TECHNISCHE
UNIVERSITÄT
DARMSTADT

A Compass to Controlled Graph Rewriting

DEM FACHBEREICH ELEKTROTECHNIK UND INFORMATIONSTECHNIK
DER TECHNISCHEN UNIVERSITÄT DARMSTADT
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
EINES DOKTORS DER NATURWISSENSCHAFTEN (DR. RER. NAT.)
GENEHMIGTE DISSERTATION

VON

GÉZA KULCSÁR, M.SC.

GEBOREN AM

9. SEPTEMBER 1988 IN CEGLÉD, UNGARN

REFERENT: PROF. DR. RER. NAT. ANDREAS SCHÜRR
KORREFERENT: PROF. DR. ANDREA CORRADINI (UNIVERSITÄT PISA)

TAG DER EINREICHUNG: 2019-01-22
TAG DER DISPUTATION: 2019-06-07

D17
DARMSTADT 2019

The work of Géza Kulcsár was partially supported by the Corporate Research Center (CRC) 1053 *Multi-Mechanismen-Adaptation für das künftige Internet (MAKI)* of the *Deutsche Forschungsgemeinschaft (DFG)* (`https://www.maki.tu-darmstadt.de`).

Kulcsár, Géza
A Compass to Controlled Graph Rewriting
Darmstadt, Technische Universität Darmstadt
Year of publication at TUprints: 2019
URN: urn:nbn:de:tuda-tuprints-93049
URL: `https://tuprints.ulb.tu-darmstadt.de/id/eprint/9304`
Disputation date: 2019-06-07

DECLARATION OF AUTHORSHIP

The following paragraphs are in German to conform to the General Doctoral Degree Regulations of the TU Darmstadt.

Erklärungen laut Promotionsordnung

- **§ 8 Abs. 1 lit. c PromO:** Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.
- **§ 8 Abs. 1 lit. d PromO:** Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.
- **§ 9 Abs. 1 PromO:** Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.
- **§ 9 Abs. 2 PromO:** Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

I hereby grant the Real-Time Systems Lab the right to publish, reproduce and distribute my work.

Darmstadt, 2019-01-22

Géza Kulcsár

PREFACE AND ACKNOWLEDGMENTS

This thesis summarizes about two years of research endeavors carried out at the Technical University of Darmstadt in Germany and at the University of Pisa in Italy. I thank all my colleagues at our group FG Echtzeitsysteme in Darmstadt for making me feel at home during my five years in Germany as far as it was possible. I'd like to thank Andy Schürr (leader of the group and also the first referee of this thesis) for a countless things; in particular, for giving me complete freedom to pursue my interests, even if they had a habit of frequent change at some points; and, in turn, for sparking my interest and tirelessly deepen my understanding in topics related to controlled graph rewriting. I'm also greatly indebted to Malte Lochau, who is responsible for introducing me to the second big pillar of the present work: the theory of process algebra. Malte was an immense help, an unwearying discussion partner and frequent co-author throughout the years spent with developing RePro, the process calculus being the subject of this thesis. It was also Malte who, by a brave talk on a yet unripe topic, made it possible that RePro captures the attention of Andrea Corradini (becoming the second referee of the thesis), the next person my deepest gratitude goes to: his decades of experience, his sharp sight and uncompromising rigor, but also his incomparable hospitality and compassion made me leave a piece of my heart in Pisa, after three months of intense research on RePro, when, in particular, the idea of compasses has been born. Any theoretical work has to be brought to life by an adequate practical illustration: I thank my colleague Roland Kluge and others in the research project MAKI, from whom I learned a lot about wireless sensor networks.

Still, all those efforts would have remained fruitless if I don't have the perfect background for it in my life: I dedicate all my contributions to my wonderful wife Klára, whose loving company, cheerful spirit and endless care have magnified my joyful days and helped to get through the tough ones. Thank you for embarking on this journey with me—to Germany and in our whole life. I thank all other members of my colorful family for surrounding me with love and unconditional support—believing in me even when I couldn't. Above all, I thank my parents for teaching me, from the very first days I can recall, to always live my dream. I thank all my friends for just being who they are. (And a special thanks to David Tibet for manifesting the musical scenery for writing this thesis.)

Budapest, on the 21st of January 2019, the Feast of Saint Maximus the Confessor

ABSTRACT

With the growing complexity and autonomy of software-intensive systems, abstract modeling to study and formally analyze those systems is gaining on importance. Graph rewriting is an established, theoretically founded formalism for the graphical modeling of structure and behavior of complex systems. A graph-rewriting system consists of declarative rules, providing templates for potential changes in the modeled graph structures over time. Nowadays complex software systems, often involving distributedness and, thus, concurrency and reactive behavior, pose a challenge to the hidden assumption of global knowledge behind graph-based modeling; in particular, describing their dynamics by rewriting rules often involves a need for additional control to reflect algorithmic system aspects. To that end, controlled graph rewriting has been proposed, where an external control language guides the sequence in which rules are applied. However, approaches elaborating on this idea so far either have a practical, implementational focus without elaborating on formal foundations, or a pure input-output semantics without further considering concurrent and reactive notions.

In the present thesis, we propose a comprehensive theory for an operational semantics of controlled graph rewriting, based on well-established notions from the theory of process calculi. In the first part, we illustrate the aforementioned fundamental phenomena by means of a simplified model of wireless sensor networks (WSN). After recapitulating the necessary background on DPO graph rewriting, the formal framework used throughout the thesis, we present an extensive survey on the state of the art in controlled graph rewriting, along the challenges which we address in the second part where we elaborate our theoretical contributions. As a novel approach, we propose a process calculus for controlled graph rewriting, called RePro, where DPO rule applications are controlled by process terms closely resembling the process calculus CCS. In particular, we address the aforementioned challenges: (i) we propose a formally founded control language for graph rewriting with an operational semantics, (ii) explicitly addressing concurrency and reactive behavior in system modeling, (iii) allowing for a proper handling of process equivalence and action independence using process-algebraic notions.

Finally, we present a novel abstract verification approach for graph rewriting based on abstract interpretation of reactive systems. To that end, we propose the so-called compasses as an abstract representation of infinite graph languages and demonstrate their use for the verification of process properties over infinite input sets.

ZUSAMMENFASSUNG

Heutzutage, die Relevanz abstrakter Modellierungsansätze für die formale Analyse komplexer Rechnersysteme gewinnt durch wachsende Komplexität und Autonomie auch zunehmend an Bedeutung. Graphersetzungssysteme stellen einen etablierten, theoretisch fundierten Formalismus zur Modellierung der graphischen Struktur und des Verhaltens solcher Systeme dar. Ein Graphersetzungssystem besteht aus deklarativen Regeln, die als Vorlage für potentielle Änderungen während der Evolution der modellierten Systemstruktur fungieren. Darüber hinaus stellen heutige komplexe Softwaresysteme oft zusätzliche Anforderungen bezüglich Nebenläufigkeitseigenschaften und reaktivem Verhalten dar, welche oft mit der verborgenen Annahme, dass die graphische Abstraktion das System vollständig repräsentiert, kollidieren. Insbesondere fordert die Beschreibung der dynamischen Systemaspekte oft zusätzliche externe Kontrollkonstrukte zur Steuerung der Anwendung von Graphersetzungs-Regeln. Um diese Herausforderungen anzugehen, kontrollierte Graphersetzungssysteme wurden in der Literatur vorgeschlagen, versehen mit einer externen Kontrollsprache für die Beschreibung der zugelassenen Regelsequenzen. Allerdings haben bisherige Ansätze entweder einen starken Fokus auf die Implementierung ohne näher auf formale Grundlagen einzugehen, oder sie beschränken sich auf eine reine Ein-/Ausgabe-Semantik ohne tiefergehende Betrachtung reaktiven und nebenläufigen Verhaltens.

In der vorliegenden Arbeit schlagen wir daher eine ganzheitliche Theorie zur Formalisierung einer operationellen Semantik von kontrollierter Graphersetzung vor. Dabei verwenden wir etablierte Begriffe und Resultate der Theorie nebenläufiger Prozess-Kalküle als Grundlage für die Spezifikation unserer Kontrollsprache. Im ersten Teil der Arbeit illustrieren wir zunächst die vorgehende fundamentale Phänomene anhand eines simplifizierten Modells für drahtlose Sensornetzwerke (wireless sensor networks, WSN). Nachdem wir die Grundlagen der sog. DPO-Graphersetzung rekapituliert haben, präsentieren wir einen ausführlichen Überblick über bisherige Ansätze und offene Probleme im Bereich, und leiten daraus die für uns anstehende Herausforderungen ab. Im zweiten Teil präsentieren wir unsere theoretische Hauptresultate: wir schlagen ein Prozess-Kalkül namens RePro für kontrollierte Graphersetzung vor. In RePro werden Sequenzen von möglichen DPO-Regelanwendungen eingeschränkt durch die Transitionen eines zusätzlichen Kontrollprozesses, dessen Syntax und Semantik an das Prozess-Kalkül CCS angelehnt ist. Insbesondere betrachten wir die folgenden Herausforderungen: (i) RePro stellt eine formal fundierte Kontrollsprache mit einer operationellen Semantik, (ii) die auch Nebenläufigkeit und reaktives Verhalten explizit adressiert und (iii) das Definition formaler Begriffe für Prozessäquivalenz und Unabhängigkeit von Aktionen ermöglicht.

Zum Schluss wird auf diesen Grundlagen ein Ansatz zur symbolischen Verifikation dieser Prozesse vorgestellt, basierend auf das Rahmenwerk der abstrakten Interpretation (abstract interpretation). Hierfür schlagen wir sog. Kompasse (compasses) als abstrakte Repräsentation unendlicher Graphmengen vor, und demonstrieren ihre Eignung zur Verifikation gewisser Prozesseigenschaften für unendliche Eingabemengen.

CONTENTS

Part I

INTRODUCTION

1

CONCURRENT ABSTRACTIONS: GRAPH MODELS AND NETWORKS

1.1 MOTIVATION: A PLURALISTIC UNIVERSE

> **हस्तिदर्शन इवजात्यन्धाः**
>
> [...] as a number of persons born blind perceiving the elephant through touching its different parts, come to have diverse notions regarding it [...]
>
> Adi Shankaracharya, from his *bashya* on the verse 5.18.1 of the *Chandogya Upanishad*[104]

Starting from ancient times, people realized again and again that, as an inherent property of our *condition humaine*, everyday human perception and rational thinking is not able to capture the world as a whole—our picture of it is necessarily fragmentary (both in its momentary and persistent forms). It is, however, crucial to notice here that any such statement necessarily contains an implicit assumption about a unitary, universal Whole, a One, being the ultimate object of our perception.

Due to this inherent limitedness of observing that Whole, the resulting fragmentariness also descends and goes over to some smaller, seemingly more well-defined, units within the whole, i. e., even if considering a smaller set or *domain* of real-life objects, we as perceivers face the same problem of not being able to gain monolithic, integrated, homogeneous – simply: *complete* – knowledge. Moreover, during that process of contemplation, *perceivers perceive perceivers*: the fragmentariness disperses among the (*a priori* also unknown) *plurality* of perceivers. Thus, given a domain assumed to have a universal interpretation, in reality, there arises a co-existence of different fragments related to the knowledge acquired on that domain.

In our current epoch, the same phenomena are easily observable even in our ways of approaching and connecting to the *digital world*. The term *ubiquitous computing* has become a commonplace since the original vision of Mark Weiser [113]. (Also, cf. [57] on emphasizing the protean character of ubiquitous computing.) His essay coined another important term, though less frequently quoted: *embodied virtuality*. The Internet as a virtual space and, generally, each form of digital interaction has almost seamlessly fused into our perceived, everyday, commonplace reality. This is meant by virtuality being em-bodied: the connections and the devices used for connecting are everywhere, on and around us, they are not only visible but tangible and always within reach—despite their essentially non-physical ontology, their appearance is interwoven with the body of physical reality and achieved perfect mimicry.

Indeed, the un-mediated presence of digital technology leads to an interplay between the physical and the digital world: we do not only leave the physical plane to contemplate the virtual world *per se* (as it was the case in the early days of Internet), but also vice versa, elements of the digital world help us in connecting (back) to our physical reality: our carry-on mobile devices have become the single most important way to communicate with others, and we gather a large amount of environmental data through devices installed for that purpose.

Due to those circumstances – the fragmentariness of knowledge, the co-existence of those fragments, and the immediacy of technology – *computer science* has not only to develop methods providing a correct, adequate behavior of the involved devices, but also has to provide means to cope with the overall scenario described above: to have means for *representing* in a concise manner the overall knowledge about a given domain[1], and, using that representation, to faithfully capture those *concurrent* (internal or external) perceptions of the domain.

In the following, based on the above musings, we elaborate and make more specific those concepts relevant to underpin and motivate the theoretical advances described in Part II of the thesis. In particular, we give a high-level description of those networks arising on the border between people and their environment, through the example of *wireless sensor networks* (Sect. 1.2). Then, we demonstrate the use of *modeling* techniques for such abstract model representations using the example of wireless sensor networks (Sect. 1.3). Finally, we emphasize again a crucial aspect of modeling of such scenarios, motivating and guiding our technical contribution in Part II: we consider how the natural *concurrency* arising in self-organizing multi-agent systems can be embraced and faithfully reflected by well-founded modeling paradigms (Sect. 1.4).

1.2 MOBILE AD HOC NETWORKS AND TOPOLOGY CONTROL

The Latin phrase *ad hoc* (lit. “for this”, i.e., for the present conditions) has been used in various contexts, also within computer science, and, particularly, in network theory and practice. In that latter context, the notion of an *ad hoc network* has been widely studied and applied for various different networking scenarios, mostly within the context of wireless mobile networks due to their high degree of adaptability.

---

[1] To what extent is such a representation useful or meaningful, being necessarily exposed to the same fragmentariness of knowledge as the represented domain itself? This question would deserve an elaborate treatment on its own, but is out of scope here.

Indeed, *mobile ad hoc networks* (MANET) are self-configuring, decentralized networks of mobile devices; thus, the MANET concept involves some *autonomy* of the network regarding its ability to self-organize itself without any central administration [17].

