Automatic Verification of Directory-Based Consistency Protocols

Parosh Aziz Abdulla[1], Giorgio Delzanno[2], and Ahmed Rezine[3]

[1] Uppsala University, Sweden
parosh@it.uu.se
[2] Università di Genova, Italy
giorgio@disi.unige.it
[3] University of Paris 7, France
rezine.ahmed@liafa.jussieu.fr

Abstract. We propose a symbolic verification method for directory-based consistency protocols working for an arbitrary number of controlled resources and competing processes. We use a graph-based language to specify in a uniform way both client/server interaction schemes and manipulation of directories that contain the access rights of individual clients. Graph transformations model the dynamics of a given protocol. Universally quantified conditions defined on the labels of edges incident to a given node are used to model inspection of directories, invalidation loops and integrity conditions. Our verification procedure computes an approximated backward reachability analysis by using a symbolic representation of sets of configurations. Termination is ensured by using the theory of well-quasi orderings.

1 Introduction

Several implementations of consistency and integrity protocols used in file systems, virtual memory, and shared memory multi-processors are based on client-server architectures. Clients compete to access shared resources (cache and memory lines, memory pages, open files). Each resource is controlled by a server process. In order to get access to a resource, a client needs to start a transaction with the corresponding server. Each server maintains a directory that associates to each client the access rights for the corresponding resource. In real implementations these information are stored into arrays, lists, or bitmaps and are used by the server to take decisions in response to client requests, e.g., to grant access, request invalidation, downgrade access mode or to check integrity of meta-data as in programs like fsck used to check Unix-like file-systems. Typically, a server handles a set of resources, e.g. cache lines and directory entries, whose cardinality depends on the underlying hardware/software platform. Consistency protocols however are often designed to work well independently from the number of resources to be controlled, i.e., independently from a given hardware/software configuration.

The need of reasoning about systems with an arbitrary number of resources makes verification of directory-based consistency protocols a quite challenging

O. Bournez and I. Potapov (Eds.): RP 2009, LNCS 5797, pp. 36–50, 2009.

task in general. Abstraction techniques operating on the number of resources and/or the number of clients are often applied to reduce the verification task to decidable problems for finite-state (e.g. invisible and environment abstraction in [9,13]) or Petri net-like models (e.g. counting abstraction used in [16,14,24,25]).

In this paper we propose a new approximated verification technique that operates on models in which both the number of controlled resources and of competing clients is not fixed a priori. Instead of requiring a preliminary abstraction of the model, our method makes use of powerful symbolic representations of parametric system configurations and of dynamic approximation operators applied during symbolic exploration of the state-space.

Our verification method is defined for a specification language in which system configurations are modelled by using a special type of graphs in which vertexes are partitioned into client and server nodes. Client and server nodes are labelled with a set of "states" that represent the current state of the corresponding processes. Labelled edges are used both to define client/server transactions and to describe the local information maintained by each server (e.g. a directory is represented by the set of edges incident to a given server node).

Protocol rules are specified here by rewriting rules that update the state of a node and of one of its incident edges. This very restricted form of graph rewriting naturally models asynchronous communication mechanisms. Furthermore, we admit here guards defined by means of universally quantified conditions on the set of labels of edges of a given node. This kind of guards is important to model operations like inspection of a directory or invalidation cycles without need of abstracting them by means of atomic operations like broadcast in [16,14]. In order to reason about *parameterized formulations* of consistency protocols we consider here systems in which the size of graphs (number of nodes and edges) is not bounded a priori.

The advantage of working with conditional graph rewriting is twofold. On one side it gives us enough power to formally describe each step of consistency protocols like the full-map coherence protocol [20] in a very detailed way. On the other side it allows us to define (and implement) our verification method at a very abstract level by using graph transformations.

Related Work. Parameterized verification methods based on finite-state abstractions have been applied to safety properties of consistency protocols and mutual exclusion algorithms. Among these, we mention the *invisible invariants* method [9,21] and the *environment abstraction* method [13]. Counting abstraction and Petri net-like analysis techniques are considered, e.g., in [16,14,24,25].

Differently from all the previous works, our algorithm is based on graph constraints that allow us to symbolically represent infinite-sets of configurations without need of preliminary finitization of parameters like number of clients, servers, resources, and size of directories. We apply instead dynamic approximation techniques to deal with universally quantified global conditions. We recently used a similar approach for systems with flat configurations (i.e. words) and with a single global context [7]. The new graph-based algorithm is a generalization of the approach in [7]. Indeed, the symbolic configurations we used in [7] can be viewed as graphs with a single server node and no edges, since global conditions are tested directly on the current process states.

Furthermore, the approximation we propose in this paper is more precise than the monotonic abstraction used to deal with global conditions in our previous work [3,4,5,6,1] (i.e. deletion of processes that do not satisfy the condition). Indeed, consistency property like reachability of a server in state *bad* in the case study presented in Section 4 always return false positives using monotonic abstraction (by deleting all edges that are not in Q we can always move to *bad*). In synthesis the new approach can be viewed as an attempt of introducing more precise approximated verification algorithms for parameterized systems while retaining good features of approaches like counting and monotonic abstraction in [16,3] like termination properties based on the theory of well-quasi orderings.

Concerning verification algorithms for graph rewriting systems, we are only aware of the works in [18,22]. We use here different type of graph specifications (e.g. we consider universal quantification on incoming edges) and a different notion of graph-based symbolic representation (i.e. a different entailment relation) with respect to those applied to leader election and routing protocols in [18,22].

2 A Client/Server Abstract Model

To represent configurations of client/server protocols, we define a special kind of bipartite graphs. Let Λ_s be a finite set of *server node labels*, Λ_c a finite set of *client node labels*, and Λ_e a finite set of *edge labels*. Furthermore, for $n \in \mathcal{N}$ let $\overline{n} = \{1, \ldots, n\}$. A c/s-graph is a tuple $G = (n_c, n_s, E, \lambda_c, \lambda_s, \lambda_e)$, where $\overline{n_s}$ is the set of server nodes, $\overline{n_c}$ is the set of client nodes, $E \subseteq \overline{n_s} \times \overline{n_c}$ is a set of edges connecting a server with a set of clients, and a client with at most one server (i.e. for each $j \in \overline{n_c}$ we require that there exists at most one edge incident in j in E), and $\lambda_c : \overline{n_c} \rightarrow \Lambda_c$, $\lambda_s : \overline{n_s} \rightarrow \Lambda_s$, and $\lambda_e : E \rightarrow \Lambda_e$ are labelling functions.

