International Journal of Foundations of Computer Science
Vol. 22, No. 4 (2011) 761–782
© World Scientific Publishing Company
DOI: 10.1142/S0129054111008416

AUTOMATIC VERIFICATION OF DIRECTORY-BASED CONSISTENCY PROTOCOLS WITH GRAPH CONSTRAINTS

PAROSH AZIZ ABDULLA

Uppsla University, Department of Information Technology,
Box 337, 751 05 Uppsala, Sweden
parosh@it.uu.se

GIORGIO DELZANNO

Università di Genova, via Dodecaneso 35, 16146 Genov, Italy
giorgio@disi.unige.it

AHMED REZINE*

University of Uppsala,
Department of Information Technology, Box 337, 751 05 Uppsala, Sweden
rahmed@it.uu.se

Received 31 December 2009
Accepted 13 August 2010
Communicated by Oliver Bournez and Igor Potapov

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,

*Corresponding author.

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. 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 and 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 challenging problem. 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 [8, 12]) or Petri net-like models (e.g. counting abstraction used in [15, 13, 23, 24]).

In this paper we propose a new approximated verification technique that operates on models in which the number of controlled resources and the number 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 modeled by using a special type of graphs in which nodes are partitioned into client and server nodes. Node labels represent the current state of the corresponding processes. Labeled 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. Furthermore, we admit 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 the inspection of a directory or invalidation cycles without need of abstracting them by means of atomic operations like broadcast in [15, 13]. 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 verification problem we consider here is called pattern reachability. Specifically, we fix an infinite set of initial configurations (e.g. in which clients and servers are in their initial state) and a finite set of *bad patterns* that represent violations to safety properties (e.g. a graph in which a server node is connected to two different clients that share the corresponding resource). The pattern reachability problem consists in checking if a configuration that contains a bad pattern is reachable from one of the initial configurations.

To attack this problem, we propose an approximated verification algorithm based on the notion of *graph constraint*. A graph constraint is a symbolic representation of an infinite sets of configurations. More precisely, a graph constraints G represents the set of configurations that contains G as a subgraph. Graph constraints can naturally be used to represent bad patterns, i.e., to locally represent a violations of a given safety property. Furthermore, they can be used to define a symbolic computation of predecessor configurations. Predecessors of graph constraints are defined by means of graph transformations. The resulting set of operations can be used to explore backwards the state space of a directory based protocol. To handle universally quantified guards so as to ensure the termination of the analysis it is necessary however to apply approximations during the computation of predecessors. We include the approximation in the symbolic computation of predecessors in the following way. Each server node of a graph constraint contains as a label a set of admitted edges, called padding set, that connect the nodes to the rest of the graph. Every time a transition applied backward adds a new edge to the node, its label is added to the padding set. This way we abstract away the number of egdes with that label connected to the nodes. Indeed if a label is in the padding set then the node may have $k \geq 0$ egdes with that label in the denotation of the graph constraint. However, if a label does not belong to the padding set, then the node cannot have edges with that label in the denotation of the graph constraint. Thus, associating a padding set to each node allow us to symbolically represent the precondition of a transition with universally quantified conditions on edge labels (we restrict the padding set of a node accordingly to the guard of the transition). Termination of the resulting approximated symbolic backward exploration algorithm is obtained by applying the theory of well-structured transition systems [2, 16].

We have implemented a prototype version, SYMGRAPH [25], of our approximated verification algorithm and tested on a model of the *full-map cache coherence protocol* described in [19]. The protocol is defined for a multiprocessor with shared memory and local caches. The memory controller maintains a directory for each memory line with information about its use. The directory is used to optimize the invalidation and downgrade phase required when a processor sends a new request for exclusive or shared use. For this case study we consider pattern reachability problems for checking reachability of patterns that represent violation to mutual exclusion and consistency properties.
The prototype is available at the URL

`http:\\www.disi.unige.it\person\DelzannoG\Symgraph`

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 [19] in a very detailed way. On the other side it allows us to define our verification method at a very abstract level by using graph transformations.

1.1. *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 [8, 20] and the *environment abstraction* method [12]. Counting abstraction and Petri net-like analysis techniques are considered, e.g., in [15, 13, 23, 24].

Differently from previous work we are aware of, our algorithm is based on graph constraints that allow us to symbolically represent infinite-sets of configurations without need of fixing parameters like the number of clients, servers, resources, and the 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 [6]. The new graph-based algorithm is a generalization of the approach in [6]. Indeed, the symbolic configurations we used in [6] 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 [4] (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 5 always return false positives using monotonic abstraction (by deleting all edges that are not in Q we can always move to *bad*). For this type of property, it is essential to attach a padding set to each node in the symbolic representation of a set of configurations. In synthesis the new approach can be interpreted as a more precise approximated verification algorithm for parameterized systems compared to previous work based on counting and monotonic abstraction in [15, 3, 4].

