Spin the Groove: Specification of Link Reversal Routing via Graph Transformations

Giorgio Delzanno and Riccardo Traverso

DIBRIS, Università di Genova, Italy

Abstract. We present a formal specification of the Gafni-Bertsekas algorithm, a Link Reversal Routing protocol, via Graph Transformation Systems (GTS) as provided in the GROOVE tool. Validation tools like GROOVE can naturally exploit systems symmetries both at the specification and state-exploration level. To exploit the most recent features of the tool like data fields, we study a height-based version of Gafni-Bertsekas that requires both graph transformation and comparison and updates of tuple of integers. Our case-study can be used as test-case to compare graph-based tools like GROOVE with other validation methods based on emunerative or symbolic search like SPIN and UPPAAL.

1 Introduction

Following a recently established connection between graph transformations and algorithmic verification [11,1,13], in this paper we explore the application of the GROOVE tool [9], consisting of a visual specification language, a simulator, and a model checker, to validate an case-study taken from the class of Link Reversal Routing algorithms. Link Reversal Routing (LRR) algorithms are one possible choice for route maintenance in Ad Hoc Networks. These protocols localize the effect of topology changes in order to react only when necessary. LRR protocols are based on graph update sequences to ensure that, in case of link failures, new routes to a given destination node can be automatically restored. Verification and validation of this class of protocols, and more in general of protocols for Ad Hoc Networks, is a challenging problem both in the case of finite- or infinite-state verification [7,2,4,5,3,8,14,16,15].

GROOVE has been developed as a support for (object-oriented) program (Java) and model (UML) transformations via the definition of executable graph production rules. Although protocol validation is not a central application, it has been considered e.g. in [12] a work presented at SPIN 2006. Indeed, GROOVE provides a model checker for (CTL/LTL) temporal properties that can be checked against the labelled transition system associated to (all possible derivations of) a graph grammar.

The former is a typical modelling feature of GROOVE that must be hardcode in languages like Promela/SPIN and UPPAAL. The latter can be modelled in recent extensions of GROOVE with manipulation of data fields, whereas it is a built-in feature in Promela/SPIN and in the UPPAAL specification language.

Our case-study represents a an trivial test-case for the usability and flexibility of the GROOVE tool in the application domain of protocol validation.

Outline In Section 2 we introduce the main concepts underlying the GROOVE tool. In Section 3 we describe Link Reversal Routing and the Gafni-Beserktas protocol. In Section 4 we describe in detail the specification of the protocol in GROOVE and the results of our analysis. In Section 5 we draw some conclusions and compare with other approaches for the specific classes of protocol considered here.

2 The GROOVE Tool

GROOVE [9] consists of a graphical interface tool (Simulator) that allows graphical editing of rules and graphs, and integrates the functionality of the Generator and Model Checker. The state space is stored as a Labelled Transition System (LTS), where each state is a graph and transitions are labelled by the rule applications. The strategy according to which the state space is explored can be set as a parameter. The Model Checker verifies properties expressed in branching or linear temporal logic in which propositions can be expressed via graph patterns. If a property does not hold, the Model Checker returns a counter-example. State space exploration in GROOVE is based on a graph representation of system states and on graph transformations to represent the state updates. Graphs transformations rule specify the following features: a pattern that must be present in the host graph in order for the rule to be applicable, where graph matching is used to select the pattern in the current configuration; sub-patterns that must be absent in the host graph in order for the rule to be applicable; nodes and edges to be deleted/added/merged from/to/in the graph. In the following section we formally define graph transformation rules.

2.1 Basic Notions

To simplify the definition of graph transformation systems we follow the style of [6,10]. A graph $G = \langle N, E, L\rangle$ consists of a finite set N of nodes, a finite set $E \subseteq N \times N$ of edges, and a labelling function L of nodes and edges. We use $\mathcal{G}$ to denote the set of all graphs, ranged over by $G, H, \ldots$.

Definition 1. *A graph matching $m : \mathcal{G} \to \mathcal{G}$ is a graph morphism that preserves node and edge labels, i.e., for $G = \langle N, E, L\rangle$ and $G' = \langle N', E', L'\rangle$, if $e = \langle n, n'\rangle \in E$, then $e' = \langle m(n), m(n')\rangle \in E'$, $L(n) = L'(m(n))$, $L(n') = L'(m(n'))$, and $L(e) = L'(e')$.*

A graph transformation rule specifies how the system evolves when going from one state to another.

Definition 2. *A graph transformation rule $p \in R$ is identified by its name ($Np \in \mathcal{N}$, where $\mathcal{N}$ is a global set of rule names) and consists of a left-hand-side graph (L_p), a right-hand side graph (R_p), and a set of so-called negative application conditions (NAC_p, which are super-graphs of L_p).*

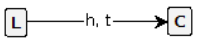
Definition 3. *A Graph Production System (GPS) $P = \langle I, R \rangle$ consists of a graph I representing the initial state of the system and a set of graph transformation rules R.*

The application of a graph transformation rule p transforms a graph G, the source graph, into a graph H, the target graph, by looking for an occurrence of L_p in G (specified by a graph matching m that cannot be extended to an occurrence of any graph in NAC_p) and then replacing that occurrence with R_p, resulting in H. Such a rule application is denoted as $G \rightarrow_{p,m} H$. Each GPS $P = \langle R, I \rangle$ specifies a (possibly infinite) state space which can be generated by repeatedly applying the graph transformation rules on the states, starting from the initial state I.

