A Logic-based Approach to Verify Distributed Protocols

Giorgio Delzanno

DIBRIS, University of Genova, Italy

Abstract. We present a framework for the specification of distributed protocols based on a logic-based presentation of bipartite graphs. For the considered language, we define assertions that can be applied to arbitrary configurations. We apply the language to model the distributed version of the Dining Philosopher Protocol. The protocol is defined for asynchronous processes distributed over a graph with arbitrary topology. To validate the protocol, we apply permutation schemes, transformation rules, and inductive verification.

1 Introduction

Verification of distributed systems with parametric dimension is a challenging task. In this setting protocols are defined on generic configurations of a network, e.g., for an arbitrary number of nodes and arbitrary connection topology. Protocol rules may depend on the current network configuration. For instance, in [14] Namjoshi and Trefler introduced a distributed variant of the dining philosopher protocol in which individual nodes interact asynchronously via shared buffers. Global conditions formulated on the state of visible buffers are used to regulate the access to resources shared between adjacent nodes.

In this paper we present a formalization of distributed protocols with all above mentioned features based on a logic-based presentation of bipartite graphs defined over two classes of nodes: agent and edges. A agent corresponds to a network agent. An edge corresponds to a potential link between multiple agents. We use bipartite graphs in order to model both complex topologies and asynchronous operations in a more natural way with respect to standard transition systems. Point-to-point links can be obtained by using conditions defined on update rules. Our specification language is based on a predicated logic with quantification defined over a finite set of typed relations without function symbols. The interpretation domain of relations is defined over an infinite set of agents and edges. In this setting we can model handshaking between two agents via intermediate steps in which agents first connect to edges and then, by updating the relations connecting them, start an interaction between them. In our model we admit multiple connections to the same edge.

The technical contribution of the paper is as follows. We first define the syntax and formal semantics of the BLog specification language. The language is based on conditional update rules. Conditions are expressed as first order quantified formulas defined over binary predicates without function symbols. Update rules are defined by sets of atomic formulas that specify the ground atoms that have to be removed from the current configuration and those that have to be added to the new one. BLog can be viewed as a fragment of richer languages for representing evolving databases like DCDS [8]. Differently from DCDS, designed for updates of relational databases, we are interested here in a minimal fragment of relational logic to reason about evolving bipartite graphs. Furthermore, we focus our attention on a special class of decision problems that can be viewed as parametric versions of reachability problems. More specifically, in order to reason about families of distributed systems we consider reachability problems existentially quantified over initial configurations as in previous work on formal verification of parameterized systems. Furthermore, we isolate a fragment of BLog in which we show that the considered decision problem is undecidable. The proof is based on an encoding of Turing machines on a special type of bipartite graphs in which we used special links to order the representation of tape cells.

We apply our language to model the distributed version of the Dining Philosopher Protocol proposed in [15]. The protocol is defined for asynchronous processes distributed over a graph with arbitrary topology. We model the protocol by considering dynamic reconfigurations of the topology. To verify the correctness of our model, we apply a proof method that combines two kinds of reasoning. We first apply source to source transformations based on permutation schemes that lead to derived rules. Permutation schemes are obtained via a proof theoretic analysis of computations. Permutation rules can be applied to derive meta rules obtained by composing sequences of rule applications of the original protocol. We use deductive reasoning to prove mutual exclusion on the resulting model. Invariants are expressed using parametric formulas, an extension of the assertional language used in BLog needed to express properties like parametric reachability. The resulting method is based on proof techniques that are complementary to compositional reasoning [15] and symbolic backward exploration [7].

1.1 Related Work

We focus our attention on a class of formal models for concurrent and distributed systems in which synchronization is achieved by using broadcast communication, a less standard communication primitive than rendez-vous or point-to-point communication. Rendez-vous communication involves a fixed a priori number of agents (e.g. a sender and a receiver in point-to-point communication). Properties of rendez-vous communication have been investigated deeply in the field of automated verification, see e.g. [6]. Interactions in models with broadcast communication are usually defined by allowing a finite but arbitrary number of agents to react to a given message or signal. This type of communication is particularly useful to define protocols for replicated systems, e.g. cache coherence protocols, algorithms with global conditions, e.g., simultaneous resets of local variables, and communication in an open environment like an Ad Hoc or

