# Toward model-based correct-by-construction development of distributed topology control algorithms (Appendix)

Roland Kluge
roland.kluge@es.tu-darmstadt.de
Real-Time Systems Lab, TU Darmstadt
Darmstadt, Germany

Marcel Blöcher
bloecher@dsp.tu-darmstadt.de
Distributed Systems Programming Group, TU Darmstadt
Darmstadt, Germany

Andy Schürr
andy.schuerr@es.tu-darmstadt.de
Real-Time Systems Lab, TU Darmstadt
Darmstadt, Germany

## ABSTRACT

Wireless Sensor Networks consist of small, distributed, battery-powered sensor devices and constitute a crucial component of the emerging Internet of Things. A topology control algorithm increases the lifetime of a Wireless Sensor Network by inactivating redundant energy-inefficient links. Safety-critical applications of Wireless Sensor Networks demand that a topology control algorithm fulfills certain safety and liveness properties. We previously proposed a methodology to constructively ensure safety properties based on a centralized specification using graph transformation. In this paper, we present a complementary iterative, model-based approach for verifying and restoring liveness properties. We employ a vertex-centric, interleaved computation model based on per-node local views to characterize liveness issues such as livelocks and starvation. Using state-of-the-art techniques for identifying conflicts and dependencies among graph transformation rules, we show how to identify liveness-threatening interaction and conflict sequences. We show how to eliminate liveness-threatening conflicts and dependencies by systematically introducing critical regions. We illustrate our approach using the class of triangle-based topology control algorithms.

## CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; • **Networks** → *Formal specifications*; *Logical / virtual topologies*; • **Hardware** → *Wireless devices*;

## KEYWORDS

Distributed algorithm, Model-driven engineering, Wireless sensor network , Conflict and dependency analysis, Safety and liveness

**ACM Reference format:**
Roland Kluge, Marcel Blöcher, and Andy Schürr. 2018. Toward model-based correct-by-construction development of distributed topology control algorithms (Appendix). In *Proceedings of International Conference on Model Driven Engineering Languages and Systems, Copenhagen, Denmark, October 2018 (MODELS'18)*, 13 pages.
https://doi.org/10.475/123_4

## 1 INTRODUCTION

This document is an appendix to the paper "Toward model-based correct-by-construction development of distributed topology control algorithms".

A *wireless sensor network* (WSN) is a communication system that consists of battery-powered sensor nodes, which communicate via wireless links and adjust their transmission ranges to connect to relevant neighbor nodes. Within the emerging Internet of Things [41], WSNs are important for safety-critical application scenarios (e.g., structural health monitoring [20]). The *topology* of a WSN is a graph that represents the devices and their interconnecting links as nodes and weighted edges, respectively. A *topology control (TC) algorithm* identifies all links in the topology that are costly w.r.t. energy consumption and, at the same time, inessential for the purpose of the WSN (e.g., forwarding data to a central sink node); all other links are marked as essential. Based on this classification, each sensor node reduces its transmission range to be able to use all essential links while ignoring inessential ones. A TC algorithm must certain fulfill safety and liveness properties. An important safety property is that the essential links form a connected graph if the underlying network is connected. Liveness properties comprise deadlock, livelock, and starvation freedom [38].

Traditionally, a TC algorithm is developed in two major phases. In the specification phase, safety properties of the TC algorithm are proved based on a formal model. In the implementation phase, the TC algorithm is implemented manually and evaluated inside a network simulator and/or using a real-world testbed. Liveness properties are usually not modeled but only tested for during the implementation phase. The manual effort during the latter step is error prone, time consuming, and, actually, unnecessary, given that Model-Driven Engineering provides formal and technical frameworks to specify and analyze TC algorithms. For instance, tools for static analysis [5, 36] and configurable code generation [21, 39] are state of the art.

Previous work shows that safety properties of centralized TC algorithms can be specified conveniently using graph constraints [22, 23]. To this end, a topology is modeled as attributed graph and a TC algorithm is specified using programmed graph transformation (GT) [10, 12]. The GT-based specification can then be refined mechanically such that it preserves the specified constraints [19]. The TC algorithm specification serves as input for generating platform-specific code for simulation and testbed [21, 23].

A centralized-global perspective prove correctness of the TC algorithms w.r.t. safety properties (Figure 1a). However, this perspective is only suitable for preliminary simulation experiments that access the topology via a global-knowledge oracle. For testbed

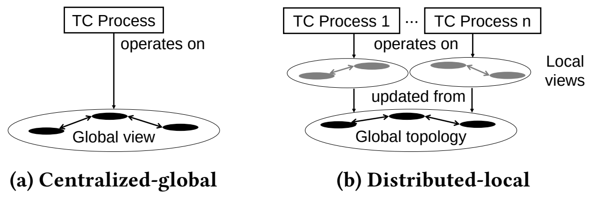


**Figure 1: Execution types of a TC algorithm**

and advanced simulation experiments, no such oracle exists. Instead, each node runs a node-local instance of the TC algorithm based on a local view of the topology (Figure 1b). The local view contains only nodes and links that are, e.g., at most two hops away, where a *hop* is the traversal of a link. These restrictions apply for the following reasons: First, a centralized execution incurs prohibitive communication overhead due to the required coordination messages. Second, storing the global topology exceeds the limited working memory of a sensor node, entails additional protocol messages to exchange the local views of the topology across the entire network, and leads to an ever-outdated global view because the topology is dynamic, i.e., links disappear and reappear regularly, e.g., due to moving obstacles. The required distributed-local perspective gives rise to potential liveness issues, which should be identified statically during the specification phase to avoid the repeated time-consuming programming of the testbed devices during the implementation phase. Existing formalisms for identifying liveness issues in distributed systems (e.g., CCS [29] or Petri nets [14]) become highly challenging when it comes to analyzing systems with dynamic structures, such as WSNs.

In this paper, *we propose a systematic model-based approach to designing correct-by-construction distributed TC algorithms.* Our approach constructively integrates safety properties (w.r.t. local views) and allows to iteratively ensure liveness properties. Only the per-node happened-before relationship given by the programmed GT specification restricts the execution order of topology modifications. Therefore, our approach does not require clock synchronization across nodes. We introduce background information on correct centralized TC algorithms in Section 2. Then, Section 3 presents the major contributions of this paper.

- We characterize distributed TC algorithms using a vertex-centric, interleaved computation model based on a per-node local view of the topology and characterize liveness issues such as livelocks and starvation (Section 3.1).
- We introduce a systematic, iterative approach for identifying liveness issues using conflict and dependency analysis (Sections 3.2 and 3.3).
- We propose a refinement strategy to eliminate liveness-preventing conflict and dependency sequences by selectively inserting guarded regions for critical links (Section 3.4).

Section 4 surveys related work, and Section 5 concludes this paper. Triangle-based TC algorithms [23] serve as running example[1].

[1]We provide supplementary material at https://github.com/Echtzeitsysteme/models2018-distributedtc.

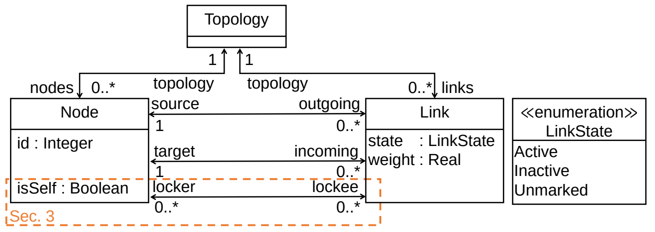


**Figure 2: Topology metamodel**

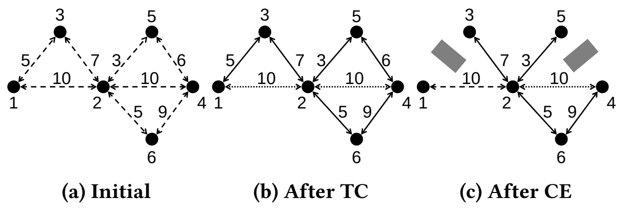


**Figure 3: Sample topologies (CE: context event, ■: obstacle)**

## 2 CENTRALIZED TOPOLOGY CONTROL

In this section, we present background on modeling correct centralized TC algorithms using metamodeling for specifying domain concepts, graph constraints for specifying safety properties, and (programmed) graph transformation for specifying (the order of) topology modifications [22, 23].

**Metamodeling topologies:** We employ metamodeling [40] to specify WSN domain concepts. In this paper, a metamodel consists of classes (shown as rectangular boxes) and associations between classes (shown as directed edges). A class may contain zero or more typed attributes, and each end of an association is labeled with the role and multiplicity of the association.

**Example:** Figure 2 shows the metamodel of our running example. Elements inside the dashed box are introduced during Section 3. A Topology contains zero or more nodes and links. A Node $n_1$ has an integer-valued id attribute, a Boolean isSelf attribute, and zero or more incoming and outgoing incident links. A Link $\ell_{12}$ has a source node $n_1$, a target node $n_2$, a real-valued weight $w(\ell_{12})$ (abbreviated with $w_{12}$), and an enumeration type state attribute. The enumeration type LinkState captures that a Link $\ell_{12}$ may be in one of three states $s(\ell_{12})$: If $\ell_{12}$ is Active ($s(\ell_{12}) = A$), it must remain available after reducing the transmission power. If $\ell_{12}$ is Inactive ($s(\ell_{12}) = I$), it should not be used to transmit messages. If $\ell_{12}$ is Unmarked ($s(\ell_{12}) = U$), it needs to be processed by the TC algorithm. Additionally, a link is locked by zero or more nodes (locker), and a node locks zero or more links (lockee).

Figure 3 shows three sample topologies of the running example. A node is depicted as circle with its identifier as label (e.g., ●1). A link is depicted as arrow-headed line whose style indicates the link state. Active, inactive, and unmarked links are depicted with solid black (1●⟶●2), dotted black (1●⋯>●2), and dashed arrows (1●- ->●2). Solid gray arrows represent links of unknown state (1●⟶●2). In this paper, we assume that, for each link $\ell_{12}$, the link $\ell_{21}$ exists.

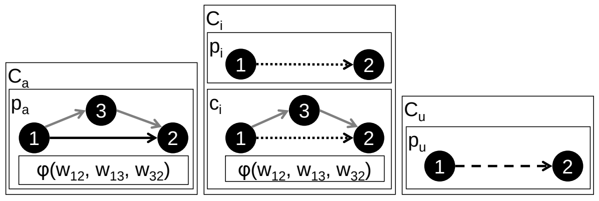
**TC processes:** Traditionally, a TC algorithm processes the *entire* topology whenever the input topology changes due to context events. In this paper, we consider three types of *context events*: A link may be removed (e.g., if an obstacle moves between its incident nodes), a link may be added (e.g., if the obstacle disappears), or the weight of a link may change (e.g., if its incident nodes move). The effect of a context event is typically small compared to the size of the topology. To take advantage of this observation, our goal is to support the development of *TC processes*, which *react* to context events, and, therefore, consist of two types of components [23]: (i) The *batch TC algorithm* accepts a partially marked input topology and returns a fully marked output topology. (ii) For each type of context event, a *context event handler* updates the topology incrementally to reflect the effect of the context event.