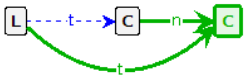
Definition 4. *A graph transition system $T = \langle S, \rightarrow, I \rangle$ generated by $P = \langle R, I \rangle$ consists of a set $S \subseteq \mathcal{G}$ of states, an initial state $I \in S$, and a transition relation $\rightarrow \in S \times R \times [\mathcal{G} \rightarrow \mathcal{G}] \times S$, such that $\langle G, p, m, H \rangle \in \rightarrow$ iff there is a rule application $G \rightarrow_{p,m} H'$ with H' isomorphic to H.*

2.2 Example

To illustrate how graph productions are specified in the GROOVE visual language we consider an example in which graphs represent dynamically created linked lists with tail insertion (put) and head removal (get). In the initial configuration we use two nodes as sentinels to denote the empty list. The first node has two forward pointers (h=head, t=tail) both pointing to the second node.

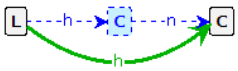


The *put* operation inserts a new node pointed by the tail pointer. The GROOVE visual language adopts colored nodes and egdes to denote deletion and addition of edges, nodes, and label updates. Indeed, the rule is specified as follows.



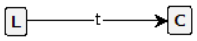
The dashed line denotes the deletion of the old t-edge. A deletion acts both as a guard (the edge being removed has to exist) and as a postcondition (the edge is removed from the graph). The thick lines denote the addition of two new edges. This notation can be expanded into a graph production containing a graph L_p with two nodes L and C connected via a t-edge in the left-hand side, and a graph R_p with three nodes L, C and C, with an n-edge connecting the last two nodes and a t-edge connecting the first and last node. The L_p graph is removed and the nodes and edges of R_p are created at its place. The nodes of R_p are linked to the nodes of L_p via a further graph morphism (usually denoted by using extra numerical labels to put in relation nodes in the left- and right-hand side).

Deletion of a cell is specified via the following rule.

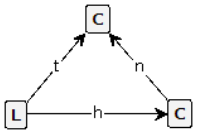


Dashed lines denote removal of old h- and n-edges. The thick line denotes the addition of a new h-edge. Clearly, the two productions assume that in the transformed graph there exists only one h-edge and only one t-edge (this property is invariant under applications of the productions).

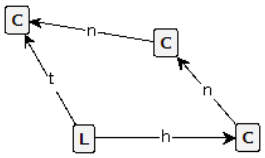
In the initial configuration the only fireable transition is *put* with the following matching pattern (*get* requires at least two n edges).



The application of the *put* rule produces the following new configuration.



We can now apply either *put* or *get*. The application of *get* (the whole graph is a match) leads to the initial configuration. Alternatively, a second application of *put* produces the following graph.



Again both rules have matching patterns and the transformation can continue either back to the previous state or to a list with an additional cell. Therefore, the considered graph production system generates lists of arbitrary length.

2.3 Extensions

In the latest version of GROOVE there are several specification facilities. Node labels can either be node types or flags. Tags are used to model a boolean condition, which is true for a node if and only if the flag is present.

Data Fields and Operations To specify data fields ranging over basic types like booleans, integers, and strings, we can use node attributes. Attributes are treated as special edges that do not point to a standard node, but to a node that corresponds to a data value. Operations over data fields are specified as node relations (evaluated automatically in data fields corresponding to the results).

Universal Quantification Universal quantification is another interesting feature of the input language. A universally quantified (sub)rule is a rule that is applied to all sub-graphs that satisfy the relevant application conditions, rather than just a single one as in the standard case. The use of universally quantified rules allows to naturally define parametric transformations (thus saving space in both the input model and in the state space). Universal quantification can be nested with existentially quantified. The documentation of the feature is not very detailed, so we used it in a restricted way.

Priorities and Control GROOVE provides different form of control in the rewriting process. The first method is based on priorities. Low-priority rules may only be applied if no higher-priority rule is applicable. A more sophisticated mechanism is to use in the control language. It can be used to define an order of application of the rules of that system (with constructs like looping, random choice, recursive functions). We will illustrate some of the extended features of GROOVE in the description of the specification of our case-study.

2.4 Simulation and Model Checking

GROOVE provides a GUI with a Simulator for the step-by-step visualization of the behaviour of a system, in which it is possible to highlight the matching pattern for a specific rule. Rules are partitioned in accord to the associated priorities. The simulator guides the execution via the corresponding rule ordering.

The verifier built in the GROOVE Simulator builds the LTS of a given graph production system in form of a reachability graph. Based on this representation of the state-space, the GROOVE Model Checker supports verification of CTL and LTL specifications that can be defined over graph patterns. This is achieved by using rule names as propositions. Specifically, the left-hand side of a formula (and the corresponding NAC) is used to check for the presence of a given sub-pattern in the current configuration. Consider a rule with name *bad*, whose left-hand side denotes a bad pattern (e.g. a cycle in a graph). Then, the firing of *bad* denotes the occurrence of the bad pattern in the reachability graph. Formulas are built over predicates defined over rule names, temporal operators like A, E (for CTL only), F, G, X (for CTL/LTL), and of their Boolean combinations (and/or/negation). A CTL formula like AG !*bad* can then be used to specify the safety property "the bad pattern can never be reached". In our linked list example we could specify bad patterns like self-loops with h- and t-edges (unreachable in our model). As a final remark, since the state-space of a model is potentially infinite, the verifier provides different strategies to find counterexamples like breadth-, depth-first, and bounded-depth search.