As a direct consequence, the *topology* of such a network is highly dynamic. Those dynamics are influenced by three factors:

- *Mobility:* Inherently, the devices of such a network are typically non-stationary, also influencing their possible connections (i.e., the network topology).
- *Instability of wireless connections:* As a wireless connection cannot rely on any protected transmission media, it is more exposed to environmental effects causing the links to become instable.
- *Proactive topology optimization:* As link maintenance is a central concern in MANET, often, monitoring and control mechanisms are applied to optimize the topology according to various quality criteria. The corresponding topology decisions might naturally collide with the stochastic topology dynamics arising due to the other two factors above.

According to the observations above, ad hoc networks demonstrate in an adequate manner the general observations in Sect. 1.1: In a MANET, the network participants (devices, agents, etc.) are simultaneously the perceivers of (their own view on) the network, and the objects being perceived by others. Thus, autonomy implies a diffusion of object and subject.[2]

The actual kind of devices and connection technologies used heavily depends on the application contexts where a MANET is employed. There is a diverse range of scenarios, differing to a large extent w.r.t. the applied hardware as well as software solutions and technologies, still, each adhering to the aforementioned MANET principles. Typical application scenarios include *vehicular ad hoc networks* to enhance communication between traffic participants and roadside units, *smartphone ad hoc networks* to facilitate a direct communication in the absence of traditional network infrastructure or for privacy reasons, *Internet-based ad hoc networks* for extending Internet services themselves, and many others.

In the following, we describe in detail another typical instance of mobile ad hoc networks: *wireless sensor networks* (WSN). The choice of WSN as a concrete scenario is motivated by the fact that *topology control* (i.e., proactive topology optimization) has been thoroughly studied in the case of WSN, and the main principles behind topology decisions aptly demonstrate the inherent abstractional representation

---

[2] We point out that the term *ad hoc*, although fixed in the technical jargon, is rather counterintuitive in this case: stronger autonomy would rather imply a stronger ability to cope with any environmental situation, not only with “this” (*hoc*).

qualities of our *model-based* approach as elaborated in the upcoming Sect. 1.3.

Wireless sensor networks (WSN) are employed in various contexts such as environmental data collection, smart buildings and cities, zoological and agricultural monitoring, construction monitoring, military applications, etc. [17, 72]. In a WSN, autonomous sensor *nodes* are deployed in a given environment in order to fetch and aggregate measurement data.

The sensors autonomously organize their communication to share and distribute the data being collected. The communication is performed through wireless channels, i.e., communication *links*, where the overall *topology* (i.e., the network graph) emerges as the result of both the physical context of the deployed nodes (node proximity, wireless connection possibilities) and the consensual accumulation of decisions of single nodes about their own link preferences. In fact, maintaining each possible communication link might be unnecessarily energy-consuming for nodes and thus have negative effects on WSN lifetime.

Topology control is a diversely applicable and widely used technique to optimize a WSN topology according to some quality criteria. According to a survey of Li et al. [72], the two major optimization objectives in the field of WSN topology control are the following:

- *Coverage:* For coverage optimization techniques, the goal is to influence the spatial configuration of sensor nodes to improve how well the physical target area is monitored by the WSN. There are various metrics being used for this purpose, e.g., blanket, barrier or sweep coverage (for details, refer to [72]).
- *Power management:* Sensors in WSN are typically low-cost and, accordingly, low-capacity devices both in terms of calculations as well as battery lifetime. After a WSN has been deployed, usually, no new nodes are added to the network and thus, the overall WSN *lifetime* is determined by the lifetime of the single nodes the WSN consists of [30]. In any WSN application, the lifetime of the network is a crucial aspect of their effective use; therefore, it is a central concern in WSN research to propose dynamic techniques for nodes in order to save battery lifetime. To this end, the major technique is to reduce the number of links maintained by the nodes.

More recently, in parallel to the growing scale of application scenarios and, thus, also of accumulated data, a third major topology control objective arose [54, 56]:

- *Data freshness:* In contrast to the aforementioned objectives, data freshness is concerned with the measurement of data accumulated in a WSN instead of its physical deployment and configuration. Data freshness refers to the potential accuracy of data distributed among sensor nodes; thus, optimizing towards data freshness might involve establishing new routing paths. Therefore, such content-centric topology control mechanisms would necessarily interfere with, e.g., power management considerations if applied simultaneously in a single WSN.

In case of the first two optimization objectives, a common underlying criterion is to preserve some basic connectivity properties of the network. However, a crucial difference lies in the operations applied to the network in order to optimize: For coverage, nodes are directly forced to change their physical location, whereas in the case of power management, locality might be abstracted away as it is the status of physically available links which gets changed. When performing those optimization operations, mostly affecting single nodes and their local context, connectivity preservation usually involves *constraints* formulated over the whole network.[3]

Thus, connectivity constraints shed light on an inherent, implicit discrepancy of the assumptions behind topology control, very much in the spirit of our observations in Sect. 1.1: Whereas topology control decisions are necessarily based on an assumed knowledge on the *overall* status (comprising node properties, location, topology, etc.) of the WSN, this complete knowledge is never at hand in reality and decisions thus have to be made on the basis of a local knowledge fragment. This discrepancy might be mildened, but not fully eliminated by describing topology control operations for single nodes based solely on their local context: even there, the assumptions behind a decision might not hold anymore at the time of its execution.

In the following, we mainly focus on the power management aspect of topology control, as in that case, more details of the physical setting and the hardware can be left out of consideration by still retaining a satisfyingly realistic level of network representation (cf. also the upcoming Sect. 1.3). In particular, we examine topology control algorithms whose aim is to achieve a save on battery lifetime in nodes by reducing link redundancy [44, 111, 112]. Within this family of algorithms, the complexity of redundancy criteria largely varies, even if we consider a more specific setting, *pattern-based* redundancy detection, where the presence of a given geometrical link structure, usually a triangle of links, indicates redundancy and results in an operation eliminating that configuration. The aforementioned triangle elimination constitutes the basic principle of the *kTC* family of topology control algorithms. In the case of kTC, the goal is explicitly to reduce the number of links attached to each node whenever possible, by

---

[3] Connectivity constraints might be either *hard* or *soft* in a given application context, distinguishing if deterioration is tolerated to some degree. E.g., it is usually a hard constraint that each node can reach each other along some path; whereas, for smaller components, one might consider some further metric constraints on the length of those paths as a soft criterion. For details, cf. [65, 107].

detecting link triangles in the neighborhood of nodes and inactivating one of their links [103]. The decision on which edge to eliminate is based on a weighting which, in turn, relies on physical measurements in a real-life implementation.

In the field of WSN, it is inevitable to study topology control algorithms and the overall (potential or predictable) network behavior also *a priori*, i.e., before the actual WSN deployment, in order to uncover mistakes in the algorithm design or to detect potential conflicts with environmental events. One possibility is to deploy real WSN hardware in a controlled simulation testbed; however, for many scenarios, even such a simulation is rather resource-consuming, with results having limited relevance to an actual real-life deployment. Therefore, there is a necessity to develop WSN *analysis* techniques which are completely hardware-independent, but are still able to faithfully capture relevant non-physical aspects of WSN behavior.

Thus, after having recalled the motivation and purpose of topology control in general and narrowing the scope to fit the concrete example presented in the thesis, in the following section, we present a graph-based model for WSN and a rule-based description of topology control within that model.

1.3 MODELING AS ABSTRACTIONS OVER REAL-LIFE DOMAINS

As motivated above, analyzing the behavior of a real-life system domain, like that of a WSN, requires an abstract representation of the domain itself as well as formal techniques for reasoning about WSN behavior and, in particular, topology control. *Model-Driven Engineering* (MDE) is a software engineering paradigm which advocates the use of models, i.e., *structured* domain representations, as artifacts on which the analysis can be carried out directly. Therefore, in general, MDE requires modeling techniques which are adaptable to any analysis scenario w.r.t. the level of abstraction involved, also providing a formally founded reasoning methodology for both single domain states and their evolution over time.

Representing a structure in a formally (i.e., even mathematically) well-founded way amounts to relying on some kind of *graph* structure. A graph (i.e., a net of nodes and edges in between) immediately provides an abstraction over real-life domains: graph nodes correspond to real-life objects and edges represent (any kind of) relations between them. Moreover, this correspondence between reality and abstraction becomes very apparent if the domain being the subject of modeling is a network, as in our WSN example above. In the following, we describe a concrete WSN modeling case study which will be used throughout the thesis for demonstration purposes.

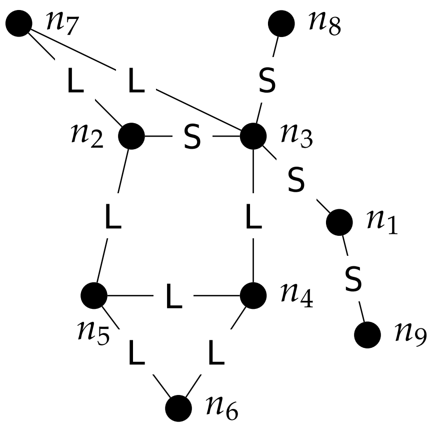


Figure 1.1: Example Topology

1.3.1 *Topology Representation*

We consider an autonomous sensor network consisting of homogeneous sensor nodes without any centralized control, thus, each node in our model represents the same kind of device.[4] Although sensor nodes distinguish themselves through a high degree of mobility, capturing (patterns of) their location changes in a model necessarily involves sophisticated stochastic methods, which belong to a different modeling approach than the subject of the present thesis. Thus, we provide a WSN model where node locations are not directly reflected.[5]

Regarding the communication links, in the thesis, we rely on a widely used WSN modeling assumption: *unit disk graph* (UDG). According to UDG, each sensor node has a uniform transmission range and there is a bidirectional communication link between two nodes whenever they are mutually in the transmission range of each other. However, for presenting an abstract version of the *kTC* topology control algorithm (originally relying on edge weighting based on physical properties), we consider different link lengths; for demonstrating the kTC mechanism, it suffices to consider *short* (S) and *long* (L) links. At this point, we are already able to present a (yet preliminary) topology model, shown in Figure 1.1, where the black dots represent nodes and (undirected) edges between them are bidirectional communication links, labeled with their respective lengths.

The purpose of topology control and, in particular, kTC is to monitor the topology and proactively change which available communication links should be *actively* maintained, i.e., used for communication

---

[4] Note that we exploit the ambiguity of the coinciding meanings of the term *node* here and refer by it both to the sensor nodes itself and to the graph nodes which represent them. However, if a distinction is needed, we specify which kind of node is meant.

[5] Still, we are able to capture the effect of node movements on the topology by introducing link creation and deletion events into the description of the environment as shown in the next section.

purposes. Therefore, an edge of the topology graph represents the physical possibility to establish a connection, whereas the current status of the link from a network perspective is represented by a further edge attribute: *active* (`a`) indicates the link is currently used for communication, *inactive* (`i`) denotes links that are not in use, i.e., they should currently not be maintained according to topology control. Topology control mechanisms, also kTC, often rely on an additional marking principle, reducing the risk that crucial links get inactivated without verifying the decision. For this purpose, we introduce an intermediate edge status: links with the status *unclassified* (`u`) can still be used for communication, but require status revision by topology control (e.g., due to environmental changes). In turn, as an algorithm design principle, topology control performs each activation and inactivation step through this intermediate status. Thus, our model includes six different edge types in total, denoted `S;a`, `S;i`, `S;u`, `L;a`, `L;i` and `L;u`.

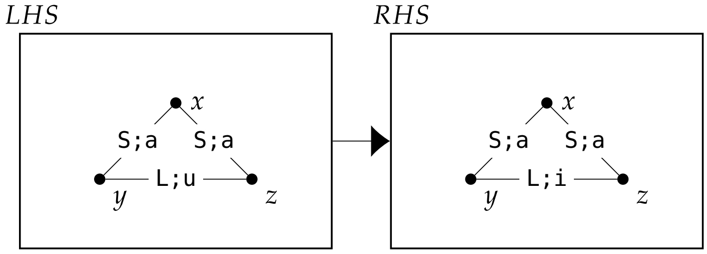
To address topology constraints whose scope goes beyond physical link patterns, topology control usually considers a *two-layered* model of WSN. In this setting, the aforementioned physical edges constitute the so-called *underlay* network layer, while the *overlay* consists of *virtual edges*. In a real-life WSN, those virtual edges can be used to implement complicated transport and routing mechanisms, potentially even connecting the WSN to the Internet; in turn, the overlay concept might also reflect data freshness criteria as introduced above. However, in this thesis, it is sufficient to conceive of virtual edges as the representations of further constraints on active underlay paths: there should be at least one underlay path of active (or at least unclassified) edges between the end nodes of a virtual edge (where virtual edges are also considered bidirectional). Thus, although representing data freshness optimization mechanisms is not covered to a full extent in the thesis, the concept is abstractly represented through virtual edges.

The origins of this simplistic WSN model go back to our earlier work in the field of model-based reasoning about WSN [65]. In turn, that line of research reached maturity through the work of Kluge et al. [58].

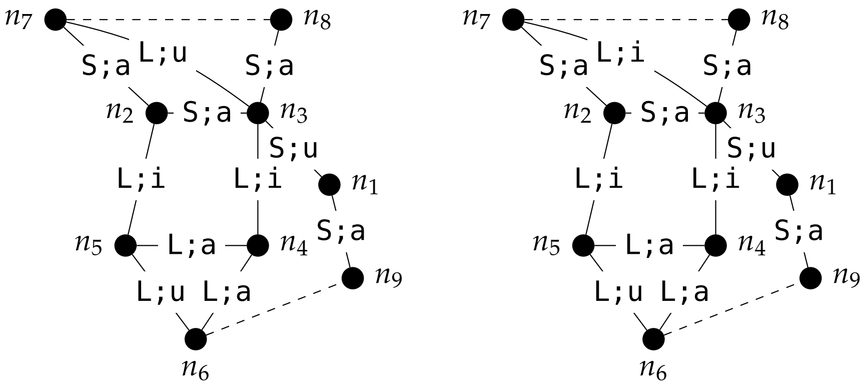
Fig. 1.2b shows our topology example enriched with edge statuses and virtual edges (dashed lines): while the virtual edge n_7n_8 is satisfied, there is no active path for n_6n_9 as there is at least one `i`-edge on each possible path.

1.3.2 *Dynamic Behavior*

In the previous subsection, we described the *static* component of our model, i.e., the way we represent a given state of a topology in an abstract manner using graph structures. In the following, we



(a) kTC Rule for Inactivating an Unclassified Edge



(b) Example Topology with Edge Statuses and Virtual Edges

(c) Example Topology after Rule Application at Triangle $n_2n_3n_7$

Figure 1.2: WSN Topology Example and kTC Edge Inactivation Rule

intuitively describe the *rule-based* specification of the network behavior itself, i.e., the *dynamics* of our model.