A TC process must fulfill *safety properties*, which specify when a link may or should be active or inactive. To support marking the topology in an incremental way, we define two levels of safety properties. A topology is *weakly consistent* if it contains no falsely marked links according to the specified safety properties. A topology is *strongly consistent* if it is weakly consistent and contains no unmarked links. For example, a fully unmarked topology is weakly consistent (Figure 3a). A *correct TC algorithm* eventually transforms a weakly consistent input topology into a strongly consistent output topology. A *correct context event handler* preserves weak consistency and successfully performs the corresponding context event (e.g., removes the given link). A *TC process is correct* if its TC algorithm and context event handlers are correct.

In contrast to a batch algorithm, a TC process never terminates. The analogous concept to termination for processes is called *stabilization* [2]. A *topology has stabilized* if it is strongly consistent and no context events are pending.

**Example:** The running example of this paper is the class of *triangle-based TC algorithms*. Such algorithms inactivate links based on finding $\varphi$-triangles, i.e., triangles $(\ell_{12}, \ell_{13}, \ell_{32})$ in the input topology that fulfill a predicate $\varphi(\mathrm{w}_{12}, \mathrm{w}_{13}, \mathrm{w}_{13})$. For example, the triangle-based TC algorithm *kTC* [33] inactivates a link if and only if it is the weight-maximal link inside a triangle and if the weight of this link is at least k times larger than the weight of the weight-minimal link in the same triangle. The parameter k controls the aggressiveness of the link inactivation. Therefore, for kTC, $\varphi(\mathrm{w}_{12}, \mathrm{w}_{13}, \mathrm{w}_{13}) \Leftrightarrow \mathrm{w}_{12} > \max(\mathrm{w}_{13}, \mathrm{w}_{12}) \wedge \mathrm{w}_{12} > \mathrm{k} \cdot \min(\mathrm{w}_{13}, \mathrm{w}_{12})$. Figures 3a and 3b show an input-output pair of kTC topologies for $\mathrm{k} = 1.5$. Figure 3c shows the topology that results from two obstacles (see ■) causing link removal events for $\ell_{13}$, $\ell_{31}$, $\ell_{45}$, and $\ell_{54}$. The context event handler determines that $\ell_{12}$ and $\ell_{21}$ may no longer be inactive because the context event destroyed the only related $\varphi$-triangles. In contrast, $\ell_{24}$ and $\ell_{42}$ remain inactive because each link is still part of a $\varphi$-triangle. In a subsequent iteration of the TC algorithm, $\ell_{12}$ and $\ell_{21}$ will be activated.

**Patterns and constraints:** We use graph patterns [10, 19, 32] to characterize (un-)desired topologies. A *pattern* consists of (i) a *pattern graph*, whose nodes and edges are called *node variables* and *link variables* and serve as placeholders for nodes and links, respectively, and (ii) a set of relational *attribute constraints* over the attributes of the nodes and links represented by the variables. A *match m of a pattern p in a topology G* is an injective mapping from the node and link variables of $p$ to the nodes and links of G that preserves source and target nodes. This means that the source and target node variables of a link variable must be mapped to the source and target nodes of the image of the link variable. Additionally, the attribute constraints of $p$ must be fulfilled when replacing the node and link variables with the matched nodes and links.



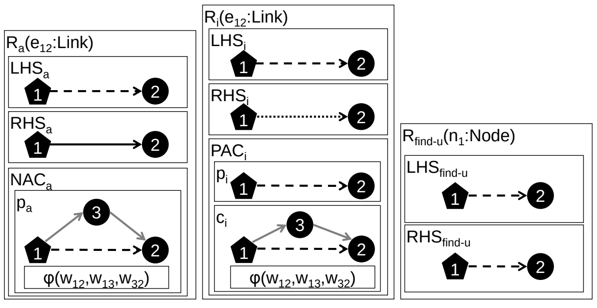
**(a) $\mathbf{C_a}$** **(b) $\mathbf{C_i}$** **(c) $\mathbf{C_u}$**

**Figure 4: Consistency constraints**

A pattern $p_2$ is an *extension of a pattern* $p_1$ if the variable graph of $p_1$ is a subgraph of $p_2$. A match $\mathrm{m}_2$ is an *extension of a match* $m_1$ if each mapping of $\mathrm{m}_1$ is also contained in $\mathrm{m}_2$. A *graph constraint* consists of one *premise pattern* and zero or more *conclusion patterns*, which all extend the premise pattern. A *negative graph constraint* has no conclusion pattern, whereas a *positive graph constraint* has at least one conclusion pattern. A *topology fulfills a graph constraint* if each match of the premise pattern can be extended to a match of at least one conclusion pattern. Therefore, a topology fulfills a negative graph constraint if it does not contain any match of the premise pattern of the constraint.

**Example:** Figure 4 shows the three graph constraints that characterize consistent topologies of triangle-based TC algorithms. A node variable is depicted as solid black circles containing its label, and a link variable is depicted as arrow-headed lines. We use e as symbol for link variables to distinguish them from links. Attribute constraints are shown in a framed box below the pattern graph. The pattern graphs of $\mathrm{p_a}$ and $\mathrm{c_i}$ represent the structural pattern, and the ternary predicate $\varphi$ represents the attribute constraints of triangle-based TC algorithms. Graph constraints allow us to formalize safety properties: A *weakly consistent topology* fulfills $\mathrm{C_a}$ and $\mathrm{C_i}$. A *strongly consistent topology* fulfills $\mathrm{C_a}$, $\mathrm{C_i}$, and $\mathrm{C_u}$. Figure 3a shows a weakly and Figure 3b shows a strongly consistent topology.

**Topology modifications:** We specify modifications of a topology using graph transformation rules. A *graph transformation (GT) rule* consists of a left-hand side (LHS) pattern, a right-hand side (RHS) pattern, and a set of positive or negative application conditions (PACs, NACs). Attribute constraints in the RHS pattern are attribute assignments (i.e., equality constraints with one left-hand side argument). An application condition is a constraint whose premise extends the LHS pattern. A GT rule is *applicable at a match of its LHS* if any extension of the match to a match of a PAC premise can be further extended to a match of a conclusion pattern of the same PAC and the match cannot be extended to any match of a NAC premise. A *GT rule is applied at a match* by (i) *removing* all node and links that are matched by variables that only appear in the LHS pattern, (ii) *creating* fresh nodes and links for all variables that only appear in the RHS pattern, and (iii) *assigning* attribute values according to the RHS attribute constraints. All other nodes and links are *preserved*.



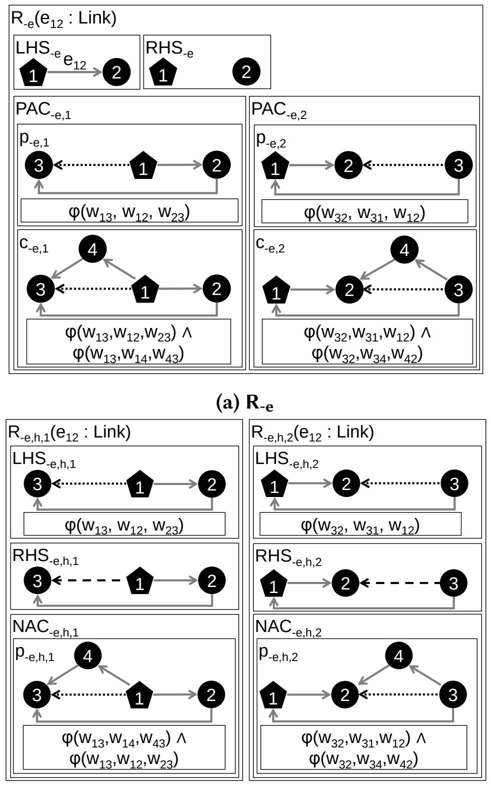
(a) $\mathbf{R_a}$ (b) $\mathbf{R_i}$ (c) $\mathbf{R_{find\text{-}u}}$

**Figure 5: TC rules**

We distinguish three types of rules: (i) A *context event rule* specifies an elementary modification of the topology that is caused by the environment (e.g., the removal of a link) and may violate weak consistency. (ii) A *context event handler rule* of a particular context event rule unmarks a link that would otherwise cause a violation of weak consistency. (iii) Finally, a *TC rule* specifies an atomic action for processing unmarked links inside a TC algorithm.

**Example:** Figure 5 shows the TC rules of the running example. Until Section 3, pentagon-shaped (e.g., ⬟) and circle-shaped (e.g., ❶) node variable are equivalent. The *activation rule* $R_a$ activates a given link $e_{12}$ if it is not the weight-maximal link in a $\varphi$-triangle (see $NAC_a$). The *inactivation rule* $R_i$ inactivates a given link $e_{12}$ if is the weight-maximal link in a $\varphi$-triangle (see $PAC_i$). The *unmarked-link-identification rule* $R_{find\text{-}u}$ determines whether a node $n_1$ exists that has at least one unmarked outgoing link. This rule does not alter the topology because LHS and RHS are equal. Figure 6 shows the *link removal rule* $R_{-e}$ and the corresponding context event handler rules $R_{-e,h,1}$, $R_{-e,h,2}$. The rule $R_{-e}$ may only be applied if the rule application does not destroy the only extension of a match of $p_i$ to a match of $c_i$. The two PACs correspond to situations were $e_{12}$ maps to $e_{13}$ and $e_{32}$ of $C_i$. The rule $R_{-e,h,1}$ prevents that the inactive-link constraint $C_i$ is violated by unmarking a link $e_{13}$ if removing the link $e_{12}$ would destroy the only match of the conclusion of $C_i$ that corresponds to $e_{13}$. The rule $R_{-e,h,2}$ prevents a similar situation where the to-be-removed $e_{12}$ corresponds to $e_{32}$ in $C_i$. Figure 7 shows the link addition rule $R_{+e}$ and the corresponding context event handlers $R_{+e,h,1}$, $R_{+e,h,2}$. Figure 8 shows the link weight modification rule $R_{mod\text{-}w}$ and the corresponding context event handlers $R_{mod\text{-}w,h,1}$, $R_{mod\text{-}w,h,2}$, $R_{mod\text{-}w,h,3}$, $R_{mod\text{-}w,h,4}$.

