# A Methodology for Designing Dynamic Topology Control Algorithms via Graph Transformation

Roland Kluge, Gergely Varró, and Andy Schürr

Real-Time Systems Lab, TU Darmstadt

**Abstract.** This paper presents a constructive, model-driven methodology for designing dynamic topology control algorithms. The proposed methodology characterizes valid and high quality topologies with declarative graph constraints and formulates topology control algorithms as graph transformation systems. Afterwards, a well-known static analysis technique is used to enrich graph transformation rules with application conditions derived from the graph constraints to ensure that this improved approach always produces topologies that (i) are optimized wrt. to a domain-specific criterion, and (ii) additionally fulfill all the graph constraints.

**Keywords:** topology control, graph constraints, static analysis

## 1 Introduction

In the telecommunication engineering domain, wireless sensor networks (WSNs) [14] are a highly active research area. For instance, WSNs are applied to monitor physical or environmental conditions with distributed, autonomous, battery-powered measurement devices that cooperatively transmit their collected data to a central location. To extend the battery lifetime of these measurement devices, topology control (TC) [14] is carried out on WSNs to inactivate redundant communication links by reducing their transmission power. The most significant requirements on TC algorithms include the ability to (i) handle continuously changing network topologies, (ii) operate in a highly distributed environment, in which each node can only observe and modify its local neighborhood, and (iii) still guarantee important local and global formal properties for their neighborhood and the whole network, respectively.

The design and implementation of TC algorithms are, therefore, challenging and elaborate tasks, especially if a high quality of service is a non-functional requirement of the overall WSN. In a typical development setup, several variants of different TC approaches have to be prepared and quantitatively assessed in an iterative process. In each iteration, (i) a new variant must be individually designed and implemented for a distributed environment, (ii) the preservation of required formal properties (e.g., connectivity) must be proved, and (iii) performance measurements must be carried out in a corresponding runtime environment (testbed or simulator). This last point also assumes an interaction between

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-21155-8_15

the TC algorithm and the runtime environment. More specifically, the runtime environment alters the topology, which has to be repaired by the TC algorithm in an *incremental* manner, namely, by retaining unaltered parts of the topology as much as possible.

The main challenge with the individual design of each new algorithm variant is that it strongly relies on the experience of highly qualified experts. To enable a more systematic and well-engineered approach, model-driven principles [1] can be applied to the development of TC algorithms, as in the case of many other success stories [17]. More specifically, topologies can be described by graph-based models, and possible local modifications in the topology can be specified declaratively with graph transformation (GT) rules [13]. Although this approach provides a well-defined procedure for repairing the topology even in a distributed environment, it cannot ensure that all required formal properties hold for the repaired topology.

A well-known, constructive, static analysis technique [7] has been established in the GT community to formulate structural invariants and to guarantee that these invariants hold. In this setup, graph constraints specify positive or negative patterns, which must be present in or missing from a valid graph, respectively. An automated process then derives additional rule application conditions from graph transformation rules and graph constraints to ensure that the application of the enriched new rules never produces invalid graphs. Although this technique has already been employed in scenarios where graph constraints represent invariants that must hold permanently [10], its applicability in the TC domain is hindered by the fact that topology modifications performed by the runtime environment may temporarily (and unavoidably) violate graph constraints.

In this paper, we propose a new, constructive, model-driven methodology for designing TC algorithms by graph transformation. We demonstrate the approach on the kTC algorithm [15]. More specifically, we define graph constraints from algorithm-specific formal and quality requirements to characterize valid and high quality topologies. Such desirable topologies are to be produced by a TC algorithm, which is formulated as a graph transformation system. We iteratively refine this transformation by applying the constructive approach of [7], which enriches the rules at compile-time with additional application conditions, derived from the graph constraints. Finally, we prove that our improved GT-based TC algorithm terminates and always produces connected topologies that fulfill all the graph constraints.

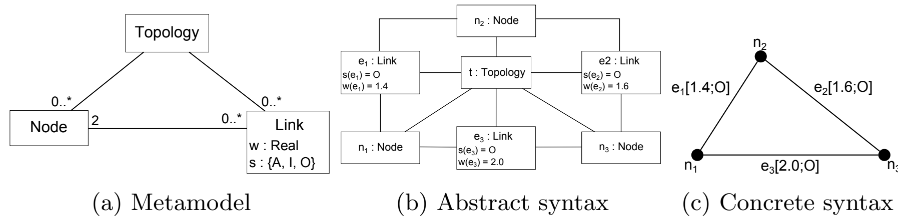
Section 2 introduces modeling and TC concepts. Section 3 describes graph constraints and their application to describe invariants of TC algorithms. Section 4 illustrates GT concepts on a basic TC algorithm. Section 5 delineates the construction methodology that enriches the generic TC algorithm with graph constraints. Section 6 summarizes related work, and Section 7 concludes our paper with a summary and an outlook.

## 2 Modeling Concepts and Topology Control

This section introduces fundamental modeling and topology control concepts.

### 2.1 Basic Modeling Concepts

A *metamodel* describes the basic concepts of a domain as a graph. In this paper, network topologies are used as a running example, whose metamodel is shown in Figure 1a. *Classes* represent the nodes of the metamodel, and *associations* represent the edges between classes. An association end is labeled with a *multiplicity*, which restricts the number of target objects that can be reached by navigating along an association in the given direction. *Attributes* (depicted in the lower part of the classes) store values of primitive or enumerated types.