Indeed, as modeling describes a domain abstractly by summarizing the structure of each potential instance of that domain, dynamic behavior is necessarily described by using rules, i.e., abstract behavioral patterns applicable to each possible domain instance. *Model transformation* is a paradigm considering rule-based specifications of model dynamics as well as execution techniques for those rules. Considering models with graph-like structure just as in the case of our WSN example, *graph rewriting* provides a mathematically rigorous foundation for modeling and model transformation.[6] Graph rewriting itself is the main topic of this thesis: In Part II, we present extensions to the existing theory along with new results, motivated by the considerations and examples in the present chapter. Although we first recapitulate the theory of graph rewriting in the next chapter, we provide an intuition for rule-based network behavior in the following. Thereby, we give names to basic topology control operations (written in italic) for later reference.

The three dynamic components of a WSN *underlay* model are:

---

[6] In turn, graph rewriting can be seen as a general model transformation framework, as a large variety of system structures and semantics can be expressed abstractly in a graphic form, regardless of their original shape and implementation details.

- **Node behavior:** The movement of nodes is modeled through the changes they induce in the link topology. In particular, a *Create Link* operation represents that a new connection becomes available due to nodes getting close enough, and a *Delete Link* operation represents that two nodes lost sight of each other.
- **Link behavior:** In addition, independently of node movement, some environmental events might make it necessary to revise the status of a link. This is represented by the *Unclassify* operation, turning an active or inactive link to unclassified.
- **Topology control:** According to the triangle elimination principle in the kTC algorithm as introduced in Sect. 1.2, the *Resolve Unclassified Link* operation inactivates an unclassified long link if there is an alternative active 2-hop path between its end nodes, where both links are short.[7]

The dynamic components of a WSN *overlay* model comprise the following:

- **Administrator interaction:** Modeling the interaction of an administrator with the network comprises a *Create Virtual Link* operation for creating a new requirement by setting a virtual link, as well as a *Delete Virtual Link* operation for removing a virtual link if it is not needed anymore.
- **Routing maintenance:** For any virtual link, the *Search Active Path* operation should first look if there is a path between the end nodes consisting solely of active (or unclassified) links. Afterwards, all the unclassified edges on such a path should get activated (*Activate Unclassified on Path*) in order to avoid the risk of path inactivation. Or, if there are only paths with inactive edges, then those should get unclassified to enable their potential activation (*Unclassify on Path*).

To conclude this chapter, we give an intuitive example for specifying the topology control operation *Resolve Unclassified Link* using graph rewriting. Figure 1.2a represents the rule to express the operation mentioned above, being applicable to any topology conforming to our model. The rule consists of a *left-hand side* (LHS) and a *right-hand side* (RHS). The *application* of this rule to a given topology model consists in the following:

(i) **Match phase:** Searching an occurrence (a so-called *match*) of the LHS in the topology.

---

[7] Note that this is a rather strict variant of kTC; there are formulations where the links on the alternative path do not have to be active, if longer alternative paths are also tolerated. However, having a more relaxed resolution mechanism usually involves more obligations when reasoning about correctness.

(ii) **Deletion phase:** Removing elements from the match if the corresponding rule element is not present in the RHS.

(iii) **Creation phase:** Adding to the topology elements corresponding to elements in the RHS which are not present in the LHS.

Thus, the application of the rule in Figure 1.2a looks for a triangle with an unclassified long edge yz and two active short edges xy and xz. Then, the unclassified long edge corresponding to yz gets replaced by an inactive long edge. For example, applying the rule to the topology in Figure 1.2b results in the topology shown in Figure 1.2c: the edge n_3n_7 turned inactive as the triangle $n_3n_7n_2$ is a match of the LHS. On the contrary, even if n_5n_6 is an `L;u`-edge too, $n_5n_6n_4$ is not a match as there is no short edge present in the triangle.

1.4 CONCURRENCY: EMBRACING PLURALISM

Looking at the WSN modeling example in the previous section, we might immediately observe that nothing of our knowledge-pluralism and observational pluralism (cf. Sect. 1.1) is reflected in the abstraction: when shifting from the domain to be represented towards a rigorous representation of it, the immanent multiplicity of agents (observers *and* actors) diffuses into an *apparently* unified, in fact rather petrified, monolithic representation, capable of evolution but devoid of any of its original inherent ambiguity.

Indeed, this discrepancy has been, directly or indirectly, discovered by philosophers of pragmatism and of science on the one hand, and theoretical computer scientists on the other hand, from different perspectives and sometimes in a broader context. Classical modeling theory, rooted in the pragmatic school of philosophy, advocates a view that a model is essentially a mapping of some parts of the world (i.e., of what we called a domain) into a purpose-defined (i.e., pragmatic) representational framework [55, 97, 105].

Here, we see an example of a widely practiced engineering methodology having relevant and deep philosophical consequences—and/or vice versa. Another such branch of computer science is concurrency theory, which is in itself to a large extent a modeling activity: although proposing a formal model to capture concurrency of events is feasible (and is indeed done) without explicitly articulating an underlying reference to the human perception, such an understanding of the notion of concurrency is necessarily present. The case of Carl Adam Petri, whose namesake the famous Petri nets are, is particularly interesting: in his late work, he became explicit on his general take on knowledge and models [89]. Petri refused the widespread view of a model as an abstractional mapping, and instead emphasized that a model is essentially a *translation* of an informal (but potentially already pluralistic, diffuse, chaotic, etc.) shared understanding of a domain into a formal representation of the same knowledge base. We want to follow Petri in embracing pluralism in modeling.[8]

Catching up on the consequences of the interplay between concurrency and modeling in the work of Petri, Giorgio De Michelis even arrives to the conclusion that Petri nets "reflect the irreducible presence of the observer" [78]. Indeed, as also demonstrated by our general observations in Sect. 1.1, the observer and the boundaries of the observing capabilities stand in the center of any take on concurrency:

- From an empirical, positivistic perspective, concurrency boils down to the question of *(in)distinguishability* of events from the perspective of the observer, along the lines of the famous Carnapian notion of *empirical indifference* [15].
- From a strictly computational perspective, concurrency is observable through the notion of *mutual exclusion* of Dijkstra [31], claimed by Leslie Lamport in a Turing Lecture in 2015 to constitute the beginning of concurrency theory [66].
- From a communicational perspective as apparent also in our motivational scenario of ubiquitous computing, concurrency is living together with *distributedness*: the observation horizon is limited, the context of system components might be unknown, still, they have to be prepared for *reacting* to relevant events and, thus, never lose awareness.

This last, reactive and distributed, perspective is that of Robin Milner, whose work is heavily relied on also in our main technical contribution in Part II. Being the (co-)author of several major process calculi, in his later work, he tried to address the challenge of ubiquitous computing by a unifying, yet pluralism-embracing formal framework [80].[9] On a general note, Milner proposes to split the modeling task into the co-existing regions of *space* (i.e., where things are) and *motion* (i.e., how they interact). Thus, as a major consequence, interaction should not be affected by causal constraints introduced by space, even after translation into the model domain.

Note that this is in accordance with the paradigm shift proposal of Petri as well: for faithfully representing such systems, it is not sufficient to forge selected aspects into an abstractional mapping, but rather the model should be a reflection of the domain including its distributedness, with all its consequences.

---

[8] Note that Petri rejected a philosophical interpretation of his work and was an ardent empiricist; for him, Petri nets directly demonstrate his epistemology and no further philosophical mediation is needed. Still, formalisms like Petri nets deserve the attention of philosophers of science and of knowledge in general.

[9] This framework is that of *bigraphs*, still used for concurrent system modeling, but the details are out of scope here.

To summarize, and to emphasize the motivational focus for the work we undertake in the present thesis: There is an inherent discrepancy between graph-based modeling approaches, which build on a monolithic world-view, and describing distributed systems by concurrency. A unifying approach for graph-based modeling of such systems, with a goal of taking both aspects into account, therefore has to possess a clear vocabulary of the relevant notions. We conclude the section by enumerating those central notions of concurrency, with a unifying theoretical explanation to each, as a philosophical guideline for the rest of the thesis.

- *Parallelism:* The most basic form of interaction. Parallelism does not have to involve real distributedness or causal independence; nevertheless, some separated components are present with a capacity to interact.

- *Concurrency:* Causal independence; from an observational point of view, concurrency is the empirical indifference of events. Some components are concurrent if they are able to act such that no temporal distinction is possible; thus, there is no causal interaction between their events.

- *Distribution:* The notion of distribution has a different premise than the above ones and pervades both: here, some observer or even specifier (i.e., an observer actively designing some part of the system) has an awareness of pluralism and bases his decisions on that awareness.

- *Reactiveness:* Here, each component is designed with the explicit goal of constantly anticipating interaction with a potentially unknown context of further distributed components. This involves an adaptive operation mechanism and, ideally, a permanent functioning.

1.5 THESIS OUTLINE

The rest of the thesis is structured as follows.

The remainder of Part I, presenting introductory material, consists of Chapters 2 and 3. Part II, the main theoretical part, contains Chapters 5 to 8, as well as a concluding Chapter 9.

- Chapter 2, titled BACKGROUND, contains a short summary on the history and divisions of the literature on graph rewriting, and mainly serves as a repository of the formal foundations of a specific branch of graph rewriting we use throughout the thesis, *algebraic Double-Pushout (DPO) graph rewriting*.

- Chapter 3, titled CONTROLLED GRAPH REWRITING: STATE OF THE ART, contains again a historical summary on how and why the idea of *control* permeated rewriting approaches. In its main part, it is an analytic survey of major existing controlled graph-rewriting approaches, some rather formal, some rather practical, with the aim of distilling challenges for advancing the field of controlled graph rewriting by observing trends and deficits in state of the art; those challenges conclude the chapter.

- Chapter 4, Control Processes for Graph Rewriting, is the first part of our main contribution, a process calculus for controlled graph rewriting called RePro. This chapter describes the calculus of pure *control processes*.

- Chapter 5, RePro: A Calculus for Controlled Graph Rewriting, presents the core of our RePro definition: the combination of process-algebraic semantics and algebraic graph rewriting.

- Chapter 6, Properties of the RePro Calculus discusses the following theoretical facets of RePro:
  - Section 6.1 explores RePro from a process-theoretic perspective, focusing on equivalence notion, particularly trace equivalence and bisimulation.
  - Section 6.2 reasons about independence notions coming from both underlying theories. In particular, *parallel independence* of graph rewriting [38] and *asynchronous transition systems* [83] are considered.
  - Section 6.3, in contrast, reasons about the expressiveness of RePro as a control language.

- Chapter 7, titled Analysis of Controlled Graph-Rewriting Processes, describes a novel abstract verification approach for controlled graph rewriting. After introducing the *abstract interpretation* framework which we utilize to finitely represent infinite graph languages, we proceed as follows:
  - Section 7.1 provides an example from the WSN domain to illustrate our novel notion for graph language characterization, called *compasses*.
  - Section 7.2 formally defines compasses and their semantic equivalence classes.
  - Section 7.3 presents the main results on how to accommodate compasses into an abstract interpretation framework for RePro: we show that temporal properties of processes can indeed be verified in an abstract manner such that they are preserved by any conforming concrete RePro processes.

- Chapter 8, Discussion and Related Work, gives a thorough overview on the potentials and limitations of RePro in the larger context of related approaches.

- Finally, Chapter 9 concludes the thesis and outlines future work.

2

BACKGROUND

2.1 ORIGINS OF ALGEBRAIC GRAPH REWRITING

Graph rewriting (or *graph transformation*) originated as early as the 60s, as a product of efforts towards a rigorous theory for handling non-linear data structures of increasing complexity [98]. In fact, graph rewriting can be seen as a generalization of term rewriting to graph-like structures, i.e., for domains whose objects are not appropriately described by terms. Formally, there is a diverse range of approaches to describe and reason about rewriting of graph-like structures.

A common characteristics of all (not just graphic) rewriting approaches is their rule-based nature. A *rewriting system* or *grammar* is a collection of declarative rewriting rules, where each approach adheres to some extent to a *replacement* behavior of rules: whenever some structure is found in an object, then the rule describes how to replace it with something else, yielding a rewritten object. This is called the *application of a rule*. These general characteristics are also shared by approaches to graph rewriting.

The present thesis is built on the foundations of *algebraic graph rewriting*, specifically in its *Double-Pushout* (DPO) variant. The frames of the thesis only allow for a short enumeration of approaches other than the algebraic one.

- *Replacement grammars* are the most straightforward generalizations of term-rewriting systems: labels within the graph are divided into terminals and non-terminals, where non-terminals get replaced by rule applications with arbitrary further graph structures, such that the rule also exactly specifies the connections of the new elements to the preserved ones. There are two major sub-divisions of replacement grammars:
  - *node replacement grammars* apply the above principle to single nodes of a graph, whereas
  - *hyperedge replacement grammars* consider a more general setting and allow for the replacement of hyperedges.
- The *logical* approach coined by Bruno Courcelle captures (classes or properties of) graphs as *monadic second-order* formulas and, in turn, their evolution by the rewriting of those formulas.
- The theoretical framework of *2-structures* has been proven to be appropriate to study some phenomena in graph rewriting,

prominently those related to graph and rule composition and decomposition.

Algebraic graph rewriting has been proposed in the 70s through the endeavors of Hartmut Ehrig, Michael Pfender and Hans J. Schneider (TU Berlin) to provide a generalization of Chomsky grammars for graphs in the setting of *category theory* [36]. The major distinguishing feature, in comparison to replacement grammars, is the absence of the context-free distinction of symbols—instead, each graph structure plays an equal role in rewriting derivations. Throughout the years, there have been proposed numerous applications and cross-fertilizations in other fields of computer science [37]. It is also within this setting that parallelism and synchronization in graph rewriting as well as different flavors of concurrent graph-rewriting semantics has been considered [6], resonating to our thoughts presented in Sect. 1.4 and, thus, making the algebraic approach a particularly appropriate choice of foundation for the present thesis.

