Parameterized Verification of Topology-sensitive Distributed Protocols goes Declarative

Sylvain Conchon[1], Giorgio Delzanno[2], and Angelo Ferrando[3]

[1] LRI, Université Paris-Sud, sylvain.conchon@lri.fr

[2,3] DIBRIS, University of Genova, {giorgio.delzanno,angelo.ferrando}@unige.it

Abstract. We show that Cubicle [10], an SMT-based infinite-state model checker, can be applied as a verification engine for GLog, a logic-based specification language for topology-sensitive distributed protocols with asynchronous communication. Existential coverability queries in GLog can be translated into verification judgements in Cubicle by encoding relational updates rules as unbounded array transitions. We apply the resulting framework to automatically verify a distributed version of the Dining Philosopher mutual exclusion protocol formulated for an arbitrary number of nodes and communication buffers.

1 Introduction

Automated verification of distributed systems is a difficult task for standard model checkers [8,9]. Protocols designed to operate in distributed systems are often defined for an arbitrary number of nodes, arbitrary connection topology, and asynchronous communication. protocol rules typically depend on the current network configuration (e.g, presence of a communication link, state of all connections, etc). Several formal languages have been proposed to specify this class of systems, e.g., communicating state machines, automata, process algebraic languages, (graph) rewriting, etc. In this setting safety properties can be nicely formulated by lifting decision problems based on reachability and coverability, in which the initial configuration is typically fixed a priory, to formulations that are existentially quantified over an infinite set of initial configurations. Existentially quantified coverability problems have been considered in [13,14,15,7,6] in order to reason on parameterized formulation of distributed protocols with broadcast communication. The coverability decision problem [1] is typically used to formulate reachability of bad configurations independently from the number of components of a system. Therefore, a constructive way to solve an existentially quantified coverability problem for a formal specification of a distributed algorithm provides a characterization of initial configurations from which it is possible to reach a bad configuration (e.g. an anomaly in the protocol). Existentially quantified coverability problems turn out to be undecidable even for systems with a static communication topology and very basic interaction primitives like atomic broadcast communication [1,12,13,14,8]. As mentioned before, communicating state machine (automata), adopted in [13,14,15], and graph rewriting, adopted

in [6], are two examples of formal description languages for this kind of systems. communicating state machines can be considered a standard design and protocol specification language adopted in several verification tools like Uppaal and Spin. Graph rewriting systems are well-suited for representing topology-sensitive rules as shown by the examples of the Groove tool suite. In several case-studies, protocol rules require complex guards that require the inspection of the state of nodes, links, paths, vicinity etc. and tables to store information collected during the execution of protocol phases. The combination of these features seems to require more general specification formalisms. To this aim, in [11] we proposed to adopt a logic-based declarative language, named GLog, a fragment of both DCDCs [11] and MSR [22]. GLog can be viewed as a logic-based presentation of graph update rules with global conditions expressed using quantified first order formulas. GLog is based on a quantified predicate logic in a finite relational signature with no function symbols. Configurations are represented here as sets of ground atomic formulas (instances of unary and binary predicates). Update rules consist of a guard and two sets of first order predicates that define resp. deletion and addition of state components. Differently from specification languages based on extension of Petri nets like transfer and broadcast protocols, guards are checked atomically but update transitions have only local effect. In other words, we forbid simultaneously update of the state of all nodes in a graph. Update rules can be applied to update a global configuration node by node and to operate on the vicinity of a node by restricting updates to given predicates. Termination of an update subprotocol can then be checked via a global condition. Similar specification patterns have been applied to model non-atomic consistency protocol and mutual exclusion protocols with non-atomic global conditions. GLog has been applied to manually analyze distributed protocols in [11]. In the present paper we show that Cubicle [10,17], an SMT-based infinite-state model checker based on previous work by Ghilardi et al. [4], can be applied as automated verification engine for existentially quantified coverability queries in GLog. In Cubicle parameterized systems can be specified as unbounded arrays in which individual components can be referred to via an array index. The Cubicle verification engine performs a symbolic backward reachability analysis using an SMT solver for computing intermediate steps (preimage computation, entailment and termination test) and applies overapproximates predecessors using upward closed sets as in monotone abstractions [2,3]. A peculiar feature of Cubicle w.r.t. MCMT [4] is that the tool can handle unbounded matrices. This is particularly relevant when modeling topology-sensitive protocols as done in GLog using binary relations defined over component identifiers. Furthermore, existentially quantified coverability decision problems in GLog can directly be mapped into Cubicle. More specifically, the encoding transforms GLog update rules into array-based update formulas in Cubicle. Classes of initial configurations are specified by using partial specifications of initial configurations in Cubicle verification judgements. Infinite sets of bad configurations can be expressed using unsafe configurations in Cubicle verification judgements.