(a) Metamodel (b) Abstract syntax (c) Concrete syntax

Fig. 1: Topology metamodel and a sample topology in abstract/concrete syntax

A *topology* is a graph that consists of *nodes* and *(communication) links* between nodes. As specified in Figure 1a, a link connects exactly 2 nodes, and a node can be an endpoint of zero or more incident links. A link $e$ has an algorithm-specific *weight* $w(e)$ and *state* $s(e)$ attribute, whose role will be explained together with the corresponding topology control concepts, shortly.

**Example.** Figures 1b and 1c depict a sample topology with three nodes ($\mathsf{n}_1$,$\mathsf{n}_2$,$\mathsf{n}_3$) and three links ($\mathsf{e}_1$,$\mathsf{e}_2$,$\mathsf{e}_3$) forming a triangle in abstract and concrete syntax, respectively. In the rest of the paper, the concrete syntax notation is used, which denotes nodes and links by black circles and solid lines, respectively. Each link is labeled with its name, followed by its weight and state in brackets.

### 2.2 Topology Control

Topology control (TC) is the discipline of adapting WSN topologies to optimize network metrics such as network lifetime [14]. The nodes in a WSN topology are often battery-powered sensors, which limits the lifetime of the network. For each node, TC selects a logical neighborhood, which is a subset of the nodes within its transmission range. Afterwards, each node may reduce its transmission power to reach its farthest logical neighbor. The weight attribute $w(e)$ of a link $e$ describes

the cost of communicating across this link. In this paper, we use the distance of the end nodes of a link as a weight metric.

Figure 2 shows the interaction of an evolving network, represented as a stream of topology change events (e.g., link weight change, addition or removal of nodes or links[1]), and a topology control algorithm, which takes an input topology and produces an output topology, which is a subgraph of the input topology. In this setup, a *batch TC algorithm* reconsiders every link in the topology in each execution, irrespective of the actual topology change events, while an *incremental TC algorithm* only reevaluates the modified parts of the topology. In this paper, we assume that no topology change events occur while the TC algorithm is running.

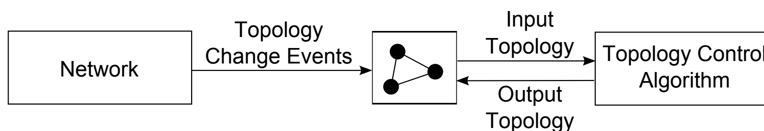



Fig. 2: Modification of the topology by network evolution and TC algorithm

We introduce a *state* attribute for links to handle topologies in transition and to describe batch and incremental TC algorithms uniformly. A link is *active* (A) or *inactive* (I) if it is included in or excluded from the output topology by the TC algorithm, respectively. A link is *outdated* (O) if it has not yet been categorized as active or inactive by the TC algorithm. A topology control algorithm tries to activate or inactivate outdated links, while topology change events may outdate links. Note that the set of active links represents the output topology, which is always a subgraph of the whole input topology.

**Important properties of topologies and TC.** Every TC algorithm must (i) terminate without outdated links and (ii) preserve topology connectivity. The first property ensures that each link in the input topology is definitely part or not part of the output topology. A topology is *connected* if the subgraph induced by its active and outdated links is connected. This entails that the output topology is connected if and only if its subgraph induced by the active links is connected.

**Example.** Figure 3 illustrates an incremental variant of the kTC algorithm [15], which inactivates exactly those links in the sample topology that are the longest link in a triangle and that are at least $k$-times longer than the shortest link in the same triangle. We always assume $k > 1$.

Initially, all links are outdated ①. The first execution of kTC ($k = 1.5$) activates or inactivates all links ②. A move of node $n_4$ might trigger a weight change on link $e_4$, which outdates this link ③. As link $e_4$ is no longer the longest link in the triangle ($n_2$,$n_3$,$n_4$), the next (incremental) iteration of kTC activates $e_2$, inactivates $e_4$, and retains the state of links $e_1$, $e_3$, and $e_5$ ④.

[1] Such events occur, e.g., when nodes move and join or leave the network.

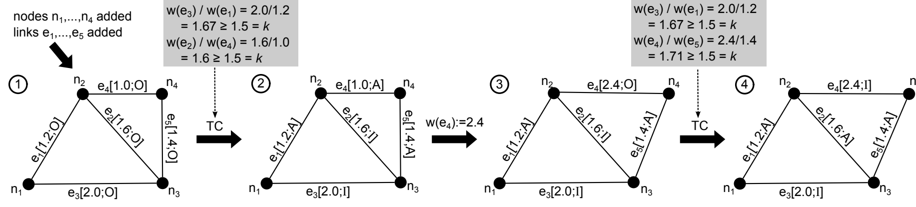



Fig. 3: Incremental topology control with kTC

## 3 Characterizing Topologies with Graph Constraints

As a first step in our constructive TC design methodology, formal properties and quality requirements of the TC algorithm are analyzed, and graph constraints are defined as invariants to characterize valid and high quality topologies.

A *pattern* is a graph consisting of node and link variables together with a set of attribute constraints. A *node (link) variable* serves as a placeholder for a node (link) in a topology. An *attribute constraint* is a predicate over attributes of node and link variables. A *match of a pattern* $P$ *in a topology* $G$ maps the node and link variables of $P$ to the nodes and links of $G$, respectively, such that this mapping preserves the end nodes of the link variables and all attribute constraints are fulfilled.

A *graph constraint* consists of a *premise* pattern and a *conclusion* pattern such that (i) the premise is a subgraph of the conclusion, and (ii) the attribute constraints of the conclusion imply the attribute constraints of the premise. A *positive (negative) graph constraint* $\mathcal{P}$ $(\mathcal{N})$ *is fulfilled on a topology* $G$ if each match of the premise in the topology $G$ can (cannot) be extended to a match of the conclusion in the same topology $G$.