Wireless network. Algorithmic verification of models with broadcast communication started receiving more attention after the introduction of the Broadcast Protocols of Emerson and Namjoshi [3]. FAST [2,24], TRex [23], LASH [25], and MIST [17] are tools that can handle counter abstractions of network models given in formal of vector addition systems and their extensions. MAP [4] is a tool based on transformations of constraint logic programs that can be applied to infinite-state systems with linear configurations and relations over data variables. MCMT [18] is a symbolic backward reachability engine based on SMT solvers that can handle parameterized systems with linear configurations. The MCMT tool is based on the EPR fragment of first order logic with arrays and applies different types of heuristics including invariant generation to reduce the state space. PFS [21] and UNDIP [22] are tool specifically devised to handle parameterized systems. AUGUR 2 [11] is a tool devised for the analysis of Graph Transformation Systems using approximated unfoldings based on Petri nets. The approximated systems can then be verified using regular expressions, first order logic and coverability checking techniques. AUGUR 2 does not handle global operations like those needed for modeling broadcast communication. PETRUCHIO [13] is a tool that extracts a Petri net representation from specifications of dynamic networks based on the π-calculus. Boom [20] is a tool that applies symbolic algorithms, see, e.g., [10,12,1,9], to verify counter abstractions of multithreaded programs. The algorithms behind the tool go beyond backward search. Indeed they combine several types of heuristics like those based on dynamic generation and refinement of over-approximations (defined in terms of upward closed set of states). PCW [19] is a tool that applies ordered counter abstraction [5], a refinement of monotonic abstraction with CEGAR, for the verification of parameterized systems. In this setting over-approximations are refined by using stronger and stronger orderings that can be used to define upward closed sets that "forbid" specific patterns (e.g. they forbid sets of points defined by a given equation). UNCOVER [16,26] is a tool that performs a symbolic backward reachability analysis for GTS with universally quantified conditions. The tool exploits a generalization of monotonic abstraction to quantifications over graph patterns as a heuristic to manipulate infinite sets of configurations using minimal constraints (given in form of graphs) only. UNCONVER can be viewed as the counterpart of UNDIP and PFS for systems in which configurations have a graph structure.

Differently from [16,26] and [14], our specification logic can be applied to define invariants involving an arbitrary but fixed number of nodes. For this purpose, we use assertions with parametric formulas. Global conditions can be defined using universally quantified formulas. The use of SMT solvers for graph-based specification is an interesting direction to explore.

2 BLog

In this section we will define a logic-based presentation of evolving bipartite graphs called BLog.

BLog formulas are based on a simple relational calculus that can be used to express updates of configurations defined by sets of ground atoms. Ground atoms define relations, i.e., labelled links, between agents and edges. More formally, let P be a finite set of names of binary relations, and V be a denumerable set of variables partitioned in two sets V_a and V_e, namely variables ranging over agents and edges, respectively. Furthermore, let $\mathcal{N}$ and $\mathcal{E}$ be denumerable sets of agent and edge identifiers. We consider here typed binary predicates in which the first argument is of agent type and the second one is of edge type. Our logic has no function symbols but can be instantiated with elements fro$\mathcal{N}$ and $\mathcal{E}$.

Definition 1. *An atomic formula is a formula $p(x, y)$, where $p \in P$, $x \in V_a$, and $y \in V_e$. A ground formula is a formula $p(n, e)$, where $n \in \mathcal{N}$, and $e \in \mathcal{E}$. A literal is either an atomic formula or the negation $\neg A$ of an atomic formula A. A formula is a Boolean combination of atomic formulas.*

Definition 2. *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 will be used as application conditions of rules. 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. We will consider free variables occurring in conditions to restrict the range of identifiers of a configuration.

Definition 3. *A rule has the following form $\langle C, D, A\rangle$, where C is a quantified formula, D and A are two sets of predicates with variables in V, and such that $FV(A) \cup FV(D) \subseteq FV(C)$.*

Configurations can be viewed as models in which to evaluate a formula.

Definition 4. *A configuration is a set Δ of ground atomic formulas.*

Definition 5. *We use $\Delta \models A$ to define the satisfiability relation of a quantified formula A s.t. $FV(A) = \emptyset$. The relation is defined by induction as follows.*

- *$\Delta \models A$, if $A \in \Delta$;*
- *$\Delta \models A \wedge B$, if $\Delta \models A$ and $\Delta \models B$;*
- *$\Delta \models \neg A$, if $\Delta \not\models A$;*
- *$\Delta \models \forall X.A$ (resp. $\forall E.A$), if $\Delta \models A[n/X]$ (resp. $A[e/X]$) for each $n \in \mathcal{N}$ (resp. $e \in \mathcal{E}$);*
- *$\Delta \models \exists X.A$ (resp. $\exists E.A$), if $\Delta \models A[n/X]$ (resp. $A[e/X]$) for some $n \in \mathcal{N}$ (resp. $e \in \mathcal{E}$).*

To define instantiations of free variables, we consider injective mappings. We use injective mappings in order to give a unique interpretation of a formula like

$p(X,E) \wedge p(Y,F)$, i.e., X and Y refer to distinct agents and E and F to distinct edges. We allow multiple occurrences of the same variable to implicitly model equality constraints.

Definition 6. *An interpretation is an injective mapping σ from $V_a \cup V_e$ to $\mathcal{N} \cup \mathcal{E}$ s.t. $\sigma(V_a) \subseteq \mathcal{N}$ and $\sigma(V_e) \subseteq \mathcal{E}$.*

Definition 7. *Given a configuration Δ, we say that the quantified formula A is satisfied in Δ, if there exists an interpretation σ s.t. $\Delta \models A\sigma$ is satisfiable.*

Example 1. The condition $\forall X.\ p(X,E) \supset q(X,E)$, when evaluated on a configuration Δ, is satisfiable if there exists an edge e such that for all instances $p(n,e) \in \Delta$, there exists an atom $q(n,e) \in \Delta$. Based on these considerations, the rule

$$\langle link(X,Y) \wedge \forall Z.link(Z,Y) \supset \neg own(Z,Y), \emptyset, \{own(X,Y)\}\rangle$$

defines a transition in which for a pair n,e such that $link(n,e)$ occurs in the current configuration and such that there are not other agents n' s.t. $own(n',e)$ is in the current configuration, $own(n,e)$ is added to the successor configuration. In the operational semantics we assume that all atomic formulas that are not deleted are transferred to the successor configuration.