In turn, two major variants of the algebraic approach emerged, sharing their outlines but differing in formalization details and practical consequences of those.[1]

- The *Double-Pushout* (DPO) approach [36], the first one proposed by the "Berlin school", is based on a strict separation of *deletion* and *creation* within the application of a single rule (also in this strict order). Thereby, deletion is not allowed to produce non-graphic structures, e.g., by deleting a node without deleting the adjacent edges. Due to its well-studiedness in a number of aspects relevant to the thesis, we build on the DPO approach in the following and recapitulate its formal details in the upcoming Section 2.2.

- The *Single-Pushout* (SPO) approach [73] instead follows a simpler rule application schema, thus resulting in more side effects, e.g., edges being implicitly deleted along with their adjacent nodes.

From the next section on throughout the thesis, by graph rewriting we usually specifically mean the algebraic Double-Pushout (DPO) approach.

2.2 THE DOUBLE-PUSHOUT (DPO) APPROACH: RULES, DERIVATIONS, INDEPENDENCE

We introduce the fundamental definitions of graph rewriting according to the algebraic *Double-Pushout* (DPO) approach: (typed) graphs, rules

---

[1] Later on, more expressive variants of those major ones have been proposed, like the Sesqui-Pushout approach [22] or AGREE rewriting [23], whose presentation is out of scope.

and their applications, notions of parallelism: parallel rules, parallel derivations and parallel independence, as well as the Local Church-Rosser and Parallelism Theorems [38]. When defining the parallel composition of rules (parallel rules, Definition 2.3), we slightly depart from earlier definitions to fit our approach more smoothly. In this section, we make extensive use of notions from elementary category theory; for a short summary of those, we refer the reader to the appendix of [38]; a more elaborate treatment can be found in a number of textbooks, e.g., in [4].

Obviously, the definition of a *graph* plays a central role in the formal framework for graph rewriting. Particularly, as algebraic (and, thus, also DPO) graph rewriting is formalized in a categorical setting, a function-based definition works the best for defining *graph morphisms* as structure-preserving mappings between graphs. The categorical setting also allows for a neat formalization of node and edge *typing*, where any graph can be typed over a given *type graph* by providing a morphism to the type graph, whose elements, in turn, represent types. Note that typing is similar to, but more expressive than, usual graph labeling, as the type graph might also constrain type adjacency.

Definition 2.1 (Graphs and Typed Graphs)**.** *A (directed)* graph *is a tuple* $G = \langle N, E, s, t \rangle$*, where* N *and* E *are finite sets of nodes and edges, and* $s, t : E \to N$ *are the source and target functions. The components of a graph* G *are often denoted by* N_G*,* E_G*,* s_G*,* t_G*. A* graph morphism $f : G \to H$ *is a pair of functions* $f = \langle f_N : N_G \to N_H, f_E : E_G \to E_H \rangle$ *such that* $f_N \circ s_G = s_H \circ f_E$ *and* $f_N \circ t_G = t_H \circ f_E$*. A graph morphism* $f : G \hookrightarrow H$*, is a* monomorphism*, indicated by the hooked arrow* $\hookrightarrow$*, if both* f_N *and* f_E *are injective; it is an* epimorphism *if both* f_N *and* f_E *are surjective; it is an* isomorphism *if both* f_N *and* f_E *are bijective.*

Graphs G *and* H *are isomorphic, denoted* $G \simeq H$*, if there is an isomorphism* $f : G \to H$*. We denote by* $[G]$ *the class of all graphs isomorphic to* G*, and we call it an* abstract graph*. We denote by* **Graph** *the category of graphs and graph morphisms, by* $|\mathbf{Graph}|$ *the set of its objects, that is all graphs, and by* $[|\mathbf{Graph}|]$ *the set of all abstract graphs.*

The category of typed graphs *over a* type graph T *is the slice category* $(\mathbf{Graph} \downarrow T)$*, also denoted* $\mathbf{Graph}_T$ *[18]. That is, objects of* $\mathbf{Graph}_T$ *are pairs* (G, t) *where* $t : G \to T$ *is a* typing *morphism, and an arrow* $f : (G, t) \to (G', t')$ *is a morphism* $f : G \to G'$ *such that* $t' \circ f = t$*.*

As an example for a type graph, consider the type graph T_{Top} for WSN topologies (cf. Sect. 1.3) in Figure 2.1. As our WSNs have a homogeneous node set, T_{Top} has a single node. The six loop edges represent the WSN link types presented in Sect. 1.3 (`length;status`, Long-Short and active-inactive-unclassified, respectively), identified by labels for readability. Along the thesis, we will work with typed graphs, thus, when clear from the context, we omit the word "typed" and the typing morphisms. In particular, each WSN example throughout the thesis contains graphs implicitly typed over T_{Top}, with edge

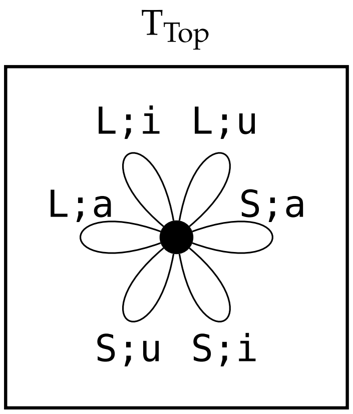


Figure 2.1: Type Graph T_{Top}

types often indicated as labels. Figure 1.2 already contained some examples of graphs typed over T_{Top} with this notation.

A (DPO) *graph-rewriting rule* is a generic description of a rewriting operation on graphs; categorically, it is a span (i.e., a pair of morphisms sharing their source) of (typed) graphs. A DPO rule (or often shortly rule in the following) follows the mechanism already informally introduced in Sect. 1.3: if we find a match (morphism) of the left-hand side of a rule to an *input* graph, then the further components of the rule describe what to delete and what to create to yield the *output* graph. In particular, a rule $(L \xleftarrow{l} K \xrightarrow{r} R)$ (with $L, K, R \in |\mathbf{Graph}|$) has L as *left-hand side* (LHS), K as *interface* and R as *right-hand side* (RHS).

For modeling a system or a domain using graph rewriting, usually, a set of rules (typed over a common type graph) is provided, declaratively describing the overall (evolutionary) system behavior. Such a collection is called a *graph-rewriting system*. Although we use our own established terminology throughout the thesis, we use some different terms for this notion in some parts, due to historical reasons and in order to maintain alignment to the literature.

First, we adopt the alternative term *graph transformation system*, and particularly the abbreviation GTS, for a graph-rewriting system, due to its omnipresence in the literature. Second, sometimes and most prominently in the historical overview in Chapter 3, the term *grammar* is used to denote a collection of rewriting rules; thus, a graph grammar is essentially a GTS (sometimes understood as including a start graph).

Formally, in addition to fixing the type graph and providing the rule spans, a GTS also gives names to the rules for easier identification.

Definition 2.2 (DPO Rule, Graph Transformation System)**.** *A* (T-typed DPO graph-rewriting) rule *is a span* $(L \xleftarrow{l} K \xrightarrow{r} R)$ *in* $\mathbf{Graph}_T$ *where* l *is mono. The graphs* L, K, *and* R *are called the* left-hand side, *the* interface, *and the* right-hand side *of the rule, respectively. A* graph transformation system (GTS) *is a tuple* $\mathcal{G} = \langle T, \mathcal{R}, \pi \rangle$, *where* T *is a type graph,* $\mathcal{R}$ *is a finite set of* rule names, *and* π *maps each rule name in* $\mathcal{R}$ *into a rule.*

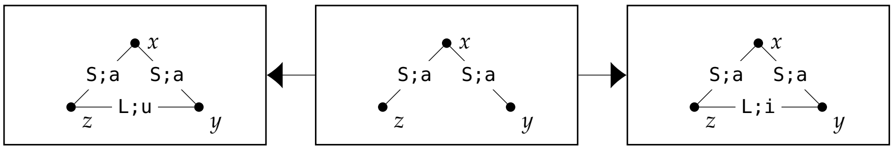


Figure 2.2: DPO Rule p_{kTC} for Resolving Unclassified Links by Inactivation

For example, the rule corresponding to the *Resolve Unclassified Link* operation in our WSN scenario, already specified in an informal notation in Figure 1.2a, is shown in Figure 2.2 as a proper DPO rule p_{kTC} (with graph names omitted).

In the following, we usually assume that $\mathcal{G} = \langle T, \mathcal{R}, \pi \rangle$ denotes an arbitrary but fixed GTS and omit explicit references to its ingredients.

In a concurrent scenario like that of WSNs, parallel composition of rules takes a central role. Although different notions have been proposed in the graph-rewriting literature for rule parallelism (cf. [6]), a usual and straightforward way of composing rules is to "glue" them together disjointly in a well-defined manner. The categorical framework allows for such a definition of the parallel rules by taking the coproduct of the corresponding spans.

Definition 2.3 (Parallel Rules)**.** *Given a GTS* $\mathcal{G} = \langle T, \mathcal{R}, \pi \rangle$*, the set of* parallel rule names $\mathcal{R}^*$ *is the free commutative monoid generated by* $\mathcal{R}$*,* $\mathcal{R}^* = \{p_1 | \ldots | p_n \mid n \geq 0, p_i \in \mathcal{R}\}$*, with monoidal operation "* $|$ *" and unit* ε*. We use* ρ *to range over* $\mathcal{R}^*$*. Each element of* $\mathcal{R}^*$ *is associated with a span in* $\mathbf{Graph}_T$*, up to isomorphism, as follows:*

1. $\varepsilon\colon (\emptyset \leftarrow \emptyset \rightarrow \emptyset)$*, where* $\emptyset$ *is the empty graph;*
2. $p\colon (L \xleftarrow{l} K \xrightarrow{r} R)$ *if* $p \in \mathcal{R}$ *and* $\pi(p) = (L \xleftarrow{l} K \xrightarrow{r} R)$*;*
3. $\rho_1 | \rho_2\colon (L_1 + L_2 \xleftarrow{l_1 + l_2} K_1 + K_2 \xrightarrow{r_1 + r_2} R_1 + R_2)$ *if* $\rho_1\colon (L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1)$ *and* $\rho_2\colon (L_2 \xleftarrow{l_2} K_2 \xrightarrow{r_2} R_2)$*, where* $G + H$ *denotes the* coproduct *(i.e., the disjoint union) of graphs* G *and* H*, and if* $g : G \rightarrow G'$ *and* $h : H \rightarrow H'$ *are morphisms, then* $g + h : G + H \rightarrow G' + H'$ *denotes the obvious mediating morphism.*

For $\rho \in \mathcal{R}^*$*, we denote by* $\langle \rho \rangle$ *the set of rule names appearing in* ρ*, defined inductively as* $\langle \varepsilon \rangle = \emptyset$*,* $\langle p \rangle = \{p\}$ *if* $p \in \mathcal{R}$*, and* $\langle \rho_1 | \rho_2 \rangle = \langle \rho_1 \rangle \cup \langle \rho_2 \rangle$*.*

Clearly, the same rule name can appear several times in a parallel rule name. Note that the effect of a parallel rule is clear by the above construction: the parallel rule span $(L_1 + L_2 \xleftarrow{l_1 + l_2} K_1 + K_2 \xrightarrow{r_1 + r_2} R_1 + R_2)$ is uniquely defined *up to isomorphism* by the coproduct construction and applications of those isomorphic rule copies would result in isomorphic outputs. However, the rule application mechanism of graph rewriting is based on concretely given graphs (cf. Definition 2.5). Therefore, we exploit a technique of "canonization" proposed in [21]

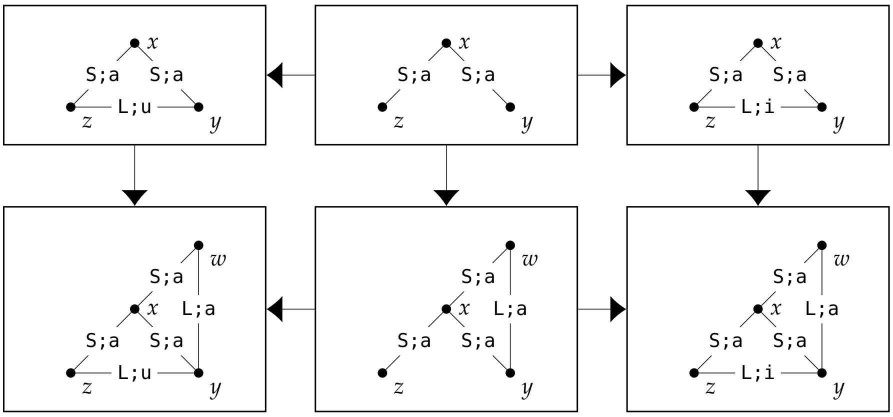


Figure 2.3: An Application of p_{kTC}

to guarantee that the parallel rule span is indeed given in terms of concrete graphs and morphism.

Given a category **C**, a *skeleton subcategory* **S** of **C** is a full, isomorphism-dense subcategory in which no two distinct objects are isomorphic [2]. Isomorphism-dense means that each object in **S** is isomorphic to some object of **C**. The existence of a skeleton subcategory of **C** follows from the Axiom of Choice.

Definition 2.4 (Canonical Graphs)**.** *The category* **CanGraph** *of canonical graphs, is an arbitrarily chosen but fixed skeleton subcategory of* **Graph**, *equipped with a functor* $Can : \mathbf{Graph} \to \mathbf{CanGraph}$ *such that* $Can \circ I = Id_{\mathbf{CanGraph}}$ *and for each graph* G, $Can(G) \simeq G$, *where* $I : \mathbf{CanGraph} \to \mathbf{Graph}$ *is the inclusion functor. For a graph* $G \in |\mathbf{Graph}|$ *we call* $Can(G)$ *its* canonical graph.

It follows also that for each isomorphic pair of graphs $G \simeq H$, $Can(G) = Can(H)$ [6]. We overload *Can* also for the analogous definition for typed graphs. In the Appendix of [6] an effective procedure is presented for constructing the canonical graph of a finite labelled graph, following the ideas proposed in [75].

Given a functor *Can* as in Definition 2.4, any parallel rule $\rho_1|\rho_2$ has $(Can(L_1 + L_2) \xleftarrow{Can(l_1+l_2)} Can(K_1 + K_2) \xrightarrow{Can(r_1+r_2)} Can(R_1 + R_2))$ as associated span, based on an arbitrary choice of concrete coproduct objects and morphisms. However, as a slight abuse of notation, we keep referring to the parallel span as $(L_1 + L_2 \xleftarrow{l_1+l_2} K_1 + K_2 \xrightarrow{r_1+r_2} R_1 + R_2)$ for the sake of notational clarity, denoting a concrete canonical representative.