In the rest of the paper we use the operations on c/s-graphs defined in Fig. 1. A client/server system is a tuple $S = (I, R)$ consisting of a (possibly infinite) set I of c/s-graphs (initial configurations), and a finite set R of rules. We consider here a restricted type of graph rewriting rules to model both the interaction between clients and servers and the manipulation of directories viewed as the set of incident edges in a given server nodes.

The rules have the general form $l \Rightarrow r$ where l is a pattern that has to match (the labels and structure) of a subgraph in the current configuration in order for the rule to be fireable and r describes how the subgraph is rewritten as the effect of the application of the rule. In this paper we are interested in modelling asynchronous communication patterns. Thus, we consider the following patterns: the empty graph $\cdot$ (it matches with any graph); $\langle\!\langle \ell \rangle\!\rangle$ that denotes an isolated client node with label ℓ; $(\!(\ell)\!)$ that denotes a server node with label ℓ, $[\![\ell]\!] \overset{\sigma}{\longleftrightarrow}$ that denotes a client node with label ℓ and incident edge with label σ; $(\!(\ell)\!) \overset{\sigma}{\longleftrightarrow}$ that denotes a server node with label ℓ and an incident edge with label σ.

Furthermore, we also admit a special type of rules in which the rewriting step can be applied to a given server node if a universally quantified condition

Given a graph $G = (n_c, n_s, E, \lambda_c, \lambda_s, \lambda_e)$, we define:

- $\mathsf{edges}(G) = E$, $\mathsf{edges}_s(i, G) = \{e \mid e = (i, j) \in E\}$ for $i \in \overline{n_s}$, and $\mathsf{edges}_c(j, G) = \{e \mid e = (i, j) \in E\}$ for $j \in \overline{n_c}$; $\mathsf{label}_e(e, G) = \lambda_e(e)$ for $e \in E$ and $\mathsf{label}_e(i, G) = \{\lambda_e(e) \mid e \in \mathsf{edges}_s(i, G)\}$ for $i \in \overline{n_s}$;
- $\mathsf{add}_e(e, \sigma, G) = (n_c, n_s, E \cup \{e\}, \lambda_c, \lambda_s, \lambda'_e)$ where $\lambda'_e(e) = \sigma$, $\lambda'_e(o) = \lambda_e(o)$ in all other cases;
- $\mathsf{update}_e(e \leftarrow \sigma, G) = (n_c, n_s, E, \lambda_c, \lambda_s, \lambda'_e)$ where $\lambda'_e(e) = \sigma$, and $\lambda'_e(o) = \lambda_e(o)$ in all other cases;
- $\mathsf{del}_e(e, G) = (n_c, n_s, E', \lambda_c, \lambda_s, \lambda'_e)$, where $E' = E \setminus \{e\}$, $\lambda'_e(o) = \lambda_e(o)$ for $o \in E'$.
- $\mathsf{nsize}_c(G) = n_c$, and $\mathsf{label}_c(i, G) = \lambda_c(i)$ for $i \in \overline{n_c}$,
- $\mathsf{add}_c(P, G) = (n_c + 1, n_s, E, \lambda'_c, \lambda_s, \lambda_e)$ where $\lambda'_c(n_c + 1) = P$ and $\lambda'_c(o) = \lambda_c(o)$ in all other cases;
- $\mathsf{update}_c(i_1 \leftarrow P_1, \ldots, i_m \leftarrow P_m, G) = (n_c, n_s, E, \lambda'_c, \lambda_s, \lambda_e)$ where $\lambda'_c(i_k) = P_k$ for $k : 1, \ldots, m$, and $\lambda'_c(o) = \lambda_c(o)$ in all other cases;
- $\mathsf{del}_c(i, G) = (n_c - 1, n_s, E', \lambda'_c, \lambda_s, \lambda'_e)$ where, given the mapping $h_i : \overline{n_c} \to \overline{n_c - 1}$ defined as $h_i(j) = j$ for $j < i$ and $h_i(j) = j - 1$ for $j > i$, $E' = \{(k, h_i(l)) \mid (k, l) \in E\}$, $\lambda'_c(k) = \lambda_c(p)$ for each $k \in \overline{n_c - 1}$ such that $k = h_i(p)$ and $p \in \overline{n_c}$, $\lambda'_e((k, l)) = \lambda_e((k, q))$ for $(k, l) \in E'$ such that $l = h_i(q)$ for $q \in \overline{n_c}$, $\lambda'_x(o) = \lambda_x(o)$ in all other cases for $x \in \{e, c\}$;
- nsize_s, label_s, add_s, update_s, and del_s are defined for server nodes in a way similar to the client node operations.

Fig. 1. Definition of basic graph operations

on the labels of the corresponding incident edges is satisfied. Specifically, we consider the rule schemes illustrated in Fig. 2, where ℓ and ℓ' are node labels of appropriate type, σ and σ' are edge labels, and $\forall Q$ is a condition with $Q \subseteq \Lambda_e$.

With the first two types of rules, we can non-deterministically add a new node to the current graph (e.g. to dynamically inject new servers and clients). With rule *start_transaction*, we non-deterministically select a server and a client (not connected by an already existing edge) add a new edge between them in the current graph (e.g. to dynamically establish a new communication). With rules of types *client/server_steps*, we update the labels of a node with label ℓ and one of its incident edges (non-deterministically chosen) with label σ (e.g. to define asynchronous communication protocols). With rule *test*, we update the node label of a server node i only if all edges incident to i have labels in the

$$
\begin{array}{llr}
\cdot \;\Rrightarrow\; \langle\!\langle \ell \rangle\!\rangle & & (\mathit{new\_client\_node}) \\
\cdot \;\Rrightarrow\; (\!(\ell)\!) & & (\mathit{new\_server\_node}) \\
\langle\!\langle \ell \rangle\!\rangle \;\Rrightarrow\; [\![\ell']\!] \xleftrightarrow{\sigma} & & (\mathit{start\_transaction}) \\
(\!(\ell)\!) \xleftrightarrow{\sigma} \;\Rrightarrow\; (\!(\ell')\!) \xleftrightarrow{\sigma'} & & (\mathit{server\_step}) \\
[\![\ell]\!] \xleftrightarrow{\sigma} \;\Rrightarrow\; [\![\ell']\!] \xleftrightarrow{\sigma'} & & (\mathit{client\_step}) \\
(\!(\ell)\!) \;\Rrightarrow\; (\!(\ell')\!) \;:\; \forall Q & & (\mathit{test}) \\
[\![\ell]\!] \xleftrightarrow{\sigma} \;\Rrightarrow\; \langle\!\langle \ell' \rangle\!\rangle & & (\mathit{stop\_transaction})
\end{array}
$$