As a case-study, we consider the distributed version of the Dining Philosopher mutual exclusion protocol (DDP) recently studied in [19,16,11]. The protocol deals with an arbitrary, finite number of nodes and buffers that act as single-place communication channels, and arbitrary link topology between nodes and buffers. Ownership of buffers is specified using asynchronous rules. Global conditions over linked buffers are used as enabling conditions for acquiring access to resources shared among neighbors. The GLog formal specification of DDP is mapped to a Cubicle verification problem in a natural way. In our preliminary experiments, Cubicle verified the correctness of the protocols in negligible time. Furthermore, as expected it reports potential error traces when introducing network reconfiguration rules (e.g. dynamic link creation and deletion) unrelated to the state of the corresponding involved nodes. The application of declarative specification languages and SMT-based engine seems a very promising research line for dealing for a larger class of distributed algorithms.

Contents The paper is organized as follows: we first present the GLog declarative language and, in that context, the existential coverabiliity problem; we then introduce Cubicle and exhibit a general encoding of existential coverability into Cubicle; we then discuss the experimental evaluation on the DDP case-study, and, finally, discuss other examples and future directions.

2 GLog

GLog [11] formulas are based on a simple relational calculus that can be used to express updates of sets of ground atoms. A set of ground atoms can be interpreted as the current state or configuration of the system we are modeling. Update rules contain a formula working as a condition and deletion and addition sets that specify ground atoms to be deleted and added to the current state. More formally, let P be a finite set of names of (unary and binary) predicate names, $\mathcal{N}$ a denumerable set of node identifiers equipped with a total order $<$, V be a denumerable set of variables. Predicates in P are used to model current configurations. In addition to predicates in P, we interpret the binary relation lt as the total order $<$ in our model. Our logic has no function symbols but can be instantiated with elements from $\mathcal{N}$. An atomic formula is either a formula $p(x)$, $lt(x, y)$ or $p(x, y)$, where $p \in P$, $x, y \in V \cup \mathcal{N}$ A ground atom is a either a $p(n)$, $lt(n, m)$, or $p(n, m)$, where $n, m \in \mathcal{N}$. A literal is either an atomic formula or the negation $\neg A$ of an atomic formula A. A formula is a first order formula built on literals, namely, any literal is a formula, conjunctions, disjunctions, universally and existentially quantified formulas are still formulas. Multiple occurrences of the same variable implicitly model equality constraints. The set of free variables of a formula F, namely $FV(F)$, is the minimal set satisfying $FV(p(x, y)) = \{x, y\}$, $FV(A \vee B) = FV(A) \cup FV(B)$, $FV(A \wedge B) = FV(A) \cap FV(B)$, $FV(\neg A) = FV(A)$, $FV(\forall v.A) = FV(A) \setminus \{v\}$, and $FV(\exists v.A) = FV(A) \setminus \{v\}$. Given $S = \{F_1, \ldots, F_n\}$, we define $FV(S) = FV(F_1) \cup \ldots \cup FV(F_n)$. Quantified formulas we will be used as application conditions of rules.

Configurations, Interpretations and Update Rules As mentioned before a set of ground atoms will be used to model a configuration. Formally, a configuration is a finite set Δ of ground atomic formulas with predicates in P. A configuration implicitly defines a graph in which directed edges are represented by atomic formulas whose predicate name acts as edge label. Configurations can also be viewed as models in which to evaluate a conditions. An interpretation is a mapping σ from V to $\mathcal{N}$. We use here a fixed interpretation of variables. The interpretation domain however consists of a denumerable set of node identifiers. For a formula F we use $F\sigma$ as an abbreviation for $\hat{\sigma}(F)$, where $\hat{\sigma}$ is the natural extension of σ to terms. For a set $S = \{A_1, \ldots, A_n\}$, we use $S\sigma$ to denote the set $\{A_1\sigma, \ldots, A_n\sigma\}$. Update rules consists of conditions defined by quantified formulas with no function symbols, a deletion and an addition set. The deletion (resp. addition) set defines the set of ground atoms that have to be cancelled from (resp. added to) the current configuration. A rule has the following form $\langle C, D, A\rangle$, where C is a quantified formula, D and A are two sets of atomic formulas with variables in V and predicates in P, and such that $FV(A) \cup FV(D) \subseteq FV(C)$. A protocol $\mathcal{P}$ is defined as a set of rules.