A *rule application* is a formal representation of the effect of a rule described above: applying a rule ρ to a graph G consists in (i) looking for a match $m : L_\rho \to G$, and potentially (ii) deleting some parts of G as described by the morphism $L \xleftarrow{l} K$ and (iii) creating new graph elements according to the morphism $K \xrightarrow{r} R$. Categorically, a rule

$$\begin{tikzcd}
R_1 \arrow[d, "n_1"'] & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d, "k_1"] & L_1 \arrow[dr, "m_1"] \arrow[drrr, dashed, "d_1"] & & L_2 \arrow[dl, "m_2"'] \arrow[dlll, dashed, "d_2"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] \arrow[d, "k_2"] & R \arrow[d, "n_2"] \\
H_1 & D_1 \arrow[l, "g_1"'] \arrow[rr, "f_1"] & & G & & D_2 \arrow[ll, "f_2"'] \arrow[r, "g_2"] & H_2
\end{tikzcd}$$

Figure 2.4: Parallel Independence of Rule Applications $H_1 \overset{\delta_1}{\Leftarrow} G \overset{\delta_2}{\Rightarrow} H_2$

application corresponds to a pair of pushout diagrams (hence the name Double-Pushout approach).

Definition 2.5 (Rule Application, Derivations)**.** *Let G be a graph, let $\rho : (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ be a possibly parallel rule, and let m be a* match*, i.e., a (possibly non-injective) graph morphism $m : L \rightarrow G$. A* DPO rule application *from G to H via ρ (based on m) is a diagram δ as in the diagram below, where both squares are pushouts in $\mathbf{Graph}_T$.*

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(PO)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(PO)"] & R \arrow[d, "n"] \\
G & D \arrow[l, "f"'] \arrow[r, "g"] & H
\end{tikzcd}$$

In this case we write $G \overset{\delta}{\Rightarrow} H$, with denoting the underlying rule of δ by $\rho(\delta)$ and its underlying match by $m(\delta)$. We denote by $\mathcal{D}$ the set of DPO diagrams, ranged over by δ. For a rule $p \in \mathcal{R}$ and a graph G, we write $G \overset{p}{\not\Rightarrow}$ if there is no DPO diagram δ such that $G \overset{\delta}{\Rightarrow} H$ for some graph H.

A (parallel) derivation *φ from a graph G_0 is a finite sequence of rule applications $\varphi = G_0 \overset{\delta_1}{\Rightarrow} G_1 \cdots G_{n-1} \overset{\delta_n}{\Rightarrow} G_n$, via $\rho_1, \ldots, \rho_n \in \mathcal{R}^*$. A derivation is* linear *if $\rho_1, \ldots, \rho_n \in \mathcal{R}$.*

As an example, Figure 2.3 shows an application of p_{kTC} to a smaller input graph instead of our running example due to reading convenience (and omitting graph and morphism names).

Regarding the *semantics* of graph-rewriting systems, the most prevalent interpretation is based on derivations and formulated relatively to a given *start graph*: the semantics of a GTS on a graph G is the set of derivations starting from G. Operationally, this might be conceived of as a transition system where states are graphs, and transitions are valid rule applications from the given GTS to their source graph, having their output graph as target. This interpretation, even if usually remaining informal, can be rightly labeled as the *standard semantics* of graph rewriting, a term which we use throughout the thesis.

Besides semantics, another important aspect of graph-rewriting theory is the analysis of rule applications w.r.t. their connections and

$$\begin{tikzcd}
& & G \arrow[dll, Rightarrow, "\delta_1"'] \arrow[drr, Rightarrow, "\delta_2"] \arrow[dd, Rightarrow, "\delta"] & & \\
H_1 \arrow[dr, Rightarrow, "\delta_2'"'] & & & & H_2 \arrow[dl, Rightarrow, "\delta_1'"] \\
& H_{12} \arrow[r, phantom, "\simeq"] & H \arrow[r, phantom, "\simeq"] & H_{21} &
\end{tikzcd}$$

Figure 2.5: Local Church-Rosser and Parallelism Properties of Rule Applications $H_1 \overset{\delta_1}{\Leftarrow} G \overset{\delta_2}{\Rightarrow} H_2$

properties. Due to our concurrent setting, the notion of *parallel independence* is of special importance to us. Intuitively, two rule applications starting from the same graph are parallel independent if they can be sequentialized arbitrarily with isomorphic results. This property is captured categorically by the following definition [38].

Definition 2.6 (Parallel Independence). *Given two rules* $\rho_1 : (L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1)$ *and* $\rho_2 : (L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2)$ *and two matches* $L_1 \xrightarrow{m_1} G \xleftarrow{m_2} L_2$ *in a graph* G*, the resulting rule applications* δ_1 *and* δ_2 *are* parallel independent *if there exist arrows* $d_1 : L_1 \to D_2$ *and* $d_2 : L_2 \to D_1$ *such that* $m_1 = f_2 \circ d_1$ *and* $m_2 = f_1 \circ d_2$ *as in Figure 2.4, where the double squares represent the two rule applications* $H_1 \overset{\delta_1}{\Leftarrow} G \overset{\delta_2}{\Rightarrow} H_2$.

As recalled by the next result, two parallel independent rule applications can be applied in any order to a graph G obtaining the same resulting graph, up to isomorphism. Furthermore, the same graph can be obtained by applying to G the parallel composition of the two rules, at a match uniquely determined by the coproduct construction.

Proposition 2.1 (Local Church-Rosser and Parallelism Theorems [38]). *Given two parallel independent rule applications* $H_1 \overset{\delta_1}{\Leftarrow} G \overset{\delta_2}{\Rightarrow} H_2$ *with matches* $m_1 : L_1 \to G$ *and* $m_2 : L_2 \to G$*, there exist the following matches:*

(i) $m_1' : L_1 \to H_2$ *with* $m_1' = g_2 \circ d_1$ *as in Figure 2.4,*

(ii) $m_2' : L_2 \to H_1$ *with* $m_2' = g_1 \circ d_2$ *as in Figure 2.4, and*

(iii) $m : L_1 + L_2 \to G$*, where* m *is the arrow uniquely determined by* m_1 *and* m_2 *(as in Figure 2.4) through the coproduct construction*

such that there are rule applications

(i) $H_1 \overset{\delta_2'}{\Rightarrow} H_{12}$ *over* ρ_2 *based on* m_2'*,*

(ii) $H_2 \overset{\delta_1'}{\Rightarrow} H_{21}$ *over* ρ_1 *based on* m_1'*, and*

(iii) $G \overset{\delta}{\Rightarrow} H$ *over* $\rho_1|\rho_2$ *based on* m

such that H_{12}, H_{21} *and* H *are pairwise isomorphic, as shown in Figure 2.5.*

3

CONTROLLED GRAPH REWRITING: STATE OF THE ART

In this chapter, after providing some background on the concepts of *controlled graph rewriting* (CGR), we recapitulate and informally analyze a number of CGR approaches.

3.1 HISTORY OF (CONTROLLED) GRAMMARS

Grammars, i.e., sets of rewriting rules have been a major subject of study since the beginnings of computer science.[1] Famously, it was Chomsky who coined the treatment of grammars in a formal context by introducing his hierarchy of fundamental importance [16].

Although delving into the details of the Chomsky hierarchy and related research is out of scope here, there are two major approaches to the classification of grammars, distinguished if grammars are characterized by the form of their rules or by properties of the (words in the) language they describe (i.e., generate). Although the original approach of Chomsky primarily falls into the former category (with connected results regarding the latter approach), language properties also have essential importance if a grammar is used to describe, e.g., syntax. Indeed, there is a direct connection between the form of the rules used in a grammar and the arising language structure.

Another aspect of the study of grammars is the language generation process itself. Usually, this process starts from a given starting *state* and continues sequentially until some termination criteria are reached. In each step within that sequence, a rule of the grammar is *applied* to rewrite the current state, yielding a subsequent one. In turn, (a subset of) those states represent the words or elements of the language we describe. Thereby, in any formal context, it is essential to give answers to the following questions.

(i) Which arising states belong to the language we describe?

(ii) How and when might a rule be applied to a state?

(iii) Which rules are available in the next step and which rule sequences arise?

Regarding question (i), we just remark here that DPO graph rewriting (cf. Chapter 2) departs from the majority of grammatic approaches

---

[1] ...and before: the first known grammarian, Panini (4th c. BC) already made use of a rule-based description of the syntax and morphology of classical Sanskrit; some even likened his approach to the Turing machine [40].

where the alphabet is divided into terminals and non-terminals such that the absence of non-terminals is used as a natural inclusion criterion.[2] Instead, each reachable state (even those in non-terminating sequences) is part of the language associated with the grammar. The answer to question (ii) has been thoroughly described in Chapter 2.

In the present thesis, we focus on phenomena connected to question (iii) above. In traditional approaches, independently of the underlying state structure, it is usually assumed that any rule can be applied in any state for which a step is defined. *Controlled grammars* have been proposed to cope with situations where this assumption might have negative consequences in the given context and, thus, more accurate and explicit control on rule availability is desired or even necessary [28]. Despite substantial differences, each approach to controlled grammars shares a conceptual feature: the presence of an external *control grammar* over the alphabet of rule names, i.e., names associated with rewriting rules of the original grammar.

The situation is nothing different in the context of controlled graph rewriting. Already in 1978, Horst Bunke proposed to enhance graph grammars with control structures (also coining the alternative terminology *programmed* graph rewriting) [13]. Although the topic gained on relevance as graph rewriting has found its way into real-world software engineering applications, mostly within the frames of *model-driven* software engineering, there was not much work done towards a unifying formal foundation of control in graph rewriting. There is a number of competing approaches, almost each of them being complemented by practical tool support. A part of those approaches come equipped with an explicit formal semantics, while others are completely pragmatical and tooling-centered with control semantics remaining implicit. In turn, in the following section, we recall a number of relevant major CGR approaches, grouping them based on how their control semantics is conceptually given. To maintain the connection of state-of-the-art to our contribution presented in Part II, we present languages whose underlying rewriting formalism is rooted in the algebraic approach to graph rewriting—note, however, that there is a number of alternatives w.r.t. the theoretical foundations of graph rewriting; for a comprehensive overview, refer to [98].

The first group is that of approaches having a *structural operational semantics* (SOS), i.e., an inductive transition semantics formally adhering to the principles originally proposed by Plotkin [90]. The second group reflects an attitude typical for graph rewriting in general, where semantics is given in terms of a relation on graphs, consisting of input-output pairs of (sequences of) graph-rewriting steps. As a control language with an inductive syntax is also present, these *input-output*

---

[2] Note that, however, there are further, widely used approaches to graph rewriting which follow the idea of (non-)terminal symbols, such as vertex replacement grammars or hyperedge replacement grammars [98].

semantics are also provided in an inductive manner. The third group contains those practical approaches for which no formal semantic foudnation is given *a priori*, and the behavior is implicitly defined by the implementation itself. These are tools enabling to create CGR applications, thus, obviously having an underlying, implementation-defined semantics—the aim of such a distinction is just to underline that providing a formal semantic domain is not among the goals of such approaches and, indeed, meaningful practical CGR activities might arise even in the absence of a formal semantic definition.

Our selection of related work presents two of the arguably most prominent approaches in each of the above categories:

1. Structural operational semantics
    a) Graph Programs (GP, Sect. 3.2.1)
    b) PORGY (Sect. 3.2.2)
2. Input-output semantics
    a) PROGRES (Sect. 3.2.3)
    b) Graph Transformation Units (GTU, Sect. 3.2.4)
3. Implementation-based semantics
    a) HENSHIN (Sect. 3.2.5)
    b) GROOVE (Sect. 3.2.6)

3.2 EXISTING APPROACHES TO CONTROLLED GRAPH REWRITING

In this section, we recall the goals and properties of each mentioned major CGR approaches. Thereby, in each case, we elaborate on the following aspects:

- Origins and goals
- Control constructs
- WSN example: feasibility and limitations
- Definition of the control semantics
- Analysis and verification facilities

3.2.1 *Graph Programs*

ORIGINS AND GOALS. *Graph Programs* (GP) have been proposed and first implemented by Detlef Plump, Sandra Steinert and Greg Manning around 2007-2009 as “a rule-based, nondeterministic programming language for solving graph problems at a high level of abstraction, freeing programmers from handling low-level data structures” [91].

Further on, motivated by case studies conducted with the first version of the language, Plump and many others created a usability-focused extended version called GP 2 [92]. As our main interest are the fundamental language characteristics, we will concentrate on features of the original proposal.

Note that as a slight abuse of terminology, later on in Sect. 6.3, we are going to use the term Graph Programs and the abbreviation GP for a related, but smaller and differently defined language, as it was presented in a 2001 paper of Habel and Plump [49]. In contrast to GP as above, the aim of that paper was to provide a formal reference language being minimal, yet complete.

As emphasized by the above quote, the intention of GP is to provide high-level programming facilities by enabling to directly operate on graphs, which in turn might represent complicated data structures in a compact, still well-founded manner. In turn, a lot of examples and case studies published by the authors deal with algorithms for deciding complicated graph properties (e.g., being a tree, connectedness, linearity), which are not decidable by uncontrolled grammars (without changing the graph structure).

CONTROL CONSTRUCTS. The leading design principle of GP is slenderness: instead of giving the user a diversity of configuration options, GP is focusing on highly reusable essentials. A *graph program* (i.e., a program in GP) is given in a textual syntax, composed of rule names from a given set of graph-rewriting rules as well as operators representing control. The execution of a graph program requires an *input graph*; each step of the execution consists in performing a rule application as well as proceeding in the graph program, such that the choice of the rule conforms to the graph program. The execution produces an *output graph* if the program successfully terminates; otherwise, it goes to a failure state or does not terminate at all.

The following control constructs are present in the core of GP (with GP ranging over graph programs and p over rule names):

- *Non-deterministic choice:* This is the most basic form of a graph program, and also the basis for further constructs. Syntactically, a choice consists of a set of rule names, written as

  $$GP := \{p_1, p_2, \dots\}$$

  The semantics is that one of those rules gets (non-deterministically) chosen and applied to the current graph state. (The deterministic application of a single rule is indeed a special case of a choice with one member.) Graph programs of this form as GP above are called *elementary*.