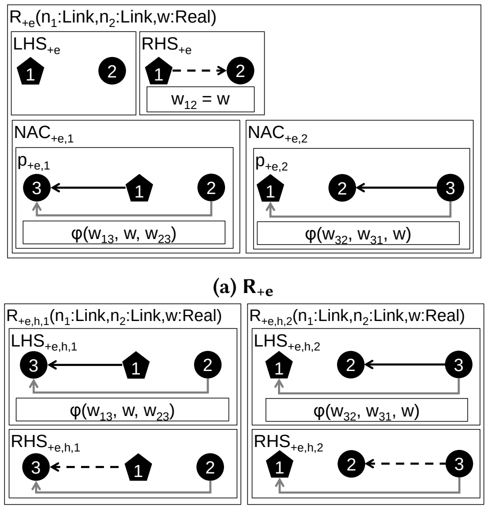
**Control flow:** We specify the order topology modifications using Story-Driven Modeling (SDM) [12], a dialect of programmed GT that borrows from UML activity diagrams. A *story diagram* is a graph that specifies the control flow of an operation and consists of *activity nodes* and *activity edges*. An activity edge may be labeled with a success ([S]) or failure guard ([F]). An activity node can be of one of four types: start node, stop nodes, story node, or operation node. The unique *start node* of a story diagram has exactly one outgoing, unguarded activity edge. Each of the zero or more *stop nodes* has at least one incoming activity edge and no outgoing activity edges. A *story node* contains a GT rule application, and an *operation node* contains an invocation of a story diagram. A story or



(a) $\mathbf{R_{-e}}$

(b) $\mathbf{R_{-e,h,1}}$ (c) $\mathbf{R_{-e,h,2}}$

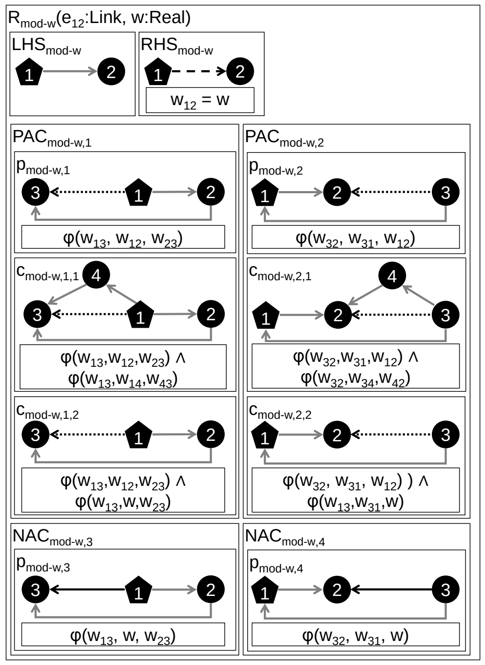
**Figure 6: Link removal rule with handlers**



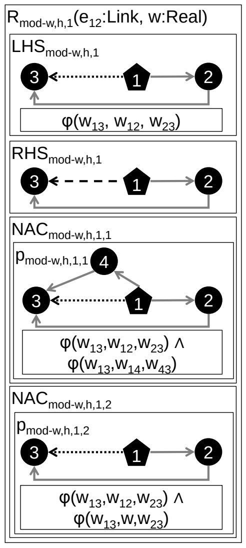
(a) $\mathbf{R_{+e}}$

(b) $\mathbf{R_{+e,h,1}}$ (c) $\mathbf{R_{+e,h,2}}$

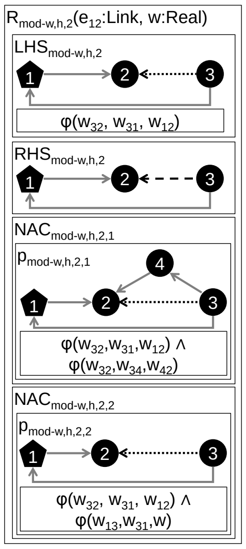
**Figure 7: Link addition rule with handlers**



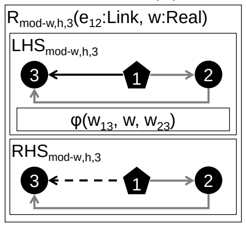
(a) $\mathbf{R_{mod\text{-}w}}$



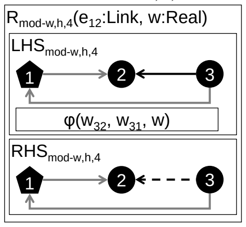
(b) $\mathbf{R_{mod\text{-}w,h,1}}$



(c) $\mathbf{R_{mod\text{-}w,h,2}}$

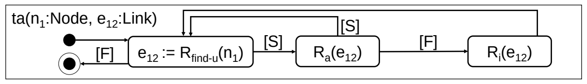


(d) $\mathbf{R_{mod\text{-}w,h,3}}$

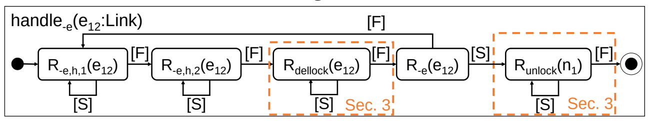


(e) $\mathbf{R_{mod\text{-}w,h,4}}$

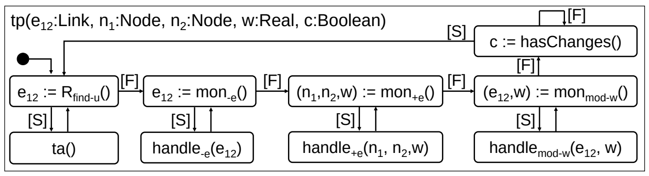
**Figure 8: Link weight modification rule with handlers**



(a) TC algorithm ta



(b) Link removal handler $\mathbf{handle_{-e}}$

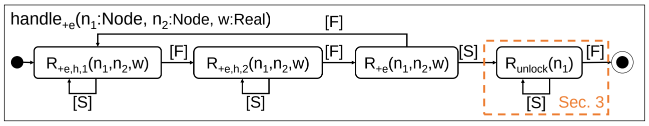


(c) TC process tp

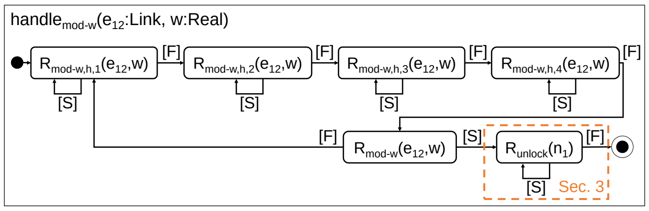
**Figure 9: Story diagrams**

operation node has at least one incoming activity edge and either one unguarded outgoing activity edge or two outgoing activity edges labeled with [S] and [F], respectively. A story diagram is *executed* as follows: The execution begins at the start node and continues along activity edges until arriving at a stop node. The execution of story and operation nodes is similar. Upon arriving at a story node, the contained graph transformation rule is applied (if possible). Upon arriving at an operation node, the contained operation is invoked. In case of two outgoing activity edges, if the rule application was successful or if the operation invocation returned a non-null result, the execution continues along the [S]-edge, else along the [F]-edge. In case of one one outgoing activity edge, the execution continues along this edge after the rule application or operation invocation. An *operation (node or link) variable* represent local variables of the story diagram. These variables are declared in parentheses next to the operation name and are used to pass information to the operation or among rule and operation invocations. An operation variable can be assigned based on parameters of a successfully applied rule or invoked operation (denoted with :=) and passed to rule applications or operation invocations (denoted with parentheses).

**Example:** Figure 9a shows ta, the specification of the TC algorithm. In ta, unmarked links are identified using $R_{find\text{-}u}$ and either activated (by applying $R_a$) or (if $R_a$ was inapplicable) inactivated as long as possible. The algorithm terminates if no more unmarked links exist. The operation variable $e_{12}$ is used to pass the identified unmarked link from $R_{find\text{-}u}$ to $R_a$ and $R_i$. Figure 9b shows the story diagram of the context event handler $handle_{-e}$ for $R_{-e}$. First, the rules $R_{-e,h,1}$ and $R_{-e,h,2}$ are executed as long as possible to anticipate all violations of $C_i$ that would otherwise result from removing $e_{12}$. Afterwards, $e_{12}$ is removed by applying $R_{-e}$. The story node containing $R_{unlock}$ is introduction in Section 3.4. Figure 9c shows tp, the specification of a continuous TC process. The *monitoring operations* $mon_{-e}$, $mon_{+e}$, $mon_{mod\text{-}w}$ detect pending context events, i.e., whether a link $e_{12}$ in the underlying topology no longer exists ($mon_{-e}$), a link from $n_1$ to $n_2$ is missing ($mon_{+e}$), or if the weight of a link $e_{12}$ has changed to w ($mon_{mod\text{-}w}$). A context event is *pending*



(a) Link addition handler ($\text{handle}_{+e}$)



(b) Link weight modification handler ($\text{handle}_{\text{mod-w}}$)

**Figure 10: Additional story diagrams in long version**

if it has not been processed by a context event handler yet. The operation hasChanges returns true if an unmarked link or pending context event exists.

Figures 10a and 10b shows the context event handler for link additions and link weight modifications, respectively.

tp is correct for the following reasons. All rules preserve weak consistency. Therefore, it remains to prove that ta terminates and enforces $C_u$, and that each context event handler terminates after performing the corresponding context event. First, ta enforces $C_u$ because it only terminates if no more unmarked links exist, and it always terminates because, for a given unmarked link $e_{12}$, either $R_a$ or $R_i$ is applicable due to the complementary application conditions. Second, $\text{handle}_{-e}$ may only terminate if the given link is removed, and it terminates because $R_{-e,h,1}$ and $R_{-e,h,2}$ may be applied for each link in the topology at most once, and $R_{-e}$ is guaranteed to be applicable afterwards. The same arguments apply to $\text{handle}_{+e}$ and $\text{handle}_{\text{mod-w}}$.

## 3 DISTRIBUTED TOPOLOGY CONTROL

This section contains the core contributions of this paper, which constitute answers to the following questions.

**Characterization of concurrency (Section 3.1):** Which important properties of distributed TC algorithms need to be represented in the specification (i.e., metamodel, constraints, GT rules, story diagrams)? What is a suitable computation model for specifying concurrent computation (atomicity, causality)? How can the per-node local views represented properly? Which liveness issues may occur?

**Liveness analysis and restoration (Sections 3.2 to 3.4):** Is a TC algorithm specification that is correct under centralized-global execution also correct under distributed-local execution? If not, which systematic steps are necessary to restore correctness?

### 3.1 Characterization of Concurrency

We begin with a characterization of concurrency in WSNs.