Operational Semantics To fix an operational semantics for our language we need a support for the interpretation of relations and variables. We use $\Delta \models A$ to define the satisfiability relation of a quantified formula A s.t. $FV(A) = \emptyset$. Let $A[n/X]$ denote the formula obtained by replacing each free occurrence of X with n. The relation is defined by induction as follows. $\Delta \models p(n)$, if $p(n) \in \Delta$, $\Delta \models lt(n, m)$, if $n < m$, $\Delta \models p(n, m)$ for $p \in P$, if $p(n, m) \in \Delta$, $\Delta \models A \wedge B$, if $\Delta \models A$ and $\Delta \models B$, $\Delta \models \neg A$, if $\Delta \not\models A$, $\Delta \models \forall X.A$, if $\Delta \models A[n/X]$ for each $n \in \mathcal{N}$, and $\Delta \models \exists X.A$, if $\Delta \models A[n/X]$ for some $n \in \mathcal{N}$. Given a configuration Δ, we say that the quantified formula A is satisfied in Δ, if there exists an interpretation σ s.t. $A\sigma$ is satisfiable. In order to apply a rule $\langle C, D, A\rangle$ to Δ, there must be an interpretation σ that satisfies the quantified formula C. The same interpretation σ is then applied to the atomic formulas in D and A. The resulting sets of atoms, say D' and A' respectively, are deleted from and added to Δ, respectively.

The operational semantics of a protocol $\mathcal{P}$ is given by a transition system $T_{\mathcal{P}} = \langle \mathcal{C}, \rightarrow\rangle$, where $\mathcal{C}$ is the set of possible configurations, i.e., finite subsets of ground atoms with predicates in P, and $\rightarrow \subseteq \mathcal{C} \times \mathcal{C}$ is a relation defined as follows. For $\Delta, \Delta' \in \mathcal{C}$ and a rule $\langle C, D, A\rangle \in \mathcal{P}$, $\Delta \rightarrow \Delta'$ if there exists σ s.t. $\Delta \models C\sigma$ and $\Delta' = (\Delta \setminus D\sigma) \cup A\sigma$. A computation is a sequence of configurations $\Delta_0\Delta_1 \ldots$ s.t. $\Delta_i \rightarrow \Delta_{i+1}$ for $i \geq 0$. We use $\rightarrow^*$ to denote the reflexive and transitive closure of $\rightarrow$. In a single step of the operational semantics a rule is evaluated in the current configuration by taking a sort of closed-world assumption, i.e., ground atomic formulas that do not occur in a configuration are evaluated to false. Furthermore, ground atomic formulas that are not deleted are transferred from the current to the successor configuration. The latter property can be viewed then as a sort of frame axiom. It is important to notice that, in general, a configuration Δ has several possible successors. Indeed, depending of the chosen interpretation of

free variables the same rule can be applied to different subsets of ground atoms contained in the same configuration. Furthermore, the choice of the rules to be applied at a given step is non-deterministic.

As an example, we consider possible application of GLog to the specification of distributed protocols. The key ingredient of the specification language is the combination of complex conditions and update rules to reason on graphs in which predicates can be viewed as labels of links between agents and communication buffers. We have shown that we can also add labels to individual agents and buffers, e.g., to represent their current state. Update rules can be used to dynamically reconfigure the graph, i.e., change labels, topology and add or delete agents. The separation between agents and buffers is convenient to model asynchronous communication. For instance, let us consider a protocol in which two agents need to establish a connection via a shared buffer.

- An agent n_1 of type A connects to a buffer e_1 in idle state (the buffer is free) and sets the state of the buffer to *ready*.
- An agent n_2 of type B connects to e_1 in state *ready* and changes the state to *ack*.
- Agent n_1 sends message m by changing the state of e_1 to msg_m.
- Agent n_2 receives message m and updates the state of the channel to *ack* for further communications.

The protocol can be specified as follows. We use unary predicates to associate states to edges. *send* messages are non-deterministically generated. An initial configuration has the form $idle(b_1), \ldots, idle(b_k)$, where $b_i < b_j$for $i \neq j$, $i, j : 1, \ldots, k$. For the sake of simplicity, we do not model the state of agents but only their capabilities (req, rec, send).