Concerning verification algorithms for graph rewriting systems, we are only aware of the works in [17, 21]. 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 [17, 21].

2. A Client/Server Abstract Model

In this section we introduce an abstract model for client/server protocols in which configurations are bipartite graphs and transition rules are specified using conditional graph rewriting. We first introduce configurations, called c/s-graphs, and then show the class of conditional rewriting rules that we consider.

2.1. *Configurations: c/s-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 $\overline{n} = \{1, \dots, n\}$. 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),
- $\lambda_c : \overline{n_c} \to \Lambda_c$, $\lambda_s : \overline{n_s} \to \Lambda_s$, and $\lambda_e : E \to \Lambda_e$ are labelling functions.

Given a c/s-graph $G = (n_c, n_s, E, \lambda_c, \lambda_s, \lambda_e)$, we define the following set of graph operations:

- $\mathsf{edges}(G) = E$;
- $\mathsf{edges}_s(i, G) = \{e \mid e = (i, j) \in E\}$ for $i \in \overline{n_s}$;
- $\mathsf{edges}_c(j, G) = \{e \mid e = (i, j) \in E\}$ for $j \in \overline{n_c}$;
- $\mathsf{label}_e(e, G) = \lambda_e(e)$ for $e \in E$;
- $\mathsf{label}_e(i, G) = \{\lambda_e(e) \mid e \in \mathsf{edges}_s(i, G)\}$ 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;
- $\mathsf{update}_c(i_1 \leftarrow P_1, \dots, i_m \leftarrow P_m, G) = (n_c, n_s, E, \lambda'_c, \lambda_s, \lambda_e)$ where $\lambda'_c(i_k) = P_k$ for $k : 1, \dots, m$, 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 $x \in \{e, c\}$;

The operations nsize_s, label_s, add_s, update_s, and del_s are defined for server nodes in a way similar to those of client nodes.

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

Fig. 1. Rewriting rules with conditions on egdes.

2.2. *Transitions: c/s system*

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 node.

The rules have the general form $l \Rrightarrow r$: 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; r describes how the subgraph is rewritten as the effect of the application of the rule. For defining the enabling conditions, 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.

The previous conditions are used to model asynchronous communication patterns. 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 on the labels of the corresponding incident edges is satisfied. Specifically, we consider the rule schemes illustrated in Fig. 1, 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 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 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. A server can however check the set of its 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.

2.3. *Transition relation*

Let G be a c/s-graph. The formula $\forall Q$ is satisfied in server node i if $\mathsf{label}_e(i, G) \subseteq Q$. Given a rule r in R, the operational semantics is defined via a binary relation $\Rightarrow_r$ on c/s-graphs such that $G_0 \Rightarrow_r G_1$ 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 $\Rightarrow$ as $\bigcup_{r \in R} \Rightarrow_r$.

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*

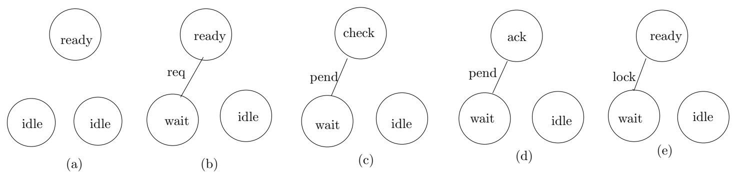

Fig. 2. Example of execution.

of a resource after invalidating the client that is currently using it.

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

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)*:*

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

The first five steps are also drawn in Fig. 2

3. Verification as Pattern Reachability

In this paper we are interested in verification problems that can be expressed as reachability of graphs that contain specific patterns (subgraphs). Patterns can be used to represent bad configurations of a protocol. For instance, 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 $h_c : \overline{n_c} \to \overline{m_c}$ and $h_s : \overline{n_s} \to \overline{m_s}$ 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 $\{G' | \ G \preceq G'\}$. 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). Unfortunately, it is not possible to completely solve this problem.