For a set $S = \{A_1, \ldots, A_n\}$, we use $S\sigma$ to denote the set $\{A_1\sigma, \ldots, A_n\sigma\}$.

2.1 Transition System

A protocol $\mathcal{P}$ is a set of rules. The operational semantics of $\mathcal{P}$ 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, 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 $\Delta' = (\Delta \setminus D\sigma) \cup A\sigma$.

Definition 8. *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 $\rightarrow$.

Example 2. Let $\mathcal{P}$ be a protocol that contains the rule $\forall X.\ link(X,E) \supset own(X,E)$. Consider $\Delta = \{link(n_1,e_1), link(n_2,e_1), link(n_3,e_3), own(n_3,e_3)\}$, we first notice that Δ does not satisfy the condition

$$link(X,Y) \wedge \forall Z.link(Z,Y) \supset \neg own(Z,Y)$$

with the interpretation $\sigma = [n_3/X, e_3/Y]$. Indeed, we have that

- $\Delta \not\models link(n_3,e_3) \wedge \forall Z.link(Z,e_3) \supset \neg own(Z,e_3)$;
- $\Delta \models link(n_3,e_3)$ and $\Delta \not\models \forall Z.link(Z,e_3) \supset \neg own(Z,e_3)$;
- $\Delta \not\models \forall Z.link(Z,e_3) \supset \neg own(Z,e_3)$;
- $\Delta \models link(n_1,e_3) \supset \neg own(n_1,e_3)$, since $own(n_1,e_3) \notin \Delta$, $link(n_1,e_3) \in \Delta$;
- $\Delta \models link(n_2,e_3) \supset \neg own(n_2,e_3)$, since $own(n_2,e_3) \notin \Delta$, $link(n_2,e_3) \in \Delta$;

– $\Delta \not\models \mathit{link}(n_3, e_3) \supset \neg \mathit{own}(n_3, e_3)$, since $\mathit{own}(n_3, e_3), \mathit{link}(n_3, e_3) \in \Delta$.

However, Δ satisfies the condition $\mathit{link}(X, Y) \wedge \forall Z.\mathit{link}(Z, Y) \supset \neg \mathit{own}(Z, Y)$ with the interpretation $\sigma = [n_1/X, e_1/Y]$. Indeed, we have that

– $\Delta \not\models \mathit{link}(n_1, e_1) \wedge \forall Z.\mathit{link}(Z, e_1) \supset \neg \mathit{own}(Z, e_1)$;
– $\Delta \models \mathit{link}(n_1, e_1)$ and $\Delta \not\models \forall Z.\mathit{link}(Z, e_1) \supset \neg \mathit{own}(Z, e_1)$;
– $\Delta \not\models \forall Z.\mathit{link}(Z, e_1) \supset \neg \mathit{own}(Z, e_1)$;
– $\Delta \models \mathit{link}(n_1, e_1) \supset \neg \mathit{own}(n_1, e_1)$, since $\mathit{own}(n_1, e_1) \notin \Delta$, $\mathit{link}(n_1, e_1) \in \Delta$;
– $\Delta \models \mathit{link}(n_2, e_1) \supset \neg \mathit{own}(n_2, e_1)$, since $\mathit{own}(n_2, e_1) \notin \Delta$, $\mathit{link}(n_2, e_1) \in \Delta$;
– $\Delta \models \mathit{link}(n_3, e_3) \supset \neg \mathit{own}(n_3, e_1)$, since $\mathit{own}(n_3, e_1) \notin \Delta$, $\mathit{link}(n_3, e_1) \in \Delta$.

Starting from $\Delta_0 = \Delta$ we obtain then a successor configuration

$$\Delta_1 = \Delta_0 \cup \{\mathit{own}(n_3, e_3)\}$$

Now, let $\mathcal{P}$ be a protocol that contains the rule

$$\langle \mathit{link}(X, Y), \{\mathit{own}(X, Y)\}, \emptyset \rangle$$

Starting from Δ_1, we can reach Δ_0. Indeed, we have that $\Delta_0 \models \mathit{link}(n_3, e_3)$, and that $\Delta_1 = \Delta_0 \setminus \{\mathit{own}(n_3, e_3)\}$.

In a single step of the operational semantics a rule is evaluated in the current configuration by taking a sort of closed-word assumption, i.e., ground atoms that do not occur in a configuration are evaluated to false. Furthermore, 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.