Fig. 2. Rewriting rules with conditions on egdes

set $Q \subseteq \Lambda_e$. With rule *stop_transaction*, we non-deterministically select a client node with label ℓ and incident edge with label σ, and delete such an edge from the current graph (e.g. to terminate a conversation).

It is important to remark that the a server has not direct access to the local state of a client. Thus, it cannot check conditions on the global sets of its current clients in an atomic way. For checking global conditions a server can however check the set of it incident edges, i.e., a local snapshot of the current condition of clients. A consistency protocol should guarantee that the information on the edges (directory) is consistent with the current state of clients.

2.1 Transition Relation

Let G be a c/s-graph. The formula $\forall Q$ is satisfied in server node i if $\mathsf{label}_e(i, G) \subseteq Q$. The operational semantics is defined via a binary relation $\Rrightarrow_r$ on c/s-graphs such that $G_0 \Rrightarrow G_1$ if and only if one of the following conditions hold:

- r is a *new_client_node* rule and $G_1 = \mathsf{add}_c(\ell, G_0)$;
- r is a *new_server_node* rule and $G_1 = \mathsf{add}_s(\ell, G_0)$;
- r is a *server_step* rule and there exist nodes i and j in G_0 with edge $e = (i, j) \in \mathsf{edges}(G)$ such that $\mathsf{label}_s(i, G_0) = \ell$, $\mathsf{label}_e(e, G_0) = \sigma$, $G_1 = \mathsf{update}_e(e \leftarrow \sigma', \mathsf{update}_s(i \leftarrow \ell', G_0))$;
- r is an *client_step* rule and there exist nodes i and j in G_0 with edge $e = (i, j) \in \mathsf{edges}(G)$ such that $\mathsf{label}_n(j, G_0) = \ell$, $\mathsf{label}_e(e, G_0) = \sigma$, $G_1 = \mathsf{update}_e(e \leftarrow \sigma', \mathsf{update}_c(j \leftarrow \ell', G_0))$;
- r is a *start_transaction* rule, there exists in G_0 a client node j with no incident edges in E such that $\mathsf{label}_c(j, G_0) = \ell$, and $G_1 = \mathsf{add}_e((i, j), \sigma, \mathsf{update}_c(j \leftarrow \ell', G_0))$ for a server node i;
- r is a *stop_transaction* rule, there exist nodes i and j in G_0 such that $\mathsf{label}_c(j, G_0) = \ell$, $e = (i, j) \in \mathsf{edges}(G_0)$, $\mathsf{label}_e(e, G_0) = \sigma$, and $G_1 = \mathsf{del}_e(e, \mathsf{update}_c(j \leftarrow \ell', G_0))$.
- r is a *test* rule, there exist node i in G_0 such that $\mathsf{label}_s(i, G_0) = \ell$, $\mathsf{label}_e(i, G_0) \subseteq Q$, and $G_1 = \mathsf{update}_s(j \leftarrow \ell', G_0)$.

Finally, we define $\Rrightarrow$ as $\bigcup_{r \in R} \Rrightarrow_r$.

Example 1. As an example, consider a set of labels Λ_n partitioned in the two sets $\Lambda_c = \{idle, wait, use\}$ and $\Lambda_s = \{ready, check, ack\}$, and a set of edge labels $\Lambda_e = \{req, pend, inv, lock\}$. The following set R of rules models a client-server protocol (with any number of clients and servers) in which a server grants the use of a resource after invalidating the client that is currently using it.

$$\begin{array}{l}
(r_1)\ \langle\!\langle idle \rangle\!\rangle \ \Rrightarrow\ \llbracket wait \rrbracket \xleftrightarrow{req} \\
(r_2)\ (\!(ready)\!) \xleftrightarrow{req} \ \Rrightarrow\ (\!(check)\!) \xleftrightarrow{pend} \\
(r_3)\ (\!(check)\!) \xleftrightarrow{lock} \ \Rrightarrow\ (\!(check)\!) \xleftrightarrow{inv} \\
(r_4)\ (\!(check)\!) \ \Rrightarrow\ (\!(ack)\!) \ :\ \forall\{pend, req\} \\
(r_5)\ (\!(ack)\!) \xleftrightarrow{pend} \ \Rrightarrow\ (\!(ready)\!) \xleftrightarrow{lock} \\
(r_6)\ \llbracket use \rrbracket \xleftrightarrow{inv} \ \Rrightarrow\ \langle\!\langle idle \rangle\!\rangle \\
(r_7)\ \llbracket wait \rrbracket \xleftrightarrow{lock} \ \Rrightarrow\ \llbracket use \rrbracket \xleftrightarrow{lock}
\end{array}$$

With rule r_1 a client non-deterministically creates a new edge connecting to a server. With rule r_2 a server processes a request by changing the edge to *pending*, and then moves to state *check*. With rule r_3 a server sends invalidation messages to the client that is currently using the resource (marked with the special edge *lock*). With rule r_4 a server moves to the acknowledge step whenever all incident edges have state different from *lock* and *inv*. With rule r_5 a server grants the pending request. With rule r_6 a clients releases the resource upon reception of an invalidation request. With rule r_7 a waiting client moves to state *use*.

