φ_1 holds at state s, then Φ_2 must eventually hold in all executions starting at s. Liveness properties can be checked algorithmically via nested *greatest* and *least fixpoint computations.* Verification of liveness for Vector Addition Systems with transfers arcs (or with test for zero in the guards) is *undecidable* [EFM99]. However, constraint-based model checkers can still be used as incomplete verification procedures using heuristics and approximation techniques to enforce termination [BGP99].

The table in Fig. 4 summarizes the practical results we obtained via the Action Language Verifier. We performed our experiments on two different models of the client-server protocol described in Section 2. The model 'B' of Fig. 4 is the abstract protocol of Fig. 3. The model 'I' is a refinement of model 'B' in which the atomic invalidation broadcast is replaced by the *invalidation loop* formulated at the abstract level as shown in Fig. 5. Note that this formulation needs *guards* with tests for zero. Tests for zero break the decidability of the verification of safety properties [Del00], i.e., approximations might be necessary in order to verify AG-formulas for the model 'I'. For both models, we considered the CTL properties listed in Fig. 6. The parameters of the experimental evaluation were the following: 'UA' denotes the use of approximations in the fixpoint computations; 'UF' denotes the use of approximate forward state exploration (see explanation below); 'Strategy' denotes the strategy used to check the properties, namely the sequence of steps (f=forward exploration, EF=least fixpoint, EG=greatest fixpoint) annotated with the number of iterations needed for each of them, e.g., $EF(4)$ means that the least fixpoint is reached in four iterations;

Property	Model	UA	UF	Strategy	Memory (Mbytes)	Time (secs.)
$P1-2$	B			$EF(4)$	10.2	0.60
$P1-2$	B		$\surd$	f(7), $EF(1)$	9.7	0.52
$P3$	B			$EG(3)$, $EF(5)$	14.1	2.37
$P3$	B		$\surd$	f(7), $EG(3)$, $EF(1)$	10.7	0.68
$P4$	B			$EG(3)$, $EF(8)$	25.6	9.34
$P4$	B		$\surd$	f(7), $EG(3)$, $EF(1)$	11.2	0.74
$P5$	B	$\surd$		$EG(4)$, $EF(11)$	14.1	3.01
$P5$	B	$\surd$	$\surd$	f(7), $EG(3)$, $EF(1)$	10.4	0.61
$P1-2$	I			$EF(4)$	10.4	0.59
$P1-2$	I		$\surd$	f(6), $EF(1)$	9.8	0.50
$P3$	I	$\surd$		$EG(3)$, $EF(5)$	11.9	2.01
$P3$	I		$\surd$	f(6), $EG(3)$, $EF(1)$	10.6	0.65
$P4$	I		$\surd$	f(6), $EG(3)$, $EF(1)$	11.6	0.81

Fig. 4. Experimental results obtained on a SUN ULTRA 10 workstation with 768 Mbytes of main memory, running SunOS 5.7.

$$ServeE \to GrantE : x_S = 0 \land x_E = 0$$
$$ServeE \to ServeE : \ x_S \geq 1 \land x'_N = x_N + 1 \land x'_S = x_S - 1$$
$$ServeE \to ServeE : \ x_E \geq \land x'_N = x_N + 1 \land x'_E = x_E - 1 \land \neg ex'$$

Fig. 5. The invalidation loop in state $ServeE$.

'Memory' and 'Time' denote the total resource consumption for the application of the corresponding strategy.

Some explanations for Fig. 4 are in order. Let us start from the model 'B'. As expected, we verified the safety properties $P1$ and $P2$ of Section 2, i.e., the CTL formula $P1-2$ of Fig. 6 without need of any approximation. We also verified the liveness properties $P3$ and $P4$ without using approximations. Since our method works on abstract models in which we forget identifiers of clients, the liveness property $P4$ must be read as *if more than k clients are waiting, then at least k clients will get the desired access*, i.e., a sort of freedom from *global deadlocks* for the original concrete protocol. Fixpoint computations for the liveness property $P5$ does not converge without approximation techniques. However, we were able to prove the property using truncated fixpoint computations and widening.