**Local views:** One particularity of TC is that the nodes operate on the graph that they are part of [34]. Additionally, instead of accessing a global topology, each node maintains a *local view*, i.e., an individual local copy of all links and nodes that are at most, e.g., two hops away from the node. Therefore, one Topology instance exists for each node and the local views of neighboring nodes overlap. Inside the local view of $n_X$, this node is marked as *self node* ($\text{isSelf}(n_X) = \textit{true}$), and all other nodes are marked as remote nodes $n_Y \neq n_X$, $\text{isSelf}(n_Y) = \textit{false}$. The metamodel isSelf attribute of Node class reflects this property.

**Local responsibility:** Each node is responsible for updating certain links in the topology: (i) A node may only mark its outgoing links because a node can only control its own transmission range. (ii) A node may unmark arbitrary links because unmarking a link does not entail a change of the transmission range but indicates that the link should be re-marked. (iii) A node handles the context events that affect its outgoing links. To reflect local responsibility, each rule contains exactly one *self-node variable*, denoted by a pentagon (e.g., ⬟1) in Figures 5 and 6. A self-node variable is a regular node variable for which an additional attribute constraint $\text{isSelf}(n_X) = \textit{true}$ exists. For each circular-shaped remote-node variables $n_Y \neq n_X$, an attribute constraint $\text{isSelf}(n_Y) = \textit{false}$ is added.

**Computation model:** We employ an interleaved computation model [11, Sec. 3.1] to model that TC processes are running concurrently on all WSN nodes. This means that no two rule applications happen at the same point in time. A rule application is an atomic computation step in our model. A TC process may only be interrupted prior to a rule application or operation invocation. The local views of all nodes are synchronized: The effect of a rule application performed on one local view is visible in the local views of all other nodes in the next time step. The control flow specification of a TC process induces a per-node happened-before relationship [25] of GT rule applications. Apart from this restriction, any interleaving of rule executions is possible.

**Liveness issues:** Deadlocks, livelocks, and starvation are the three major types of *liveness issues* in concurrent programming, which describe situations in which a process does not or may not make progress [38]. A *process makes progress* if it eventually reaches *progress statements*, which reflect a meaningful execution checkpoint. We define all rule applications and operation invocations in tp as progress statements. A *deadlock* is a situation in which multiple processes lack progress because they are waiting for actions of the other processes. A *livelock* is similar to a deadlock, but, here, each process does not wait for but continuously reacts to actions of other processes without making progress visible from outside. *Starvation* describes a situation in which a process lacks progress due to an unfavorable scheduling order.

To sum up, deadlocks and livelocks are situations in which non-termination is guaranteed whereas starvation is a situation in which termination is not guaranteed. Deadlocks are not possible with our notion of concurrent programmed GT because no blocking operations exist. However, livelocks and starvation are still possible.

**Example:** The example shown in Figure 11 illustrate the incorrect behavior of tp under concurrent execution. The figure depicts the local views of and rules applications of the TC processes running on $n_1$ and $n_3$ using swim-lane notation. Node $n_1$ is executing ta, and $n_3$ is executing $\text{handle}_{-e}$. Curved arrows across swim lane borders indicate a switch of the running process (→). Horizontal arrows within a swim lane specify the per-node rule application order, similar to the instruction pointer in traditional programming languages

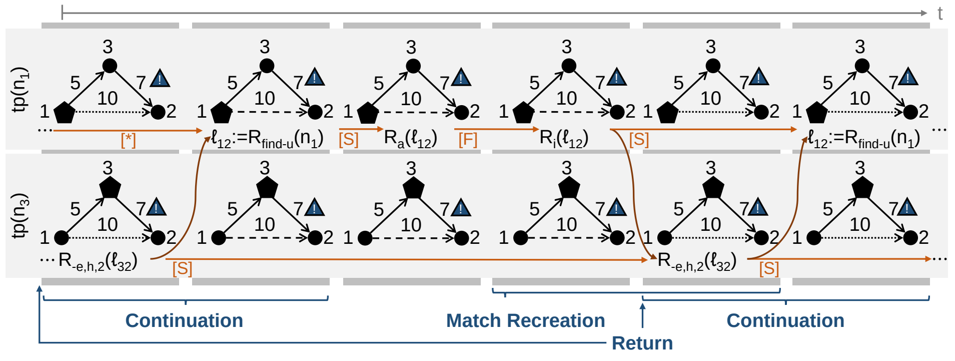


Figure 11: Starvation of TC processes on two nodes under concurrent execution

(→). The execution begins with a fully marked topology and a pending removal of $\ell_{32}$ (⚠). We assume that tp($n_3$) is scheduled initially, preparing the removal of $\ell_{32}$ by unmarking $\ell_{12}$. The next rule application in tp($n_3$) is again $R_{\text{-e,h,1}}$ ($\ell_{32}$). Now, we assume that tp($n_1$) is scheduled, detecting via $R_{\text{find-u}}(n_1)$ that $\ell_{12}$ is unmarked, trying $R_a$ unsuccessfully, and applying $R_i$ to $\ell_{12}$ successfully. The next rule application to be tried in tp($n_1$) is now $R_{\text{find-u}}(n_1)$. If $n_3$ is scheduled again, $\ell_{12}$ will become unmarked again. Therefore, neither process may make progress in case of an unfavorable scheduling order. Still, handle$_{\text{-e}}$ may terminate if $n_3$ is able to remove $\ell_{32}$ without being interrupted. The discussed situation represents starvation because non-termination is possible but not guaranteed. A livelock could be provoked, e.g., if ta and handle$_{\text{-e}}$ immediately reacted to each other.

The observed incorrect behavior originates from three properties. (i) **Loops:** The specifications of ta and handle$_{\text{-e}}$ contain loops: $R_{\text{find-u}}$ is the *condition* of an *[S]-loop*, which is executed as long as the condition rule is applicable, and $R_{\text{-e}}$ is the condition of an *[F]-loop*, which is executed as long as $R_{\text{-e}}$ is inapplicable. (ii) **Continuation:** Applying a rule (e.g., $R_i(\ell_{12})$) in one process leads to a new match of a loop condition in another process (e.g., $R_{\text{-e,h,2}}(\ell_{32})$). (iii) **Return:** The execution of both processes returns to the state of the original continuation. If the rule that causes the continuation destroys its match (e.g., $R_{\text{-e,h,2}}$), return the match is recreated in the meantime (e.g., by $R_i(\ell_{12})$).

## 3.2 Overview of Approach

We now propose a systematic, iterative approach to determine and resolve liveness issues in TC process specifications (Figure 12). We assume that the TC process is correct in under centralized-global execution and that all rules preserve consistency, which can be validated and ensured using the constructive rule refinement approach presented in [8, 19]. This assumption reduces the proof of correctness to a proof of termination for the TC algorithm and the context event handlers. Furthermore, we may assume that a finite number of context events is pending.

We begin with generalizing the finding from the previous example. Starvation or livelocks occur if applying a rule $R_1$ in tp($n_X$) causes the continuation of a loop (with condition rule $R_2$) in tp($n_Y$) that would not have occurred without an application of $R_1$. A new iteration of an [S]-loop results if a new match of $R_2$ is created by applying $R_1$ (e.g., an inactivated link). This situation is called a *dependency from $R_1$ to $R_2$* [5, 10]. Conversely, a new iteration of an [F]-loop results if the single match of $R_2$ is destroyed by applying $R_1$. This situation is called a *conflict of $R_1$ with $R_2$* [5, 10]. The term *interaction* subsumes dependencies and conflicts.

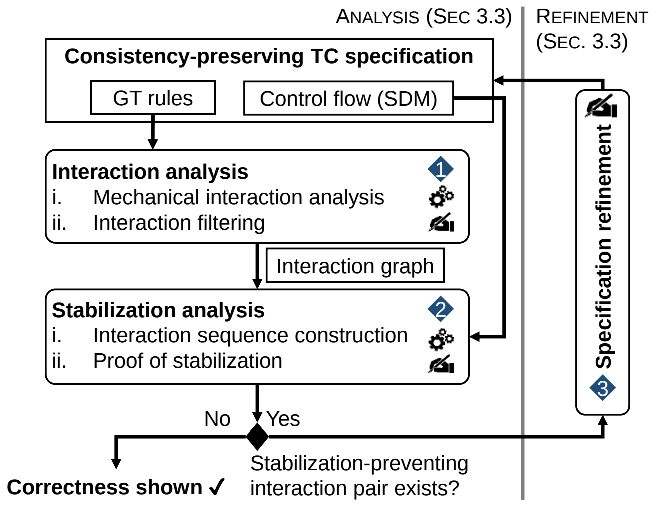

Analysis (Sec 3.3)
Refinement (Sec. 3.3)
Consistency-preserving TC specification
GT rules
Control flow (SDM)
Interaction analysis
i. Mechanical interaction analysis
ii. Interaction filtering
1
Interaction graph
Stabilization analysis
i. Interaction sequence construction
ii. Proof of stabilization
2
Specification refinement
3
No
Yes
Correctness shown ✓
Stabilization-preventing interaction pair exists?

Figure 12: Overview of approach

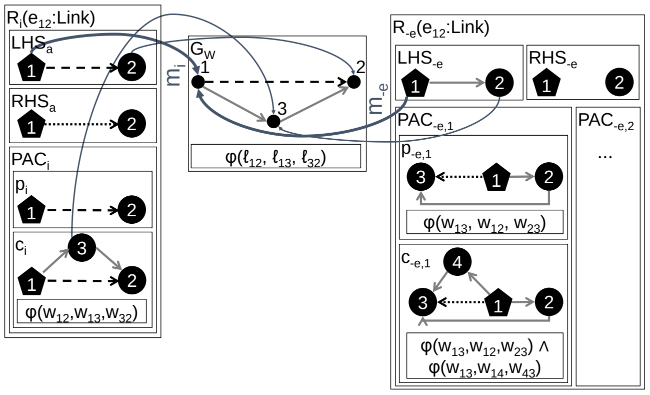
The first step in our proposed approach is the rule-level *interaction analysis* (1 in Figure 12). Here, we employ Critical Pair Analysis, an established, too-supported methodology for determining all potential interactions within a set of GT rules [5, 10]. The resulting interaction set is usually conservative and contains false-positives because, e.g., Henshin [4] cannot handle arbitrary attribute constraints. Therefore, the mechanically determined interactions should be filtered by hand based on additional knowledge. The result of the interaction analysis is an *interaction graph*, whose nodes represent rules and whose labeled edges represent interactions. The edge label corresponds to the type of interaction.

The second step is the *stabilization analysis* (2), which uses the interaction graph, control flow information, and assumptions of the computation model to determine potential stabilization-preventing interaction sequences. Based on the identified potential interaction sequences, we need to prove whether the TC process stabilizes. Either none of the interaction sequences prevents stabilization or at least one stabilization-preventing interaction sequences exists.