For a set S of configurations, we define the following sets:

$$\begin{aligned} \mathit{Post}(S) &= \{\Delta' \mid \exists \Delta \in S,\ \Delta \to \Delta'\} \\ \mathit{Pre}(S) &= \{\Delta' \mid \exists \Delta \in S,\ \Delta' \to \Delta\} \end{aligned}$$

We use $\mathit{Post}^*(S)$ (resp. $\mathit{Pre}^*(S)$) to denote the reflexive-transitive closure of Post (resp. Pre).

2.2 Derived and Unary Predicates

In the rest of the section we will introduce formulas that define special conditions on agents and edges. The formula $\mathit{isoN}(n) = \neg \exists X.(\vee_{r \in P} r(n, X))$ specifies an isolated agent n, i.e., that is not involved in any relation in P. Similarly, the formula $\mathit{isoE}(e) = \neg \exists X.(\vee_{r \in P} r(X, e))$ specifies an edge that is not involved in any relation in P. Conditions like $\mathit{isoN}(n)$ and $\mathit{isoE}(e)$ are building blocks for the definition of other types of derived predicates. This pattern can be applied anytime it is necessary to avoid interference when constructing new predicates and relations. For instance, if $isoE(e)$ holds, then we define rules to add predicates like $q(n, e)$ that associate a state/label q to agent n. Similarly, if $isoN(n)$ holds, then we can add predicates like $q(n, e)$ to associate a state/label to edge e. For the sake of simplicity, in the rest of the paper we use unary predicates, e.g. $q(X)$ and $q(E)$, instead of binary predicate in which one of the arguments is used only as placeholder (e.g. associated to an isolated agent/edge) as in the previous examples.

3 Decision Problem

We consider 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 agents and edges. For this purpose, we introduce the $\exists$-reachability problem defined as follows.

Definition 9. *Given a protocol $\mathcal{P}$, a set of target configurations T and a possibly infinite set of initial configurations I, $\exists$-reachability 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 $\Delta \subseteq \Delta_1$.*

In other words $\exists Reach(\mathcal{P}, I, T)$ holds if there exists a configuration $\Delta_0 \in I$ s.t. $\Delta_0 \rightarrow^* \Delta_1$ and $\Delta \subseteq \Delta_1$ for some $\Delta \in T$.

In the rest of the paper we will generalize the problem by considering patterns defined via existentially quantified formulas. Generalized $\exists$-reachability is defined as follows.

Definition 10. *Given a protocol $\mathcal{P}$ and a formula Φ with free variables in S and a set of initial configurations I, generalized $\exists$-reachability is satisfied for $\mathcal{P}$, I, and Φ, written $\exists Reach(\mathcal{P}, I, \Phi)$, if there exists an interpretation σ, a configuration Δ s.t. $\Delta \models \Phi\sigma$, and a configuration Δ_1 s.t. $\Delta_1 \in Post^*(I)$ and $\Delta \subseteq \Delta_1$.*

3.1 Undecidability

The $\exists$-Reachability problem turns out to be undecidable. A proof can be constructed by reducing the halting problem for Turing machines to $\exists$-reachability. The idea of the construction is as follows. Let $M = \langle \Sigma, Q, \delta, q_0, F \rangle$ be a Turing machine in which Q is the set of control states, Σ is the tape alphabet that includes the special blank symbol B, $F \subseteq Q$ is the set of final states, q_0 is the initial state, and $\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R\}$ is the transition relation. Given an input word $w \in \Sigma^*$ without blanks, we first define a bipartite graph that encodes w. We use here three types of relations:

- The atomic formula $q(n_1, e_1)$ is used to represent the state of the head pointing to cell n_1.
- The atomic formula $\ell(n_1, e_1)$ is used to represent a cell n_1 containing symbol ℓ.

– The atomic formula $next(n_2, e_1)$ is used to link, via edge e_1, a cell to its successor cell identified by agent n_2.

Creation of the word $w = a_1 \dots a_n$ and initialization of the tape head are defined by the rule $(\wedge_{i=1}^{n} isoN(X_i) \wedge isoE(E_i), \emptyset, A)$, where

$$A = \{q_0(X_1, E_1), a_1(X_1, E_1), next(X_2, E_1), a_2(X_2, E_2), \dots, a_n(X_n, E_n)\}$$

This rule selects n isolated agents and edges and defines a zig-zag graph in which we keep the information about letters and pointers to the adjacent cells.

To simulate a transition $\delta(q_1, a) = \langle q_2, b, R\rangle$ we use the rules $r_1 = \langle C_1, D_1, A_1\rangle$ and $r_2 = \langle C_2, D_2, A_2\rangle$, where

– $C_1 = q_1(X_1, E_1) \wedge a(X_1, E_1) \wedge next(X_2, E_1) \wedge \bigvee_{\ell \in \Sigma} \ell(X_2, E_2)$,
– $D_1 = \{q_1(X_1, E_1), a(X_1, E_1)\}$ and $A_1 = \{q_2(X_2, E_2), b(X_2, E_2)\}$.
– $C_2 = q_1(X_1, E_1) \wedge a(X_1, E_1) \wedge isoN(X_2) \wedge isoE(E_2) \wedge \neg\exists Z.next(Z, E_1)$,
– $D_2 = \{q_1(X_1, E_1), a(X_1, E_1)\}$,
– $A_2 = \{next(X_2, E_1), q_2(X_2, E_2), b(X_2, E_2)\}$.