R	C	D	A
1	$idle(B) \wedge \neg req(A, B)$	$\{idle(B)\}$	$\{ready(B), req(A, B)\}$
2	$ready(B) \wedge \neg rec(A, B)$	$\{ready(B)\}$	$\{ack(B), rec(A, B)\}$
3	$true$	$\{\}$	$\{send(A, B, M)\}$
4	$ack(B) \wedge send(A, B)$	$\{ack(B), send(A, B)\}$	$\{msg(B, M)\}$
5	$msg(B, M) \wedge rec(A, B)$	$\{msg(B, M), rec(A, B)\}$	$\{idle(B)\}$

In rule 1 a buffer B is locked by a non-deterministically generated request $req(A, B)$ from sender agent A (a variable). In rule 2 a buffer B is locked by a non-deterministically generated request $rec(A, B)$ from receiver agent A (a variable). Rule 3 nondeterministically generates a send action from agent A. Rule 4 synchronizes a send action from agent A with a buffer locked by the same agent. The (non deterministically generated) message M is stored in the buffer. Rule 5 synchronizes and consumes a message in the buffer with the receiver agent, releasing the buffer.

The model provides other form of interactions. For instance, we can model ordered buffers by forming lists of messages attached to a given edge as in the representation of the tape of the Turing machine.

We can also model synchronous communication as in the following example

C	$link(A,B) \wedge s_1(A) \wedge link(E,B) \wedge s_2(E)$
D	$\{s_1(A), link(A,B), link(E,B), s_2(E)\}$
A	$\{link(A,B), s'_1(A), link(E,B), s'_2(E)\}$

Here $s(A)$ and $s'(A)$ denote agent A resp. in state s and s', $s_1(E)$ and $s'_1(E)$ denote agent E resp. in state s_1 and s'_1, and $link(A,B)$ and $link(Y,B)$ denote links to a common buffer B.

2.1 Existential Coverability

We consider here decision problems that generalize the standard notion of reachability between configurations. The key point is to reason about an infinite set of initial configurations in order to prove properties for protocol instances with an arbitrary number of nodes. For a set S of configurations, we first define the $Post$ and Pre operators as follows $Post(S) = \{\Delta' \mid \exists \Delta \in S,\ \Delta \rightarrow \Delta'\}$ and $Pre(S) = \{\Delta' \mid \exists \Delta \in S,\ \Delta' \rightarrow \Delta\}$. We use $Post^*(S)$ (resp. $Pre^*(S)$) to denote the reflexive-transitive closure of $Post$ (resp. Pre).

We now introduce the $\exists$-coverability problem as follows.

Definition 1 ($\exists$-coverability). *Given a protocol $\mathcal{P}$, a set of target configurations T and a possibly infinite set of initial configurations I, $\exists$-coverability is satisfied for $\mathcal{P}$, I and T, written $\exists Reach(\mathcal{P}, I, T)$, if there exists $\Delta \in T$ and a configuration Δ_1 s.t. $\Delta_1 \in Post^*(I)$ and $\Delta \subseteq \Delta_1$.*

By expanding the definition of $Post^*$, $\exists Reach(\mathcal{P}, I, T)$ holds if there exists a configuration $\Delta_0 \in I$ s.t. $\Delta_0 \rightarrow^* \Delta_1$ and $\Delta \subseteq \Delta_1$ for some $\Delta \in T$. The target T can be interpreted as a pattern to match or avoid in computations starting from initial configurations. If the set I consists of configurations consisting of an arbitrary, finite number of components than $\exists$-coverability formally describes a parameterized verification decision problem for specifications given in GLog. The $\exists$-coverability problem turns out to be undecidable [11].

3 From GLog to Cubicle

Cubicle is a model checker that can be applied to verify safety properties of array-based systems, a syntactically restricted class of parametrized transition systems with states represented as arrays indexed by an arbitrary number of processes [10,17]. Cache coherence protocols and mutual exclusion algorithms are typical examples of such systems. Cubicle model-checks by a symbolic backward reachability analysis on infinite sets of states represented by specific simple formulas, called cubes. Cubicle is written in OCaml. The SMT solver is a tightly integrated, lightweight and enhanced version of Alt-Ergo [20]; and its parallel implementation relies on the Functory library [21]. Cubicle input language is a typed version of Murphi similar to the one of Uclid. A system is described in Cubicle by: (1) a set of type, variable, and array declarations; (2) a formula for the initial states; and (3) a set of transitions. It is parametrized by a set of process identifiers,

denoted by the built-in type proc. Standard types int, real, and bool are also built in. Additionally, the user can specify abstract types and enumerations with simple declarations like `type data` and `type msg = Empty | Req | Ack`. As an example consider the following declaration.

```
var Turn : proc
array Want[proc] : bool
array Crit[proc] : bool

init (z) { Want[z] = False && Crit[z] = False }

unsafe (x y) {
   Crit[x] = True && Crit[y] = True }
```

The system state is defined by a set of global variables and arrays. The initial states are defined by a universal conjunction of literals characterizing the values for some variables and array entries.

```
init (z) { Want[z] = False && Crit[z] = False }
```