Proposition 2. *PRP is undecidable.*

Proof. Control state reachability for counter machines can be reduced to PRP. We use server node s_i to control the i-th counter. We use client processes with label aux_1 connected to a server with edges labelled 1 to represent the current value of the corresponding counter. We use a special client process c to keep track of the current control location and to fire the operations on counters. For instance, to simulate the instruction $L : \ inc(c_i); goto \ L'$, we proceed as follows. In state L client process c starts a transaction with server s_i. During the transaction, s_i, activated by c, accepts a new connection with an aux process, i.e., s_i changes the corresponding edge label to 1. We assume here that we can produce an arbitrary number of aux processes, and that they are always ready to connect to any server (they are created with state aux_0 and change state into aux_1 when connected to a server with an

$Instruction: \ell_1 : \ inc(c_i); goto\ \ell_2$

$(i_1)\ \langle\!\langle \ell_1 \rangle\!\rangle \ \Rrightarrow\ [\![wait_{\ell_2}]\!] \xleftrightarrow{req\_inc_i}$

$(i_2)\ (\!(c_i)\!) \xleftrightarrow{req\_inc_i} \ \Rrightarrow\ (\!(inc_i)\!) \xleftrightarrow{pend}$

$(i_3)\ (\!(inc_i)\!) \xleftrightarrow{0} \ \Rrightarrow\ (\!(grant_i)\!) \xleftrightarrow{1}$

$(i_4)\ (\!(grant_i)\!) \xleftrightarrow{pend} \ \Rrightarrow\ (\!(c_i)\!) \xleftrightarrow{ack}$

$(i_5)\ [\![wait_{\ell_2}]\!] \xleftrightarrow{ack} \ \Rrightarrow\ \langle\!\langle \ell_2 \rangle\!\rangle$

$Instruction: \ \ell_1 : \ dec(c_i); goto\ \ell_2$

$(d_1)\ \langle\!\langle \ell_1 \rangle\!\rangle \ \Rrightarrow\ [\![wait_{\ell_2}]\!] \xleftrightarrow{req\_dec_i}$

$(d_2)\ (\!(c_i)\!) \xleftrightarrow{req\_dec_i} \ \Rrightarrow\ (\!(dec_i)\!) \xleftrightarrow{pend}$

$(d_3)\ (\!(dec_i)\!) \xleftrightarrow{1} \ \Rrightarrow\ (\!(grant_i)\!) \xleftrightarrow{0}$

$(d_4)\ (\!(grant_i)\!) \xleftrightarrow{pend} \ \Rrightarrow\ (\!(c_i)\!) \xleftrightarrow{ack}$

$(d_5)\ [\![wait_{\ell_2}]\!] \xleftrightarrow{ack} \ \Rrightarrow\ \langle\!\langle \ell_2 \rangle\!\rangle$

$Instruction: \ \ell_1 : \ test(c_i); goto\ \ell_2$

$(t_1)\ \langle\!\langle \ell_1 \rangle\!\rangle \ \Rrightarrow\ [\![wait_{\ell_2}]\!] \xleftrightarrow{req\_test_i}$

$(t_2)\ (\!(c_i)\!) \xleftrightarrow{req\_test_i} \ \Rrightarrow\ (\!(test_i)\!) \xleftrightarrow{pend}$

$(t_3)\ (\!(test_i)\!) \ \Rrightarrow\ (\!(grant_i)\!) \ : \ \forall\{pend, 0\}$

$(t_4)\ (\!(grant_i)\!) \xleftrightarrow{pend} \ \Rrightarrow\ (\!(c_i)\!) \xleftrightarrow{ack}$

$(t_5)\ [\![wait_{\ell_2}]\!] \xleftrightarrow{ack} \ \Rrightarrow\ \langle\!\langle \ell_2 \rangle\!\rangle$

$Rules\ for\ \text{“}aux\text{''}\ processes$

$(a_1)\ \cdot \ \Rrightarrow\ \langle\!\langle aux_0 \rangle\!\rangle$

$(a_2)\ \langle\!\langle aux_0 \rangle\!\rangle \ \Rrightarrow\ [\![aux_1]\!] \xleftrightarrow{0}$

Fig. 3. Encoding of counter machines.

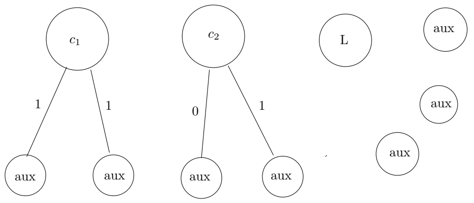

Fig. 4. Encoding of a configuration of a two counter machine with location L and counters $c_1 = 2$, and $c_2 = 1$.

edge labelled 0). After the connection is established, s_i sends an acknowledge back to c. c now stops the transaction and updates its label to L'. Notice that client c migrates from one server to another during the simulation of instructions.