**Demonstration on kTC.** For our running example, we specify three constraints: two algorithm-specific constraints of kTC and a third constraint that forbids outdated links in the output topology.

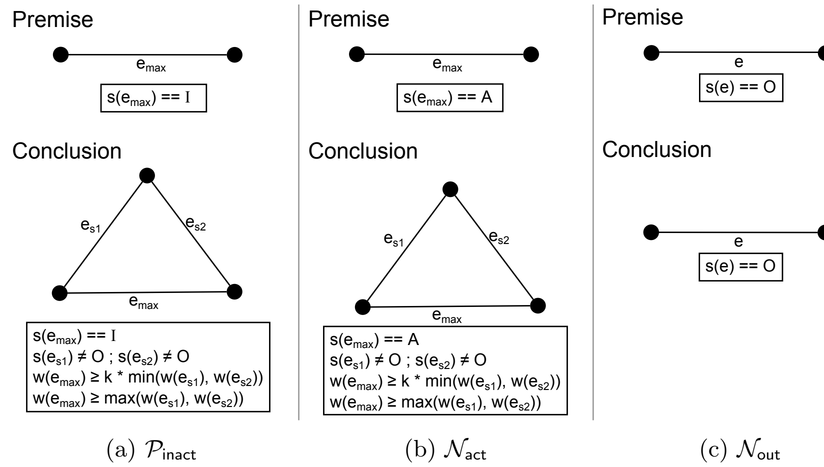
kTC inactivates a link *if and only if* this link is the longest link in a triangle and if it is additionally at least $k$-times longer than the shortest link in the same triangle. This equivalence yields the following two constraints:

- ⇒ The *inactive-link constraint* $\mathcal{P}_{\mathsf{inact}}$, depicted in Figure 4a, states that *each* inactive link $\mathsf{e}_{\mathsf{max}}$ must be part of a triangle in which (i) $\mathsf{e}_{\mathsf{max}}$ is the longest link, (ii) $\mathsf{e}_{\mathsf{max}}$ is at least $k$-times longer than the shortest link, and (iii) $\mathsf{e}_{\mathsf{s1}}$ and $\mathsf{e}_{\mathsf{s2}}$ are either active or inactive.
- ⇐ The *active-link constraint* $\mathcal{N}_{\mathsf{act}}$, depicted in Figure 4b, states that *no* active link $\mathsf{e}_{\mathsf{max}}$ may be part of a triangle in which (i) $\mathsf{e}_{\mathsf{max}}$ is the longest link, (ii) $\mathsf{e}_{\mathsf{max}}$ is at least $k$-times longer than the shortest link, and (iii) $\mathsf{e}_{\mathsf{s1}}$ and $\mathsf{e}_{\mathsf{s2}}$ are either active or inactive.

Due to the attribute constraints that require $\mathsf{e}_{\mathsf{s1}}$ and $\mathsf{e}_{\mathsf{s2}}$ to be active or inactive, the active-link constraint may only be violated and the inactive-link

constraint may only be fulfilled if all links in the triangle are either active or inactive. Therefore, topology change events never violate the active-link constraint $\mathcal{N}_{\mathsf{act}}$ because no new match of its conclusion may arise. In contrast, the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$ may be violated by a topology change event, for instance, if an inactive link belongs to exactly one match of the conclusion of $\mathcal{P}_{\mathsf{inact}}$ and if any of the links $\mathsf{e}_{\mathsf{s1}}$ or $\mathsf{e}_{\mathsf{s2}}$ gets outdated.

Finally, the *outdated-link constraint* $\mathcal{N}_{\mathsf{out}}$, depicted in Figure 4c, describes the general requirement that the (output) topology shall not contain any outdated links. Any outdated link causes a violation of $\mathcal{N}_{\mathsf{out}}$ because its premise and conclusion are identical.




(a) $\mathcal{P}_{\mathsf{inact}}$ (b) $\mathcal{N}_{\mathsf{act}}$ (c) $\mathcal{N}_{\mathsf{out}}$

Fig. 4: Two algorithm-specific constraints and one general graph constraint

## 4 Specifying Topology Control with Programmed Graph Transformation

In the second step of our design methodology, topology change events and TC operations are described by graph transformation (GT) rules. The dynamic behavior of TC algorithms is specified by programmed graph transformation [4], which carries out basic topology modifications by applying graph transformation rules whose execution order is defined by an explicit control flow.

**Programmed Graph Transformation Concepts.** A *mapping from pattern $P$ to pattern $P'$* maps a subset of node and link variables of pattern $P$ to a subset of node and link variables of pattern $P'$, respectively, such that this mapping preserves the end nodes of the link variables. A *graph transformation rule* consists of a *left-hand side (LHS)* pattern, a *right-hand side (RHS)* pattern, *negative*

*application condition (NAC)* patterns, and mappings from the LHS pattern to the RHS and NAC patterns. To enable meaningful attribute assignments, the predicate in the attribute constraints of the RHS pattern can only be an equation with an attribute of a single node or link variable on its left side.

A *GT rule is applicable to a topology $G$* if a match of the LHS pattern exists in the topology that cannot be extended to a match of any NAC pattern. The *application of a GT rule at a match of its LHS pattern to a topology $G$ produces a topology $G'$* by (1) preserving all nodes (links) of the topology that are assigned to a node (link) variable of the LHS pattern, which *has* a corresponding node (link) variable in the RHS pattern (black elements without additional markup); (2) removing those nodes (links) of the topology that are assigned to a node (link) variable of the LHS pattern, which has *no* corresponding node (link) variable in the RHS pattern (red elements with a '−−' markup); (3) adding a new node (link) to the topology for each node (link) variable of the RHS pattern, which has no corresponding node (link) variable in the LHS pattern (green elements with a '++' markup); and (4) assigning node (link) attributes (operator ':=') in such a manner that the attribute constraints of the RHS pattern are fulfilled.