Now, let us consider an initial graph G_0 with one server node with label *ready* and two client nodes with label *idle*. Then, the following sequence (of graphs) represents an evolution of the graph system (G_0, R):

$$\begin{array}{l}
G_0 = \langle\!\langle idle\rangle\!\rangle, \langle\!\langle idle\rangle\!\rangle, (\!(ready)\!) \Rightarrow \langle\!\langle idle\rangle\!\rangle, [\![wait]\!] \xleftrightarrow{req} (\!(ready)\!) \Rightarrow \\
[\![wait]\!] \xleftrightarrow{req} (\!(ready)\!) \xleftrightarrow{req} [\![wait]\!] \Rightarrow [\![wait]\!] \xleftrightarrow{pend} (\!(check)\!) \xleftrightarrow{req} [\![wait]\!] \Rightarrow \\
[\![wait]\!] \xleftrightarrow{pend} (\!(ack)\!) \xleftrightarrow{req} [\![wait]\!] \Rightarrow [\![wait]\!] \xleftrightarrow{lock} (\!(ready)\!) \xleftrightarrow{req} [\![wait]\!] \Rightarrow \\
[\![use]\!] \xleftrightarrow{lock} (\!(ready)\!) \xleftrightarrow{req} [\![wait]\!] \Rightarrow [\![use]\!] \xleftrightarrow{lock} (\!(check)\!) \xleftrightarrow{pend} [\![wait]\!] \Rightarrow \\
[\![use]\!] \xleftrightarrow{inv} (\!(check)\!) \xleftrightarrow{pend} [\![wait]\!] \Rightarrow \langle\!\langle inv\rangle\!\rangle, (\!(check)\!) \xleftrightarrow{pend} [\![wait]\!] \Rightarrow \\
\langle\!\langle inv\rangle\!\rangle, (\!(ack)\!) \xleftrightarrow{pend} [\![wait]\!] \Rightarrow \langle\!\langle inv\rangle\!\rangle, (\!(ready)\!) \xleftrightarrow{lock} [\![wait]\!] \Rightarrow \\
[\![inv]\!], [\![ready]\!] \xleftrightarrow{lock} [\![use]\!]
\end{array}$$

2.2 Pattern Reachability

In this paper we are interested in studying reachability of graphs containing specific patterns (subgraphs). Patterns can be used to represent bad configurations of a graph system. In Example 1 any graph containing the pattern $[\![use]\!] \xleftrightarrow{\sigma} (\!(ready)\!) \xleftrightarrow{\sigma'} [\![use]\!]$, for $\sigma, \sigma' \in \Lambda_e$, represents a violation to the exclusive use of a resource controlled by a server node.

To formally define the notion of pattern, we introduce an ordering $\preceq$ on c/s-graphs such that $G \preceq G'$ iff $n_c = \mathsf{nsize}_c(G) \leq m_c = \mathsf{nsize}_c(G')$, $n_s = \mathsf{nsize}_s(G) \leq m_s = \mathsf{nsize}_s(G')$, and there exist injective mappings $h_c : \overline{n_c} \rightarrow \overline{m_c}$ and $h_s : \overline{n_s} \rightarrow \overline{m_s}$ such that

- $\mathsf{label}_c(i, G) = \mathsf{label}_c(h_c(i), G')$ for $i : 1, \ldots, n_c$,
- $\mathsf{label}_s(i, G) = \mathsf{label}_s(h_s(i), G')$ for $i : 1, \ldots, n_s$,
- for each $e = (i, j) \in \mathsf{edges}(G)$, $e' = (h_s(i), h_c(j)) \in \mathsf{edges}(G')$ and $\mathsf{label}_e(e, G) = \mathsf{label}_e(e', G')$.

A set of c/s-graphs $U \subseteq C$ is *upward closed* with respect to $\preceq$ if $c \in U$ and $c \preceq c'$ implies $c' \in U$. For a c/s-graph G, we use $\widehat{G}$ to denote the upward closure of G, i.e., the set $\{G' \mid G \preceq G'\}$. For sets of c/s-graphs $D, D' \subseteq C$ we use $D \Rightarrow D'$ to denote that there are $G \in D$ and $G' \in D'$ with $G \Rightarrow G'$.

The *Pattern Reachability Problem* for graph systems is defined as follows:

Pattern Reachability Problem (PRP)

Instance

- A graph system $\mathcal{P} = (I, R)$.
- A finite set C_F of c/s-graphs

Question $G_0 \Rightarrow^* \widehat{C_F}$ for $G_0 \in I$?

Typically, $\widehat{C_F}$ (which is an infinite set) is used to characterize sets of *bad* configurations which we do not want to occur during the execution of the system. In such a case, the system is safe iff $\widehat{C_F}$ is not reachable. Therefore, checking safety properties amounts to solving PRP (i.e., to the reachability of upward closed sets). In [7] we show that control state reachability for counter machines can be reduced to PRP. From this property, it follows that PRP is undecidable.

3 Approximated Verification Algorithm

In this section we propose an approximated verification algorithm based on the notion of *graph constraints*, a special symbolic representation of an infinite sets of c/s-graphs.

A *graph constraint* (gc) is a graph $\Psi = (n_c, n_s, E, \rho_c, \rho_s, \rho_e)$, with client nodes $\{1, \ldots, n_c\}$, server nodes $\{1, \ldots, n_s\}$, edges in $E \subseteq \overline{n_s} \times \overline{n_c}$, and labels defined by maps $\rho_c : \overline{n_c} \rightarrow \Lambda_c$, $\rho_s : \overline{n_s} \rightarrow (\Lambda_s \times 2^{\Lambda_e})$, and $\rho_e : E \rightarrow \Lambda_e$.

Notice that, in a graph constraint Ψ, the label of a server node is a pair (ℓ, Q) where ℓ is a node label and $Q \subseteq \Lambda_e$ is a subset of edge labels, called *padding set.* In this section we adapt the operations on c/s-graphs to graph constraints. Specifically, given $\Psi = (n_c, n_s, E, \rho_c, \rho_s, \rho_e)$, $i \in \overline{n_s}$, $j \in \overline{n_c}$, $\rho_s(i) = (\ell, Q)$, $\rho_c(j) = \ell'$, and $e \in E$, then $\mathsf{label}_s(i, \Psi) = \ell$, $\mathsf{label}_p(i, \Psi) = Q$, $\mathsf{label}_c(j, \Psi) = \ell'$, and $\mathsf{label}_e(e, \Psi) = \rho_e(e)$. The other operations are defined as for c/s-graphs.

For a graph constraint Ψ to be well-formed (wfgc), we require that $\mathsf{label}_e(i, \Psi) \subseteq \mathsf{label}_p(i, \Psi)$ for each $i \in \overline{n_s}$.