Decrement and zero- test are simulated in a similar way. In particular, decrement can be simulated by requesting s_i to disable the connection with an *aux* process (e.g. for brevity to update the edge label from 1 to 0 or with some more transition to disconnect from s_i), and for the zero-test s_i checks that all of its incident edges have state different from 1. For this purpose, s_i can use a *test* rule.

A counter c_i with value k is represented by the graph in which the server for c_i is connected via edges labelled by 1 to k auxiliary processes (other auxiliary processes can be connected to the same server but the corresponding edges must be labelled 0).

We give an example in Fig. 4. The rules used in the encoding are shown in Fig. 3. □

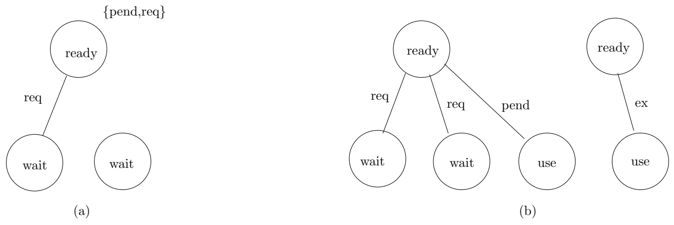

Fig. 5. A graph constraint G (a), and a c/s-graph in $[\![G]\!]$ (b).

4. 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.

4.1. *Graph constraints*

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 $\rho_c : \overline{n_c} \to \Lambda_c$, $\rho_s : \overline{n_s} \to (\Lambda_s \times 2^{\Lambda_e})$, and $\rho_e : E \to \Lambda_e$.

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 $h_c : \overline{n_c} \to \overline{m_c}$ and $h_s : \overline{n_s} \to \overline{m_s}$ 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 $[\![\Psi]\!] = \{G \mid G\ is\ a\ c/s-graph,\ \Psi \preceq G\}$.

In Fig. 5 we give an example of wfgc (a), and of one of its instances (b).

4.2. *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 Ψ we now define a relation $\rightsquigarrow_r$ working on wfgc's that we use to overapproximate the set $[\![\Psi]\!] \cup pre_r([\![\Psi]\!])$. We consider here the union of these two sets in order to be able to discard graph constraints that denote graphs already contained in $[\![\Psi]\!]$. 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 [25]. Specifically, for graph constraints Ψ, with $n_s = \mathsf{nsize}_s(\Psi)$ and $n_c = \mathsf{nsize}_c(\Psi)$, and Ψ', and a rule $r \in R$, the relation $\Psi \rightsquigarrow_r \Psi'$ is defined as follows:

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

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

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

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

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

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

  Although not explicitly present, we assume here that the edge (i,j) with label σ' is in the upward closure of Ψ (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.
- $i \in \overline{n_s}$, $\mathsf{label}_s(i,\Psi) = \ell'$, $\sigma' \in \mathsf{label}_p(i,\Psi)$, and

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

  where $Q = \mathsf{label}_p(i,\Psi) \cup \{\sigma\}$, and ℓ'' is non-deterministically chosen from Λ_c. Although not explicitly present, we assume here that both the client node $n_c + 1$ (with some label taken from Λ_c) and the edge (i, n_c+1) with label σ are in the upward closure of Ψ. We add them to Ψ 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 $\{\Psi' \mid \Psi \overset{r}{\rightsquigarrow} \Psi',\ r \in R\}$. From a case analysis of the definition of the predecessor operatore, we obtain the following property.

Proposition 3. $([\![\Psi]\!] \cup pre([\![\Psi]\!])) \subseteq ([\![\Psi]\!] \cup [\![\Psi \rightsquigarrow]\!])$.

4.3. *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 $h_c : \overline{n_c} \to \overline{m_c}$ $h_s : \overline{n_s} \to \overline{m_s}$ 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 4. *Given* Ψ *and* Ψ', $\Psi \sqsubseteq \Psi'$ *implies* $[\![\Psi']\!] \subseteq [\![\Psi]\!]$.

Proof. Let G be a configuration in $[\![\Psi']\!]$. Now let $G_{\Psi'}$ be the configuration obtained by removing the padding sets from the labels of server nodes. By definition, $G_{\Psi'}$ is a subgraph of G. Furthermore, all edges in G connected to server nodes in $G_{\Psi'}$ and such that they do not occur in $G_{\Psi'}$ must have labels contained in the padding set specified in Ψ'. Now we observe that $\sqsubseteq$ corresponds to the subgraph relation with the additional containment condition (third condition) on padding sets. Thus, by definition of $\sqsubseteq$, G_Ψ is a subgraph $G_{\Psi'}$. This implies that G_Ψ is a subgraph of G. Furthermore, by the containment conditions in $\sqsubseteq$ we have that the labels of edges connected to server nodes in G_Ψ (thus in $G_{\Psi'}$) are still contained in the corresponding padding sets in Ψ. It follows then that G is also in the denotation of Ψ. □

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 $\Psi \sqsubseteq \Psi'$.