- *Sequence:* Any two graph programs can be combined in a sequence, i.e., if the execution of the first graph program is successful, the output graph is passed as input to the second program in the sequence. Syntactically, graph programs GP_1 and GP_2 are joined in a sequence by the semicolon operator:

$$GP_1; GP_2$$

- *Maximal iteration:* This is the only kind of loop employed in graph programs: the execution is iterated *as long as possible*, i.e., the output of an iteration is taken as the input of the next one until an output gets produced on which the program cannot be executed successfully; then, that output graph is the result of the maximal iteration. Consequently, such a program always terminates successfully. For a program GP, its maximal iteration is written as

  $$GP!$$

  There is a limitation of the use of maximal iteration: it cannot be performed on a maximally iterated program directly, i.e., $GP!!$ is not a valid graph program.

- *If-then-else:* Conditional branching, widely known as simply *if-then-else* for the well-known programming keywords associated with it, is an essential construct in any imperative programming language. Informally speaking, when arriving at such a branching point, the decision on where the execution continues depends on the evaluation of a condition (*if*); whenever the condition is evaluated as true, the *then* branch of the program is taken, otherwise the execution continues with the *else* branch. Although the *then* and the *else* blocks are disjoint, the execution might join again at a later point and continue as a common branch.

  Specific to graph programs is how the condition is evaluated: although the condition itself is also a full-fledged graph program which gets executed when the condition is reached, its output does not influence the further program run. If the condition program might be executed successfully on the input graph, that graph is passed to the *then* branch, otherwise to the *else* branch (assuming that the condition program terminates). The well-known keywords also appear in the syntax of GP, an *if-then-else* written as

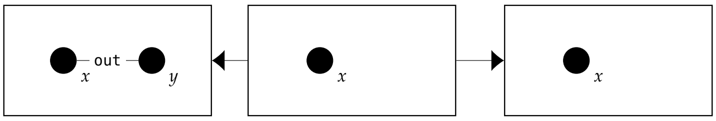
  $$\texttt{if } GP_{cond} \texttt{ then } GP_{true} \texttt{ else } GP_{false}$$

As an example to see those constructs at work, we examine a simple graph program checking if the (directed) input graph is a forest.[3]

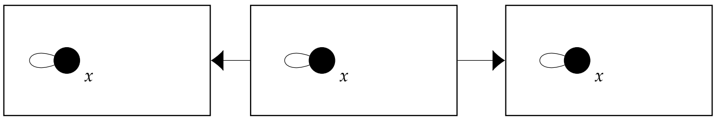
The underlying graph-rewriting system consists of three rules, depicted in Figure 3.1: `RemoveLeaf` deletes a node without outgoing

---

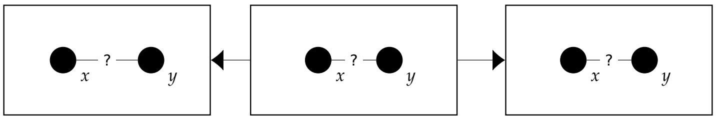
[3] Example found at `https://www.cs.york.ac.uk/plasma/wiki/index.php?title=GP_(Graph_Programs)`, last accessed Nov 11, 2018.



(a) `RemoveLeaf`



(b) `Loop`



(c) `Edge`

Figure 3.1: DPO Rules for the `CheckIfForest` Graph Program

edges, while `Loop` and `Edge` simply check the existence of a loop edge and a non-loop edge, respectively, without transforming the graph. The corresponding graph program specification is called `CheckIfForest`, having also `NotNull` as sub-program. The literals `No` and `Yes` are used as return values, as here, we are not interested in the output graph, but a property of the input graph.

$$\texttt{NotNull} := \{\texttt{Loop}, \texttt{Edge}\}$$

$$\texttt{CheckIfForest} := \{\texttt{RemoveLeaf}\}!;\ \texttt{if}\ \texttt{NotNull}\ \texttt{then}\ \texttt{No}\ \texttt{else}\ \texttt{Yes}$$

WSN EXAMPLE AND LIMITATIONS. Next, we consider the following question: To what extent are the language features of GP usable for modeling concurrent and reactive behavior, as motivated by our WSN example in Chapter 1?

The main challenges of such a complex behavior modeling task just as WSN are the following:

(i) to control the rule application sequences in order to guarantee correct algorithmic behavior,

(ii) while allowing for reactive specifications, interacting with the environment in a non-terminating manner, and

(iii) to reflect the concurrent and distributed system structure, being faithful to causal (in-)dependencies.

Arguably, GP (and each of the following major CGR approaches) lives up to challenge (i): the usual control constructs such as those presented above suffice to design imperative algorithms over graph-rewriting rules, both from a theoretical and practical perspective, while challenges (ii) and (iii) might pose a more severe problem to state-of-the-art CGR approaches such as GP.

Regarding (ii), we should consider the difference between iterativity and reactiveness. In particular, an iterated graph program with mixed-in environmental actions comes close to simulate a reactive system; however, graph programs are still *designed to terminate*. For example, as a simplified view of WSN for exposing the problem, imagine elementary graph programs GP_{node} for *node behavior* and GP_{TC} for *topology control*, respectively (cf. Sect. 1.3), as well as their intendedly reactive combination

$$(GP_{node}; GP_{TC})!$$

Then, although each execution path of this program represents an interactive and iterative interplay between a WSN and its surroundings (here, only addressing the node aspect), this graph program will end up in a *failure* state as soon as either component is unable to answer a preceding action of the other one. Instead, what we would like to see is that both system and environment are capable of *waiting* for the other side; they might get *stuck* but never terminate by failure. Such behaviors can only be captured appropriately by a notion of *parallelism* which is absent in GP.

As for challenge (iii), the issue is connected to the one raised for challenge (ii). In the above example, the necessary sequentialization of system and environment behavior introduces a causal dependency to them which is not justified conceptually. Again, causal independence of different system or model components, a central topic in concurrency theory, are only expressed faithfully by means of parallelism.

CONTROL SEMANTICS. The goal of graph programs is to either enumerate the possible output graphs for a given input, i.e., to work as a graph language *generator*, or even just to verify successful termination for a set of inputs, i.e., to work as a graph language *acceptor*. For both of those roles, it is sufficient to provide a semantic domain of relations between input graphs and output graphs. Nevertheless, for elaborating on the semantic effect of syntactic operators, Plump and Steinert propose the semantics of GP using an SOS-style inference rule system.[4]

A graph program state $\langle GP, G\rangle$ consists of a graph program term GP and a graph instance G. If the program is elementary, the semantics produces a terminal state $\langle GP_\epsilon, H\rangle$ after one step, being an output graph (i.e., an empty program GP_ϵ with a terminal graph instance H) if one of the rules are applicable to the graph. Otherwise, the program terminates in a dedicated failure state $\langle fail, G\rangle$.

The semantics of the further operators are built upon these basic steps. We defer the formal presentation of the actual semantic def-

---

[4] Note that we use the same style in our main contribution in Part II, even if in a different motivational context.

inition until Sect. 8.2, after the corresponding presentation of our own semantics, as it is instructive to examine them side-by-side more closely. We just remark here that the way semantics is handled in GP can be seen, intuitively, being "halfway" towards process algebra: an (unlabeled) transition system is provided with an expressive semantics, although without direct support for labeling-based concepts, e.g., trace analysis or Kripke structures for model checking.

ANALYSIS AND VERIFICATION. As indicated by the closing remark of the previous paragraph, while most CGR approaches employ for verification purposes some form of model checking based on transition systems, GP has a unique take on verification [94] by providing a correctness proof system inspired by Hoare logic [52].

Thereby, the correctness notion of GP is interpreted as the preservation of graph properties during program execution. More precisely, for verification purposes, programs are equipped with a *precondition* and a *postcondition*, both being properties of graphs; a program is said to be *correct* if for each input graph fulfilling the precondition, each output graph fulfills the postcondition.

The basic construction for the deductive proof system is that for any single rule and a given postcondition, it is possible to calculate the weakest precondition based on the rule-application semantics. A failing graph program has the literal *false* as single valid postcondition. All the other operators are handled by building upon those basic conclusions. The Hoare-style verification of GP provides a powerful deductive property verification procedure; however, we do not delve into further details here, as it contrasts with our own goals of analyzing graph-rewriting processes in an explicitly operational manner.

3.2.2 PORGY

ORIGINS AND GOALS. PORGY has been conceived and introduced in 2009-2010 as "a visual environment that allows users to define graphs and graph rewriting rules, and to experiment with a graph rewriting system in a visual and interactive way" [42]. To that end, PORGY offers specification, debugging, simulation and analysis facilities within the tool of the same name. However, PORGY is proposed in a slightly different context than usual for algebraic graph rewriting because of two reasons. First, PORGY directly builds on results from *term rewriting* and aims at reusing concepts in a more general graphic setting. Second, the applications of PORGY are focused on some specific scenarios such as biochemical reactions and social networks [41, 110], where the techniques of *port graph rewriting* are claimed to facilitate the specification of graph-rewriting systems. In short, a port graph is a graph with a third intermediate component: edges connect to nodes at specific points, the so-called ports. In the present thesis, we omit the presentation of port graph rewriting as those are not essential for recalling the control capabilities of PORGY.

CONTROL CONSTRUCTS. To a large extent, the constructs in PORGY resemble those of GP presented above. A control term in PORGY is called a *strategy* and is built from the following constructs (with S ranging over strategies and T over rule names):

- *Choice:* As in GP, we have the possibility to provide a set of which one element will be selected for execution. In contrast to GP and similarly to a process-algebraic choice operator, that set might consist of other arbitrary strategies (and not only single rules as in the case of GP), but also of single rules. Syntactically, the choice set is written as the parameter list of a function *ppick*, e.g.,

  $$ppick(S_1, \ldots, S_n)$$

  In newer versions of PORGY, choice has become *probabilistic*, i.e., an additional distribution function can be provided for guiding the selection process; however, this detail is irrelevant to the considerations in the thesis.

- *Sequence:* Any strategies can be combined into a sequence by the semicolon operator as in GP by writing, e.g.,

  $$S_1; S_2$$

- *Iteration:* Compared to GP, PORGY offers a larger variety of control loops: *while* loops ($while(S)do(S')$) and optional loop counters. In turn, PORGY also offers a *maximal* iteration construct, simply by setting no upper bound for the number of iterations by writing, again in a function-like syntax,

  $$repeat(S)$$

  for a strategy S.

- *If-then-else:* Just as GP, PORGY also has a conditional branching construct with the same behavior as GP:

  $$\texttt{if } S_{cond} \texttt{ then } S_{true} \texttt{ else } S_{false}$$

Note that the above constructs correspond to the language GP with slight generalizations. In addition, we have a restricted notion of *parallelism* present in PORGY, although only at the level of rules and not for arbitrary strategies.

- *Rule-level parallelism:* Any rule T in PORGY might either be a plain rule from the given graph-rewriting system or consist of the *parallel composition* of other (potentially parallel) rules:

  $$T := (T_1 \,||\, T_2)$$

  According to the rule-application semantics of PORGY, T can be applied if there are *disjoint* matches of T_1 and T_2 in the current input graph. Note that this notion is, thus, more restrictive than the usual parallel independence of DPO graph rewriting (cf. Definition 2.6), where overlapping of matches is allowed as long as the single rule applications do not destroy the other match by deleting something from it. (Note that in principle, parallel rule semantics might even agree on deleting some elements, although this is not considered in the literature.)

WSN EXAMPLE AND LIMITATIONS. Essentially, as PORGY control constructs directly reflect the capabilities of GP to a large extent, similar considerations can be met about using PORGY for the graph-based modeling of WSN. The rule-level parallelism does not provide additional expressive power w.r.t. this case study; on the contrary, explicitly parallelizing rules from different parts of the model is against the spirit of separating conceptually different behaviors.

However, there is another additional expressiveness feature which improves on the alignment of a (fictive) WSN model to our modeling requirements: as choice is available over arbitrarily complex strategies, e.g., S_{node} for node behavior and S_{TC} for topology control, the overall strategy

$$repeat(ppick(S_{node}, S_{TC}))$$

does not contain an artificial sequential dependency, even if concurrency and distribution cannot be addressed in PORGY.

CONTROL SEMANTICS. Fernandez et al. provide an SOS-style description of a transition system for PORGY programs [41]. Similarly to GP, a program configuration consists of a strategy and a graph instance. A peculiarity of the PORGY semantics, in contrast to conventional transition-system semantics with non-deterministic branching, is that here, each state is a multi-set of configurations, thus, even if there are different subsequent rule applications due to multiple matches or non-deterministic choices, each of those possibilities are summarized in a single subsequent transition-system state.

Besides the aforementioned difference, the induction logic is similar to that of GP: the basic rule of the inductive semantics generates the set of each subsequent configurations, and the rules for further operators build upon that basic rule. We discuss the semantics of PORGY in detail, along with GP and our own formal semantics (introduced in Chapter 5) in Section 8.2.

ANALYSIS AND VERIFICATION. The goal of specifying controlled graph-rewriting systems in PORGY is not so clearly articulated as in the case of GP. The creators of PORGY continuously provide typical application scenarios where their specification style excels. However, despite the aforementioned formal semantics, it is rather fuzzy what it means to "experiment" with a graph rewriting system and to what extent the semantics contributes to the results.

The PORGY tool provides an attractive user interface for specifying (port graph) rewriting rules and to generate the corresponding transition systems, displayed in an intuitive manner. Although in some cases, the goal of the user might simply be to observe a system model specification at work, unfortunately, PORGY does not go beyond that by pinning down formal verification criteria (such as Hoare properties in the case of GP).

3.2.3 PROGRES

ORIGINS AND GOALS. PROGRES has been called forth at the advent of modern-day software engineering in the 90s, where the growing demand for support of large-scale, integrated software development solutions led researchers from several fields of computer science to define a comprehensive conceptual and practical framework for this aim, within the project IPSEN [84].

As such systems heavily rely on evolving visual components with a wish for controlling that evolution, controlled graph grammars have been considered. The controlled graph grammar language and tool PROGRES has reached maturation within the IPSEN project, providing a solid technical background on the internal modeling and specification aspects. However, earlier versions of PROGRES have been proposed and discussed before, and a comprehensive formal semantics has been presented in the PhD thesis of Andy Schürr [99].

PROGRES also had a corresponding tool with both visual and textual syntaxes and comprehensive control capabilities. Unfortunately, due to maintenance issues, tool development has been discontinued; nonetheless, the PROGRES concept (and its semantic baseline) lives on in the *Story-Driven Modeling* (SDM) language as part of the tool suites FUJABA [85] and EMOFLON [70].