Control flow is specified in our approach with an activity diagram based notation in which each activity node contains a graph transformation rule. A *regular activity node* (denoted by a single framed, rounded rectangle) with one unlabeled outgoing edge applies the contained GT rule *once* to *one* arbitrary match. A regular activity node with an outgoing [Success] and [Failure] edge applies the contained GT rule and follows the [Success] edge if the rule is applicable at an arbitrary match, and it follows the [Failure] edge if the rule is inapplicable. A *foreach activity node* (denoted by a double framed rounded rectangle) applies the contained GT rule to *all* matches and traverses along the optional outgoing edge labeled with [EachTime] for each match. When all the matches have been completely processed, the control flow continues along the [End] outgoing edge. Black and green node and link variables are *bound* by a successful rule application. Subsequent activity nodes can reuse nodes and links that have been bound by an earlier rule application.

**Example.** Figure 5 depicts GT rules for each of the five possible topology change events – link weight change ($\mathsf{R}_{\mathsf{chg}}$), link addition ($\mathsf{R}_{\mathsf{addLink}}$), link removal ($\mathsf{R}_{\mathsf{remLink}}$), node addition ($\mathsf{R}_{\mathsf{addNode}}$), and node removal ($\mathsf{R}_{\mathsf{remNode}}$) – and the two TC rules, link activation ($\mathsf{R}_{\mathsf{act}}$) and link inactivation ($\mathsf{R}_{\mathsf{inact}}$). In Figure 5, $\mathsf{x}_1$ denotes the new weight $w(e)$ of link $e$. Modified and new links are marked as outdated.

Figure 6 shows a basic TC algorithm that activates all outdated links.[2] Link variable $\mathsf{e}_{\mathsf{O}}$ is bound by $\mathsf{R}_{\mathsf{loop}}$ and reused in $\mathsf{R}_{\mathsf{act}}$ and $\mathsf{R}_{\mathsf{inact}}$. Check marks inside the gray boxes indicate which constraints must be fulfilled in the input topology (precondition[3]) and which constraints will be fulfilled in the output topology (postcondition).

[2] The inactivation rule $\mathsf{R}_{\mathsf{inact}}$ is deliberately unreachable and only shown for completeness.
[3] The example in Section 3 contains a discussion why certain preconditions may be violated.

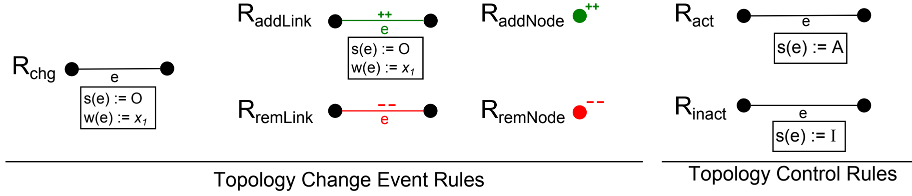



Fig. 5: GT rules describing topology change events and TC rules

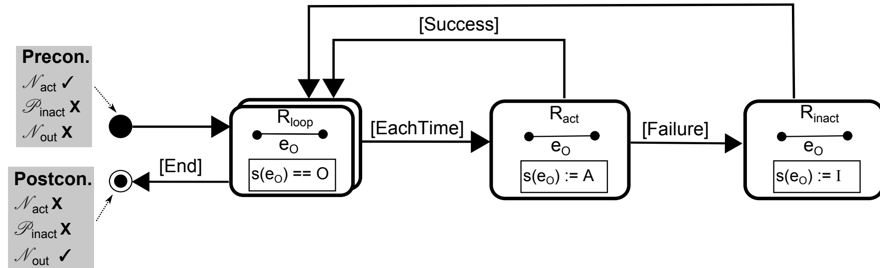



Fig. 6: Basic TC algorithm

## 5 Enriching the Graph Transformation with Constraints

The third step of our design methodology is to enrich graph transformation rules step-by-step with additional application conditions that are derived from the graph constraints. The resulting topology control algorithm always produces output topologies that fulfill all the graph constraints.

Our approach is demonstrated again on an incremental variant of kTC. Figure 7 serves as an overview, particularly during the subsequent sections. The final transformation is produced by iteratively refining the generic TC algorithm of Figure 6 according to the following procedure.

**Section 5.1** To ensure that the active-link constraint $\mathcal{N}_{\mathsf{act}}$ is still fulfilled in the output topology, a NAC derived from the active-link constraint $\mathcal{N}_{\mathsf{act}}$ is attached to the activation rule $\mathsf{R}_{\mathsf{act}}$ and the inactivation rule $\mathsf{R}_{\mathsf{inact}}$. This step produces the refined rules $\mathsf{R}^{(2)}_{\mathsf{act}}$ and $\mathsf{R}^{(2)}_{\mathsf{inact}}$.

**Section 5.2** To ensure that the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$ is fulfilled in the output topology, two steps are necessary: (i) The LHS pattern of the inactivation rule $\mathsf{R}^{(2)}_{\mathsf{inact}}$ is extended by the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$, resulting in the refined rule $\mathsf{R}^{(3)}_{\mathsf{inact}}$. (ii) A NAC derived from the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$ is attached to a new (preprocessing) rule $\mathsf{R}_{\mathsf{pre}}$.

**Section 5.3** To ensure that the outdated-link constraint $\mathcal{N}_{\mathsf{out}}$ is fulfilled in the output topology, the LHS pattern of a new (NAC match eliminating) rule $\mathsf{R}_{\mathsf{elimNAC}}$ is constructed from the common NACs of the activation and inactivation rules $\mathsf{R}^{(2)}_{\mathsf{act}}$ and $\mathsf{R}^{(3)}_{\mathsf{inact}}$.