The former rule updates the relation associated to the encoding of the cell pointed to by the head assuming that there exists another cell at its right. If there are no representations of cells at its right, i.e., the relation $next$ is not defined, the latter rule can be applied to generate a representation of a blank cell. To simulate a transition $\delta(q_1, a) = \langle q_2, b, L\rangle$ we use the rules $r_3 = \langle C_3, D_3, A_3\rangle$ and $r_4 = \langle C_4, D_4, A_4\rangle$ where

– $C_3 = q_1(X_2, E_2) \wedge a(X_2, E_2) \wedge next(X_2, E_1) \wedge \bigvee_{\ell \in \Sigma} \ell(X_1, E_1)$,
– $D_3 = \{q_1(X_2, E_2), a(X_2, E_2)\}$ and $A_3 = \{q_1(X_1, E_1), b(X_2, E_2)\}$.
– $C_4 = q_1(X_1, E_1) \wedge a(X_1, E_1) \wedge isoN(X_2) \wedge isoE(E_2) \wedge \neg\exists F.next(X_1, F)$,
– $D_4 = \{q_1(X_1, E_1), a(X_1, E_1)\}$,
– $A_4 = \{next(X_1, E_2), q_2(X_2, E_2), b(X_1, E_1), B(X_2, E_2)\}$.

The former rule updates the relation associated to the encoding of the cell pointed to by the head assuming that there exists another cell at its left. If there are no representations of cells at its left, the latter rule can be applied to generate a representation of a blank cell.

The set of initial configurations I consists of an arbitrary number of isolated agents and edges, i.e., it contains a configuration $\Delta_{n,m}$ for each possible number n of agents and m of edges. Let us indicate with $\overline{\Delta}$ the configuration of the Turing machine represented by Δ.
Namely, if

$$\Delta = \{a_1(n_1, e_1), next(n_2, e_1), a_2(n_2, e_2), \dots, a_k(n_k, e_k), q_i(n_j, e_j)\}$$

then $\overline{\Delta} = a_1 \dots a_{j-1} q_i a_j \dots a_k$. By construction, the application of a rule produced by the encoding preserve the well-formedness of the encoding of configurations. The following properties then holds.

Proposition 1. *Let $\mathcal{P}_M$ be the encoding of the Turing machine M and $\Delta_0 \in I$. If $\Delta_0 \rightarrow^* \Delta_1$, then there exists a computation in M from $\overline{\Delta_0}$ to $\overline{\Delta_1}$.*

Indeed, by construction, each step of a computation in $\mathcal{P}_M$ mimics the application of transitions in M. Δ_0 contains finitely many agents and edges, the size of visited part of tape in the computation in M is bounded by the cardinality of Δ_0.

Proposition 2. *Let $\mathcal{P}_M$ be the encoding of the Turing machine M and $\gamma_0 = q_0 w$ an initial configuration of M on input w. If γ_1 is reachable from γ_0, then there exists $\Delta_0 \in I$ and Δ_1 s.t. $\overline{\Delta_0} = \gamma_0$, $\overline{\Delta_1} = \gamma_1$, and $\Delta_0 \rightarrow^* \Delta_1$.*

If γ_1 is reachable from γ_0, then there is a finite upperbound k on the number of cells visited by the machine. Let Δ_0 be an initial configuration with at least n (isolated) agents and edges. By applying a creation rule, we can generate a configuration Δ_0' that represents the initial configuration γ_0. By construction of $\mathcal{P}_M$, a step from γ_i to γ_{i+1} can be simulated by the application of a rule in $\mathcal{P}_M$. The choice of Δ_0 ensures us that we can always select isolated agents and edges when needed, i.e., when the machine moves to a blank cell to the left or right of the visited part of the tape.

The following property then holds.

Theorem 1. *Let $\mathcal{P}_M$ be the encoding of the Turing machine M. The Halting problem for M can be reduced to $\exists$-Reachability in $\mathcal{P}_M$ by taking the formula $\Phi = \bigvee_{q \in F} q(X, Y)$ as representation of subconfigurations denoting a final state.*

4 Modelling

In this section we consider possible application of BLog 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 bipartite graphs in which predicates can be viewed as labels of links between agents and edges. We have shown that we can also add labels to individual agents and edges, 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 edges 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 an edge 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 *readyack*.
- Agent n_1 sends message m by changing the state of e_1 to msg_m.
- Agent n_2 receive message m and updates the state of the channel to *readyack* for further communications.