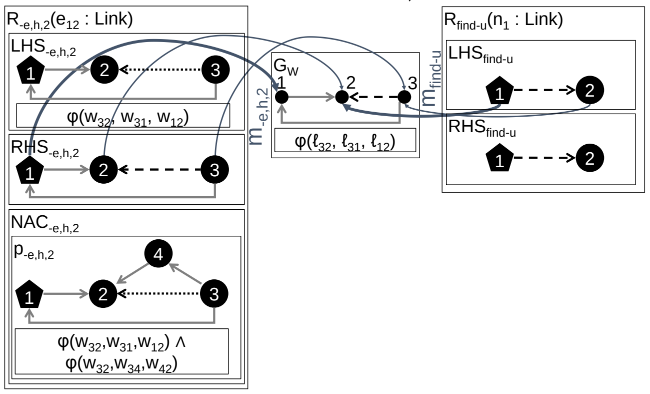
The identified stabilization-preventing interactions serve as starting point for refining the specification (3). Afterwards, the analysis phase is conducted again.

## 3.3 Analysis Phase

We use the running example to illustrate the analysis phase.
**Interaction analysis (1):** Interactions among rules are a necessary precondition for liveness issues. Therefore, we determine all interactions among the considered six rules $R_{\text{find-u}}$, $R_a$, $R_i$, $R_{\text{-e,h,1}}$, $R_{\text{-e,h,2}}$, and $R_{\text{-e}}$. The interaction analysis returns for each pair of rules $(R_X, R_Y)$ the sets of conflicts $R_X \xrightarrow{c} R_Y$ and dependencies $R_X \xrightarrow{d} R_Y$. A *conflict* $R_X \xrightarrow{c} R_Y$ consists of a witness topology $G_W$ together with matches $m_X$ of $LHS_X$ and $m_Y$ of $LHS_Y$ in G for which $R_X$ and $R_Y$ are both applicable initially and, after applying $R_X$, $R_Y$ is no longer applicable. Here, $G_W$ is a situation in which applying $R_X$ prevents the application of $R_Y$. A *dependency* $R_X \xrightarrow{d} R_Y$ consists of a witness topology $G_W$ together with a matches $m_X$ of the $RHS_X$ and match $m_Y$ of $LHS_Y$ in $G_W$ for which $R_Y$ becomes inapplicable when reverting $R_X$ at $m_X$. Here, $G_W$ reflects a situation where applying $R_X$ enables an application of $R_Y$. A *local conflict* $R_X \xrightarrow{cl} R_Y$ *or dependency* $R_X \xrightarrow{dl} R_Y$ maps the self-node variables of $R_X$ and $R_Y$ to the same node in $G_W$. A *remote conflict* $R_X \xrightarrow{cr} R_Y$ *or dependency* $R_X \xrightarrow{dr} R_Y$ is a non-local interaction. Local interactions represent interactions that happen within one TC process and can be analyzed using the control flow specification, whereas remote interactions relate to GT rule applications in different processes. A *same-match conflict* $R_X \xrightarrow{cm} R_X$ is a local conflict where both involved rules *and* matches are identical, respectively.
**Example:** Figure 13a shows the local conflict $R_i \xrightarrow{cl} R_{\text{-e}}$. The matches of each interaction are depicted as curved arrows that connect nodes variables to nodes. This interaction is a conflict because $R_i$ and $R_{\text{-e}}$ are applicable to $G_W$, and, after applying $R_i$ at $m_i$, $R_{\text{-e}}$ is inapplicable. The reason is that $PAC_{\text{-e},1}$ is not fulfilled if $\ell_{12}$ is inactive because no second $\varphi$-triangle for $\ell_{12}$ exists in $G_W$. This conflict is local because $m_i$ and $m_{\text{-e}}$ map the self-node variables of $R_i$ and $R_{\text{-e}}$ to $n_1$ in $G_W$. Figure 13b shows the remote dependency $R_{\text{-e,h},2} \xrightarrow{dr} R_{\text{find-u}}$, which causes the continuation in Figure 11. This interaction is a dependency because $R_{\text{find-u}}$ is applicable to $G_W$ and, after reverting $R_{\text{-e,h},2}$ at $m_{\text{-e,h},2}$, $R_{\text{find-u}}$ is inapplicable. This dependency is remote because $m_{\text{-e,h},2}$ maps the self-node variable of $R_{\text{-e,h},2}$ to $n_1$, and $m_{\text{find-u}}$ maps the self-node variable of $R_{\text{find-u}}$ to $n_3$.
**Stabilization analysis (2):** The stabilization analysis begins with determining interactions $R_r \xrightarrow{[c|d]} R_l$ for which an application of $R_r$ may cause the loop with condition $R_l$ to continue. If the loop is an [S]-loop, only dependencies are relevant ($R_r \xrightarrow{d} R_l$), and if the loop is an [F]-loop, only conflicts are relevant ($R_r \xrightarrow{c} R_l$). The first column of Table 1 lists the four loops of our example. In our running example, each continuation rule $R_r$ has a same-match conflict. A reiteration is only possible if the match of $R_r = R_{r,1}$ is recreated by an application of another rule $R_{r,2}$. A dependency $R_{r,2} \xrightarrow{d} R_{r,1}$ is a necessary precondition for observing a match recreation. Obviously, this process can be repeated if $R_{r,2}$ has a same-match conflict.



**(a) Local conflict $R_i \xrightarrow{cl} R_{\text{-e}}$**



**(b) Remote dependency $R_{\text{-e,h},2} \xrightarrow{dr} R_{\text{find-u}}$**

**Figure 13: Local and remote interactions**

To sum up, the stabilization analysis starts with collecting potential stabilization-preventing *interaction sequences*: We start with a continuation interaction $R_{r,1} \xrightarrow{[c|d]} R_l$ and may continue adding recreation interactions $R_{r,x+1} \xrightarrow{d} R_{r,x}$ as long as $R_{r,x} \xrightarrow{cm} R_{r,x}$ exists (*: [l|r]):

$$R_{r,n} \xrightarrow{d^*} R_{r,n-1} \xrightarrow{d^*} \cdots \xrightarrow{d^*} R_{r,1} \xrightarrow{d^*} R_l \text{ for [S]-loops,}$$
$$R_{r,n} \xrightarrow{d^*} R_{r,n-1} \xrightarrow{d^*} \cdots \xrightarrow{d^*} R_{r,1} \xrightarrow{c^*} R_l \text{ for [F]-loops.}$$

Calculating interaction sequences provides additional information for the subsequent proof of stabilization. In the presence of same-rule dependencies, interaction sequences can become arbitrarily long. Therefore, it is sensible to start with continuation interactions and extend the interaction sequences as needed. We build interaction sequences of length two because all continuation rules in our example have same-match conflicts.

The following two criteria help to filter the calculated interaction sequences. First, we neglect all interaction sequences that contain a context event recreation rule $R_{r,x}$ because (by assumption) a finite number of context events is pending. Therefore, each interaction involving a context event rule can be observed finitely often. Second, we neglect all interactions sequences that contain a local interaction that involves progress. For instance, whenever the local conflict $R_i \xrightarrow{cl} R_{\text{-e}}$ (Figure 13a) can be observed, the TC process applying both rules makes progress because its execution hits the

**Table 1: Interaction sequences before refinement (*: [l|r])**

| **Loop ($\mathbf{R_l}$)** | | **Interaction seqs. ($\mathbf{R_{r,2} \xrightarrow{d^*} R_{r,1} \xrightarrow{[c\|d]^*} R_l}$)** |
|---|---|---|
| $R_{\text{-e}}$ | [F] | $(R_{\text{-e,h,2}} \xrightarrow{dr} R_i \xrightarrow{cr} R_{\text{-e}})$, |
| $R_{\text{-e,h,1}}$ | [S] | $\emptyset$ |
| $R_{\text{-e,h,2}}$ | [S] | $(R_{\text{-e,h,2}} \xrightarrow{dr} R_i \xrightarrow{dr} R_{\text{-e,h,2}})$, |
| $R_{\text{find-u}}$ | [S] | $(R_i \xrightarrow{dr} R_{\text{-e,h,2}} \xrightarrow{dr} R_{\text{find-u}})$, |

progress statement $\text{mon}_{\text{-e}}$. Of the 69 original interaction sequences, 59 (10) can be neglected due to the first (second) criterion. Both sets of neglected interaction sequences overlap, leading to the three remaining interaction sequences shown in Table 1. The final step of the stabilization analysis is to prove whether or not the identified interaction sequences may prevent stabilization. Figure 11 corresponds to $(R_i \xrightarrow{dr} R_{\text{-e,h,2}} \xrightarrow{dr} R_{\text{find-u}})$. For $(R_{\text{-e,h,2}} \xrightarrow{dr} R_i \xrightarrow{cr} R_{\text{-e}})$, the situation is similar (Figure 14): Again, $tp(n_3)$ is running and handling the pending removal of $\ell_{32}$, and $tp(n_1)$ is waiting at $R_{\text{find-u}}$ in ta. In $tp(n_3)$, $R_{\text{-e,h,2}}$ is applied successfully (such that $s(\ell_{12}) = U$) and tried a second time unsuccessfully before switching to $tp(n_1)$. In $tp(n_1)$, $R_i$ is applied to $\ell_{12}$. After switching back to $tp(n_3)$, $R_{\text{-e}}$ cannot be applied due to the violated $PAC_{\text{-e,2}}$, leading to a restart of the context event handler. The third example for $(R_{\text{-e,h,2}} \xrightarrow{dr} R_i \xrightarrow{cr} R_{\text{-e,h,2}})$ is omitted here for space constraints[1]. To sum up, the stabilization analysis of our example reveals that all three interaction shown in Table 1 may cause starvation and, thereby, prevent stabilization.

## 3.4 Refinement Phase

The specification refinement (❸) ensures that the topology always stabilizes. We seek to eliminate all stabilization-preventing interactions without introducing new ones. The stabilization-preventing interaction sequences indicate that the interleaved execution of the TC algorithm and context event handler on distinct nodes causes starvation.

**Locking:** To eliminate stabilization-preventing interactions, we introduce guarded regions by locking the links that constitute the reason for the problematic interactions as follows: (i) We introduce the locker-lockee association into the metamodel (Figure 2, depicted as •—🔒↑). (ii) An unmarked link may only be marked if it is not locked ($NAC_{i,\text{unlock}}$ in Figure 15a and $NAC_{a,\text{unlock}}$ in Figure 16a). (iii) To break the stabilization-preventing interaction sequences (Table 1), $R_{\text{-e,h,2}}$ needs to lock the to-be-unmarked link $e_{12}$. Figure 15b shows the modified rule. The same applies for $R_{\text{+e,h,2}}$, $R_{\text{mod-w,h,2}}$, and $R_{\text{mod-w,h,4}}$.(iv) Before returning, each context event handler applies $R_{\text{unlock}}$ as long as possible to unlock all links locked by $n_1$ (Figures 9b and 15c). (v) Before removing a link $e_{12}$, $\text{handle}_{\text{-e}}$ removes all locks from $e_{12}$ by applying $R_{\text{dellock}}$ as long as possible (Figures 9b and 15d). Without this extension, $e_{12}$ may not be removed if it has locker-lockee associations due to the dangling edge condition [10, Sec. 1.2.1]. This extension only applies to this context event handler.