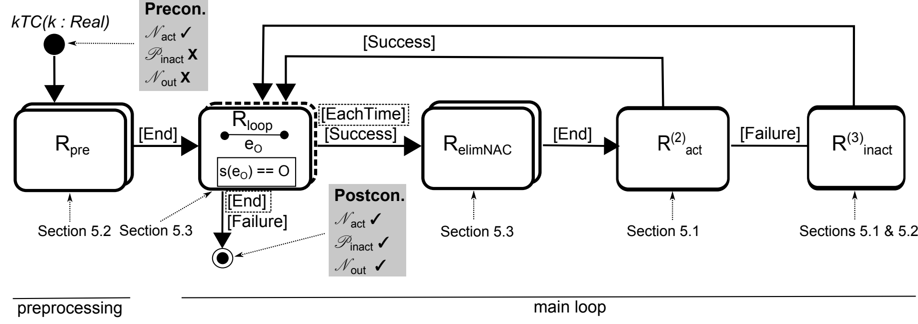



Fig. 7: kTC algorithm after performing steps of Sections 5.1 – 5.3

**Section 5.4** The connectivity of the output topology and the termination of the final version of kTC algorithm are proved.

### 5.1 Fulfilling the Active-Link Constraint

This first step ensures that the active-link constraint $\mathcal{N}_{\mathsf{act}}$ is fulfilled in the output topology. Since the input topology fulfills the constraint, it is enough to ensure that applying the activation and inactivation rules does not violate the active-link constraint $\mathcal{N}_{\mathsf{act}}$.

**Methodology.** The methodology to ensure that a GT rule fulfills a negative constraint is well-known in literature [7]: The negative constraint is translated into a number of NACs of the GT rules. Each overlap of the conclusion of the negative constraint with the RHS of the rule yields one NAC.

**Demonstration on kTC.** First, we consider the activation rule $\mathsf{R}_{\mathsf{act}}$: The conclusion of the active-link constraint $\mathcal{N}_{\mathsf{act}}$ and the RHS of the activation rule $\mathsf{R}_{\mathsf{act}}$ can be glued in three different ways such that the link $\mathsf{e}_{\mathsf{O}}$ overlaps with the link variables $\mathsf{e}_{\mathsf{max}}$, $\mathsf{e}_{\mathsf{s1}}$, or $\mathsf{e}_{\mathsf{s2}}$, respectively. This yields the three NACs $\mathsf{N}_{\mathsf{max}}$, $\mathsf{N}_{\mathsf{s1}}$, and $\mathsf{N}_{\mathsf{s2}}$. Note that the latter two NACs are *equivalent* in the sense that any match of $\mathsf{N}_{\mathsf{s1}}$ is also a match of $\mathsf{N}_{\mathsf{s2}}$, and vice versa. Figure 8 depicts the refined activation rule $\mathsf{R}^{(2)}_{\mathsf{act}}$.

Next, we consider the inactivation rule $\mathsf{R}_{\mathsf{inact}}$: The conclusion of the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$ and the RHS of the inactivation rule $\mathsf{R}_{\mathsf{inact}}$ can be glued in two ways such that the link $\mathsf{e}_{\mathsf{O}}$ overlaps the link variables $\mathsf{e}_{\mathsf{s1}}$ or $\mathsf{e}_{\mathsf{s2}}$, respectively. This yields two equivalent NACs, $\mathsf{N}_{\mathsf{s1}}$ and $\mathsf{N}_{\mathsf{s2}}$, which are isomorphic to the NACs of $\mathsf{R}^{(2)}_{\mathsf{act}}$ with the same name. Figure 9 depicts the refined inactivation rule $\mathsf{R}^{(2)}_{\mathsf{inact}}$.

Due to the added NACs, applying the refined activation rule $\mathsf{R}^{(2)}_{\mathsf{act}}$ and the refined inactivation rule $\mathsf{R}^{(2)}_{\mathsf{inact}}$ never violates the active-link constraint $\mathcal{N}_{\mathsf{act}}$.

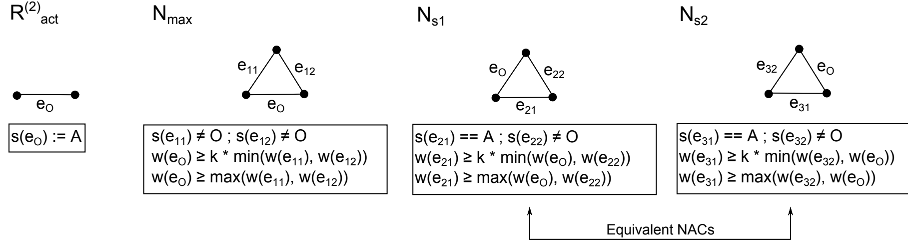



Fig. 8: Activation rule $\mathsf{R}^{(2)}_{\mathsf{act}}$ preserving active-link constraint $\mathcal{N}_{\mathsf{act}}$

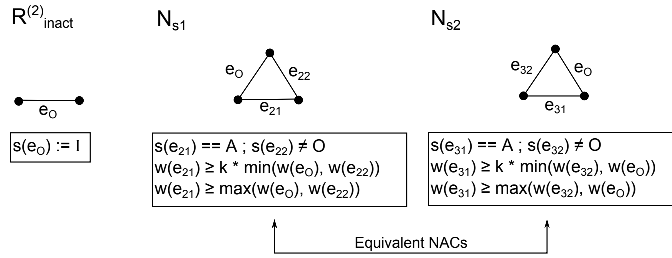



Fig. 9: Inactivation rule $\mathsf{R}^{(2)}_{\mathsf{inact}}$ preserving active-link constraint $\mathcal{N}_{\mathsf{act}}$

### 5.2 Fulfilling the Inactive-Link Constraint

To ensure that the positive inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$ is fulfilled in the output topology, two modifications are necessary: First, an additional preprocessing rule $\mathsf{R}_{\mathsf{pre}}$ ensures that any violation of $\mathcal{P}_{\mathsf{inact}}$ in the input topology is repaired. Second, a refinement of the inactivation rule $\mathsf{R}^{(2)}_{\mathsf{inact}}$ ensures that the constraint is never violated in the main loop.