The protocol can be specified as follows. We use unary predicates to associate states to edges.
Rule $req_A^1 = (C_1, D_1, A_1)$ is s.t. $C_1 = idle(E) \wedge \neg send(X, E)$,

$D_1 = \{idle(E)\}$, $A_1 = \{ready(E), req(X,E)\}$
Rule $req^1_B = (C_2, D_2, A_2)$ is s.t. $C_2 = ready(E) \wedge \neg rec(X,E)$,
$D_2 = \{ready(E)\}$, $A_2 = \{readyack(E), rec(X,E)\}$.
Rule $req^2_A = (C_3, D_3, A_3)$ is s.t. $C_3 = readyack(E) \wedge send(X,E)$,
$D_3 = \{readyack(E), send(X,E)\}$, $A_3 = \{msg_m(E)\}$
Rule $req^3_A = (C_4, D_4, A_4)$ is s.t. $C_4 = msg_m(E) \wedge rec(X,E)$,
$D_4 = \{msg_m(E), rec(X,E)\}$, $A_4 = \{idle(E)\}$

An initial state consists of a bipartite graph defined by the configuration of the following form $idle(e_1), \ldots, idle(e_k)$, where $e_i \neq e_j$for $i \neq j$, $i, j : 1, \ldots, k$.

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 by using rules like (C, D, A) defined as:

– $C = link(X,E) \wedge s_1(X,F) \wedge link(Y,E) \wedge s_2(Y,G)$
– $D = \{s_1(X,F), link(X,E), link(Y,E), s_2(Y,G)\}$
– $A = \{link(X,E), s'_1(X,F), link(Y,E), s'_2(Y,G)\}$

where $s_1(X,F)$ is a relation used to denote state of agent X, $link(X,E), link(Y,E)$ is a relation used to share a common buffer E, and $s_2(Y,G)$ is a relation used to denote state of agent Y, etc.

5 Distributed Dining Philosophers

We consider here a distributed version of the dining philosopher mutual exclusion problem presented in [14]. 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.

5.1 Formal Specification of DDP

In this section we present a formal specification of the DPP protocol. Network configurations are expressed as BLog 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 a agent to a possibly shared buffer. To model dynamic reconfigurations, we can non-deterministically add and remove *link* predicates between pairs of agents and edges. To model buffers with states, we use special relations in which agents are initially isolated. Asynchronous communication is modelled as in the previous example, i.e., agents interact only via a common edge. Communication between two agents is not atomic. Instead of modelling identifiers and buffers with data, we introduce a special relation *own* that is used to model ownership of a given edge to which a agent is linked. Ownership is normed in the same way as the labelling of buffers in the original protocol, i.e., an agent can acquire ownership only if the edge is not owned by other agents.

In our framework we model this behavior using the following predicates and rules.
Rule $link = (C_1, D_1, A_1)$ is defined s.t. $C_1 = \neg link(X, E)$, $D_1 = \emptyset$, $A_1 = \{link(X, E)\}$.
Rule $unlink = (C_2, D_2, A_2)$ is defined s.t. $C_2 = link(X, E)$, $D_2 = \{link(X, E)\}$, $A_2 = \emptyset$.
Rule $getE = (C_3, D_3, A_3)$ is s.t. $C_3 = link(X, E) \wedge \forall Z. \neg own(Z, E)$, $D_3 = \emptyset$, $A_3 = \{own(X, E)\}$.
Rule $relE = (C_4, D_4, A_4)$ is s.t.: $C_4 = D_4 = \{own(X, E)\}$, $A_4 = \emptyset$.
Rule $acquire = (C_5, D_5, A_5)$ is s.t. $C_5 = idle(X) \wedge \forall E.(link(X, E) \supset own(X, E))$, $D_5 = \{idle(X)\}$, $A_5 = \{busy(X)\}$.
Rule $release = (C_6, D_6, A_6)$ is defined s.t. $C_6 = D_6 = \{busy(X)\}$, $A_6 = \{idle(X)\}$.

An initial state consists of a bipartite graph defined by a configuration of 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 $k \geq 1$.

Correctness Mutual exclusion for the considered protocol can be formulated in its more general form, i.e., for any number of agents, as the negation of an $\exists$-reachability problem. More specifically, let Φ be the formula

$$busy(N) \wedge busy(M) \wedge link(N, G) \wedge link(M, G)$$

that denotes a configurations in which two agents share the same channel. We have that $\exists Reach(\mathcal{P}_M, I, \Phi)$ holds if and only if mutual exclusion does not hold for DDP.

6 Verification via Transformation Rules

In this section we define a sort of canonical form for computations in $\mathcal{P}_{ddp}$ obtained via permutation and deletion schemes that we can be uses to synthesize derived rules that are equivalent w.r.t. our correctness criteria.

6.1 Permutation and Deletion Schemes

We first consider deletion and permutation properties of rule applications within a given computation. We focus our attention on *link* and *unlink* rules, i.e., dynamic reconfigurations of the graph topology. Let $\theta = \Delta_0 \Delta_1 \Delta_2$ be a computation in $\mathcal{P}_{ddp}$ and let r_1, r_2 be the transitions applied in each step.

- We first observe that if r_2 is an instance of the *unlink* rule applied to n_1, e_1, and r_1 is an instance of rule *link* applied to the same pair, they can be eliminated since $\Delta_0 = \Delta_2$.
- If r_1 is an instance of the *getE* rule applied to agent n_1, e_1, and r_2 is an application of *relE* applied to the same pair, then they can be eliminated since $\Delta_0 = \Delta_2$.
- We now observe that *unlink* can be permuted with every other rule applied to different pairs to its left in a computation. Namely, if r_2 is an instance of the *unlink* rule applied to n_1, e_1, and r_1 is a rule that is not applied to n_1, e_1, then we can permute the application of the two rules and obtain a new computation leading to the same configuration.
- If r_1 is an instance of the *link* rule applied to n_1, e_1, and r_2 is an application of any other rule on a different pair, then we can permute the applications of the two rules and obtain a new computation leading to the same configuration.