The refined rule set results in 86 unfiltered and only 2 filtered interaction sequence (Table 2).

**Table 2: Interaction sequences after refinement (*: [l|r])**

| **Loop ($\mathbf{R_l}$)** | | **Interaction seqs. ($\mathbf{R_{r,2} \xrightarrow{d^*} R_{r,1} \xrightarrow{[c\|d]^*} R_l}$)** |
|---|---|---|
| $R_{\text{-e}}$ | [F] | $(R_i \xrightarrow{dr} R_{\text{-e,h,2}} \xrightarrow{cr} R_{\text{-e}})$, |
| $R_{\text{-e,h,1}}$ | [S] | $\emptyset$ |
| $R_{\text{-e,h,2}}$ | [S] | $\emptyset$ |
| $R_{\text{find-u}}$ | [S] | $(R_i \xrightarrow{dr} R_{\text{-e,h,2}} \xrightarrow{dr} R_{\text{find-u}})$, |

**Deadlock freedom:** Introducing a locking mechanism into the specification gives rise to potential deadlocks. One necessary precondition for a deadlock is circular waiting. The control flow of tp enforces that a process requests and releases locks only during context event handling and waits for the release of locks only during ta. This means that a process that waits for a lock never holds a lock at the same time. Therefore, the proposed locking mechanism is free of deadlocks.

**Proof strategy:** In the following, we show that, for a finite number of pending context events, each interaction in the remaining interaction sequences may occur a finite number of times. We assume a *fair scheduler* [13, p. 7], which ensures that each TC process that may apply a GT rule is eventually scheduled. Without this assumption, stabilization cannot be proved because a locked link may lead to an infinite execution of ta. We show the following sufficient preconditions for stabilization for the running example. (i) **Readiness:** If the topology has not stabilized yet, at least one process is ready to apply a topology-modifying rule. (ii) **Finiteness:** For a finite number of pending context events, each GT rule is applicable finitely many times.

**Readiness:** We show that at least one GT rule can be applied if the topology has not stabilized yet using a case distinction. At least one link is unmarked or one context event is pending because the topology has not stabilized yet. (i) If an unmarked, unlocked link exists, this link can be either activated by $R_a$ or inactivated by $R_i$. (ii) If an unmarked, locked link exists, a context event handler is currently handling a pending context event. (iii) If at least one context event is pending, some node will start a context event handler eventually because a node that holds a lock never waits for other locks to be released (i.e., no cyclic waiting). (iv) The handling of a pending context event always terminates because the context event handling rules are constructed to be applicable if and only if the corresponding context event rule is inapplicable (and vice versa) [21], and the control flow of $\text{handle}_{\text{-e}}$ ensures that the context event handler rules and the lock-removing $R_{\text{dellock}}$ are tried if the context event rule is inapplicable (and vice versa). If $R_{\text{unlock}}$ is or becomes inapplicable, the context event handler terminates and the handled context event is no longer pending.

**Finiteness:** To show that the second precondition holds, we prove that the rule set fulfills the *descending chain condition* [26] according to a *potential function* $\nu$, which assigns to a topology G a potential $\nu(G) \in \mathbb{N}^4$, and a well-founded order relation $\succ$ on $\mathbb{N}^4$. The descending chain condition holds if $\nu$ decreases with each rule application to a topology $G_1$ with a resulting topology $G_2$: $\nu(G_1) \succ \nu(G_2)$.

We define $\nu$ as follows: $\nu(G) = (N_{ce}, N_{ul}, N_l, N_m)$ where $N_{ce}$ is the number of pending context events, $N_{ul}$ is the maximum number

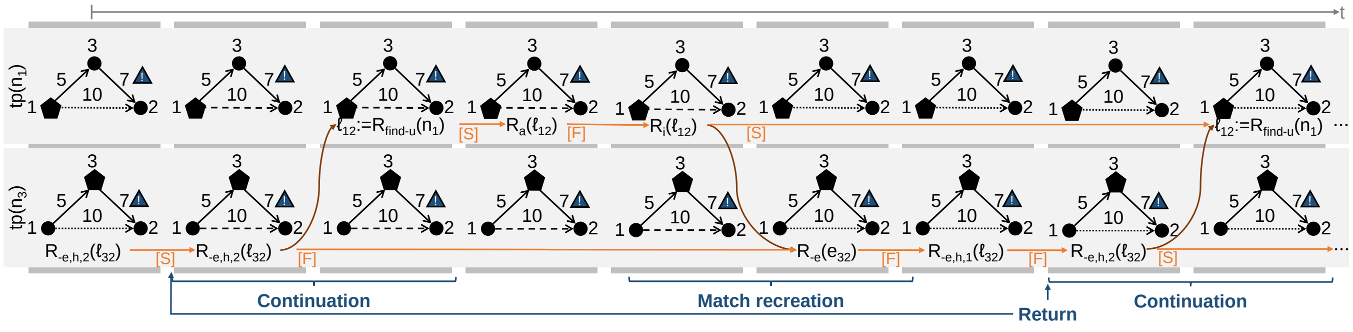
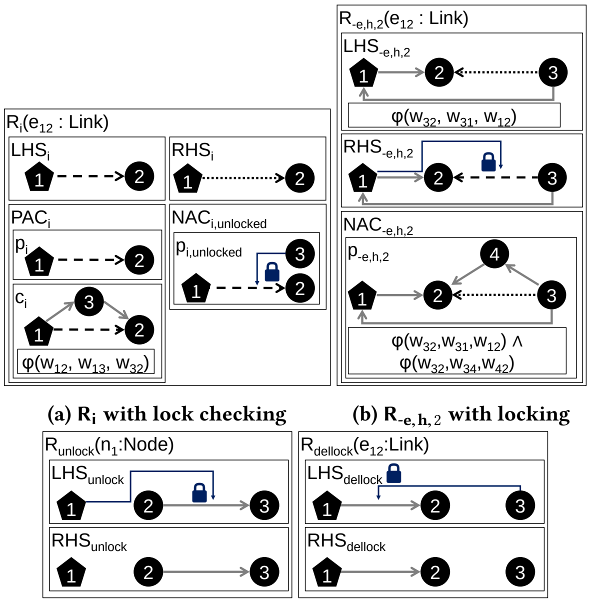


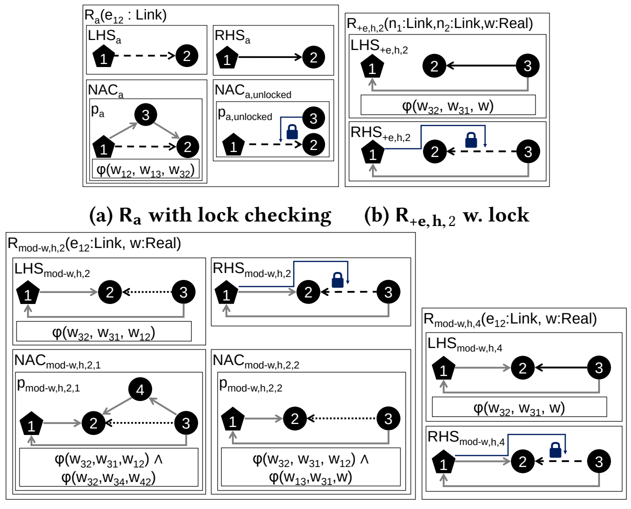
**Figure 14: Examples of incorrect behavior due to $(R_{\text{-e,h},2} \xrightarrow{\text{dr}} R_{\text{i}} \xrightarrow{\text{dr}} R_{\text{-e}})$**



(a) $\mathbf{R_i}$ **with lock checking** (b) $\mathbf{R_{\text{-e,h},2}}$ **with locking**

(c) $\mathbf{R_{unlock}}$ (d) $\mathbf{R_{dellock}}$

**Figure 15: Rule set after refinement with locking**



(a) $\mathbf{R_a}$ **with lock checking** (b) $\mathbf{R_{+e,h,2}}$ **w. lock**

(c) $\mathbf{R_{mod\text{-}w,h,2}}$ **with locking** (d) $\mathbf{R_{mod\text{-}w,h,4}}$ **w. lock**

**Figure 16: Refined rules with locking (long version)**

of links that need to be locked before performing the pending context events, $N_{\text{l}}$ is the number of locked links, and $N_{\text{m}}$ is the number of unmarked links. We define the ordering relation $>$ as the canonical order on $\mathbb{N}^4$:

$$\begin{aligned}
&\nu((N_{\text{ce},1}, N_{\text{ul},1}, N_{\text{l},1}, N_{\text{m},1})) > \nu((N_{\text{ce},2}, N_{\text{ul},2}, N_{\text{l},2}, N_{\text{m},2}))\\
&\Leftrightarrow N_{\text{ce},1} > N_{\text{ce},2}\\
&\vee \left(N_{\text{ce},1} = N_{\text{ce},2} \wedge N_{\text{ul},1} > N_{\text{ul},2}\right)\\
&\vee \left(N_{\text{ce},1} = N_{\text{ce},2} \wedge N_{\text{ul},1} = N_{\text{ul},2} \wedge N_{\text{l},1} > N_{\text{l},2}\right)\\
&\vee \left(N_{\text{ce},1} = N_{\text{ce},2} \wedge N_{\text{ul},1} = N_{\text{ul},2} \wedge N_{\text{l},1} = N_{\text{l},2} \wedge N_{\text{m},1} > N_{\text{m},2}\right)
\end{aligned}$$

Table 3 summarizes the influence of applying a rule R on $\nu$. We need to make additional assumptions about the context of certain rule applications to determine the exact influence on $\nu$. Only new pending context events may increase the value of $\nu$ (not shown in the table). This effect does not threaten stabilization because, by assumption, the number of considered context events is limited and so is the maximum value of $\nu$. Note that $\text{R}_{\text{find-u}}$ is the only rule that does not influence $\nu$. This does not threaten stabilization either because the scheduler is fair. Therefore, at least one $\nu$-influencing GT rule is eventually applied if $\text{R}_{\text{find-u}}$ is applied.