**Methodology.** Due to the page limitation, we present an adaptation of the methodology described in [7], which is tailored to our scenario: We consider GT rules without deletion that modify an attribute of a single link.

If a topology fulfills a positive graph constraint before applying a rule, two steps are necessary to ensure that the topology still fulfills the constraint after the rule application: (i) If applying the rule produces a new match of the premise of the constraint, then the RHS of the rule needs to ensure that the conclusion of the constraint holds. (ii) Applying the rule may not violate the constraint by destroying any match of the conclusion of the constraint that existed prior to the rule application.

This means that, first, the LHS pattern of each rule is extended with the conclusion of the constraint if its RHS overlaps the premise of the constraint. Second, it is analyzed whether the modified rule may destroy any existing match of the conclusion of the constraint.

**Demonstration on kTC.** The input topology may violate the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$, which necessitates a new preprocessing rule $\mathsf{R}_{\mathsf{pre}}$, which outdates all inactive links that violate the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$. The NAC $\mathsf{N}_{\mathsf{pre}}$ of rule $\mathsf{R}_{\mathsf{pre}}$, shown in Figure 10, matches exactly those links that violate $\mathcal{P}_{\mathsf{inact}}$.

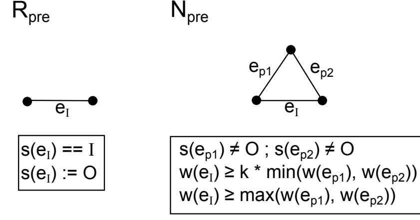



Fig. 10: Preprocessing rule $\mathsf{R}_{\mathsf{pre}}$, which outdates all inactive links violating $\mathcal{P}_{\mathsf{inact}}$

Next, we show that the activation rule $\mathsf{R}^{(2)}_{\mathsf{act}}$ remains unchanged: The RHS of rule $\mathsf{R}^{(2)}_{\mathsf{act}}$ does not overlap with the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$. Additionally, applying the rule does not violate the constraint because it activates a link, which obviously cannot destroy any match of the conclusion of $\mathcal{P}_{\mathsf{inact}}$.

Finally, we describe the refinement of the inactivation rule $\mathsf{R}^{(2)}_{\mathsf{inact}}$: The link variable $\mathsf{e}_{\mathsf{max}}$ in the premise of the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$ and the link $\mathsf{e}_{\mathsf{O}}$ in the RHS of $\mathsf{R}^{(2)}_{\mathsf{inact}}$ overlap. Therefore, the LHS of $\mathsf{R}^{(2)}_{\mathsf{inact}}$ is extended by an image of the conclusion of the constraint. The refined inactivation rule $\mathsf{R}^{(3)}_{\mathsf{inact}}$ is depicted in Figure 11.

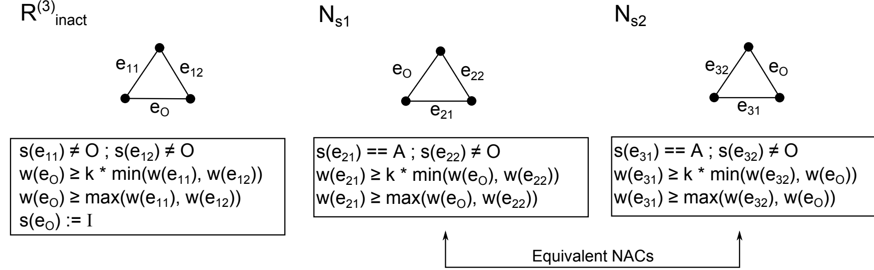



Fig. 11: Inactivation rule $\mathsf{R}^{(3)}_{\mathsf{inact}}$ preserving $\mathcal{N}_{\mathsf{act}}$ and $\mathcal{P}_{\mathsf{inact}}$

Due to the additional preprocessing rule $\mathsf{R}_{\mathsf{pre}}$, which ensures that the topology fulfills the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$ at the beginning of the main loop, and the refined inactivation rule $\mathsf{R}^{(3)}_{\mathsf{inact}}$, the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$ is fulfilled in the output topology as well.

### 5.3 Fulfilling the Outdated-Link Constraint

To ensure that the outdated-link constraint $\mathcal{N}_{\mathsf{out}}$ is fulfilled in the output topology, a new GT rule is added in this section because the activation rule $\mathsf{R}_{\mathsf{act}}^{(2)}$ and inactivation rule $\mathsf{R}_{\mathsf{inact}}^{(3)}$ share the NACs $\mathsf{N}_{\mathsf{s1}}$ and $\mathsf{N}_{\mathsf{s2}}$, which may block the activation *and* inactivation of outdated links in some topologies. Due to the equivalence of the NACs $\mathsf{N}_{\mathsf{s1}}$ and $\mathsf{N}_{\mathsf{s2}}$, we only consider $\mathsf{N}_{\mathsf{s1}}$ in the following.