Let Ψ be a wfgc, and G be a c/s-graph. In order to define the denotation of a wfgc Ψ we introduce the relation $\preceq$ such that, given a c/s-graph G, $\Psi \preceq G$ iff $n_c = \mathsf{nsize}_c(\Psi) \leq m_c = \mathsf{nsize}_c(G)$, $n_s = \mathsf{nsize}_s(\Psi) \leq m_s = \mathsf{nsize}_s(G)$, and there exist injective mappings $h_c : \overline{n_c} \rightarrow \overline{m_c}$ and $h_s : \overline{n_s} \rightarrow \overline{m_s}$ such that

- $\mathsf{label}_c(i, \Psi) = \mathsf{label}_c(h_c(i), G)$ for $i : 1, \ldots, n_c$,
- $\mathsf{label}_s(i, \Psi) = \mathsf{label}_s(h_s(i), G)$ and $\mathsf{label}_e(h_s(i), G') \subseteq \mathsf{label}_p(i, \Psi) = Q$ for $i : 1, \ldots, n_s$;
- for each $e = (i, j) \in \mathsf{edges}(\Psi)$, $e' = (h_s(i), h_c(j)) \in \mathsf{edges}(G)$ and $\mathsf{label}_e(e, \Psi) = \mathsf{label}_e(e', G)$.

The denotation of a graph constraint Ψ is then defined as $[\![\Psi]\!] = \{G \mid G\ \textit{is a c/s-graph},\ \Psi \preceq G\}$.

Approximated Predecessor Relation The set of predecessors of a set S of c/s-graphs computed with respect to a rule r is defined as

$$pre_r(S) = \{G \mid G \Rightarrow_r S\}$$

Given a wfgc $\varPsi$ we now define a relation $\leadsto_r$ working on wfgc's that we use to overapproximate the set $[\![\varPsi]\!] \cup pre_r([\![\varPsi]\!])$. We consider here the union of these two sets in order to be able to discard graph constraints that denote graphs already contained in $[\![\varPsi]\!]$. For brevity, we describe here the computation of predecessors for rules of the form *server-step*, *client-step*, and *test*. The complete definition is given in [26]. Specifically, for graph constraints $\varPsi$, with $n_s = \mathsf{nsize}_s(\varPsi)$ and $n_c = \mathsf{nsize}_c(\varPsi)$, and $\varPsi'$, and a rule $r \in R$, the relation $\varPsi \leadsto_r \varPsi'$ is defined as follows:

server-step: r is the rule $(\!(\ell)\!) \overset{\sigma}{\longleftrightarrow} \quad \Rrightarrow \quad (\!(\ell')\!) \overset{\sigma'}{\longleftrightarrow}$ and one of the following conditions hold

- $i \in \overline{n_s}$, $j \in \overline{n_c}$, $e = (i,j) \in \mathsf{edges}(\varPsi)$, $\mathsf{label}_s(i,\varPsi) = \ell'$, $\mathsf{label}_e(e) = \sigma'$, and

  $$\varPsi' = \mathsf{update}_e(e, \sigma, \mathsf{update}_s(i \leftarrow (\ell, Q), \varPsi))$$

  where $Q = label_p(i) \cup \{\sigma\}$.

  In this case we update the label of an existing edge (i,j) and of the node i with the labels σ and ℓ, respectively. They represent the preconditions for firing the rule. Furthermore, we augment the padding set of i with label σ. Notice that here we apply an approximation, i.e., as soon as we add σ we allow any number of occurrences of edges with label σ but we do not count them. The label of client node j is not modified.
- $i \in \overline{n_s}$, $j \in \overline{n_c}$, $\mathsf{edges}(j,\varPsi) = \emptyset$ (j has no incident edges), $\mathsf{label}_s(i,\varPsi) = \ell'$, $\sigma' \in \mathsf{label}_p(i,\varPsi)$, and

  $$\varPsi' = \mathsf{add}_e((i,j), \sigma, \mathsf{update}_s(i \leftarrow (\ell, Q), \varPsi))$$

  where $Q = \mathsf{label}_p(i,\varPsi) \cup \{\sigma\}$.

  Although not explicitly present, we assume here that the edge (i,j) with label σ' is in the upward closure of $\varPsi$ (this can happen only if j is not involved in other explicit edges). We add the edge (i,j) with label σ since its presence is a precondition for the firing the rule. Furthermore, we update the label of i as in the first case.
- $i \in \overline{n_s}$, $\mathsf{label}_s(i,\varPsi) = \ell'$, $\sigma' \in \mathsf{label}_p(i,\varPsi)$, and

  $$\varPsi' = \mathsf{add}_e((i, n_c+1), \sigma, \mathsf{add}_c(\ell'', \mathsf{update}_s(i \leftarrow (\ell, Q), \varPsi)))$$

  where $Q = \mathsf{label}_p(i,\varPsi) \cup \{\sigma\}$, and ℓ'' is non-deterministically chosen from $\varLambda_c$. Although not explicitly present, we assume here that both the client node $n_c + 1$ (with some label taken from $\varLambda_c$) and the edge (i, n_c+1) with label σ are in the upward closure of $\varPsi$. We add them to $\varPsi$ since their presence is a precondition for the firing of r. We update the label of i as in the other two cases. Notice that the dimension of the graph constraint is increased by one, since we insert the new node $n_c + 1$.

For this kind of rules, there are two remaining cases to consider (the edge and the server node, or the edge and both server and client nodes are not explicitly present in Ψ). However these cases give rise to graph constraints that are redundant with respect to Ψ. Thus, we can discard them without loss of precision (we recall that our aim is to symbolically represent $[\![\Psi]\!] \cup pre_r([\![\Psi]\!])$).

client-step: r is the rule $[\![\ell]\!] \xleftrightarrow{\sigma} \;\Rightarrow\; [\![\ell']\!] \xleftrightarrow{\sigma'}$ and one of the following conditions hold

– $i \in \overline{n_s}$, $j \in \overline{n_c}$, $e = (i, j) \in \mathsf{edges}(\Psi)$, $\mathsf{label}_c(j, \Psi) = \ell'$, $\mathsf{label}_e(e) = \sigma'$, and

$$\Psi' = \mathsf{update}_e(e, \sigma, \mathsf{update}_s(i \leftarrow (\mathsf{label}_s(i, \Psi), Q), \mathsf{update}_c(j \leftarrow \ell, \Psi)))$$

where $Q = \mathsf{label}_p(i, \Psi) \cup \{\sigma\}$.