3 Link Reversal Routing Algorithms

Link Reversal Routing (LRR) algorithms are designed for large, dynamically changing networks, in which topology changes are too frequent to make flooding

of routing informations a viable solution. The main goal is to quickly repair a corrupted route with a new valid, but not necessarily optimal, one. The adaptivity and scalability of LRR make them suitable for ad hoc networks. We assume to work on networks in which nodes are connected via bidirectional channels (i.e. the communication layer is an undirected graph). LRR works with an overlay network used to identify routes to a specific destination node. The overlay network is defined via a Directed Acyclic Graph (DAG) with exactly one destination node (a node with only incoming links). Other nodes have either incoming and outgoing links or just outgoing links. When the last outgoing link of a node breaks, the node starts route maintenance, e.g., it reverses all incoming edges. After reversing edges, the maintenance procedure is recursively applied to the surrounding nodes. The algorithm stabilizes after finitely many steps if the graph is not partitioned. All the described algorithms are loop- and deadlock-free and establish multiple routes to one destination.

3.1 The Gafni-Bertsekas algorithm

The Gafni-Bertsekas algorithm is an LRR protocol with two different methods to handle the nodes with no outgoing links. In the *full reversal* method, the nodes which need to execute route maintenance reverse the direction of all their links. Thus, after that the nodes have only outgoing links. In the *partial reversal* method, reversal is done more efficiently. Specifically, every node, except the destination, keeps a list of the reversed links between it and its neighbour. When a neighbour of a node reverses the direction of the link between the nodes, the node writes the name of the neighbour down. As the node loses its last outgoing link, all other links than the already reversed ones are reversed. After that the list is emptied. If all the links have been reversed i.e. the list contains all the neighbour nodes, all the links are transformed from incoming to outgoing and the list is emptied. The reversals are informed by sending an update packet to the neighbours. If the update causes the neighbour to lose its last outgoing link, the reversing transaction continues with that node. The partial reversal method is usually more efficient than the full reversal method.

The partial reversal method can be implemented using *heights*. Every node u has a tuple of values $\langle \alpha_u, \beta_u, u \rangle$, where α_u is a non negative integer, and β_u is an integer. Initially, α_u is 0 for every node u. Tuples are totally ordered using the lexicographic ordering. Virtual edges (i.e. pointing towards the destination node) are directed from higher tuple to lower tuples, i.e., for every pair u and v of nodes, $(\alpha_u, \beta_u, u) > (\alpha_v, \beta_v, v)$ if and only if the virtual edge between them is directed from u to v. Route maintenance is triggered when a node u has no more incoming edges, i.e., the node is a local minimum. Let N_u be the set of neighbour nodes of node u. The node tries to repair the configuration by updating the value of α_u with a value that is larger than the minimum value of the α's for nodes in N_u,

$$\alpha'_u = (min_{v \in N_u} \alpha_v) + 1$$

The new value of α'_u may reverse edges directed towards nodes v s.t. α_v is equal to α'_u, i.e., edges reversed in a previous step. Thus, we also set the new value of

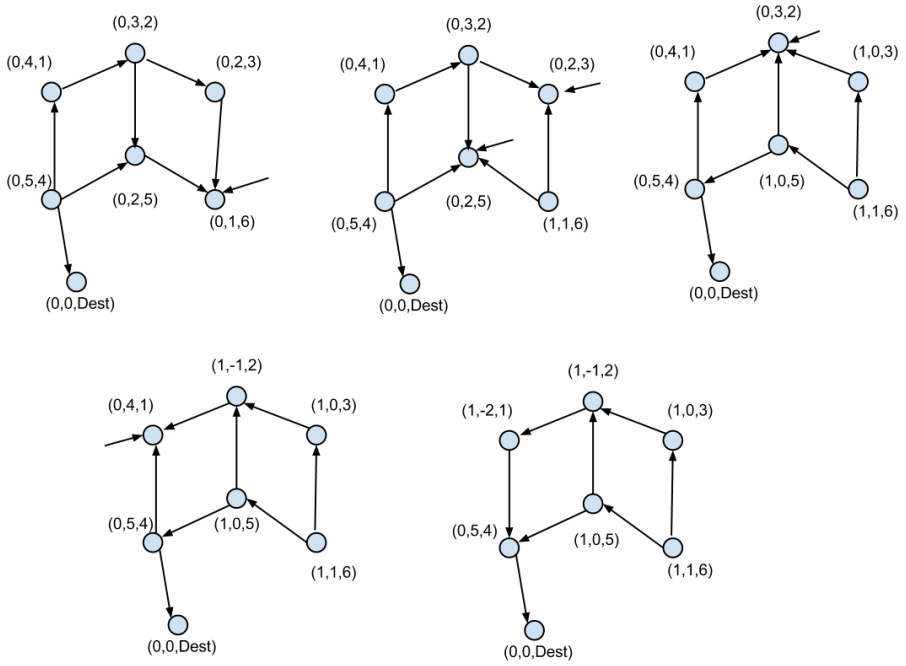


Fig. 1. Execution of Height-based Reversal.