A state of our example consists of a process identifier Turn and two boolean arrays Want and Crit; a state is initial iff every cell of both arrays are set to false Transitions are given in the usual guard/action form and may be parameterized by (one or more) process identifiers. Guards are expressed via `required` expressions. They are quantified formulas. Quantification is defined only over variables of type *proc*. As an example, consider the following rule.

```
transition req (i)
requires { Want[i] = False }
              { Want[j] := case
                 | i = j : True
                 | _: Want[j]  }
```

The transition `req(i)` is enabled if there exists index `i` (a process) such that `Want[i]=false`. Its effect is to set `Want` to true for index `i` and leave the array unchanged in all other positions. A system execution is defined by an infinite loop that at each iteration: (1) non-deterministically chooses a transition instance whose guard is true in the current state; and (2) updates state variables according to the action of the fired transition instance.

Infinite sets of unsafe states (bad configurations) are defined by using $unsafe$ constraints. For instance, the judgement

```
unsafe (x y) {
   Crit[x] = True && Crit[y] = True }
```

specifies the infinite set of arrays $Crit$ (with any size) in which there there exist two cells with value $True$.

The Cubicle verification engine is based on symbolic backward exploration. Cubicle operates over sets of existentially quantified formulas called cubes. Formulas containing universally quantified formulas (generated during the computation of predecessors) are over-approximated by existentially quantified formulas. The class of formulas manipulated by the backward reachability loop of Cubicle in not closed by pre-image in presence of universally quantified guards. To handle such formulas, Cubicle implements a safe but over-approximate pre-image computation. Given a cube $\exists\bar{i}.\Phi$ and a guard G of the form $\forall\bar{j}.\Psi(\bar{j})$, the pre-image replaces G by the conjunction $\bigwedge_{\sigma\in\Sigma(\bar{j},\bar{i})}\Psi(\bar{j})\sigma$ of instances over the permutation of $\Sigma(\bar{j},\bar{i})$. In other words, in order to handle universally quantified guards, Cubicle applies monotone abstraction [2,3] and over-approximates predecessors via upward-closed sets of configurations. The search procedure maintains a set V and a priority queue Q resp. of visited and unvisited cubes. Initially, let V be empty and let Q contain the cubes representing bad states. At each iteration, the procedure selects the highest-priority cube Φ from Q and checks for intersection with the formula denoting the initial configurations (satisfiability of conjunction of Φ and formulas in the initial conditions). If the test fails, it terminates reporting a possibile error trace. If the test passes, the procedure proceeds to the subsumption check, i.e., implication between formulas. If subsumption fails, then add Φ to V, compute all cubes in $pred_t$ (for every t), add them to Q, and move on to the next iteration. If the subsumption check succeeds, then drop Φ from consideration and move on. The algorithm terminates when a safety check fails or Q becomes empty. When an unsafe cube is found, Cubicle actually produces a counterexample trace. Safety checks, being ground satisfiability queries, are easy for SMT solvers. The challenge is in the subsumption check because of their size and the existential implies existential logical form. Cubicle applies the heuristics described in [10] to handle subsumption.

Encoding GLog in Cubicle

In this section we present an encoding of GLog into an array-based specification language. The encoding is quite natural. The interpretation domain of variables is that of process indexes. For each unary predicate $p \in P$, we introduce a corresponding Boolean array variable `array p`.

```
array p[proc] : bool
```

For each binary predicate q, we introduce a two-dimensional Boolean array `q`

```
array q[proc,proc] : bool
```

Encoding of guards is straightforward. Free variables occurring in GLog update rules become parameters of transition definitions. A literal $q(x, y)$ [resp. $q(x)$] is mapped to the formula $q(x, y) = true$ [resp. $q(x) = true$]. A literal $\neg q(x, y)$ [resp. $\neg q(x)$] is mapped to the formula $q(x, y) = false$ [resp. $q(x) = false$]. Compound/quantified require conditions are mapped to compound/quantified formulas over literals.

Some care has to be taken in the encoding of GLog update rules. Transitions in Cubicle operate simultaneously on every cell of an array to provide support for global operations like reset and transfer. This kind of operations are not provided in GLog since the focus is on asynchronous behavior, i.e., we assume that global operations are split into several asynchronous operations equipped with guards that can be used to check for the current state of the protocol phase under consideration.