We also investigated the use of an a priori *forward* exploration of the abstract protocol reachable states (indicated as 'UF' in Fig. 4). Specifically, using widening techniques we first computed an *over-approximation* of the set of reachable states, and then used it to restrict the search-space during backward reachability. As an example, for model 'B' the approximate forward exploration (indicated as 'f' in Fig. 4) allowed us to verify all the properties faster, e.g., $P1-2$ in one iteration instead of four. By *caching* of the approximated reachable set, it should be possible to further improve the execution times of Fig. 4.

Property	Property specification in CTL
$P1-2$	$\neg EF((x_S \geq 1 \wedge x_E \geq 1) \;\; \vee \;\; x_E \geq 2)$
$P3$	$AG(x_{WS} \geq 1 \;\; \rightarrow \;\; AF(x_S \geq 1))$
$P4$	$AG(x_{WS} \geq N \;\; \rightarrow \;\; AF(x_S \geq N)),\ N \geq 1$
$P5$	$AG(x_{WE} \geq 1 \;\; \rightarrow \;\; AF(x_E \geq 1))$

Fig. 6. Specification of the properties for our case-study.

Let us consider now the model with invalidation loop. Again, to verify $P1-2$ we needed no approximations (however, note that termination is not guaranteed in this case). For property $P3$, the innermost fixpoint converged in three iterations, whereas the outermost fixpoint diverged without approximations. Using approximation techniques both fixpoints converged and we were able to verify the property. One interesting result is that we were able to verify property $P3$ without using approximations in the backward fixpoints when we combined them with the approximate forward exploration. The approximate forward exploration for model 'I' converges in six iterations and using it we can verify $P3$ more efficiently. For property $P4$, as with $P3$, the innermost fixpoint converged in three iterations, whereas the outermost fixpoint diverged without approximations. However, when we used approximation techniques although the fixpoint computations converged the results were not strong enough to verify the property. When we used approximate forward exploration backward fixpoint computations converged and we were able to verify the property. When we tried the to verify property $P5$ for model 'I', inner fixpoint computation did not converge. When we used approximations the results were not strong enough to verify the property. Even when we used approximate forward exploration, the results did not change. Hence, we were not able to verify or falsify the property $P5$ for model 'I'.

8 Conclusions and Related Works

In this paper we have shown that existing constraint technology can be successfully applied to the formal verification of protocols parametric on the number of participants. Abstract interpretation works as a bridge between protocol specifications and models that can be handled via constraint-based verification methods working on heterogeneous data like the Action Language Verifier of [BYK01]. The *counting abstraction* has been introduced in [GS92], where families of asynchronous CCS processes were verified via a reduction to Petri Nets. In [DEP99,Del00], a similar abstraction has been applied to the verification of cache coherence protocols (but not to directory-based as the protocol of [Ger00]), and concurrent systems specified as broadcast protocols [EN98]. The specification in [DEP99,Del00] allows synchronous communication but it does not admits heterogeneous data like global Boolean variables.

In [BCR01], the counting abstraction has been applied for the verification of *skeletons* of *multi-threaded* libraries. The resulting abstract models are basically Petri Nets with state. The authors analyze them using the Karp-Miller coverability construction, i.e., forward exploration with accelerations [EN98]. However, this procedure is not guaranteed to terminate in presence of broadcast communication [EFM99].