β_u to be strictly less than the minimum value of the β's for those nodes in N_u with a value for α equal to α'_u, i.e.,

$$\beta'_u = (min_{v \in \{v' \in N_u | \alpha_{v'} = \alpha'_u\}} \beta_v) - 1$$

This way the edges directed towards the those nodes will be left unchanged. If the graph is connected then the ideal algorithm is ensured to terminate and to produce a new DAG pointing to the destination node. As in the scenario considered in the original algorithm, we assume here that route maintenance is performed after the failure of a single link and terminated before the subsequent link failure (the algorithm is designed for networks with such a relation between the frequency of the two types of events). We show an example of reversal steps in Figure 1.

We remark that the full reversal algorithm can be obtained by ignoring β and changing the updates of α as follows:

$$\alpha'_u = (max_{v \in N_u} \alpha_v) + 1$$

This way all incoming edges of a sink node are reversed into outgoing edges. Only when passing from informal specifications to formal ones, we can uncover details that must be taken into account in a real implementation of the protocol.

Since the informal specification of the LRR algorithm is based on graph transformations, it seems a natural case-study for a tool like GROOVE in order to fully exploit symmetries and compactness of graph production rules.

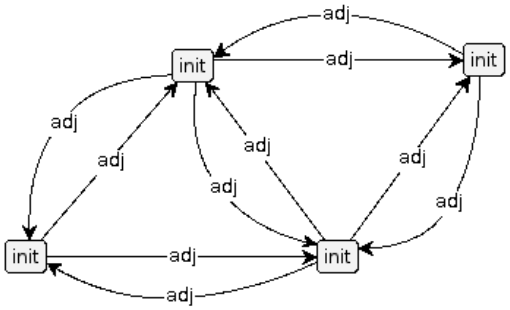


Fig. 2. Initial configuration with four nodes.

4 Formal Specification using GROOVE

In this section we describe a formal specification of the Gafni-Bersektas algorithm using the GROOVE input language. An initial configuration (e.g., Figure 2) for the protocol consists of a graph of *init* nodes, where pairs of *adj* edges represent links between adjacent nodes. In this state, nodes still do not have any identifier, α or β. The purpose of the rules shown in Figures 3 through 5 is to initialize such values and choose a destination among the nodes. They are all given the maximum priority level, in order to ensure no other rule will match before the initialization of the system is complete. First, the rule INIT-

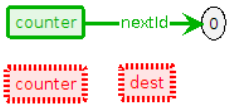


Fig. 3. INIT-COUNTER: Initialization of node states.

COUNTER introduces a new *counter* node (nodes with thick borders are added as a side effect of firing the rule, just like with thick edges) linked via a *nextId* edge to a special value node labelled by the integer 0. When writing rules, a value node can either hold a specific value from some supported data domain, or simply state the associated data domain. In this case the node may assume any value of that domain, depending on the matching. INIT-COUNTER may fire only if

the system is still uninitialized, i.e. both the *counter* and a destination node are missing. This sort of negative preconditions (NAC) is expressed through nodes with thick, dashed borders. The rule INIT-LOCALS fires for each *init* node,

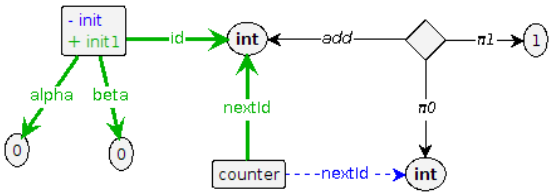


Fig. 4. INIT-LOCALS: Initialization of local variables.

setting both *alpha* and *beta* to 0, and choosing the *id* of the node according to the *counter*. The label of the node is changed to *init*1 to mark it as ready and the *counter* is increased by 1. This is achieved by first removing label *init* and then adding label *init*1. Labels added or removed as a postcondition of a rule are respectively preceded by a plus or a minus sign. The increment, just like any other expression on value nodes, has to be encoded with nodes and arcs. The diamond-shaped nodes match as tuples whose values are specified thanks to projection arcs $\pi 1, \pi 2, \cdots$, and they act as input arguments to operator edges. The latter are used to match against value nodes, which are used to hold the result of the operation. In Figure 4, rule INIT-LOCALS adds the constant 1 to the old value of *nextId*, matches an integer valued node against the result, and sets it to be the new *nextId* as well as the identifier of the node being initialized. When INIT-LOCALS is done initializing the nodes (i.e. there are no

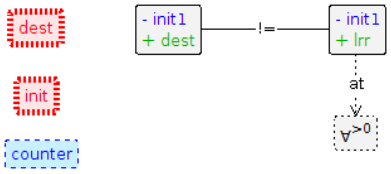


Fig. 5. INIT-DEST: Non-deterministic choice of destination node.