To encode a deletion rule, operating on the atomic formula $A(x, y)$, we use auxiliary variables u, t and a case analysis on indexes: for the case $x = u, y = t$ we add the action $A[x, y] := false$, and $A[x, y] := A[x, y]$ in all other cases. To encode an addition rule, operating on the atomic formula $A(x, y)$, we use again auxiliary variables u, t and a case analysis on indexes: for $x = u, y = t$ we add the action $A[x, y] := true$, and $A[x, y] := A[x, y]$ in all other cases.

To encode $\exists$-coverability, we also need to specify initial and unsafe configurations. Unsafe configurations can be described as in Cubicle using an existentially quantified formula over array cells. To select classes of initial states, we can use `init` declarations in which we specify only partial conditions on array cells.

4 Case Study: Distributed Dining Philosophers

We consider here a distributed version of the dining philosopher mutual exclusion problem presented in [18]. Agents are distributed on an arbitrary graph and communicate asynchronously via point-to-point channels. Channels are viewed as buffers with state. Distributed Dining Philosophers (DDP) is defined as follows. The goal is to ensure that agents can access a resource shared in common with their neighbors in mutual exclusion. The protocol from the perspective a single agent consists of the following steps:

- Initially, all agents are in *idle* state.
- When an agent A wants to get a resource, A has to acquire the control of each buffer shared with his/her neighbors.
- To acquire a channel, A marks the channel with its identifier. If the channel is already marked, A has to wait.
- A acquires the resources when all channels shared with neighbors are marked with his/her identifier.
- To release a resource, A first resets each buffer. When all buffers are reset, A moves back to idle state.

In a statically defined topology, agent A gets access to a resource when all neighbors are either idle or are waiting for acquiring some channel. Communication between two neighbors is asynchronous. Indeed, they interact by reading and writing on the shared channel. The protocol should guarantee that two agents that share the same channel cannot acquire and use a resource simultaneosly. The protocol should be robust under dynamic reconfigurations of the network.

4.1 Formal Specification of DDP

In this section we present a formal specification of the DDP protocol. Network configurations are expressed as GLog configurations. The dynamics in a protocol interaction is expressed via a finite set of update rules. We use a predicate *link* to represent connections from an agent to a possibly shared buffer. We model buffers with states using unary predicates. Asynchronous communication is modeled as in the previous example, i.e., agents interact only via a common buffer. Communication between two agents is not atomic. Instead of modeling identifiers and buffers with data, we introduce a special relation *own* that is used to model ownership of a given buffer to which a agent is linked. Ownership is normed in the same way as the labeling of buffers in the original protocol, i.e., an agent can acquire ownership only if the buffer is not owned by other agents. Ownership can be realeased when in *idle* state. We also model non-deterministic creation (in *idle* state) and deletion of links. We model this behavior using the following predicates and rules (rules have the form (C_i, D_i, A_i) for $i : 1, \ldots, 6$):

R	C	D	A
$getE$	$link(X,E) \wedge \forall Z.\neg own(Z,E)$	$\emptyset$	$\{own(X,E)\}$
$relE$	$\{idle(X), own(X,E)\}$	$\{own(X,E)\}$	$\emptyset$
$acquire$	$idle(X) \wedge \forall E.(link(X,E) \supset own(X,E))$	$\{idle(X)\}$	$\{busy(X)\}$
$release$	$\{busy(X)\}$	$\{busy(X)\}$	$\{idle(X)\}$

An initial state configuration has the following form $idle(n_1), \ldots, idle(n_k)$, where $n_i \neq n_j$ for $i \neq j$, $i, j : 1, \ldots, k$ and $k \geq 1$.

4.2 Encoding in Cubicle

Following the encoding rules specified for GLog, we can now obtain a Cubicle specification for DDP. To simplify a little bit the specification, we introduce an enumeration type `state` over node states.

```
type state = Idle | Busy
```

This way, we can use a single array `State` with cell type `state` instead of two Boolean arrays. We also need a `link` array and a `own` array to specify link and ownership relations between nodes and buffers.

```
array State[proc] : state
array Link[proc,proc] : bool
array Own[proc,proc] : bool
```

The initial configuration consists of all possible topologies in which nodes are in idle state. We also enforce the ownership relation to be false for each pair of node and buffer.

```
init (n m) {
  State[n] = Idle &&
  Own[n,m] = False }
```

This way we do not put any constraints on link topology. The bad configuration are defined by graphs of the following form.

```
unsafe (n m e) {
  State[n]  = Busy && State[m]  = Busy &&
  Link[n,e] = True && Link[m,e] = True
}
```

Two node are in mutex state while pointing to at the same buffer. The transitions are obtained via the encoding of the GLog specification into Cubicle input language shown in Appendix A. When applying Cubicle to the above described problem, the tool proves the model correct in few seconds without need to apply multicore optimizations via the Functory library. More specifically, Cubicle visits 19 nodes with at most 3 process indexes, 529 fixpoints and 176 calls to the SMT solver. Since Cubicle operates over unbounded arrays, the above result provides a formal correctness proof of the considered model for any number of nodes and links and any topology. The proof certificate can be obtained by taking the set of assertions (formulas) collected during the fixpoint computation.