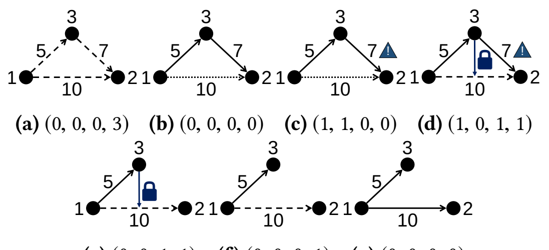
**Example:** Figure 17 shows a sequence of topologies together with their respective $\nu$ values. The initial topology consists of three unmarked links (Figure 17a). Afterwards, all links are marked and the topology stabilizes (Figure 17b). Now, the pending removal of $\text{e}_{32}$ is detected (⚠, Figure 17c). The context event handling requires one lock (Figure 17d). Afterwards, $\ell_{32}$ is removed, which reduces the number of pending context events (Figure 17e). Finally, $\text{R}_{\text{unlock}}$ unlocks and $\text{R}_{\text{a}}$ activates $\text{e}_{12}$ (Figures 17f and 17g).

## 3.5 Discussion

In the following, we discuss implementation aspects and next steps.
**Implementation:** We experimented with Henshin CPA [4] because it is established in the GT community. Unfortunately, we encountered severe runtime problems. On a Windows 7 64-bit machine with a 3.4 GHz i7-2600 CPU and 6 GB of RAM allocated to

**Table 3: Influence of rule applications on $v$**

| **Rule (with context)** | $N_{\mathbf{ce}}$ | $N_{\mathbf{ul}}$ | $N_{\mathbf{l}}$ | $N_{\mathbf{m}}$ |
|---|---|---|---|---|
| $R_i$ | 0 | 0 | 0 | -1 |
| $R_a$ | 0 | 0 | 0 | -1 |
| $R_{\text{find-u}}$ | 0 | 0 | 0 | 0 |
| $R_{\text{-e}}$ ($e_{12}$ unmarked) | -1 | 0 | 0 | 0 |
| $R_{\text{-e}}$ ($e_{12}$ locked) | -1 | 0 | -1 | 0 |
| $R_{\text{-e}}$ ($e_{12}$ marked, will lock) | -1 | -1 | 0 | 0 |
| $R_{\text{-e}}$ ($e_{12}$ marked, will not lock) | -1 | 0 | 0 | 0 |
| $R_{\text{-e,h,1}}$ | 0 | -1 | 0 | +1 |
| $R_{\text{-e,h,2}}$ | 0 | -1 | +1 | +1 |
| $R_{\text{unlock}}$ | 0 | 0 | -1 | 0 |
| $R_{\text{dellock}}$ | 0 | 0 | -1 | 0 |
| $R_{\text{+e}}$ | -1 | 0 | 0 | +1 |
| $R_{\text{+e,h,1}}$ | 0 | -1 | 0 | +1 |
| $R_{\text{+e,h,2}}$ | 0 | -1 | 0 | +1 |
| $R_{\text{mod-w}}$ (unmarked) | -1 | 0 | 0 | +1 |
| $R_{\text{mod-w}}$ (marked) | -1 | 0 | 0 | +1 |
| $R_{\text{mod-w,h,1}}$ | 0 | -1 | 0 | +1 |
| $R_{\text{mod-w,h,2}}$ | 0 | -1 | +1 | +1 |
| $R_{\text{mod-w,h,3}}$ | 0 | -1 | 0 | +1 |
| $R_{\text{mod-w,h,4}}$ | 0 | -1 | +1 | +1 |



**(a)** (0, 0, 0, 3) **(b)** (0, 0, 0, 0) **(c)** (1, 1, 0, 0) **(d)** (1, 0, 1, 1)
**(e)** (0, 0, 1, 1) **(f)** (0, 0, 0, 1) **(g)** (0, 0, 0, 0)

**Figure 17: TC sequence with $v$ values ($v = (N_{\mathbf{ce}}, N_{\mathbf{ul}}, N_{\mathbf{l}}, N_{\mathbf{m}})$)**

the CPA, analyzing all 36 rule pairs for conflicts took ca. 497 h with a median runtime of 0.3 s and a 90th-percentile runtime of 10.9 h per rule pair. We may expect that the dependency analysis takes a similar amount of time. This behavior originates from the large number of generated interaction candidates that are filtered afterwards using a pairwise list comparison (e.g., 27 640 possible and 0 actual conflicts for $R_{\text{-e,h,1}}$ and $R_i$). We are aware of two alternatives to Henshin CPA. In [5], the authors present a potentially faster Critical Pair Analysis whose implementation is not available yet. SyGrAV [7] leverages complex attribute constraints during candidate generation, but its implementation is still a prototype.

**Synchronization of local views:** In this paper, we take a shared-memory perspective on distributed TC by assuming that modifications in one local view are immediately visible in all overlapping local views. Technically, data distribution frameworks could provide the necessary synchronization of local views (e.g., OMG DDS [1], tinyDSM [31]). One strength of our approach is that correctness only relies on the per-node local view (w.r.t. safety) and happened-before relationship (w.r.t. liveness). This allows us to abstract from the data distribution framework as long as reliable synchronous communication is ensured. In a next step, we will model message exchange to overcome these assumptions and, especially, to reflect that local views may not be perfectly synchronized. Operations on remote links (e.g., locking) are realized as protocol messages and modeled with additional GT rules. The proved liveness properties only carry over if we assume synchronous reliable communication. For asynchronous communication, we need to extend the control flow to wait for acknowledgment messages from other nodes. For unreliable communication, we need to extend the control flow to cope with message loss. To reflect the stochastic nature of message transfer, stochastic [18] and probabilistic GT [24] can be used to specify the latency and reliability of the physical communication channel.

**Node failures:** We discuss shortly how to handle node failures in our approach. We assume that a node has enough time to properly handle its imminent failure (e.g., thanks to a watch dog). A node failure is a complex context event consisting of the removal of all incoming and outgoing links and the eventual removal of the isolated node. Therefore, the failure of $n_X$ can be handled as follows: (i) Each neighbor node $n_Y$ of $n_X$ handles the pending removal of $\ell_{YX}$. (ii) On $n_X$, handle$_{\text{-e}}$ is invoked for each outgoing link $\ell_{XY}$. (iii) The isolated node $n_X$ is removed.

## 4 RELATED WORK

In this section, we survey related research areas.

**Iterative interaction analysis:** In [17, 30], Critical Pair Analysis is applied to detect flaws in software specifications. In [3], Critical Pair Analysis is applied to improve the quality of medical treatment plans. All three works use AGG [36], which is the backend of Henshin CPA, and propose an iterative approach that uses the identified conflicts and dependencies as feedback to the user. In this work, we develop an iterative interaction-based analysis for communication system algorithms and propose to reduce the amount of interactions using mechanic rules before presenting them to the TC developer.

**Verification:** Verifying safety and liveness properties of distributed systems is a large and established research area. In general, process algebras (e.g., CCS [29]) and Petri nets [14] are well-suited for this purpose. However, verifying safety and liveness properties in the presence of structure and structural dynamics within the modeled system is a challenging line of research that has been tackled only recently by combining the aforementioned formalisms with graph transformation [15, 16]. Therefore, we decided to approach the challenge of designing correct-by-construction (distributed) TC algorithms using established techniques from the GT domain as also discussed in the following.

**Concurrency and distribution in GT:** In distributed GT, distributed systems are specified using a two-level graph-based model consisting of a (high-level) network graph and (low-level) local graphs per node [35, 37]. Therefore, distributed GT is a suitable starting point for introducing message passing into our scenario.

Considering concurrent rules applications has a long tradition in the GT community (e.g., [11]). Our work builds upon these fundamental results in that we assume rule applications to be atomic steps and interleaved execution semantics. Much of the classic literature focuses on characterizing concurrency in terms of trace-based equivalence classes. In contrast, we focus on a practical approach to specify TC processes in a Model-Driven Engineering approach.

In [27], global graph transformations are proposed, which specify that an entire graph is transformed in a step-wise synchronous manner. However, global graph transformation is impractical for modeling TC algorithms because nodes may but need not perform topological modifications synchronously.

Bigraphs specify distributed, reactive systems using (high-level) topographs for grouping nodes that belong to one location and (low-level) monographs for modeling links. Bigraphs and GT represent topologies and topology modifications differently [9]. Unfortunately, this makes bigraphs incompatible with our existing results on developing correct-by-construction TC algorithms.

In [6], an approach for querying distributed graph models using alternating bulk computation phases is presented. Complementary to this paper, their work eliminates runtime interactions by enforcing a high-level execution order of the entire system. One downside of their approach is the potential overhead for clock synchronization. In contrast, our approach does not assume synchronized clocks, but relies only on the per-node happened-before relationship induced by the control flow specification.

**Vertex-centric computing:** Vertex-centric computing originates from (distributed) graph processing frameworks such as Pregel [28]. As in our scenario, nodes manipulate the graph that they are part of. Even though the term vertex-centric computing is unusual in the WSN community, most WSN algorithms are vertex-centric. A TC algorithm influences possible communication links, whereas, in vertex-centric computing, communication links are not affected by manipulations of the graph.

## 5 CONCLUSION

In this paper, we proposed a model-based approach for specifying and analyzing correct distributed TC algorithms using programmed graph transformation and interaction analysis. Based on a practical, vertex-centric, interleaved, shared-memory computation model whose atomic actions are GT rule applications and whose control flow is specified using SDM, we characterized the possible liveness issues of starvation and liveness. To identify and resolve liveness issues, we proposed a systematic, iterative approach that builds on mechanically computed conflicts and dependencies. These interactions are then combined to obtain possible stabilization-preventing interaction sequences. To eliminate truly stabilization-preventing interaction sequences, we proposed to lock links that cause problematic interaction sequences selectively. Using the descending chain condition, we proved that the refined specification stabilizes in the presence of finitely many context events. Finally, we briefly reported about preliminary experiments with Henshin and outlined how to model unreliable, asynchronous communication, which is out of scope of this paper.

Our approach is not limited to WSNs: It is suitable to model and analyze distributed topology adaptation algorithm whose safety properties are expressible as graph constraints (e.g., adaptive overlays, vehicular ad-hoc networks). Next, we will tackle the step to a message-passing computation model as outlined in Section 3.5 and further evaluate the applicability of our approach based on other types of distributed topology adaptation algorithms.

## ACKNOWLEDGMENTS

This work has been funded by the German Research Foundation (DFG) as part of projects A1 and B2 within CRC 1053–MAKI.