init nodes any more), INIT-DEST (Figure 5) non-deterministically picks one of them as the destination of the routing protocol. Furthermore, every remaining *init*1 node becomes an *lrr*, an active node running the LRR protocol, and the *counter* node is removed. As in the case of edge removal, nodes being removed are marked through a thin, dashed border. Because of the universal quantification in the $\forall^{>0}$ node, the $+lrr$ one is matched against every *init*1 node but the only

one chosen as destination. In general, however, nothing forbids for two different nodes of a rule to match against the same node of a configuration. In INIT-DEST we ensure to distinguish the *lrr* nodes w.r.t. *dest* with an inequality constraint specified via an edge with label ! =. The firing of INIT-DEST marks the end of the initialization phase and the beginning of the simulation of the Gafni-Bertsekas LRR protocol. At first no virtual edges (labelled by *next*) towards the destination exist, as the nodes did not interact with each other. In such a case

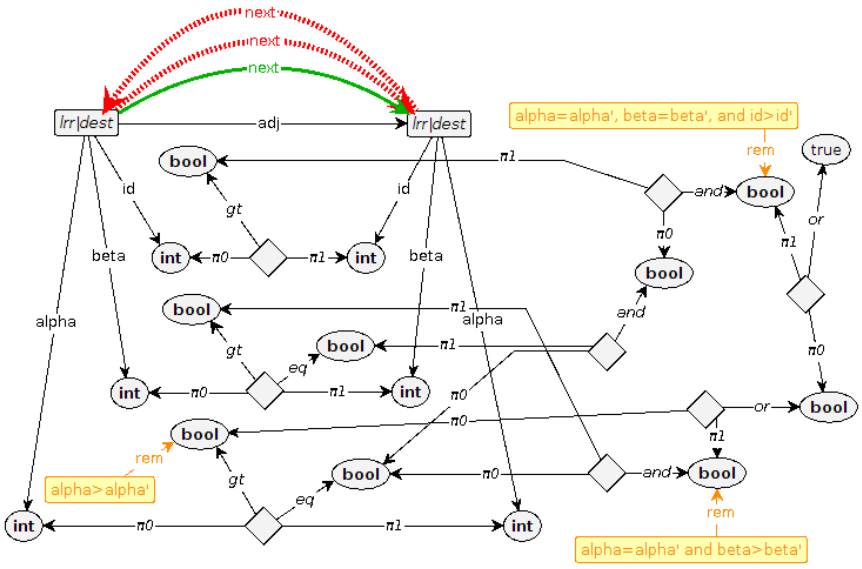


Fig. 6. NEW-LINK: Creation of a new link.

the preconditions to fire NEW-LINK (Figure 6) are satisfied. The special syntax *lrr*|*dest* matches against nodes labelled by either *lrr* or *dest*. Thick, dashed edges are treated as negative preconditions. The rule may appear as a complex one to understand, but all it does is to determine that the *lrr*|*dest* node on the left has an height lexicographically greater than the height of the one on the right. When the network is initialized and all of the *next* edges are ready, a configuration with four nodes may look like in Figure 7. This configuration also exposes an example of a sink (with $id = 1$), i.e., a node other than the destination without outgoing *next* edges which triggers the route maintenance phase. The rules in this phase have to fire in a specific order for the protocol to work, and some of them have to be fired for every possible matching. By giving strictly decreasing priorities to the rules in Figures 8 through 15 we capture exactly this behaviour. In a situation such as in Figure 7, a new *sink* can be detected through rule SINK. Since we model atomic updates to virtual edges through multiple updates to our model, we ensure the rules will work only on a single sink at a time by introducing a

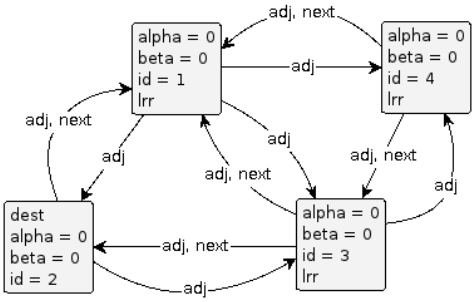


Fig. 7. Fully-initialized network with four nodes and *next* edges.

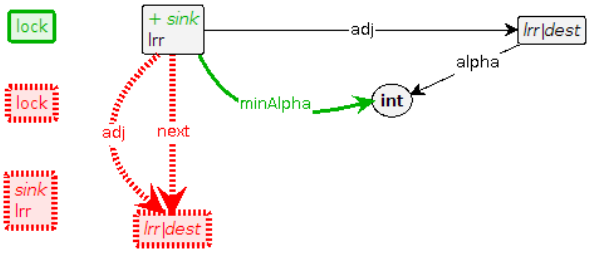


Fig. 8. SINK: Detection of a sink node.

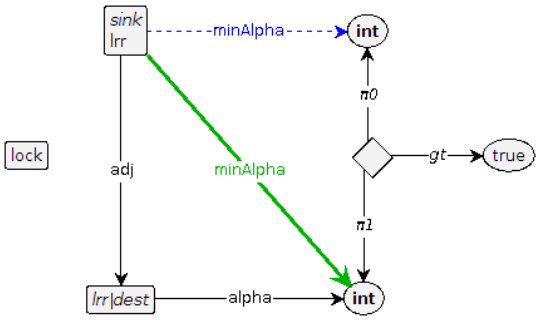


Fig. 9. ALPHA-STEP: Search for the minimum *alpha*.

lock node upon detection: SINK may fire only if *lock* is not already present and there are no *sink* nodes. Then, the node is marked with the *sink* label, and