4.3 Dynamic Reconfiguration

To model dynamic reconfigurations, we can non-deterministically add and remove *link* predicates between pairs of agents and buffers. We first consider the non-deterministic rules *link* and *unlink* defined below.

R	C	D	A
$link$	$\neg link(X,E)$	$\emptyset$	$\{link(X,E)\}$
$unlink$	$link(X,E)$	$\{link(X,E)\}$	$\emptyset$

When the model extended with the above rules is checked with Cubicle (see appendix B), the tool reports the error trace $acquire(\#1) -> link(\#1,\#3) -> get(\#2,\#3) -> acquire(\#2) -> unsafe[1]$. This trace is a real error trace. Indeed, process p_1 can acquire ownership when there are no links to buffer b_3. Since the *link* rule has no condition on p_1, a link can then be added from p_1 to b_3. However, Process p_2 can now become owner of b_3 and then move to state $Busy$. Two processes are linked to the same buffer b_3 while in state $Busy$.

We can modify the model and restrict addition of new link connected to node X only when X is in state $Idle$ as follows.

R	C	D	A
$link'$	$idle(X), \neg link(X,E)$	$\emptyset$	$\{link(X,E)\}$
$unlink$	$link(X,E)$	$\{link(X,E)\}$	$\emptyset$

In this model we assume that nodes have some form of control over connections with buffers (i.e. a new link is detected by a node in state $Idle$). When the model extended with the $link'$ and *unlink* rules is checked with Cubicle (see appendix B), the tool verifies correctness by visiting 28 nodes with at most 3 process indexes, invoking 261 times the SMT solver, and deleting 6 redundant nodes. Using dfs search, the tool verifies the property but the number of visited nodes is 39 with 414 calls to the SMT solver.

5 Conclusions

We have studied a possible application of SMT-based infinite-state model checker to the verification of topology-sensitive distributed protocols, i.e., protocols defined over network graphs and in which rules have guards and effects that depend on communicaton links. Starting from a logic-based presentation of distributed protocols based on the GLog relational update language, we have shown how to encode existential coverability queries in GLog as Cubicle verification judgements. As a case-study, we have shown that the declarative approach supported by GLog+Cubicle provides a very effective way to verify protocols operating on graphs. For instance, in previous work DDP required complex verification methodologies like assume-guarantee reasoning or ad hoc algorithms for graph rewriting systems. In the present paper DDP is verified using a very simple declarative specification and a general purpose model checker. Cubicle verifies correctness in negligible execution time (without need of multicore optimizations via the Functory library).

The proposed methodology can be applied to other types of parameterized protocols as discussed below. As additional example, consider a general scheme for reference counting protocols adopted in (distributed) operating systems. The protocol encodes a basic scheme for garbage collecting objects pointed to by an arbitrary number of agents. The system is defined by an initial set of objects Obj (all set to Null) and Ref (all set to False) relations. Objects and references to are created non-deterministically. We consider then two type of deletion: shallow and deep deletion. With shallow deletion, an object is deleted when there are no more references to it. With deep deletion, an object pointed to by a reference is deleted when there are no other pointers to it. The reference to the object is removed as well. We formally specify the protocol as shown below.

R	C	D	A
create	$\neg obj(O)$	$\emptyset$	$\{obj(O)\}$
ref	$\neg ref(A,O)$	$\emptyset$	$\{ref(A,O)\}$
shallow-delete	$obj(O) \wedge \forall A.\neg ref(A,O)$	$\{obj(O)\}$	$\emptyset$
deep-delete	$obj(O) \wedge ref(A,O) \wedge$ $\forall A1 \neq A.\neg ref(A1,O)$	$\{ref(A,O), obj(O)\}$	$\emptyset$

The first two rules non-deterministically introduce new instances of objects and reference relations. The remaining rules implement the above mentioned types of object deletion. Bad states consist of references to null objects. The scheme could be made more complicate e.g. by considering nested objects and more complicate protocols to propagate deletion commands. The encoding in Cubicle is based on the previously introduced general scheme.

Exploring other classes of distributed algorithms and properties like absences of cycles or starvation freedom are interesting directions to explore to improve the verification engine underlying the tool. More specifically, we are currently studying how to deal with routing protocols for arbitrary topologies. Another interesting direction is related to the possible application of Cubicle for verification of protocol specifications in multi agent systems [5].