In this case we update the label of an existing edge (i, j) and of the node j with the labels σ and ℓ as a precondition for the firing of the rule r. Furthermore, we add σ to the set of admitted edge labels of server node i.

– $i \in \overline{n_s}$, $j \in \overline{n_c}$, $\mathsf{edges}(j, \Psi) = \emptyset$ (j has no incident edges), $\mathsf{label}_c(j, \Psi) = \ell'$, $\sigma' \in \mathsf{label}_p(i, \Psi)$, and

$$\Psi' = \mathsf{add}_e((i, j), \sigma, \mathsf{update}_s(i \leftarrow (\mathsf{label}_s(i, \Psi), Q), \mathsf{update}_c(j \leftarrow \ell, \Psi)))$$

where $Q = p(i, \Psi) \cup \{\sigma\}$.

Although not explicitly present, we assume here that the edge (i, j) is in the upward closure of Ψ. We add the edge (i, j) with label σ since its presence is a precondition for the firing of the rule. Furthermore, we update the label of i and j as in the first case.

– $j \in \overline{n_c}$, $\mathsf{edges}(j, \Psi) = \emptyset$ (j has no incident edges), $\mathsf{label}_c(j, \Psi) = \ell'$, and

$$\Psi' = \mathsf{add}_e((n_s + 1, j), \sigma, add_s((\ell'', \Lambda_e), \mathsf{update}_c(j \leftarrow \ell, \Psi)))$$

where $\ell'' \in \Lambda_s$. Although not explicitly present, we assume here that the edge $(n_s + 1, j)$, for a new server node $n_s + 1$ with a label in Λ_s, is in the upward closure of Ψ. We add the node and the edge with label σ since its presence is a precondition for firing the rule. Furthermore, we update the label of j as in the first case.

– $i \in \overline{n_s}$, $\sigma' \in \mathsf{label}_p(i, \Psi)$, and

$$\Psi' = \mathsf{add}_e((i, n_c + 1), \sigma, \mathsf{add}_c(\ell, \mathsf{update}_s(i \leftarrow (\mathsf{label}_s(i, \Psi), Q), \Psi)))$$

where $Q = \mathsf{label}_p(i, \Psi) \cup \{\sigma\}$.

Although not explicitly present, we assume here that both the node $n_c + 1$ and the edge $(i, n_c + 1)$ are in the upward closure of Ψ. We add it with label σ to the set of edges and update the label of i including σ in the set of admitted edges. Notice that there are remaining cases (client, server, and edge are not explicitly present in Ψ). However these cases give rise to a graph constraint that is redundant with respect to Ψ. Thus, we can discard it without loss of precision (we recall that our aim is to symbolically represent $[\![\Psi]\!] \cup pre_r([\![\Psi]\!])$).

Test: r is the rule $(\!(\ell)\!) \Rrightarrow (\!(\ell')\!) \,:\, \forall Q$ and one of the following conditions hold

- $i \in \overline{n_s}$, $\mathsf{label}_s(i, \Psi) = \ell'$, $R = \mathsf{label}_p(i, \Psi) \cap Q$, $\mathsf{label}_e(e) \in R$ for each $e \in \mathsf{edges}(i, \Psi)$, and
$$\Psi' = \mathsf{update}_s(i \leftarrow (\ell, R), \Psi)$$
In this rule the padding $\mathsf{label}_p(i, \Psi)$ associated to a node i with label ℓ' plays a crucial role. We first check that the current set of labels of edges incident to i is contained into the intersection R of $\mathsf{label}_p(i, \Psi)$ and Q. If this condition is satisfied, we restrict the padding of node i to be the set R (precondition for firing the rule) and update the label of i to ℓ. This rule cannot be applied whenever there are edges in $\mathsf{edges}(i, \Psi)$ with labels not in R. If R is the empty set, then the node i must be isolated.

Given a wfgc Ψ, we define $\Psi \rightsquigarrow$ as the set $\{\Psi' \mid \Psi \overset{r}{\rightsquigarrow} \Psi',\ r \in R\}$. The following property then holds.

Lemma 1. $([\![\Psi]\!] \cup pre([\![\Psi]\!])) \subseteq ([\![\Psi]\!] \cup [\![\Psi \rightsquigarrow]\!])$.

Entailment Test We now define an entailment relation $\sqsubseteq$ used to compare denotations of graph constraints. Let Ψ and Ψ' be two wfgc such that $\mathsf{nsize}_c(\Psi) = n_c$, $\mathsf{nsize}_s(\Psi) = n_s$, $\mathsf{nsize}_c(\Psi') = m_c$, and $\mathsf{nsize}_s(\Psi') = m_s$. The relation $\Psi \sqsubseteq \Psi'$ holds iff $n_c \leq m_c$, $n_s \leq m_s$, and there exist injective mappings $h_c : \overline{n_c} \rightarrow \overline{m_c}$ $h_s : \overline{n_s} \rightarrow \overline{m_s}$ such that

- $\mathsf{label}_s(i, \Psi) = \mathsf{label}_s(h_s(i), \Psi')$ for $i \in \overline{n_s}$,
- $\mathsf{label}_c(j, \Psi) = \mathsf{label}_c(h_c(j), \Psi')$ for $j \in \overline{n_c}$,
- $\mathsf{label}_p(h_s(i), \Psi') \subseteq \mathsf{label}_p(i, \Psi)$ for $i \in \overline{n_s}$,
- for each $e = (i, j) \in E$, $e' = (h_s(i), h_c(j)) \in E'$ and $\mathsf{label}_e(e, \Psi) = \mathsf{label}_e(e', \Psi')$.

The following property then holds.

Lemma 2. *Given Ψ and Ψ', $\Psi \sqsubseteq \Psi'$ implies $[\![\Psi']\!] \subseteq [\![\Psi]\!]$.*

We naturally extend the entailment relation to finite sets of constraints as follows. Given two sets of graph constraints Φ, Φ', $\Phi \sqsubseteq \Phi'$ iff for each $\Psi' \in \Phi'$ there exists $\Psi \in \Phi$ such that $\Psi \sqsubseteq \Psi'$.