We can now reason on the previous properties in order to consider sets of equivalent computations and infer derived blocks of rules and new permutation rules. More specifically, given a computation θ we can obtain an equivalent, w.r.t. our correctness criteria, computation θ' by applying the following steps:

- We can eliminate all applications of reconfigurations, i.e., *link* and *unlink*, performed on the same pair n, e, if in between their occurrences in a computation there are no occurrences of instances of *getE* on the same pair. The property can be obtained by repeatedly applying permutation rules so as to push applications of *link* towards the first occurrence of *unlink* and then apply the corresponding deletion rules.
- We can eliminate all applications of acquire *getE* and *relE* on the same pair n, e, if there are no occurrences of *acquire* involving n in between. Again, the property can be obtained by repeatedly applying permutation rules so as to push applications of *relE* towards the first occurrence of *getE* to its left.
- We can push the application of *link* rules close to the first occurrence of a corresponding *getE* rule to its left.

Following from the previous properties, from any computation we can extract a subcomputation in which occurrences of *relE* occur only after occurrences of *acquire* on the same pair. Similarly, we can push occurrences of *unlink* so as to occur only after occurrences of *acquire*. In other words we can derive rules obtained by combining *link* and *reqE*. The resulting pattern, *link_reqE*, is used to simultaneously link an agent to an edge and acquire its ownership. Similarly, since computations in which *relE* and *unlink* on specific pairs occur after *acquire*, we can use permutations in order to cluster all occurrences of patterns *relE_unlink* occurring before an *acquire*. In other words, we can use permutations of patterns *link_reqE* with other rules or patterns operating on different agents, in order to push all their occurrences close to the first acquire on their right involving the same agent. This properties leads to patterns of the form $(link\_reqE)^* acquire$. We now observe that the pattern can be used by agent n to simultaneously acquire the ownership of a given set of edges $e_1, \ldots, e_n$, and update the local state to *busy*. We can reason on permutations of *unlink*, *relE* and *release* in a similar way, i.e., apply permutations to cluster the sequence of applications of these rules involving the same agent after an acquire. The resulting sequence can be represented by the pattern $(relE\_unlink)^* release$. In other words in the resulting patterns the *link* relation is updated in parallel with the *own* relation.

6.2 Derived Rules

The pattern $(link\_reqE)^* acquire$ cannot be represented via a single rule in our specification language, since it may involve an arbitrary number of edges. To express this pattern, we can use a family $\{r_k\}_{k \geq 0}$ of rules, where $r_k = (C_k, D_k, A_k)$ is defined as:

- $C_k = idle(X) \wedge \bigwedge_{i=1}^{k} \neg link(X, E_i) \wedge \forall Z. \bigwedge_{i=1}^{k} \neg own(Z, E_i)$,
- $D_k = \{idle(X)\}$,
- $A_k = \{busy(X), link(X, E_1), own(X, E_1), \ldots, link(X, E_k), own(X, E_k)\}$.

The rule associates agent X to an arbitrary subset of edges. The association is defined via the ownership predicate.

The pattern $(relE\_unlink)^* release$ is expressed by the infinite family $\{s_k\}_{k \geq 0}$ of rules. Rule $s_k = (C_k, D_k, A_k)$ is defined as:

- $C_k = busy(X) \wedge \bigwedge_{i=1}^{k} link(X, E_i) \wedge own(X, E_i)$,
- $D_k = \{busy(X), link(X, E_1), own(X, E_1), \ldots, link(X, E_k), own(X, E_k)\}$,
- $A_k = \{idle(X)\}$.

This rule can be applied to a subset of all edges connected to a given agent. This corresponds to a sequence of applications on *release* and *relE*.

6.3 Deductive Reasoning

To verify correctness, we have to prove that the formula

$$\Psi = \neg \exists X, Y. busy(N) \wedge busy(M) \wedge link(N, G) \wedge link(M, G)$$

is an invariant for our system. To prove the property, we will strenghten the invariant proving that

$$\Psi' = \Upsilon \wedge \Psi$$

where $\Upsilon = (\forall Z, E.link(Z,E) \supset own(Z,E))$, is still an invariant of the resulting rule. By definition Ψ' holds in any initial configuration $\Delta_{m,n}$, since initial configurations only contain agents in state *idle*. The key point is to show that Ψ' is preserved by applications of rule r_k and s_k for any $k \geq 0$.