References

1. P. A. Abdulla and G. Delzanno. Parameterized verification. *STTT*, 18(5):469–473, 2016.
2. P. A. Abdulla, G. Delzanno, N. Ben Henda, and A. Rezine. Monotonic abstraction: on efficient verification of parameterized systems. *Int. J. Found. Comput. Sci.*, 20(5):779–801, 2009.
3. P. A. Abdulla, G. Delzanno, N. Ben Henda, and A. Rezine. Monotonic abstraction: on efficient verification of parameterized systems. *Int. J. Found. Comput. Sci.*, 20(5):779–801, 2009.
4. F. Alberti, S. Ghilardi, and N. Sharygina. A framework for the verification of parameterized infinite-state systems. *Fundam. Inform.*, 150(1):1–24, 2017.
5. D. Ancona, A. Ferrando, and V. Mascardi. Parametric runtime verification of multiagent systems. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, pages 1457–1459, 2017.
6. N. Bertrand, G. Delzanno, B. König, A. Sangnier, and J. Stückrath. On the decidability status of reachability and coverability in graph transformation systems. In *RTA'12*, volume 15 of *LIPIcs*, pages 101–116. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
7. N. Bertrand, P. Fournier, and A. Sangnier. Distributed local strategies in broadcast networks. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 44–57, 2015.
8. R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.
9. R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. Decidability in parameterized verification. *SIGACT News*, 47(2):53–64, 2016.
10. S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems - tool paper. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 718–724, 2012.
11. G. Delzanno. A logic-based approach to verify distributed protocols. In *Proceedings of the 31st Italian Conference on Computational Logic, Milano, Italy, June 20-22, 2016.*, pages 86–101, 2016.
12. G. Delzanno. A unified view of parameterized verification of abstract models of broadcast communication. *STTT*, 18(5):475–493, 2016.
13. G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, pages 313–327, 2010.
14. G. Delzanno, A. Sangnier, and G. Zavattaro. On the power of cliques in the parameterized verification of ad hoc networks. In *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 441–455, 2011.
15. G. Delzanno, A. Sangnier, and G. Zavattaro. Verification of ad hoc networks with node and communication failures. In *FORTE/FMOODS'12*, volume 7273 of *LNCS*, pages 235–250. Springer, 2012.

16. G. Delzanno and J. Stückrath. Parameterized verification of graph transformation systems with whole neighbourhood operations. In *Reachability Problems - 8th International Workshop, RP 2014, Oxford, UK, September 22-24, 2014. Proceedings*, pages 72–84, 2014.
17. A. Mebsout. *Inférence d'invariants pour le model checking de systèmes paramétrés. (Invariants inference for model checking of parameterized systems)*. PhD thesis, University of Paris-Sud, Orsay, France, 2014.
18. K. S. Namjoshi and R. J. Trefler. Uncovering symmetries in irregular process networks. In *VMCAI*, pages 496–514, 2013.
19. K. S. Namjoshi and R. J. Trefler. Analysis of dynamic process networks. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 164–178, 2015.
20. `http://alt-ergo.lri.fr`.
21. `http://functory.lri.fr/`.
22. `http://www.disi.unige.it/person/DelzannoG/MSR/`.

A DDP in Cubicle

```
transition get(n e)
requires {
  Link[n,e] = True  &&
  forall_other m.   (Own[m,e] = False)
}
{
Own[m,f] := case
   | m=n && f=e : True
   |  _ : Own[m,f];
}

transition rel(n e)
requires {
 State[n] = Idle  && Own[n,e] = True
}
{
Own[m,f] := case
    | m=n && f=e : False
    |  _ : Own[m,f];
 }

transition acquire (n)
requires {
  State[n] = Idle
  &&
  forall_other g.
    (Link[n,g] = False || Link[n,g] = True && Own[n,g] = True)
```

```
}
{
State[m] := case
    | m=n : Busy
    |   _ : State[m];
 }
```

B Dynamic Reconfiguration in Cubicle

```
transition link(n m)
requires {
 Link[n,m] = False
}
{
Link[p,q] := case
    | p=n && q=m : True
    |  _ : Link[p,q];
 }

transition link'(n m)
requires {
 State[n] = Idle
 &&
 Link[n,m] = False
}
{
Link[p,q] := case
    | p=n && q=m : True
    |  _ : Link[p,q];
 }

transition unlink(n m)
requires {
 Link[n,m] = True
}
{
Link[p,q] := case
    | p=n && q=m : False
    |  _ : Link[p,q];
 }
```