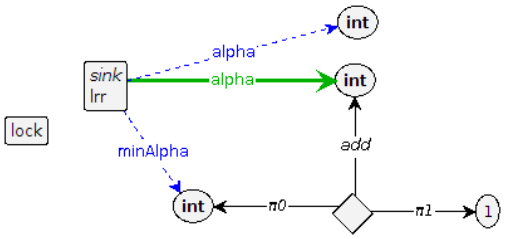


Fig. 10. ALPHA-UPD: *alpha* is updated.

minAlpha is initialized to be the same as the *alpha* of some neighbour. We will need this value later, as well as a *minBeta* at some point, in order to compute the minimum values for *alpha* and *beta* among the eligible neighbours (as previously described in Section 3). When a node officially becomes a *sink*, the first thing it

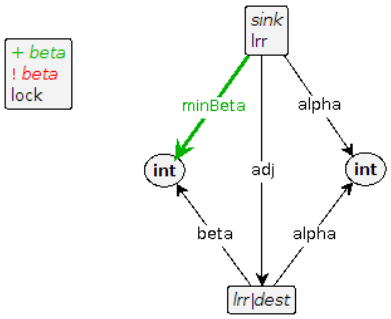


Fig. 11. BETA-INIT: Start the search for *minBeta*.

does is to search for the minimum value of *alpha* among his neighbours, via rules ALPHA-STEP and ALPHA-UPD (Figures 9 and 10). ALPHA-STEP updates *minAlpha* each time it is greater than the *alpha* of some neighbour (the *gt* operator in the rule). ALPHA-UPD, which thanks to the smaller priority can be fired only when ALPHA-STEP does not match any more, replaces the old value of *alpha* with $minAlpha + 1$. Once *alpha* is updated, the LRR protocol can proceed with the update of *beta*, which is performed via the rules BETA-INIT, BETA-STEP, and BETA-UPD of Figures 11, 12, and 13. Differently from *alpha*, *beta* has only to be compared w.r.t. the neighbours sharing the same *alpha* as the *sink*. BETA-INIT searches for a neighbour with the same *alpha*

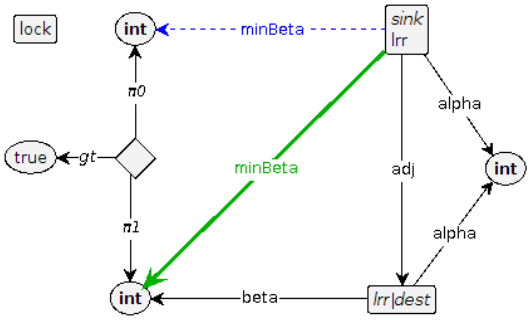


Fig. 12. BETA-STEP: Search for the minimum *beta*.

as the recently updated one in the *sink*, and initializes the *minBeta* edge. In

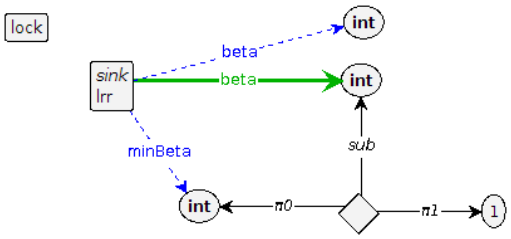


Fig. 13. BETA-UPD: *beta* is updated.

order to avoid repeating the initialization multiple times, the rule adds a *beta* label to the *lock* node and makes sure that such a label was not present before (negative preconditions on labels are preceded by an exclamation mark). Rule BETA-INIT cannot be fired when there are no neighbours with the same *alpha* as the *sink*: when it is the case, *beta* does not need to be updated. Without the *minBeta* edge introduced by BETA-INIT, the following rules BETA-STEP and BETA-UPD will skip too, as its presence is a precondition to both of them. Rule BETA-UPD roughly corresponds to ALPHA-UPD, as it updates the value of *beta* to $minBeta - 1$. At this point both *alpha* and *beta* in the *sink* have been updated, so we can proceed with the reversal of all incoming *next* edges in the *sink* which originate from neighbours with a smaller height. Rule REVERSAL of Figure 14 works lexicographically w.r.t. the heights of the *sink*'s neighbours, exactly as NEW-LINK, except it changes the orientation of *next* edges instead of adding new ones. Finally, when all *next* edges have been adjusted, the system fires REVERSAL-END, deletes the *sink* label from the node, and removes the *lock* (Figure 15). In the case of the example configuration in Figure 7, a complete run of link reversal in the sink node with $id = 1$ would result in the configuration