**NAC-elimination rule.** We propose to insert a new rule $\mathsf{R}_{\mathsf{elimNAC}}$, depicted in Figure 12, prior to the activation rule $\mathsf{R}_{\mathsf{act}}^{(2)}$, which removes all matches of NAC $\mathsf{N}_{\mathsf{s1}}$. The LHS of rule $\mathsf{R}_{\mathsf{elimNAC}}$ is an image of $\mathsf{N}_{\mathsf{s1}}$, and the RHS outdates the longest link $\mathsf{e}_{\mathsf{max}}$. Note that the outdated state propagates only toward longer links.

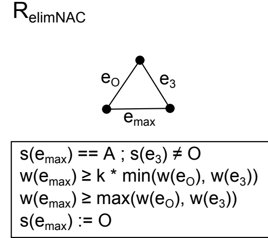



Fig. 12: NAC elimination rule $\mathsf{R}_{\mathsf{elimNAC}}$

**Loop rule.** The new NAC elimination rule $\mathsf{R}_{\mathsf{elimNAC}}$ results in additional outdated links that are not considered when $\mathsf{R}_{\mathsf{loop}}$ is first applied. Consequently, the foreach activity node around loop rule $\mathsf{R}_{\mathsf{loop}}$ changes to a regular activity node in this step, as shown earlier in Figure 7. For this reason the algorithm terminates if and only if the topology contains no more outdated links. Consequently, the output topology fulfills the outdated-link constraint $\mathcal{N}_{\mathsf{out}}$.

### 5.4 Proofs of Termination and Connectivity

The rule refinements and additions in Sections 5.1, 5.2, and 5.3 ensure that the active-link constraint $\mathcal{N}_{\mathsf{act}}$, the inactive-link constraint $\mathcal{P}_{\mathsf{inact}}$, and the outdated-link constraint $\mathcal{N}_{\mathsf{out}}$ are fulfilled in the output topology. We still have to show that the algorithm in Figure 7 terminates and that the output topology is connected.

**Theorem 1 (Termination).** *The algorithm terminates for any input topology.*

*Proof.* Consider a topology with link set $E$. The preprocessing loop $\mathsf{R}_{\mathsf{pre}}$ is executed at most once for each link, so it suffices to show that the main loop terminates.

We consider the sequence of all link states $s_i(e_1), \ldots, s_i(e_m)$ with $m := |E|$ after the $i$-th execution of $\mathsf{R}_{\mathsf{loop}}$, where the links $e_k$ are ordered according to their

weight. We compare two sequences of link states, $s_i$ and $s_j$, as follows: $s_i \prec s_j$ if and only if (i) some link $e_k$ is outdated in $s_i$ and active or inactive in $s_j$, and (ii) the states of all links shorter than $e_k$ are identical in $s_i$ and $s_j$, formally:

$$s_i \prec s_j :\Leftrightarrow \exists k, 1 \leq k \leq m : s_i(e_k) = \mathtt{O} \wedge s_j(e_k) \in \{\mathtt{A}, \mathtt{I}\}$$
$$\wedge \forall \ell, 1 \leq \ell \leq k-1 : s_i(e_\ell) = s_j(e_\ell)$$

Note that any sequence of active and inactive links is an upper bound for $\prec$.

We now show that $s_{i-1} \prec s_i$ for $i > 1$. Let $e_k$ be the link that is bound by applying the loop rule $\mathsf{R}_{\mathsf{loop}}$. The NAC elimination rule $\mathsf{R}_{\mathsf{elimNAC}}$ outdates links $e_j$ with $w(e_j) > w(e_k)$ and thus $j > k$. The activation rule $\mathsf{R}^{(2)}_{\mathsf{act}}$ or the inactivation rule $\mathsf{R}^{(3)}_{\mathsf{inact}}$ activate or inactivate $e_k$, respectively.

Therefore, $s_{i-1} \prec s_i$ because (i) the first $k{-}1$ elements of $s_{i-1}$ and $s_i$ are identical, and (ii) $s_{i-1}(e_k) = \mathtt{O}$ and $s_i(e_k) \in \{\mathtt{A}, \mathtt{I}\}$. The termination follows because any ordered sequence $s_1 \prec s_2 \prec \ldots$ has finite length.

**Theorem 2 (Connectivity).** *The output topology of the algorithm is connected if its input topology is connected.*

*Sketch of Proof.* The output topology only contains active and inactive links because the outdated-link constraint $\mathcal{N}_{\mathsf{out}}$ is fulfilled. It is thus enough to show the claim that the end nodes of each link are connected by a path of active links in the output topology. This trivially holds for the end nodes of active links.

By induction, we show that the claim also holds for all inactive links: We consider the inactive links $e_{i_1}, \ldots, e_{i_k}$ of the topology ordered by weight.

*Induction start:* The shortest inactive link, $e_{i_1}$, is part of a triangle with two shorter, active links that connect the end nodes of link $e_{i_1}$. Thus, the claim holds for link $e_{i_1}$.

*Induction step:* We now consider an inactive link $e_{i_{\ell+1}}$ with $1 \leq \ell \leq k{-}1$, which is part of a triangle with two links, $e_1$ and $e_2$. We assume that only $e_1$ is inactive.[4] Thus, there is some $s \leq \ell$ such that $e_1 := e_{i_s}$. Since the claim has been proved for all inactive links shorter than $e_{i_{\ell+1}}$, there is a path of active links between the end nodes of $e_{i_s}$. A path of active links between the end nodes of link $e_{\ell+1}$ can be constructed by joining the two paths between the end nodes of $e_1$ and $e_2$.

## 6 Related Work

We briefly present related work on verification and model-based development.