3.1 Backward Reachability

We use the relation $\rightsquigarrow$ to define a symbolic backward reachability algorithm for approximating solutions to PRP. We start with a finite set Φ_F of graph constraints denoting an infinite set of bad graph configurations. We generate a sequence $\Phi_0, \Phi_1, \Phi_2, \cdots$ of finite sets of constraints such that $\Phi_0 = \Phi_F$, and $\Phi_{j+1} = \Phi_j \cup (\Phi_j \rightsquigarrow)$. Since $[\![\Phi_0]\!] \subseteq [\![\Phi_1]\!] \subseteq [\![\Phi_2]\!] \subseteq \cdots$, the procedure terminates when we reach a point j where $\Phi_j \sqsubseteq \Phi_{j+1}$. Notice that the termination condition implies that $[\![\Phi_j]\!] = (\bigcup_{0 \leq i \leq j} [\![\Phi_i]\!])$. By Lemma 1, Φ_j denotes an over-approximation of the set of all predecessors of $[\![\Phi_F]\!]$. This means that if $(I \bigcap [\![\Phi_j]\!]) = \emptyset$, then there exists no $G \in [\![\Phi_F]\!]$ with $G_0 \Rightarrow^* G$ for $G_0 \in I$. Thus, the procedure can be used as a semi-test for checking PRP.

According to the general results in [2], the termination of our (approximated) symbolic backward reachability procedure can be ensured by proving that the entailment relation of graph constraints is a well-quasi ordering (wqo). The latter property follows from the fact that a c/s-graph with n_s server nodes and n_c client nodes can be given an alternative representation as a bag of tuples of a special form. A wfgc can be represented as a *bag* (multiset) containing the (multiset) of isolated client nodes in G together with tuples of the form (s_i, Q_i, M_i) for $i \in \{1, \ldots, n_s\}$, where

- $s_i \in \Lambda_s$ is the label of the server node i,
- $Q_i \in 2^{\Lambda_e}$ is the padding associated to i,
- if i has client nodes $j_1, \ldots, j_{k_i}$ connected to it M_i is a bag $\{p_1, \ldots, p_{k_i}\}$ such that $p_l = (\sigma_l, c_l)$, where σ_l is the label of the edge incident to node j_l and c_l is the label of node j_l.

Given bags m_1 and m_2 associated resp. to wfgc's G_1 and G_2, $m_1 \leq m_2$ holds if: each isolated client node in m_1 can be injected into an isolated client node in m_2; each tuple (s, Q, M) in m_1 can be injected into a tuple (s', Q', M') in m_2 such that $s = s'$, $Q' \subseteq Q$ and M is contained into M' (multiset containment). From closure properties of wqo's under bag and tuple composition operators, we have that $\leq$ is a wqo. Furthermore, we have that $m_1 \leq m_2$ implies $G_1 \sqsubseteq G_2$. Thus, the entailment relation of graph constraints is a well-quasi ordering (wqo).

4 A Case Study

We have implemented a prototype version, SYMGRAPH [26], of our approximated verification algorithm and tested on a model of the *full-map cache coherence protocol* described in [20]. This protocol is defined for a multiprocessor with shared memory and local caches in which the memory controller maintains a directory for each memory line with information about its use, i.e., the line is shared between different caches or used in exclusive mode by a given cache. The directory is used to optimize the invalidation and downgrade phase required when a processor sends a new request for exclusive or shared use. Memory controllers associate a special flag *mode_ex* to each line to remember when the line is in exclusive use (i.e. without need to inspect the full-map).

For reason of space, we only give the key ideas behind our model of this protocol. The initial graph configurations consist of any number of isolated client nodes with label *inv* (cache controller for a given line in state *invalid*) and server nodes in state *idle* (memory controller for a given line in state *idle*).

During its life cycle the same cache line can be associated to different memory lines. However, at any given instant a cache line is either invalid or contains a copy of a given memory block. A memory line however can be copied into several cache lines. A cache controller in invalid state sends a request for exclusive or shared access using one of the two following rules

$$\langle\!\langle inv \rangle\!\rangle \Rrightarrow \llbracket wait \rrbracket \overset{req\_ex}{\longleftrightarrow} \qquad \langle\!\langle inv \rangle\!\rangle \Rrightarrow \llbracket wait \rrbracket \overset{req\_sh}{\longleftrightarrow}$$

A cache controller in *wait* state moves to *exclusive* (*shared*) state upon reception of message *ex* (*sh*) along the edge that connects it to the memory controller as specified by the rules

$$\llbracket wait \rrbracket \overset{ex}{\longleftrightarrow} \Rrightarrow \llbracket exclusive \rrbracket \overset{ex}{\longleftrightarrow} \qquad \llbracket wait \rrbracket \overset{sh}{\longleftrightarrow} \Rrightarrow \llbracket shared \rrbracket \overset{sh}{\longleftrightarrow}$$

A cache controller moves to invalid state upon reception of a *req_inv* message as specified by the rules

$$\llbracket shared \rrbracket \overset{req\_inv}{\longleftrightarrow} \Rrightarrow \langle\!\langle inv \rangle\!\rangle \qquad \llbracket exclusive \rrbracket \overset{req\_inv}{\longleftrightarrow} \Rrightarrow \langle\!\langle inv \rangle\!\rangle$$

A cache controller in exclusive state moves to shared state upon reception of a *req_dg* message as specified by the rule

$$\llbracket exclusive \rrbracket \overset{req\_dg}{\longleftrightarrow} \Rrightarrow \llbracket shared \rrbracket \overset{sh}{\longleftrightarrow}$$

A memory controller that receives a *req_ex* message from a channel (edge) updates the label on the corresponding egde to *pend* and then moves to state *inv_loop* as specified by the rule

$$\llparenthesis idle \rrparenthesis \overset{req\_ex}{\longleftrightarrow} \Rrightarrow \llparenthesis inv\_loop \rrparenthesis \overset{pend}{\longleftrightarrow}$$