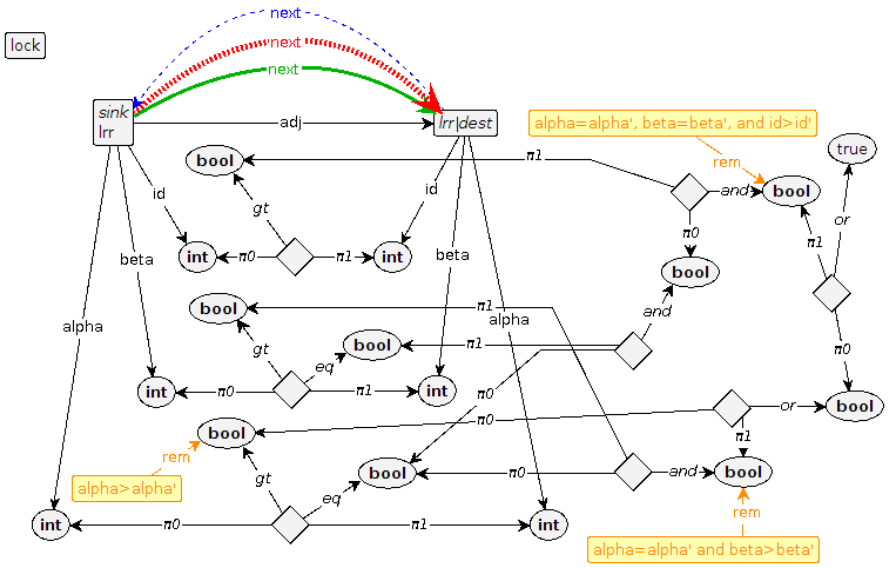


Fig. 14. REVERSAL: Reversal of virtual edges according to the new height.

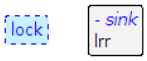


Fig. 15. REVERSAL-END: End of reversal.

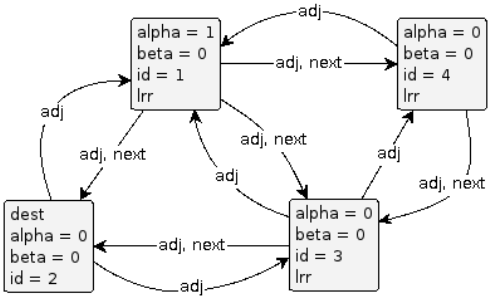


Fig. 16. Example network without sinks.

of Figure 16. Its *alpha* has been updated from 0 to 1, *beta* is still 0 (since there is no neighbour with $alpha = 1$), and all of its *next* edges have been reversed.

4.1 Automated Analysis

Based on our model, we have performed different types of analysis using the GROOVE Model Checker on the height-based model for configurations of fixed size. In particular, we have introduced link deletion rules in order to trigger the link reversal phases. On link deletions that lead to partitioning we have verified the possible non-termination of the algorithm (generation of strictly increasing sequences of α values).

The introduction of locks for the specification of reversal steps (i.e. updates of heights) is necessary in order to avoid interferences between different updates. Real implementation has to take into consideration this kind of details in order to avoid race conditions during operations like the computation of new α's and β's.

Furthermore, with a preliminary bound on maximal values of α's, we have automatically checked the absence of loops (loop-freedom) using CTL properties in which the bad pattern is a cycle of size less or equal than the size of the network. To express these patterns, we need to specify several rules to be used as targets for the exploration, one for each loop size. The logic has no spatial operators, e.g., to express reachability properties on the structure or to express absence of cycles in arbitrary topologies. Nevertheless, we can exploit sub-programs to define this kind of properties (i.e. propositions that express spatial navigation) introducing however a possible overhead in the state-space generation.

For the configuration with four nodes of Figure 2 and arbitrary link deletions, verification generates 1972 states and requires about 1 second[1] with an upper bound on α equal to 10. We considered also the automated generation of arbitrary topologies of fixed size, by introducing a link creation rule which may connect together previously disconnected nodes. With an upper bound on α equal to 8 and a maximum number of dynamic changes to the topology equal to 8, starting from the same initial configuration with four nodes as before, we obtained 56395 states in about 3 minutes. Both of the considered variants satisfied loop-freedom.

5 Conclusion

Related Work Specification and verification of routing protocols dates back to seminal work like the semi-automated correctness proof of AODV given in [2]. More recently, model checking tools (e.g. SPIN [8] and Uppaal[7]) and constraint-based engines [14,15,16] have been applied to verification of ad hoc and wireless protocols. In these approaches executions of a fixed number of agents are explored with enumerative or symbolic methods [8,7], or generating positive/negative constraints on links in a lazy manner [14,15,16]. Parameterized verification

[1] On a common laptop with a 2.53GHz CPU and 4GB of RAM.

of this kind of protocols has been studied from a theoretical point of view in [4,5,3], where decidability and complexity frontiers have been given for problems like control state reachability (reachability of a state in which a node has a certain state). The use of graph transformation systems for automated validation of dynamic systems has been proposed in [12] using GROOVE, and [11] via a symbolic backward exploration working on ideals generated by the graph minor relation. Decidability of reachability problems for Graph Transformation Systems are studied in [1].

Usability A comparison of the performance of GROOVE, used as a verification tool for dynamical systems, and SPIN is considered in [12] (SPIN 2006) for different case-studies. Since tools like SPIN and UPPAAL are based on consolidated heuristics and optimizations, we believe that performance is not the only measure to consider. In fact, in this section we will focus our attention on the usability of the different languages/tools using Gafni-Bertsekas as guiding example.

As a first observation, we first remark that graph updates are typically defined modulo graph isomorphism. Therefore, GTS provide compact specifications of dynamic systems based on structural modifications like LRR protocols (abstractly LRR can be viewed as a graph operation). Negative conditions and universal quantification further simplify the specification. Indeed, they allow complex combinations of local and global modifications of a pattern in a single rule. The visual language used in GROOVE hides the more operational view of LTS adopted in Promela and UPPAAL which require some knowledge of C-like programming languages.

Furthermore, the specification of initial topologies (graphs) and of reachability patterns using a visual language can facilitate the analysis of a protocol (and the divulgation of the results). Graph productions can also be used to generate parametric initial and target configurations as we discussed in the previous section.