**Verification.** Model checking [12] is an analysis technique used to verify particular properties of a system. If a symbolic problem description is missing, model checking tools are often limited to a finite model size. The approach in this paper constructively integrates constraints at design time so that it can be shown that constraints are fulfilled on arbitrary topologies.

[4] If both links are active, the claim follows trivially. If both links are inactive, the argument applies for each link individually.

In [7], graphical consistency constraints, which express that particular combinations of nodes and edges should be present in or absent from a graph, are translated into application conditions of GT rules. This technique has been generalized later [3] and extended to cope with attributes [2]. The basic idea is to translate consistency conditions, characterizing "valid" graphs, into application conditions of GT rules. This paper applies and extends this generic methodology for a practical and complex application scenario. We represent positive application conditions in [7] as extensions of the LHS of GT rules, which is equivalently expressive [5]. This representation is unsuitable to express global constraints such as connectivity, which requires, e.g., second-order monadic logic [7]. This paper ensures connectivity of topologies by an additional proof.

In [6], the authors distinguish four situations in which a model transformation considers consistency conditions, including the preservation and enforcement of consistency constraints. The algorithm in this paper preserves the active-link constraint, and it enforces and preserves the inactive-link constraint.

**Model-based development.** Model-based techniques have shown to be suitable to describe [16] and construct [8] adaptive systems. Formal analysis of supposed properties of complex topology adaptation algorithms has already revealed special cases in which the implemented algorithms violate crucial topology constraints [18]. In [9], model checking is applied to detect bugs and to point at their causes in the TC algorithm LMST, leading to an improved implementation thereof. This paper, in contrast, applies a constructive methodology [7] for GT to develop correct algorithms in the first place.

In [11], variants of the TC algorithm kTC [15] are developed using GT, integrating the GT tool eMoflon[5] with a network simulator. While [11] focuses on improving a concrete algorithm, this paper aims at devising a generic methodology to develop TC algorithms that fulfill the given constraints by design.

## 7 Conclusion

In this paper, we proposed a new, model-driven methodology for designing topology control algorithms by graph transformation, and demonstrated the approach on an incremental variant of the kTC algorithm. The presented procedure characterizes valid topologies with graph constraints, specifies topology control algorithms as graph transformation system, and applies a well-known static analysis technique to enrich graph transformation rules with application conditions derived from the graph constraints. The new algorithm always terminates and produces connected, valid topologies.

Future research includes interleaving the network evolution with topology control and evaluating the methodology on further topology control algorithms.

**Acknowledgment.** This work has been funded by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 – MAKI. The authors would like to thank Matthias Hollick (subprojects A03 and C01) for his valuable input.

[5] www.emoflon.org

## References


[1] Beydeda, S., Book, M., Gruhn, V.: Model-driven software development. Springer, 15th edn. (2005)

[2] Deckwerth, F., Varró, G.: Generating Preconditions from Graph Constraints by Higher Order Graph Transformation. ECEASST 67, 1–14 (2014)

[3] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)

[4] Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the Unified Modeling Language. In: Proc. of the International Workshop on Theory and Application of Graph Transformation. pp. 296–309. Springer (1998)

[5] Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae 26(3/4), 287–313 (1996)

[6] Hausmann, J.H., Heckel, R., Sauer, S.: Extended model relations with graphical consistency conditions. In: Proc. of UML 2002 Workshop on Consistency Problems in UML-based Software Development. pp. 61–74 (2002)

[7] Heckel, R., Wagner, A.: Ensuring consistency of conditional graph rewriting – A constructive approach. In: Proc. of Joint COMPUGRAPH/SEMAGRAPH Workshop. ENTCS, vol. 2, pp. 118–126. Elsevier (1995)

[8] Jacob, R., Richa, A., Scheideler, C., Schmid, S., Täubig, H.: A Distributed Polylogarithmic Time Algorithm for Self-stabilizing Skip Graphs. In: Proc. of the ACM Symposium on Principles of Distributed Computing. pp. 131–140. ACM (2009)

[9] Katelman, M., Meseguer, J., Hou, J.: Redesign of the LMST Wireless Sensor Protocol through Formal Modeling and Statistical Model Checking. In: Formal Methods for Open Object-Based Distributed Systems, LNCS, vol. 5051, pp. 150–169. Springer (2008)

[10] Koch, M., Mancini, L.V., Parisi-Presicce, F.: A graph-based formalism for RBAC. ACM Trans. Inf. Syst. Secur. 5(3), 332–365 (2002)

[11] Kulcsár, G., Stein, M., Schweizer, I., Varró, G., Mühlhäuser, M., Schürr, A.: Rapid prototyping of topology control algorithms by graph transformation. In: Proc. of the 8th Int. Workshop on Graph-Based Tools. ECEASST, vol. 68 (2014)

[12] Rensink, A., Schmidt, Á., Varró, D.: Model checking graph transformations: A comparison of two approaches. In: Proc. of the 2nd International Conference on Graph Transformations, LNCS, vol. 3256, pp. 226–241. Springer (2004)

[13] Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1: Foundations. World Scientific (1997)

[14] Santi, P.: Topology control in wireless ad hoc and sensor networks. ACM computing surveys (CSUR) 37(2), 164–194 (2005)

[15] Schweizer, I., Wagner, M., Bradler, D., Mühlhäuser, M., Strufe, T.: kTC – Robust and adaptive wireless ad-hoc topology control. In: Proc. of the 21st International Conference on Computer Communications and Networks (2012)

[16] Taentzer, G., Goedicke, M., Meyer, T.: Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In: Theory and Application of Graph Transformations, pp. 179–193. Springer (2000)

[17] Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons (2013)

[18] Zave, P.: Using lightweight modeling to understand chord. SIGCOMM Comput. Commun. Rev. 42(2), 49–57 (2012)