4.4. *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 $[\![\Phi_j]\!] = (\bigcup_{0\le i\le j} [\![\Phi_i]\!])$. From Prop 3, 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.

4.5. *Termination*

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, $m_1 \le m_2$ 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).

5. Case Studies

5.1. *Full-map cache coherence protocol*

We consider here the *full-map cache coherence protocol* described in [19]. 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. The protocol is informally defined by the following steps; we use M to denote the controller of a memory line, and C to denote the controller of a cache line.

req_ex: C sends a **req_ex** message to M for requesting exclusive use of a memory line and then moves to a waiting state.

rec_inv: Upon reception of **req_ex** and when not processing other requests M locks the line, stores the identifier of the requesting cache controller, and starts an invalidation phase. In this phase M sends an message **req_inv** to all caches in the directory marked as *shared* or *exclusive*.

ack_inv: Upon receipt of msg **req_inv** coming from M, cache C invalidates its line and sends an acknowledgment **ack_inv** to M.

grant_ex: Memory controller terminates the invalidation phase after having received acknowledgments from all cache controllers in the directory. Memory controller sends a message **grant_ex** to the cache controller who sent the **req_ex** message and adds its identifies in the directory (marked as *exclusive*). A waiting cache receives the message **grant_ex** and moves to state **exclusive**.

req_sh: C sends a **req_sh** message to M for requesting shared use of a memory line and then moves to a waiting state.

rec_dg: Upon reception of **req_sh** and when not processing other requests M locks the line, stores the identifier of the requesting cache controller, and starts a downgrade phase. In this phase M sends an message **req_sh** to all caches in the directory marked as *shared* or *exclusive*.

ack_dg: Upon receipt of msg **req_sh** coming from M, cache C (in state *shared* or *exclusive*) downgrade its state to *shared* and sends an acknowledgment **ack_sh** to M.

grant_ex: Memory controller terminates the downgrade phase after having received acknowledgments from all cache controllers in the directory. Memory controller sends a message **grant_sh** to the cache controller who sent the **req_sh** message and adds its identifier to the directory (marked as *shared*). A waiting cache receives the message **grant_sh** and moves to state **shared**.

replacement: As a result of a replacement strategy, the cache controller invalidates a cache line in state **exclusive/shared**.

We model this protocol by means of a bi-partite graph system with a set of labels Λ_n partitioned in the two sets

$$\begin{array}{ll} \Lambda_n^C = \{inv, wait, shared, exclusive\} & (cache\ nodes) \\ \Lambda_n^S = \{idle, inv\_loop, dg\_loop, ack\_ex, ack\_sh\} & (memory\ nodes) \end{array}$$

and a set of edge labels

$$\Lambda\_e = \{req\_ex, req\_sh, req\_inv, req\_dg, pending, sh, ex\}$$

We recall that in our model we assume to have an arbitrary number of memory line each one controlled by a distinct memory node with label in λ_n^S and an arbitrary number of cache lines each one controlled by one controller with label in λ_n^C.

The initial graph configurations consist then of any number of isolated nodes with label *inv* or *idle*. Furthermore, we can use the rules

$$\cdot \ \Rrightarrow \ \langle\!\langle inv \rangle\!\rangle \qquad\qquad \cdot \ \Rrightarrow \ \langle\!\langle idle \rangle\!\rangle$$

to add dynamic creation of new cache and memory line controllers. 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 in several cache lines. Thus, we are only interested in graphs in which a cache node has at most one incident edges, while there are no upper bounds on the number of edges incident to a memory node. To avoid to generate unreachable configuration, we can embed this restriction in the symbolic operations (e.g. in the construction of predecessors) for this kind of graph systems.

The graph rules that model the interaction between the controller of a cache line and that of a memory line are shown in Fig. 6. With rule c_1 and c_2, a cache controller sends resp. a *req_ex* and a *req_sh* message to a memory controller chosen non-deterministically, and then moves to a waiting state. A request is modeled here as creation of a new edge. As mentioned before, we assume that a cache node has at most one edge. The non-determinism in the choice of the server node models the fact that the association between a cache line and a memory line depends from the memory address contained in the operations issued on the local processor. Rule c_3 and c_4 model the reception of an invalidation request and the invalidation of the cache node. Rule c_5 models the reception of a downgrade request coming from a server node and the consequent passage from state *exclusive* to state *shared*. The label of the edge is changed to *sh*. Rule c_6 and c_7 model reception (in state *wait*) of the grant for exclusive (edge with label *ex*) or shared use (edge with label *sh*) for the requested memory line. Rule c_8 and c_9 model the invalidation of the cache line requested by the local processor (e.g. for replacing the current cache line with another memory line). After applying one of this rule, a cache node disconnected