Consider a configuration Δ with cardinality n and assume that $\Delta \models \Psi'$. Now consider r_k s.t. $k \leq n$. If r_k can be applied to Δ, there exists an interpretation σ s.t. $\Delta \models C_k\sigma$ where $C_k = (idle(X) \wedge \bigwedge_{i=1}^{k} \neg link(X, E_i) \wedge \forall Z. \bigwedge_{i=1}^{k} \neg own(Z, E_i))$. This implies that there exists an agent n s.t. $idle(n) \in \Delta$, and there exist $e_1, \ldots, e_n$ s.t. $link(m, e_i), own(m, e_i) \notin \Delta$ for $i : 1, \ldots, k$ and for any m occurring in Δ. The application of the rule yields a configuration Δ' defined as

$$\Delta' = \Delta \cup \{link(n, e_1), \ldots, link(n, e_k), own(n, e_1), \ldots, own(n, e_k), busy(n)\}$$

Since by assumption $own(m, e_i), link(m, e_i) \notin \Delta$ for any m occurring in Δ and $i : 1, \ldots, k$, we have that $\Delta' \models \Psi'$.

Now consider a configuration Δ with cardinality n and assume that $\Delta \models \Psi'$. Now consider s_k s.t. $k \leq n$. If s_k can be applied to Δ, there exists an interpretation σ s.t. $\Delta \models C_k\sigma$ where $C_k = (busy(X) \wedge \bigwedge_{i=1}^{k} link(X, E_i) \wedge own(X, E_i))$. This implies that there exists an agent n s.t. $busy(n) \in \Delta$, and there exist $e_1, \ldots, e_n$ s.t. $link(n, e_i), own(n, e_i) \in \Delta$ for $i : 1, \ldots, k$. The application of the rule yields a configuration Δ' defined as

$$\Delta' = \Delta \setminus \{link(n, e_1), \ldots, link(n, e_k), own(n, e_1), \ldots, own(n, e_k)\} \cup \{idle(n)\}$$

Although the rule might remove a strict subset of *link* and *own* predicates involving n, the resulting configuration still satisfies Ψ'.

Since Δ, k, and σ are chosen in arbitrary way, we have that the considered invariant Φ' is an invariant for the whole family of rules of type s_k and r_k with $k \geq 0$.

Finally, we observe that for any Δ we have that if $\Delta \models \Psi'$ then $\Delta \models \Psi$. This prove that Ψ is still an invariant for the model resulting from the transformation described in the previous sections.

7 Conclusions and Related Work

We have presented a formal language for the specification of asynchronous distributed systems based on logic-based update rules for bipartite graphs. For the considered language, we have presented a verification approach that combines transformation schemes based on permutation properties and inductive verification. The proposed approach can be applied to verify the correctness of a distributed version of the dining philosopher protocol regardless the network topology and the number of nodes. Mechanization of the considered proof method could be an interesting direction for future work.

References

1. P. Liu andT. Wahl. Infinite-state backward exploration of boolean broadcast programs. In *Formal Methods in Computer Aided Design, FMCAD '14*, pages 155–162, 2014.
2. Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. FAST: acceleration from theory to practice. *STTT*, 10(5):401–424, 2008.
3. E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*, pages 70–80, 1998.
4. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving reachability analysis of infinite state systems by specialization. *Fundam. Inform.*, 119(3-4):281–300, 2012.
5. P. Ganty and A. Rezine. Ordered counter-abstraction - refinable subword relations for parameterized verification. In *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, pages 396–408, 2014.
6. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.
7. S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6(4), 2010.
8. B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 163–174, 2013.
9. A. Kaiser, D. Kroening, and T. Wahl. Lost in abstraction: Monotonicity in multithreaded programs. In *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, pages 141–155, 2014.
10. A. Kaiser, D. Kroening, and Thomas Wahl. A widening approach to multithreaded program verification. *ACM Trans. Program. Lang. Syst.*, 36(4):14, 2014.
11. B. König and V. Kozioura. Augur 2 - A new version of a tool for the analysis of graph transformation systems. *Electr. Notes Theor. Comput. Sci.*, 211:201–210, 2008.
12. D. Kroening. Automated verification of concurrent software. In *Reachability Problems - 7th International Workshop, RP 2013, Uppsala, Sweden, September 24-26, 2013 Proceedings*, pages 19–20, 2013.
13. R. Meyer and T. Strazny. Petruchio: From dynamic networks to nets. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 175–179, 2010.
14. K. S. Namjoshi and R. J. Trefler. Uncovering symmetries in irregular process networks. In *VMCAI*, pages 496–514, 2013.
15. Kedar S. Namjoshi and Richard 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.

16. J. Stückrath. Uncover: Using coverability analysis for verifying graph transformation systems. In *Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings*, pages 266–274, 2015.
17. `https://github.com/pierreganty/mist`.
18. `http://users.mat.unimi.it/users/ghilardi/mcmt/`.
19. `http://www.ahmedrezine.com/tools/`.
20. `http://www.ccs.neu.edu/home/wahl/Research/boom-and-cutoffs.html`.
21. `http://www.it.uu.se/research/docs/fm/apv/tools/pfs/`.
22. `http://www.it.uu.se/research/docs/fm/apv/tools/undip/`.
23. `http://www.liafa.jussieu.fr/~sighirea/trex/`.
24. `http://www.lsv.ens-cachan.fr/Software/fast/`.
25. `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.
26. `http://www.ti.inf.uni-due.de/de/research/tools/uncover/`.