## REFERENCES

[1] Kai Beckmann and Marcus Thoss. 2012. A wireless sensor network protocol for the OMG Data Distribution Service. In *Intl. Workshop on Intelligent Solutions in Embedded Systems (WISES)*. IEEE, NY, USA, 45–50. https://ieeexplore.ieee.org/document/6273603/

[2] Jalel Ben-Othman, Karim Bessaoud, Alain Bui, and Laurence Pilard. 2013. Self-stabilizing algorithm for efficient topology control in Wireless Sensor Networks. *J. of Computational Science* 4, 4 (2013), 199–208. https://doi.org/10.1016/j.jocs.2012.01.003

[3] Jonas Santos Bezerra, Andrei Costa, Leila Ribeiro, and Érika Cota. 2016. Formal Verification of Health Assessment Tools: a Case Study. *ENTCS* 324 (2016), 31–50. https://doi.org/10.1016/j.entcs.2016.09.005

[4] Kristopher Born, Thorsten Arendt, Florian Heß, and Gabriele Taentzer. 2015. Analyzing Conflicts and Dependencies of Rule-Based Transformations in Henshin. In *Fundamental Approaches to Software Engineering (FASE)*, Alexander Egyed and Ina Schaefer (Eds.). Springer, Berlin, Heidelberg, 165–168. https://doi.org/10.1007/978-3-662-46675-9_11

[5] Kristopher Born, Leen Lambers, Daniel Strüber, and Gabriele Taentzer. 2017. Granularity of Conflicts and Dependencies in Graph Transformation Systems. In *Graph Transformation*, Juan de Lara and Detlef Plump (Eds.). Springer Intl. Publishing, Cham, 125–141. https://doi.org/10.1007/978-3-319-61470-0_8

[6] Márton Búr, Gábor Szilágyi, András Vörös, and Dániel Varró. 2018. Distributed Graph Queries for Runtime Monitoring of Cyber-Physical Systems. In *Fundamental Approaches to Software Engineering (FASE)*, Alessandra Russo and Andy Schürr (Eds.). Springer Intl. Publishing, Cham, 111–128.

[7] Frederik Deckwerth. 2017. *Static Verification Techniques for Attributed Graph Transformations*. Ph.D. Dissertation. Technische Universität Darmstadt, Darmstadt. http://tuprints.ulb.tu-darmstadt.de/6150/

[8] Frederik Deckwerth and Gergely Varró. 2014. Generating Preconditions from Graph Constraints by Higher Order Graph Transformation. In *Intl. Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT)*, Vol. 67. ECEASST, Dortmund, 1–14. https://doi.org/10.14279/tuj.eceasst.67.945

[9] Hartmut Ehrig. 2012. *Bigraphs meet Double Pushout*. World Scientific, Singapore, 27–40. https://doi.org/10.1142/9789812562494_0038

[10] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer, Berlin Heidelberg. https://doi.org/10.1007/3-540-31188-2

[11] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg (Eds.). 1999. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 3: Concurrency, Parallelism, and Distribution*. World Scientific, River Edge, NJ, USA. https://dl.acm.org/citation.cfm?id=320647

[12] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. 1998. Story Diagrams: A New Graph Rewrite Language based on the Unified Modeling Language. In *Intl. Workshop on Theory and Application of Graph Transformation (TAGT)*. Springer, Berlin Heidelberg, 296–309. https://doi.org/10.1007/978-3-540-46464-8_21

[13] Wan Fokkink. 2018. *Distributed Algorithms: An Intuitive Approach*. The MIT Press, Cambridge, Massachusetts.

[14] Claude Girault and Rüdiger Valk. 2001. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Springer NY, Secaucus, NJ, USA.

[15] Andy Schürr Géza Kulcsár, Malte Lochau. 2018. Graph-Rewriting Petri Nets. In *Intl. Conf. on Graph Transformation (ICGT) (LNCS)*, Vol. to appear. Springer, Berlin Heidelberg, 1–16.

[16] Malte Lochau Géza Kulcsár, Andrea Corradini. 2018. Equivalence and Independence in Controlled Graph Rewriting. In *Intl. Conf. on Graph Transformation (ICGT) (LNCS)*. Springer, Berlin Heidelberg, 1–16.

[17] Jan Hendrik Hausmann, Reiko Heckel, and Gabi Taentzer. 2002. Detection of Conflicting Functional Requirements in a Use Case-driven Approach: A Static Analysis Technique Based on Graph Transformation. In *Intl. Conf. on Software Engineering (ICSE)*. ACM, New York, NY, USA, 105–115. https://doi.org/10.1145/581339.581355

[18] Reiko Heckel, Georgios Lajios, and Sebastian Menge. 2004. Stochastic Graph Transformation Systems. In *Intl. Conf. on Graph Transformation (ICGT)*, Hartmut

Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg (Eds.). Springer, Berlin Heidelberg, 210–225.

[19] Reiko Heckel and Annika Wagner. 1995. Ensuring Consistency of Conditional Graph Rewriting – A Constructive Approach. In *Joint COMPUGRAPH/SEMAGRAPH Workshop (ENTCS)*, Andrea Corradini and Ugo Montanari (Eds.), Vol. 2. Elsevier, Amsterdam, 118–126. https://doi.org/10.1016/S1571-0661(05)80188-4

[20] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. 2007. Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks. In *ACM/IEEE Conf. on Information Processing in Sensor Networks (IPSN)*. ACM, New York, NY, USA, 254–263. https://doi.org/10.1145/1236360.1236395

[21] Roland Kluge, Michael Stein, David Giessing, Andy Schürr, and Max Mühlhäuser. 2017. cMoflon: Model-Driven Generation of Embedded C Code for Wireless Sensor Networks. In *European Conf. on Modelling Foundations and Applications (ECMFA)*, Anthony Anjorin and Huáscar Espinoza (Eds.). Springer Intl. Publishing, Marburg, Germany, 109–125. https://doi.org/10.1007/978-3-319-61482-3_7

[22] Roland Kluge, Michael Stein, Gergely Varró, Andy Schürr, Matthias Hollick, and Max Mühlhäuser. 2016. A Systematic Approach to Constructing Incremental Topology Control Algorithms using Graph Transformation. *J. of Visual Languages & Computing (JVLC)* 38 (2016), 47–83. https://doi.org//10.1016/j.jvlc.2016.10.003

[23] Roland Kluge, Michael Stein, Gergely Varró, Andy Schürr, Matthias Hollick, and Max Mühlhäuser. 2017. A Systematic Approach to Constructing Families of Incremental Topology Control Algorithms Using Graph Transformation. *Software & Systems Modeling (SoSyM)* online (2017), 1–41. https://doi.org/10.1007/s10270-017-0587-8

[24] Christian Krause and Holger Giese. 2012. Probabilistic Graph Transformation Systems. In *Intl. Conf. on Graph Transformation (ICGT)*, Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg (Eds.). Springer, Berlin Heidelberg, 311–325.

[25] Leslie Lamport. 1978. Time, Clocks and the Ordering of Events in a Distributed System. *Commun. ACM* 21 (July 1978), 558–565. https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/

[26] Igor Litovsky, Yves Métivier, and Wiesław Zielonka. 1993. The power and the limitations of local computations on graphs. In *Graph-Theoretic Concepts in Computer Science*, Ernst W. Mayr (Ed.). Springer, Berlin, Heidelberg, 333–345. https://doi.org/10.1007/3-540-56402-0_58

[27] Luidnel Maignan and Antoine Spicher. 2015. Global Graph Computations. In *Intl. Workshop on Graph Computation Models (GCM) (CEUR)*. Detlef Plump, York, UK, 34–49. http://ceur-ws.org/Vol-1403/

[28] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[29] Robert Milner. 1989. *Communication and concurrency.* Prentice Hall, New York.

[30] Marcos Oliveira, Leila Ribeiro, Érika Cota, Lucio Mauro Duarte, Ingrid Nunes, and Filipe Reis. 2015. Use Case Analysis Based on Formal Methods: An Empirical Study. In *Recent Trends in Algebraic Development Techniques*, Mihai Codescu, Răzvan Diaconescu, and Ionuț Țuțu (Eds.). Springer Intl. Publishing, Cham, 110–130. https://doi.org/10.1007/978-3-319-28114-8_7

[31] Krzysztof Piotrowski, Peter Langendoerfer, and Steffen Peter. 2009. tinyDSM: A highly reliable cooperative data storage for Wireless Sensor Networks. In *Intl. Symposium on Collaborative Technologies and Systems (CTS)*. IEEE, NY, USA, 225–232. https://doi.org/10.1109/CTS.2009.5067485

[32] Grzegorz Rozenberg (Ed.). 1997. *Handbook of Graph Grammars and Computing by Graph Transformation.* Vol. 1: Foundations. World Scientific, Singapore. https://doi.org/10.1142/3303

[33] Immanuel Schweizer, Michael Wagner, Dirk Bradler, Max Mühlhäuser, and Thorsten Strufe. 2012. kTC – Robust and Adaptive Wireless Ad-Hoc Topology Control. In *Intl. Conf. on Computer Communication and Networks (ICCCN)*. IEEE, NY, USA, 1–9. https://doi.org/10.1109/ICCCN.2012.6289318

[34] Michael Stein, Mathias Fischer, Immanuel Schweizer, and Max Mühlhäuser. 2017. A Classification of Locality in Network Research. *ACM CSUR* 50, 4, Article 53 (Aug. 2017), 37 pages. https://doi.org/10.1145/3092693

[35] Gabriele Taentzer. 1996. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* Ph.D. Dissertation. TU Berlin.

[36] Gabriele Taentzer. 2000. AGG: A Tool Environment for Algebraic Graph Transformation. In *Proc. of AGTIVE'99 (LNCS)*, Manfred Nagl, Andy Schürr, and Manfred Münch (Eds.), Vol. 1779. Springer, Berlin Heidelberg, 481–490. https://doi.org/10.1007/3-540-45104-8_41

[37] Gabriele Taentzer. 2002. Visual Modeling of Distributed Object Systems by Graph Transformation. *ENTCS* 51 (2002), 304–318. https://doi.org/10.1016/S1571-0661(04)80212-3 GETGRATS Closing Workshop.

[38] Kuo-Chung Tai. 1994. Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs. In *Intl. Conf. on Parallel Processing (ICPP)*, Vol. 2. IEEE, NY, USA, 69–72. https://doi.org/10.1109/ICPP.1994.84

[39] Gergely Varró, Anthony Anjorin, and Andy Schürr. 2012. Unification of Compiled and Interpreter-Based Pattern Matching Techniques. In *European Conf. on Modelling Foundations and Applications (ECMFA)*, Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitris Kolovos (Eds.). Springer, Berlin Heidelberg, 368–383. https://doi.org//10.1007/978-3-642-31491-9_28

[40] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. 2013. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, New York.

[41] Andrew Whitmore, Anurag Agarwal, and Li Da Xu. 2015. The Internet of Things—A survey of topics and trends. *Information Systems Frontiers* 17, 2 (2015), 261–274. https://doi.org/10.1007/s10796-014-9489-2