The above presented features are peculiar of GTSs and are less common in other specification languages for distributed systems. Indeed in Promela/SPIN and UPPAAL, it is often necessary to hardcode the network topology using global shared data structures (e.g. an adjacency matrix). This introduces a lower level of details with respect to the pure protocol logic. Similarly, selective (i.e. restricted to neighbours) point-to-point or broadcast communication must be simulated by adding conditions on send and receive operations defined on top of the considered implementation of the network topology. These details are completely abstracted away in GROOVE.

Concerning symmetries at the specification level, Promela/SPIN and Uppaal provide templates for defining parametric processes, e.g., by using arrays of processes or by instantiating templates using spawning operators. However GTS rules with negative conditions and nested universal/existential quantification seem to provide a higher level of abstraction (i.e. exploit locality at the structural level) w.r.t. to standard specification constructs. For instance, to sim-

ulate universal quantification, it seems necessary to use loops and iterations, introducing however an overhead in the state-space exploration (and the need of moving to a lower level of details).

Search strategies like different types of visits, priorities, and confluence (to enforce sequentiality of blocks of rules) are also provided in other tools (eq. deterministic constructs and committed/urgent locations in UPPAAL). The GUI of the Simulator/Model Checker provided by GROOVE shares similarities with those of UPPAAL or GUI interfaces for SPIN. However SPIN provides built-in strategies to enforce weak fairness conditions.

Symbolic representation of data relation is typical of Promela and UPPAAL specifications which also provide programming constructs like macros, while-loops, conditionals and assignments. As shown in our case-study, the most recent versions of GROOVE provide ways to specify data fields (as node attributes) and relations as well as control programs. Unfortunately, the specification of updates of data fields like those required in state transitions of communication protocols is quite complex in the input language of GROOVE (as shown in the rules for updating α and β). This is related to the fact that GROOVE has not been designed for protocol validation but to define model transformations (e.g. in UML and Java). This feature makes the specification of protocols combining structural and data updates quite difficult to write and read.

Spin the Groove The previous considerations suggest future directions of research for combining the SPIN/Promela specification style (e.g. for individual process components) inspired to programming languages like C with the level of abstraction provided by GROOVE for specifying configurations and structural update rules (e.g. providing symmetries at the level of specification). The combination of the two paradigms could give the right balance to cover different level of abstractions to reason on distributed protocols. In particular the possibility of decoupling the specification of the structural level from the specification of the individual behaviour of components seems necessary to benefit from the use of visual languages like that used in GROOVE without losing the level of details provided by Promela.

Furthermore, by exploiting the structural dimension of GTS, we believe that a great improvement for the verification task could be that of combining spatio- and temporal-specification. To be more specific, for properties involving reachability within a state graph (not in the reachability graph) it would be natural to consider more complex predicates than production rules, e.g., spatial navigation modalities. This kind of properties can be simulated via ad hoc graph productions however at the cost of a possible overhead for state-exploration.

As a final remark, lazy strategies for state-space exploration like those provided in [15] (based on constraints on presence/absence of edges) could be an interesting heuristics to improve performance of the model checking engine (orthogonal to SPIN/UPPAAL ones).

References

1. 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*, pages 101–116, 2012.
2. K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.
3. G. Delzanno, A. Sangnier, R. Traverso, and G. Zavattaro. On the complexity of parameterized reachability in reconfigurable broadcast networks. In *FSTTCS*, pages 289–300, 2012.
4. G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR*, pages 313–327, 2010.
5. G. Delzanno, A. Sangnier, and G. Zavattaro. On the power of cliques in the parameterized verification of ad hoc networks. In *FOSSACS*, pages 441–455, 2011.
6. H. Ehrig and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformations (Vol 1–3)*. World Scientific Publishing, 1997.
7. A. Fehnker, R. J. van Glabbeek, P. Höfner, A. McIver, M. Portmann, and W. L. Tan. Automated analysis of aodv using uppaal. In *TACAS*, pages 173–187, 2012.
8. A. Fehnker, L. van Hoesel, and A. Mader. Modelling and verification of the lmac protocol for wireless sensor networks. In *IFM*, pages 253–272, 2007.
9. A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova. Modelling and analysis using groove. *STTT*, 14(1):15–40, 2012.
10. R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, 2006.
11. S. Joshi and B. König. Applying the graph minor theorem to the verification of graph transformation systems. In *CAV*, pages 214–226, 2008.
12. H. Kastenberg and A. Rensink. Model checking dynamic states in groove. In *SPIN*, pages 299–305, 2006.
13. B. König. *Analysis and verification of systems with dynamically evolving structure*. PhD thesis, Universität Stuttgart, 2004.
14. A. Singh, C. R. Ramakrishnan, and S. A. Smolka. A process calculus for mobile ad hoc networks. In *COORDINATION*, pages 296–314, 2008.
15. A. Singh, C. R. Ramakrishnan, and S. A. Smolka. Query-based model checking of ad hoc network protocols. In *CONCUR*, pages 603–619, 2009.
16. A. Singh, C. R. Ramakrishnan, and S. A. Smolka. A process calculus for mobile ad hoc networks. *Sci. Comput. Program.*, 75(6):440–469, 2010.