CONTROL CONSTRUCTS. Again, the basic constructs of PROGRES are similar to the ones above. Here, for the sake of compactness, we follow the presentation from a paper of Andy Schürr, which in turn provides a reduced syntax with the intention of neatly formalizing the semantics of the full-fledged PROGRES system [100]. Thus, although the full PROGRES language and tool has more (syntactic) constructs than presented below, those constructs shown here represent an essential core of PROGRES(and we refer to that core as PROGRES below). In the following, A and B range over PROGRES control terms.

- *Choice:* As in PORGY, the choice is unrestricted, i.e., arbitrary expressions can be combined in a choice by means of a binary operator:
$$A[]B$$
- *Sequence:* As in the above cases, by writing
$$A;B$$
- *Iteration:* As for iterative behavior, PROGRES goes farther then the aforementioned languages by considering a general *recursion* construct. To that end, control expressions can be equipped with identifiers and those identifiers can appear within terms, resulting in a recursive "call" to the expression being defined. E.g., having a definition
$$Id_A = A,$$
the term
$$A;Id_A$$
iteratively repeats the execution of A as long as possible, thus representing maximal iteration. Also in general, recursion subsumes most of the aforementioned loop constructs.

However, classical *if-then-else* is missing as a syntactical ingredient. Again, we have a more general construct instead, which factors out the idea of conditional guards as a syntactic element:

- *def/undef operators:* The $def()$ operator takes terms as parameter. E.g.,
$$def(A)$$
verifies if the execution of A is able to terminate. Then, the overall execution proceeds, but the effect of A is not considered. The *negative* guard variant is expressed by the $undef()$ operator:
$$undef(A)$$
is considered to be successful whenever A cannot be executed successfully. Now, among others, conventional *if-then-else* is easily encoded as
$$(def(A_{cond});A_{true})[](undef(A_{cond});A_{false})$$

Even more interestingly, going beyond the aforementioned languages, a notion of *general parallelism* appears in PROGRES.

- *Parallelism:* PROGRES builds on a widely used semantic assumption in concurrency theory (holding in a number of process calculi), viz. that the parallel execution of two processes has the same effect as the choice between any of their sequentualizations. In PROGRES terms,

  $$A\&B$$

  means that A and then B, or, B and then A is executed *if* either of those sequentializations can be successfully executed.

WSN EXAMPLE AND LIMITATIONS. While retaining the validity of our foregoing considerations, let us consider if those two previous features, unparalleled in GP and REPRO, increase the expressiveness and alignedness to real-world systems of our WSN model. Although conditionals do not appear as general concept in WSN, the general parallel operator might improve on behavorial expressiveness.

Having PROGRES terms A_{node} and A_{TC} for node behavior and topology control, respectively, the specification

$$A_{node} \,\&\, A_{TC}$$

admits those execution paths where node behavior does not interfere with topology control, but omits those *conflicting* ones where some step of one side would hinder further execution of the other side. Thus, although PROGRES takes a step further towards general parallelism, its parallel construct still relies on an implicit causal dependency of the parallel components.

CONTROL SEMANTICS. As indicated in Sect. 3.1 and in contrast to GP and PORGY, the semantic domain of PROGRES is that of a relation on graphs, containing pairs of *input* and *output graphs* admitted by a given PROGRES term. In addition, to distinguish potentially non-terminating executions, in those pairs, the output symbol might also be the ∞ symbol.

Although, as a consequence, the formal approach to PROGRES called *fixpoint semantics* in [100], does not involve concepts of transitions and traces, still, the semantic function computes those pairs in an inductive manner, thus, yet resembling the aforementioned semantic approaches. As expected, for a term with a single rule application, the semantic relation contains those graphs as inputs on which the rule can be applied, with the yielded output as a pair. All the further constructs are defined inductively based on that basic rule application relation.

ANALYSIS AND VERIFICATION. Although PROGRES and the IPSEN project considered verification goals as part of their comprehensive software development scenario, such facilities have never been integrated into the PROGRES tool. Beyond basic well-formedness checks, Progres mainly provided a means to specify and observe graph-rewriting systems, just as we concluded in the case of Porgy.

3.2.4 *Graph Transformation Units*

ORIGINS AND GOALS. *Graph transformation units* (GTU), often simply called transformation units in the literature, has been proposed by Hans-Jörg Kreowski and Sabine Kuske in 1994 [61] and continuously studied in a number of publications thereafter. GTU is a "framework [...] which provides syntactic and semantic means for analyzing, modeling, and structuring all kinds of graph processing and graph transformation" [62].

GTU essentially reflects the main tenet of controlled graph rewriting (i.e., control structures over a given set of rules) with an additional distinguished focus on modular control structures (hence the name transformation *units*) and independence from the underlying rule-application semantics. Regarding the latter, our whole presentation of state-of-the-art in the present chapter focuses on the control level and is, thus, inherently independent of rule-application semantics. As for modularity, although the module structure of GTU necessarily appears as part of our upcoming examples, modularity represents a conceptual dimension which is orthogonal to the outline of the thesis, thus, we do not delve into details here.

CONTROL CONSTRUCTS. GTU takes a unique approach also to the specification of control structures themselves by considering two different possibilities:

- *regular expressions* over rule names, and
- so-called *stepwise controls* being a special form of finite automata with rule application instructions as transition labels.

There is no point in recalling here the details of the syntax and semantics of regular expressions; it suffices to say that regular expressions are expressive enough to subsume all the typical core constructs of the above languages up to parallelism: *choice*, *sequence* and *iteration*. For the special case of maximal iteration, GTU uses the $*$ operator. The transitions of stepwise controls are also expressive enough to subsume those constructs.

As for parallelism, Melanie Luderer in her PhD thesis discusses three different levels of parallelism for GTU, w.r.t. both aforementioned control syntaxes [74]. Those three levels are distinguished by increasing strictness from an operational perspective. Here, parallel composition of two rule applications means that instead of interleaving, the corresponding parallel rule is applied if the original applications are independent. For a pair of GTU, we have the following composition possibilities:

- *Weak parallelism* allows for parallel composition of any simultaneous rule applications, but never requires it.
- *Proactive parallelism* performs parallel composition whenever possible. Still, the single rules might also fire otherwise.
- *Synchronous parallelism* is the strictest form, blocking the execution if a rule pair cannot be composed in parallel.

In turn, Luderer addresses both control syntaxes by introducing for each parallelism level (i) fresh regular expression operators and (ii) automata composition constructs.

Note that the distinction between those levels is orthogonal to our presentation and thus, irrelevant here; as Luderer focuses on synchronous parallelism, we also assume that case in the following and denote the synchronous parallel operator by \$.

Finally, we present a simple example to see GTU at work (with regular expression control), taken from the thesis of Luderer [74]. Given any GTU called P with rules $r_1, \ldots, r_k$, we specify a compound GTU which extends P by a parameterized step counter *countdown* to bound the number of steps.

$$\begin{aligned} &\mathit{countdown}: \\ &\qquad \mathit{initial}: gr(\mathbb{N}) \\ &\qquad \mathit{rules}: \mathit{tic} \end{aligned}$$

where $gr(\mathbb{N})$ contains all single-noded graphs with a number of loop edges, and the rule *tic* deletes a loop edge. For any GTU specification, *initial* is the set of possible start graphs and *rules* is the set of rules. Note that *countdown* contains no control expression as it has only one rule. Now, we provide the compound GTU, i.e., P with a step counter.

$$\begin{aligned} &\mathit{stepcounter}(P): \\ &\qquad \mathit{import}: P, \mathit{countdown} \\ &\qquad \mathit{initial}: I_P + gr(\mathbb{N}) \\ &\qquad \mathit{rules}: r_1, \ldots, r_k, \mathit{tic} \\ &\qquad \mathit{control}: ((r_1 \mid \ldots \mid r_k) \,\$\, \mathit{tic})^* \end{aligned}$$

where *import* takes care of the modular structure by listing the subunits, I_P contains the initial graphs of P, here composed (by a coproduct construction) with the counter graphs as in *countdown*. Now, we also have a *control* expression, which is a maximal iteration of a choice over P, synchronously composed with the step counter: synchronous parallelism guarantees that each step decreases the number of loop (i.e., counter) edges and thus, only as many steps are possible as there are loop edges in the start graph.

WSN EXAMPLE AND LIMITATIONS. By retaining the validity of the aforementioned issues, we just remark that weak parallelism takes a step towards realistic WSN modeling, reflecting in the executions the possibility that actions of different parallel components might occur simultaneously, i.e., in fact in parallel from the perspective of an observer. However, the stricter forms of GTU parallelism introduce causal dependency between components and are, thus, inappropriate for WSN modeling.

CONTROL SEMANTICS. As indicated above, the two different control syntaxes, in turn, induce two different kind of semantics:

- An *input-output* semantics is given for control terms which are *regular expressions*. This approach resembles that of Progres: taking single rule applications as basic relations, the relations for arbitrary control terms are derived inductively.
- An *operational* semantics is proposed for *stepwise control*, similarly to the semantic mechanisms of GP: a state consists of an automaton state and a graph instance, where the available transitions of the automaton describe the available rules; a step consists in advancing the automaton and transforming the graph accordingly. Still, this similarity is on an informal level and GTU with stepwise control does not possess a proper SOS semantics as we have seen for GP and Porgy.

ANALYSIS AND VERIFICATION. Unfortunately, GTU does not enjoy tool support and no reasoning techniques have been proposed beyond the scope of standard graph-rewriting theory.

3.2.5 Henshin

ORIGINS AND GOALS. In contrast to the above approaches, the Henshin project has been initiated as explicitly practical and tool-oriented, with an articulated emphasis on usability and user-friendli-ness [108]. On the one hand, regarding rule-application semantics, Henshin still stands on solid ground, directly building upon the Agg algebraic graph-rewriting engine, being in turn a direct implementation of algebraic graph-rewriting theory (cf. Chapter 2). On the other hand, when it comes to control constructs, the semantics of Henshin remains implicit and only given by the implementation itself.

Henshin is available as a plug-in of the Eclipse platform and development environment.[5] The first release (0.8.0) came out in 2011. The goal of Henshin is to "provide a state-of-the-art model transformation language for the Eclipse Modeling Framework", the latter being the

---

[5] `https://www.eclipse.org/henshin`, last accessed Dec 1, 2018.

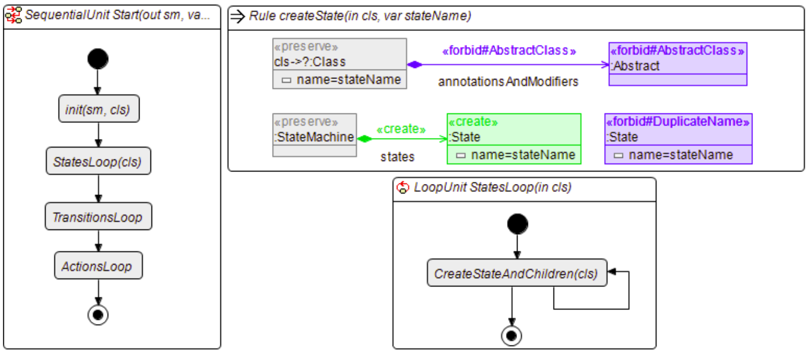


Figure 3.2: Henshin Example: Model Excerpt

underlying *Eclipse* framework for model-driven engineering. Accordingly, the core features of HENSHIN comprise the specification of model domains, transformation rules and control processes over those rules, as well as executing controlled model (i.e., graph) transformations on those models.

The syntax of HENSHIN is purely visual both for single rules and their control processes. Although we cannot illustrate the complete visual syntax of HENSHIN within the frames of present thesis, we provide a simple example for reference, found on the HENSHIN website.

The screenshot in Figure 3.2 shows an excerpt of a HENSHIN model for translating Java programs to state machines. On the left-hand side, the main initialization sequence is shown represented by a control construct called *Sequential Unit*. In turn, a Sequential Unit contains a sequence of further units, which might be single rule applications as $init(sm, cls)$ or other nested control units such as $StatesLoop(cls)$, a *Loop Unit*, whose definition is also depicted in the bottom row. This loop represents a maximal iteration of a sequence of some other rules, the first of which is depicted in the upper row of the screenshot to illustrate the rule syntax of HENSHIN. Without delving into details of the actual model semantics here, we just remark that gray elements are preserved by rule application, green ones are created, while blue elements represent a *negative application condition*, i.e., they must not be present for the rule to apply. (The red color serves for deletion, not part of this example. HENSHIN, as usual in model transformation, employs attributed graphs, hence the further inscriptions within nodes.)

CONTROL CONSTRUCTS. As seen in the above example, the control constructs of HENSHIN are called *units*. The language of units has a purely visual syntax; we omit the syntactic representation of units in the following and just describe their behavior.

- *Choice* is represented by the *Independent Unit*, allowing for nesting arbitrary further units within.

- *Sequences* of arbitrary further units are represented by a *Sequential Unit* as seen above.

- *Iteration* also appears in the usual forms: a *Loop Unit* represents a maximal iteration (cf. also Figure 3.2), while the *Iterated Unit* is a programming-like, configurable loop with counter and further termination options.

- *If-then-else* behavior is captured by a *Conditional Unit*. The last unit sort, *Priority Unit*, can be seen as a special case of nested conditional units.

WSN EXAMPLE AND LIMITATIONS. As HENSHIN provides no control-level parallelism, the same limitations hold here as in the case of GP.

CONTROL SEMANTICS. No formal semantics provided; for interpreting and using HENSHIN control units, one has to rely on their natural-language documentation and their source code.

ANALYSIS AND VERIFICATION. The *State Space Tools* module of HENSHIN provides the following features for analysis and verification purposes:

- Providing a (finite set of) start graph(s), the state space of all the possible execution paths of a given control unit can be generated and visualized.

- For analyzing properties of single states (i.e., graphs), formulas in the *Object Constraint Language* (OCL) can be specified and evaluated.

- For a more comprehensive analysis of execution paths, temporal formulas in the *modal μ-calculus* over OCL-based state predicates can be provided for state spaces; however, their evaluation is performed by binding in an external *model checker*.

Summarizing, HENSHIN provides access to the inventory of classical model checking for controlled graph-rewriting processes. A major shortcoming is the lack of a *symbolic* execution semantics, i.e., the possibility to verify properties for (potentially infinite) classes of start states.

