Automatic Verification of Directory-based Consistency Protocols with Graph Constraints

Parosh Aziz Abdulla

Uppsala University, Sweden

Giorgio Delzanno

Università di Genova, Italy

Ahmed Rezine

University of Uppsala, Sweden

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. 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 $\overline{n} = \{1, \ldots, n\}$. 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),

- $\lambda_c : \overline{n_c} \to \Lambda_c$, $\lambda_s : \overline{n_s} \to \Lambda_s$, 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;
- $\mathsf{update}_c(i_1 \leftarrow P_1, \ldots, i_m \leftarrow P_m, G) = (n_c, n_s, E, \lambda'_c, \lambda_s, \lambda_e)$ 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 $x \in \{e, c\}$;

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.

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 $l \Rightarrow r$: 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,

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

Fig. 1. Rewriting rules with conditions on egdes.

- $[\ell] \xleftrightarrow{\sigma}$ that denotes a client node with label ℓ and incident edge with label σ;
- $(\!(\ell)\!) \xleftrightarrow{\sigma}$ 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 of a resource after invalidating the client that is currently using it.*

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

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