$$
\begin{array}{l}
(c_1)\ \langle\!\langle inv \rangle\!\rangle \ \Rrightarrow\ [\![wait\_ex]\!] \overset{req\_ex}{\longleftrightarrow} \\
(c_2)\ \langle\!\langle inv \rangle\!\rangle \ \Rrightarrow\ [\![wait\_sh]\!] \overset{req\_sh}{\longleftrightarrow} \\
(c_3)\ [\![shared]\!] \overset{req\_inv}{\longleftrightarrow} \ \Rrightarrow\ \langle\!\langle inv \rangle\!\rangle \\
(c_4)\ [\![exclusive]\!] \overset{req\_inv}{\longleftrightarrow} \ \Rrightarrow\ \langle\!\langle inv \rangle\!\rangle \\
(c_5)\ \langle\!\langle exclusive \rangle\!\rangle \ \Rrightarrow\ [\![req\_dg]\!] \overset{shared}{\longleftrightarrow} sh \\
(c_6)\ [\![wait\_ex]\!] \overset{ex}{\longleftrightarrow} \ \Rrightarrow\ [\![exclusive]\!] \overset{ex}{\longleftrightarrow} \\
(c_7)\ [\![wait\_sh]\!] \overset{sh}{\longleftrightarrow} \ \Rrightarrow\ [\![shared]\!] \overset{sh}{\longleftrightarrow} \\
(c_8)\ [\![shared]\!] \overset{sh}{\longleftrightarrow} \ \Rrightarrow\ \langle\!\langle inv \rangle\!\rangle \\
(c_9)\ [\![exclusive]\!] \overset{ex}{\longleftrightarrow} \ \Rrightarrow\ \langle\!\langle inv \rangle\!\rangle
\end{array}
$$

Fig. 6. Rules for cache controllers.

$$
\begin{array}{l}
(m_1)\ [\![idle]\!] \overset{req\_ex}{\longleftrightarrow} \ \Rrightarrow\ [\![inv\_loop]\!] \overset{pend}{\longleftrightarrow} \\
(m_2)\ [\![inv\_loop]\!] \overset{ex}{\longleftrightarrow} \ \Rrightarrow\ [\![inv\_loop]\!] \overset{req\_inv}{\longleftrightarrow} \\
(m_3)\ [\![inv\_loop]\!] \overset{sh}{\longleftrightarrow} \ \Rrightarrow\ [\![inv\_loop]\!] \overset{req\_inv}{\longleftrightarrow} \\
(m_4)\ (\!(inv\_loop)\!) \ \Rrightarrow\ (\!(ack\_inv)\!) \ :\ \forall\{pend, req\_sh, req\_ex\} \\
(m_5)\ [\![ack\_inv]\!] \overset{pend}{\longleftrightarrow} \ \Rrightarrow\ [\![idle]\!] \overset{ex}{\longleftrightarrow} \\
(m_6)\ [\![idle]\!] \overset{req\_sh}{\longleftrightarrow} \ \Rrightarrow\ [\![inv\_dg]\!] \overset{pend}{\longleftrightarrow} \\
(m_7)\ [\![inv\_dg]\!] \overset{ex}{\longleftrightarrow} \ \Rrightarrow\ [\![inv\_dg]\!] \overset{req\_dg}{\longleftrightarrow} \\
(m_8)\ (\!(inv\_dg)\!) \ \Rrightarrow\ (\!(ack\_dg)\!) \ :\ \forall\{pend, sh, req\_sh, req\_ex\} \\
(m_9)\ [\![ack\_dg]\!] \overset{pend}{\longleftrightarrow} \ \Rrightarrow\ [\![idle]\!] \overset{sh}{\longleftrightarrow}
\end{array}
$$

Fig. 7. Rules for memory controllers.

from memory node m can be connected to another memory node m' using rules c_1 and c_2, and so on. The graph rules that model the interaction between the controller of a memory line and the cache controllers are shown in Fig. 7. Rules $m_1 - m_5$ model the steps needed to grant a *req_ex* request for a memory line M. In rule m_1 a memory node locks the memory line M by moving to state *inv_loop* (starting point of invalidation phase) and remembers the identity of the corresponding requesting node by marking the edge with label *pending*. Rule $m_2 - m - 4$ model the invalidation phase. We use again edges to simulate the inspection of the full-map. For each edge with label ex/sh we send a *req_inv* message to the corresponding node. Notice that we do not inspect the current state of the nodes. Invalidation is only based on edge labels. Rule m_4 corresponds to the termination of the invalidation step. We require here that all *ex* and *req_inv* messages have adequately been processes during the invalidation phase. In rule m_5 we grant exclusive use to the requesting cache controller.