In [PRZ01], an alternative method based on deductive verification has been used to verify *safety properties* of parameterized systems like the German's protocol we considered in this paper. The method of [PRZ01] uses heuristics to discover invariants for parameterized systems, and to verify that the discovered invariants are *inductive*. The method is incomplete, but fully automatic (it is based on BDDs) and sound for *safety properties*. Differently from the previously mentioned approaches, constraint-based tools like the Action Language Verifier represents an incomplete but fully automatic sound tool for checking full *CTL formulas*. We exploited this feature to automatically verify *new* safety (property $P2$ has not been studied in [PRZ01]) and *liveness* properties for our case-study. On the other hand, the specification language used in [PRZ01] allows one to associate complex data structures, e.g. arrays storing process identifiers, to individual processes. Extending our approach in order to handle parameterized system with this kind of data structures seems an interesting direction of future research.

References

[ACJT96] P. A. Abdulla, K. Cerāns, B. Jonsson and Y.-K. Tsay. General Decidability Theorems for Infinite-State Systems. In *Proc. LICS '99*, pp. 313-321, 1996.

[BCR01] T. Ball, S. Chaki, S. K. Rajamani. Parameterized Verification of Multi-threaded Software Libraries. In *Proc. TACAS '01*, LNCS 2031, pp. 158-173, 2001.

[BGP99] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM TOPLAS*, 21(4):747–789, 1999.

[BGL00] T. Bultan, R. Gerber, C. League Composite model-checking: verification with type-specific symbolic representations. *ACM TOSEM*, 9(1): 3-50, 2000.

[Bul00] T. Bultan. Action Language: A specification language for model checking reactive systems. In *Proc. ICSE '00*, pp. 335–344, 2000.

[BYK01] T. Bultan and T. Yavuz-Kahveci. Action Language Verifier. In *Proc. ASE '01*, 2001.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. POPL '77* pp. 238-252, 1977.

[CGP99] E. M. Clarke, O. Grumberg, D. Peled. Model Checking. MIT Press, December 1999.

[CUDD] Fabio Somenzi. CUDD: the CU Decision Diagram Package, Release 2.3.1. `http://vlsi.colorado.edu/~fabio/cudd/`.

[Del00] G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In *Proc. CAV '00*, LNCS 1855, pp. 53-68, 1996.

[DEP99] G. Delzanno, J. Esparza, and A. Podelski. Constraint-based Analysis of Broadcast Protocols. In *Proc. CSL '99*, LNCS 1683, pp. 50-66, 1999.

[DP99] G. Delzanno, A. Podelski. Model Checking in CLP. In *Proc. TACAS '99*, LNCS 1579, pp. 223-239, 1999.

[EN98] E. A. Emerson and K. S. Namjoshi. On Model Checking for Non-deterministic Infinite-state Systems. In *Proc. LICS '98*, pp. 70–80, 1998.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proc. LICS '99*, pp. 352–359, 1999.

[Fri00] L. Fribourg. Constraint Logic Programming Applied to Model Checking. In *Proc. LOPSTR '99*, LNCS 1817, pp. 30–41, 1999.

[Ger00] S. M. German. Personal communication.

[GS92] S. M. German, A. P. Sistla. Reasoning about Systems with Many Processes. *JACM* 39(3): 675–735 (1992)

[Hal93] N. Halbwachs. Delay Analysis in Synchronous Programs. In *Proc. CAV '93*, LNCS 697, pp. 333–346, 1993.

[Hol88] G. Holzmann Algorithms for Automated Protocol Verification. AT&T Technical Journal 69(2):32-44, 1988.

[KMP+95] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, March 1995. See also `http://www.cs.umd.edu/projects/omega/`.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proc. CAV '97*, pp. 424-435, 1997.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic, 1993.

[PRZ01] A. Pnueli, S. Ruah, L. D. Zuck Automatic Deductive Verification with Invisible Invariants. In *Proc. TACAS '01*, LNCS 2031, pp. 82-97, 2001.

[Pug92] W. Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 8:102-114, 1992.

[YKTB01] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A Library for Composite Symbolic Representations. In *Proc. TACAS '01*, LNCS 2031, pp. 52-66, 2001.