While in the *inv_loop* state, the memory controller sends an invalidation request *req_inv* to all caches connected to it with edges marked *sh* or *ex* as specified by rules

$$\llparenthesis inv\_loop \rrparenthesis \overset{ex}{\longleftrightarrow} \Rrightarrow \llparenthesis inv\_loop \rrparenthesis \overset{req\_inv}{\longleftrightarrow} \qquad \llparenthesis inv\_loop \rrparenthesis \overset{sh}{\longleftrightarrow} \Rrightarrow \llparenthesis inv\_loop \rrparenthesis \overset{req\_inv}{\longleftrightarrow}$$

The memory controller moves to state *ack_inv* after testing that all requests have been processed by the cache controllers connected to it as specified by the rule

$$\llparenthesis inv\_loop \rrparenthesis \Rrightarrow \llparenthesis ack\_inv \rrparenthesis \ : \ \forall \{pend, req\_sh, req\_ex\}$$

In state *ack_inv* the memory controller grants the access to the waiting cache controller connected to it with an edge labelled *pend* by using the rule

$$\llparenthesis ack\_inv \rrparenthesis \overset{pend}{\longleftrightarrow} \Rrightarrow \llparenthesis idle_{ex} \rrparenthesis \overset{ex}{\longleftrightarrow}$$

The state $idle_{ex}$ is used here to remember that a cache is using the line in exclusive state (the *mode_ex* flag in [20] is set to 1).

Request for shared access are treated in a similar way. However a downgrade request *req_dg* instead of an invalidation message is sent to all caches connected with an edge to the memory controller. The invalidation/downgrade loop can be avoided when the request is processed in the special state $idle_{ex}$. The complete client/server model for this protocol is given in [26].

For this case study we consider the following pattern reachability problems (PRP) that represent violation to mutual exclusion and consistency properties. For proving mutual exclusion, we consider a number of PRPs defined by taking as target set of configurations the denotations of a graph with a memory node m and two cache nodes c, c' both linked to m (to model the fact that the cache lines stored in c, c' correspond to that controlled by m) and such that c, c' and the corresponding incident edges have a conflicting state. Formally, we consider graph constraints defined as follows $G = \{1, 2, \{e = (1, 1), e' = (1, 2)\}, \rho_c, \rho_s, \rho_e\}$ where $\rho_s(1) = (\ell, \Lambda_e)$, $\ell \in \{idle, idle_{ex}\}$, $\rho_c(1) = ex$, $\rho_e(e) = ex$, and either ($\rho_c(2) = ex$ and $\rho_e(e') = ex$) or ($\rho_c(2) = sh$ and $\rho_e(e') = sh$).

We can also formulate other types of consistency properties as PRP. For instance, to check that $idle_{ex}$ corresponds to a memory (line) state in which one cache controller has exclusive access we can first add the following rule:

$$(\!(idle_{ex})\!) \Rrightarrow (\!(bad)\!) \ : \ \forall Q$$

where *bad* is a new memory label and Q is an appropriate set of edge labels (see [26]). The graph $G = \{1, 0, \emptyset, \rho_s, \emptyset, \emptyset\}$ with $\rho_s(1) = (bad, \Lambda_e)$ represents the set of violations to the consistency of the *mode_ex* flag with respect to the current state of the fullmap. Our prototype implementation of the symbolic backward procedure with graph constraints verifies the above mentioned properties automatically [26].

5 Conclusions and Related Work

We have presented a new algorithm for parameterized verification of directory-based consistency protocols based on a graph representation (graph constraints) of infinite collections of configurations. The algorithm computes an overapproximation of the set of backward reachable configurations denoted by an initial set of graph constraints. We apply the new algorithm to different versions of a non-trivial case-study discussed in [20]. We plan to investigate how to extend this approach to deal with parameterized systems in which some of the nodes play both the role of server and client in different instances of a given communication protocol.

References

1. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziz, F., Rezine, A.: Monotonic abstraction for programs with dynamic memory heaps. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 341–354. Springer, Heidelberg (2008)

2. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. LICS 1996, 313–321 (1996)
3. Abdulla, P.A., Ben Henda, N., Delzanno, G., Rezine, A.: Regular model checking without transducers. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
4. Abdulla, P.A., Ben Henda, N., Delzanno, G., Rezine, A.: Handling parameterized systems with non-atomic global conditions. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 22–36. Springer, Heidelberg (2008)
5. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
6. Abdulla, P.A., Delzanno, G., Haziza, F., Rezine, A.: Parameterized tree systems. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 69–83. Springer, Heidelberg (2008)
7. Abdulla, P.A., Delzanno, G., Rezine, A.: Approximated Context-sensitive Analysis for Parameterized Verification FORTE 2009 (2009)
8. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J.: Regular model checking made simple and efficient. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 116–130. Springer, Heidelberg (2002)
9. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
10. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003)
11. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
12. Bouajjani, A., Muscholl, A., Touili, T.: Permutation Rewriting and Algorithmic Verification. Inf. and Comp. 205(2), 199–224 (2007)
13. Clarke, E., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2005)
14. Delzanno, G.: Constraint-Based Verification of Parameterized Cache Coherence Protocols. FMSD 23(3), 257–301 (2003)
15. Emmi, M., Jhala, R., Kohler, E., Majumdar, R.: Verifying reference counted objects. In: TACAS 2009 (to appear, 2009)
16. Esparza, J., Finkel, A., Mayr, R.: On the Verification of Broadcast Protocols. LICS (1999)
17. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! TCS 256(1-2), 63–92 (2001)
18. Joshi, S., König, B.: Applying the graph minor theorem to the verification of graph transformation systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 214–226. Springer, Heidelberg (2008)
19. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. TCS 256, 93–112 (2001)
20. Pong, F., Dubois, M.: Correctness of a Directory-Based Cache Coherence Protocol: Early Experience. In: SPDP 1993, pp. 37–44 (1993)

21. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
22. Saksena, M., Wibling, O., Jonsson, B.: Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 18–32. Springer, Heidelberg (2008)
23. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. LICS 1986, 332–344 (1986)
24. Yavuz-Kahveci, T., Bultan, T.: A symbolic manipulator for automated verification of reactive systems with heterogeneous data types. STTT 5(1), 15–33 (2003)
25. Yavuz-Kahveci, T., Bultan, T.: Verification of parameterized hierarchical state machines using action language verifier. In: MEMOCODE 2005, pp. 79–88 (2005)
26. Symgraph: `http://www.disi.unige.it/person/DelzannoG/Symgraph/`