$$(c_8)\ \langle\!\langle shared \rangle\!\rangle\ \Rrightarrow\ [\![sh]\!]\ \overset{shared}{\longleftrightarrow}\ req\_rep$$
$$(c_9)\ \langle\!\langle exclusive \rangle\!\rangle\ \Rrightarrow\ [\![ex]\!]\ \overset{exclusive}{\longleftrightarrow}\ req\_rep$$

Fig. 8. Additional rules for cache controller.

Similarly, rules $m_6 - m_9$ model the step needed to gran a *req_sh* request. The difference here is that instead of invalidation request we only send downgrade request to a cache node marked as exclusive (with edge *ex*). Rule m_8 terminates this phase by checking that there are no pending *req_dg* request and that the fullmap has been completely inspected. Finally, rule m_9 is used to grant shared access to the requesting cache (that connected with the server node with a *pending* edge).

5.2. *Optimized full-map cache coherence protocol*

We now consider an optimized version [19] of the full-map coherence protocol in which 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). We describe below how the protocol changes.

rec_ex The optimization in the processing of message req_ex is the following. When $mode\_ex = 1$ M sends a message **req_inv** only to the cache marked as *exclusive.*

rec_sh The optimization in the processing of message req_sh is the following. When $mode\_ex = 1$ M sends a message **req_dg** only to the cache marked as *exclusive.*

replacement In the optimized version we need a more detailed description of this phase. The cache controller sends a replacement message to the memory controller. The memory controller resets the corresponding bit in the fullmap and sends back an invalidation request to the cache controller. If $mode\_ex = 1$ then the flag is set to 0.

We model the optimized protocol in the following way. Consider a set of labels Λ_n partitioned in the two sets. First, we add the label *req_rep* to Λ_e and the new labels $idle_{ex}, ex\_inv_1, ex\_inv_2, ex\_inv_3, ex\_inv_4$ for server nodes. A cache controller is modelled via rules $c_1, \ldots, c_7$ of Fig. 6 plus the two new rules of Fig. 8 that describe a replacement message sent to the memory controller: A memory controller is modelled via rules $(m_1) - (m_4), (m_6) - (m_{10})$ in addition to the following new rules: The new rule m_5 is used to set the *mode_ex* flag to 1 (represented by state *idle*). Rules $m_{10} - m_{11}$ are used to handle pending *req_rep* during an invalidation loop with $mode\_ex = 0$. Rules $m_{12} - m_{14}$ (for *req_ex*) and $m_{15} - m_{19}$ (for *req_sh*) model the fast invalidation phases required when $mode\_ex = 1$ (we just have to invalidate one cache controller). Rules $m_{20} - m_{21}$ model the acknowledgment to *req_rep* messages resp. when $mode\_ex = 0$ and $mode\_ex = 1$. In the latter case the flag is set to zero (i.e. $idle_{ex}$ is updated to *idle*).

$$
\begin{array}{ll}
(m_5) & [\![ack\_inv]\!] \overset{pend}{\longleftrightarrow} \;\Rrightarrow\; [\![idle_{ex}]\!] \overset{ex}{\longleftrightarrow} \\
(m_{10}) & [\![inv\_loop]\!] \overset{req\_rep}{\longleftrightarrow} \;\Rrightarrow\; [\![inv\_loop]\!] \overset{req\_inv}{\longleftrightarrow} \\
(m_{11}) & [\![inv\_dg]\!] \overset{req\_rep}{\longleftrightarrow} \;\Rrightarrow\; [\![inv\_dg]\!] \overset{req\_inv}{\longleftrightarrow} \\
(m_{12}) & [\![idle_{ex}]\!] \overset{req\_ex}{\longleftrightarrow} \;\Rrightarrow\; [\![ex\_inv_1]\!] \overset{pend}{\longleftrightarrow} \\
(m_{13}) & [\![ex\_inv_1]\!] \overset{ex}{\longleftrightarrow} \;\Rrightarrow\; [\![ex\_inv_2]\!] \overset{req\_inv}{\longleftrightarrow} \\
(m_{14}) & [\![ex\_inv_1]\!] \overset{req\_rep}{\longleftrightarrow} \;\Rrightarrow\; [\![ex\_inv_2]\!] \overset{req\_inv}{\longleftrightarrow} \\
(m_{15}) & [\![idle_{ex}]\!] \overset{req\_sh}{\longleftrightarrow} \;\Rrightarrow\; [\![ex\_inv_3]\!] \overset{pend}{\longleftrightarrow} \\
(m_{16}) & [\![ex\_inv_3]\!] \overset{ex}{\longleftrightarrow} \;\Rrightarrow\; [\![ex\_inv_4]\!] \overset{req\_inv}{\longleftrightarrow} \\
(m_{17}) & [\![ex\_inv_3]\!] \overset{req\_rep}{\longleftrightarrow} \;\Rrightarrow\; [\![ex\_inv_4]\!] \overset{req\_inv}{\longleftrightarrow} \\
(m_{18}) & (\!(ex\_inv_2)\!) \;\Rrightarrow\; (\!(ack\_inv)\!) \;:\; \forall\{pend, req\_sh, req\_ex\} \\
(m_{19}) & (\!(ex\_inv_4)\!) \;\Rrightarrow\; (\!(ack\_dg)\!) \;:\; \forall\{pend, req\_sh, req\_ex\} \\
(m_{20}) & \langle\!\langle idle\rangle\!\rangle \;\Rrightarrow\; [\![req\_rep]\!] \overset{idle}{\longleftrightarrow} req\_inv \\
(m_{21}) & \langle\!\langle idle_{ex}\rangle\!\rangle \;\Rrightarrow\; [\![req\_rep]\!] \overset{idle}{\longleftrightarrow} req\_inv
\end{array}
$$

Fig. 9. Additional rules for a memory controller.

5.2.1. *Verification problems*

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 type 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 (before or after sending a *req_rep* message) we can first add the following rule (here *bad* is a new memory label):

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

where *bad* is a new memory label and $Q = \Lambda_e \setminus \{req\_rep, ex\}$.

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 [25].

6. 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 [19]. 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] P. A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziz, and A. Rezine. Monotonic abstraction for programs with dynamic memory heaps. In Proc. CAV 2008: 341-354.

[2] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In Proc. LICS 1996: 313–321.

[3] P. A. Abdulla, N. Ben Henda, G. Delzanno, and A. Rezine. Regular model checking without transducers. In Proc. TACAS 2007: 721–736.

[4] P. A. Abdulla, N. Ben Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In Proc. VMCAI 2008: 22-36.

[5] P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In Proc. CAV 2007: 145–157.

[6] P. A. Abdulla, G. Delzanno, and A. Rezine. Approximated Context-sensitive Analysis for Parameterized Verification In Proc. FORTE 2009: 36–50.

[7] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model checking made simple and efficient. In Proc. CONCUR 2002: 116–130.

[8] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In Proc. CAV 2001: 221–234.

[9] B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In Proc. CAV 2003: 223–235.

[10] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In Proc. CAV 2004: 372–386.

[11] A. Bouajjani, A. Muscholl, and T. Touili. Permutation Rewriting and Algorithmic Verification. Inf. and Comp., 205(2): 199-224, 2007.

[12] E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In Proc. VMCAI 2006: 126-141.

[13] G. Delzanno. Constraint-Based Verification of Parameterized Cache Coherence Protocols. FMSD 23(3): 257-301 (2003)

[14] M. Emmi, R. Jhala, E. Kohler, and R. Majumdar. Verifying reference counted objects. In Proc. TACAS 2009: 262-276.

[15] J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In Proc. LICS 1999: 352-359.

[16] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! TCS 256(1-2):63–92, 2001.

[17] S. Joshi and B. König. Applying the graph minor theorem to the verification of graph transformation systems. In Proc. CAV 2008: 214–226.

[18] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. TCS 256: 93–112, 2001.

[19] F. Pong, M. Dubois. Correctness of a Directory-Based Cache Coherence Protocol: Early Experience. In Proc. SPDP 1993: 37-44

[20] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In Proc. TACAS 2001: 82–97.

[21] M. Saksena and O. Wibling and B. Jonsson. Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols. In Proc. TACAS 2008: 18–32.

[22] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In Proc. LICS 1986: 332–344.

[23] T. Yavuz-Kahveci, T. Bultan. A symbolic manipulator for automated verification of reactive systems with heterogeneous data types. STTT 5(1): 15-33, 2003.

[24] T. Yavuz-Kahveci, T. Bultan. Verification of parameterized hierarchical state machines using action language verifier. In Proc. MEMOCODE 2005: 79-88.

[25] Symgraph: `http:\\www.disi.unige.it\person\DelzannoG\Symgraph\`.