3.2.6 GROOVE

ORIGINS AND GOALS. GROOVE is a similarly tool-oriented project as HENSHIN, "centered around the use of simple graphs for modelling the design-time, compile-time, and run-time structure of object-oriented systems".[6] Here, object-orientation should be understood in a broader sense than referring to object-oriented programming structures—this use of the term essentially corresponds to model-driven engineering.

Also, the "look and feel" of Groove is very similar to Henshin, with two major differences: (i) Groove is a stand-alone tool not relying on any external frameworks or technologies, and (ii) the syntax is mixed: whereas rules are defined visually, closely resembling Henshin, control programs possess a more conventional textual syntax. We omit the presentation of rule syntax here and just provide a simple example: a control program for a Pac-Man game[7] found on the Groove website.

```
while (gameInProgress) {
ghostMove | pacmanMove;
try (eatPellet|eatPowerPellet);
try (ghostDies|pacmanDies);
}
```

Rule names are set in italic. Rule *gameInProgress* verifies a termination condition of the game, checked in the head of an outermost *while* loop. In each turn, either Pac-Man or the ghost moves as expressed by the choice operator | between the corresponding rules in the second line. The following lines contain a *try* block, essentially corresponding to an *if* statement in other languages (with an empty *else* block here).

CONTROL CONSTRUCTS. Groove has a rich body of control constructs, represented as a mixture of operators and keywords in the textual syntax. We group them in alignment of the core constructs considered for the aforementioned languages.

- *Choice* is offered both on the level of rules as well as of arbitrary control programs, although syntactically differentiated: the | operator is for rules, whereas the `choice` keyword is for programs.
- *Sequences*, as usual, are specified using the ; operator.
- *Iteration* in its maximal form is, again, syntactically differentiated for rules and programs: # and `alap`, respectively. Moreover, for single rules, non-deterministic zero-or-more and one-or-more iteration operators ($*$ and $+$, respectively) are offered; and for programs, programming-like `while` and `do` loops.
- *If-then-else* is available in both the usual flavor, as well as in a compact `try-else` form.

WSN EXAMPLE AND LIMITATIONS. The same holds as for Henshin.

---

[6] http://groove.cs.utwente.nl/, last accessed Dec 1, 2018.

[7] https://en.wikipedia.org/wiki/Pac-Man, last accessed Jan 21, 2019.

CONTROL SEMANTICS. The same holds as for HENSHIN.

ANALYSIS AND VERIFICATION. The *Simulator* module of GROOVE offers essentially the same capabilities as the *State Space Tools* of HENSHIN. The main difference is that in the case of GROOVE, the model checker is a built-in part of the GROOVE tool suite itself.

3.3 SUMMARY AND CHALLENGES

In this section, based on the informal challenges posed by a concurrent, graph-based modeling scenario such as, e.g., wireless sensor networks (Chapter 1) and the foregoing overview of state-of-the-art approaches in the field of controlled graph rewriting (CGR), we summarize our observations in unifying statements and derive challenges which have to be addressed by our contribution in Part II of the thesis. Thereby, according to our detailed presentation of state of the art, we address the aspects of *control constructs*, particularly highlighting the issue of *parallelism and concurrency*, as well as *analysis and verification*.

CONTROL CONSTRUCTS. As it has been already accentuated by the way we presented and grouped control constructs in the case of each approach, we identify a *common baseline of core CGR constructs*. It is apparent through our survey that for both theory and practice of CGR, the following constructs are of central relevance.

- *Non-deterministic choice*,
- *sequences*,
- *maximal iteration*, and
- *if-then-else*.

For judging the further constructs, we have to distinguish between the language design aspects of *expressiveness* and *usability*. Note that even expressiveness is not meant here in the strictly formal sense used in complexity theory; regarding that, we remark that even *if-then-else* might be left out from the above list, still retaining computational completeness—a topic we cover formally in Section 6.3. Rather, we mean a kind of *functional expressiveness*, i.e., what kind of systems and algorithms might be faithfully described by the language constructs. E.g., even if not necessary for completeness, *if-then-else* cannot be encoded by the other three core constructs and, thus, represents an increase in expressiveness by adding conditionals.

Another expressiveness improvement is presented by the concept of *recursion*, only present in PROGRES among the languages we discuss. General recursion subsumes each loop type and, in addition, allows for potentially non-terminating specifications with guarded termination criteria (i.e., a condition for when to continue and when to stop).

Further constructs like parameterizable, programming-like loops and another flavors of conditionals are often considered as part of the languages, but those only address the aspect of usability, as they are expressible by using the other (core) constructs, and, thus, are considered as "syntactic sugar". We do not delve into the topic of usability in model-driven engineering here—it is indeed highly non-trivial how to approach this empirical aspect of MDE, given the significantly lower amount of evidence compared to traditional software engineering; we refer the interested reader to [3].

We conclude that while an implicit baseline is present, there has been no efforts so far for fixing a unifying (meta-)theory of control in graph rewriting. We argue that a solid theoretical framework like that of SOS semantics is inevitable to undertake such a comprehensive research endeavor; we contribute to the issue by identifying the common features of existing SOS semantics of CGR in Section 8.2.

PARALLELISM AND CONCURRENCY. Recalling our general considerations in Sect. 1.4 about the notions related to parallelism and concurrency, we identified parallelism as a minimal, yet flexible basic premise: shortly, there is no concurrency without parallelism, but parallelism does not necessarily imply concurrency, as the latter notion involves causal independence, which might not be given for, e.g., a purely syntactic parallel composition construct.

In state-of-the-art CGR approaches, some restricted form of parallelism might be present, such as the composition of disjoint matches in PORGY or arbitrary interleaving in PROGRES. However, the semantics of those notions always presupposes knowledge over the other side of parallel composition and, thus, causal dependence. Concurrency is never considered in CGR, making the current state of the technique ineligible for dealing with inherently concurrent modeling scenarios like our WSN running example. In turn, more sophisticated notions like distributedness and reactiveness build on the basic assumptions of concurrency and remained, thus, also unconsidered so far.

REASONING AND VERIFICATION. Although powerful techniques of classical model checking appear and are extensively used in CGR practice, we argue that a major shortcoming is the lack of *abstract* semantics, proposed for reasoning purposes: if such a semantics is missing, each (local or temporal) property might only be verified for a finite collection of graphs. In real-life scenarios, often more is desired: e.g., for a CGR algorithm specification, one would like to reason about correctness for each potential (or for an infinite class of) input graph(s).

	Control			Reasoning			Parallelism	
	↓	↺	Rec	Ver	MC	Abs	Eq	Int
GP	☑☑	☐☐	☐☐	☑☑	☐☐	☐☐	☐☐	☐☐
Porgy	☑☑	☑☑	☐☐	☐☐	☐☐	☐☐	☑☑	☐☐
Progres	☑☑	☑☑	☑☑	☐☐	☐☐	☐☐	☑☑	☐☐
GTU	☑☐	☑☐	☐☐	☐☐	☐☐	☐☐	☑☐	☐☐
Henshin	☐☑	☐☑	☐☐	☐☑	☐☑	☐☐	☐☑	☐☐
Groove	☐☑	☐☑	☐☐	☐☑	☐☑	☐☐	☐☐	☐☐
RePro	☐☐	☐☐	☑☐	☐☐	☑☐	☑☐	☑☐	☑☐

Table 3.1: Related Work: Feature Summary

SUMMARY Table 3.1 provides an overview of the capabilities of the aforementioned state-of-the-art CGR approaches; in addition and for a comparative reference, we also included RePro, the graph-rewriting process calculus whose details constitute the upcoming main Part II of the present thesis.

The columns of the table are organized according to the above aspects of CGR theory and practice. In particular, the first three columns consider control constructs. As every language largely coincide in terms of constructs and mainly deviate in their ways to express iteration, we focus on this aspect here: the first column ↓ stands for maximal (as-long-as-possible) iteration as discussed for GP in Sect. 3.2.1, the second column (↺) stands for the explicit syntactic presence of conditional loops known from imperative programming, such as *for*, *while* and similar,[8] while the third column (Rec) refers to syntactic recursion, i.e., the possibility of naming control units and allow those names to embedded in other control units.

The second group of three columns consider the issue of Reasoning: the fourth column (Ver) refers to *verification* approaches for graph property preservation during CGR executions, the fifth column (MC) refers to classical state-based *model checking* [5] involving temporal expressions over execution paths, and the sixth column (Abs) refers to *abstract* reasoning as discussed above, i.e., the ability to perform the aforementioned reasoning tasks over an abstract domain instead of concrete graph instances, where the elements of the abstract domain refer to potentially infinite graph classes.

The third group, Parallelism, consists of the last two columns; here, the overview is confined to the basic level of parallelism as discussed in Sect. 1.4, as existing CGR theory and practice does not go beyond

---

[8] Note that even within this single aspect, the practical expressiveness of the different languages might largely differ: some offer one-two simple conditionals, while others contain more sophisticated constructs such as loop counters and complex termination criteria.

this point towards process concurrency.[9] The basic parallelism offered by some existing approaches amounts to a concise representation of *equivalent* choices between multiple action sequences. A distinction reflected in our table is if only such a construct (seventh column - Eq) or if more is available, i.e., a way to express multiple *interleaved* execution paths, even if they are not necessarily equivalent (eighth column - Int).

Each cell of the table contains two checkboxes, where the first one is ticked if there is foundational theoretical work discussing the corresponding aspect of the language, while the second one is ticked if there is an available implementation offering a corresponding construct or principle as a feature.

Beyond the particular details of each approach, we can generally observe here that RePro does not claim to generally improve on them; rather, each of the approaches represents a different facet of the complicated nature of controlled graph rewriting, with differing focuses and, of course, necessary limitations; thus, our aim with RePro is also to offer a complementary approach to the state of the art, addressing some less explored corners of CGR theory.

Based on the foregoing observations and in motivating the upcoming Part II, we summarize our analysis of the state of the art of CGR by formulating the following open challenges, referred to as **C1-3**:

- **C1: Formal foundations of control.** Although control is omnipresent in graph-rewriting practice, no serious endeavors have been made so far to establish a core theory of CGR beyond the particularities of single approaches.
- **C2: Concurrency and reactiveness.** Different forms of parallelism have been proposed for graph-rewriting in general and for CGR in particular. However, no comprehensive approach exists to faithfully represent *concurrent* and *reactive* systems in the context of graph-based system modeling.
- **C3a: Abstract reasoning.** Existing analysis and verification techniques are either focused on graph properties and constraint preservation, or employ traditional model checking to graph-rewriting systems. In any case, analysis either relies on the existence of concrete graph instance(s) [45, 108], or is based on an abstraction over graphs, but do not consider external control [20, 24, 95, 106]. That is, there is no established means to analyze control processes itself, independently of their effects for a given graph, in an abstract way. The need for such an approach calls forth a related requirement:

---

[9] More elaborate concurrency considerations are a focused distinguishing feature of RePro as discussed at various points in the thesis.

- **C3b: Operational equivalence notions.** Advanced reasoning techniques as in **C3a** rely on well-founded process-algebraic trace concepts, and adaptive, abstract operational equivalence notions (such as bisimulation) in order to handle control process behavior with factoring out irrelevant details which would obstruct abstract reasoning.

Part II

A CALCULUS OF CONTROLLED GRAPH REWRITING

4

CONTROL PROCESSES FOR GRAPH REWRITING

This chapter, together with the subsequent one, constitute a central part of the thesis, providing an introduction to our approach for specifying concurrent *controlled graph-rewriting* (CGR) processes, called RePro (for Rewriting Processes). As demonstrated in Part I and particularly, in Chapter 1, graph-based system modeling involves a need to describe *graph-rewriting algorithms*, i.e., algorithms whose atomic operations are graph-rewriting steps. The process calculus RePro aims at providing a formally rigorous means to describe such algorithms, involving concurrency, by building upon the idea of controlled graph rewriting [91, 101].

In the setting of CGR, graph-rewriting operations retain their rule application semantics as presented in Chapter 2. In addition, external *control* is added to restrict the sequences in which graph-rewriting rules are applied.

In the majority of state-of-the-art CGR approaches, this control layer takes the form of a (programming) language whose atoms are graph-rewriting rules (cf. Chapter 3). Also in RePro, the control language is expressed by the terms of a process calculus, closely resembling the CCS calculus proposed by Robin Milner [79]. However, we do not only draw inspiration from process algebra for the syntax of RePro: in this chapter, we demonstrate how a number of well-known semantic notions and properties from process algebra are reflected in or preserved by RePro. As a major contribution, using techniques from process algebra to describe CGR processes gives us a straightforward means to specify graph-rewriting algorithms involving *concurrency* (i.e., causally independent but potentially synchronized actions) as well as *reactive behavior* (i.e., intentionally non-terminating algorithms). Thus, using RePro, we are able to describe and analyze the graph-rewriting behavior of concurrent and reactive systems, as apparent in nowadays modeling scenarios (e.g., our WSN scenario used for demonstration purposes throughout the thesis).

In particular, RePro terms represent process and graph *states* and their semantics is given by a *transition relation* between those states (defined by a system of inference rules), inducing a *labeled transition system*. For further details on the underlying definitions, we refer to the upcoming Sect. 4.2.

The state notion of RePro processes involves two components:

(1) First, *control processes* are (CCS-style) term specifications for sequences of graph-rewriting rule applications; thus, in contrast to a pure graph-rewriting system according to the standard semantics,

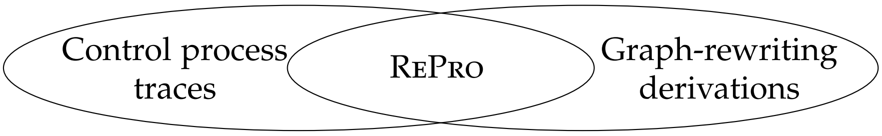


Figure 4.1: Schematic Representation of the RePro Concept

it restricts the possibilities of how rules can be sequentialized. To that end, control processes rely on the syntactical operators known from process algebra: *prefix* actions for sequencing, *choice* for non-deterministically choosing between different processes, and *parallel composition* for specifying a concurrent composition of two processes. Moreover, control processes incorporate *recursion* to enable *reactive* (i.e., recurrent and potentially non-terminating) process specifications. Furthermore, we add so-called *non-applicability conditions* as part of atomic actions to enrich graph-rewriting algorithms with conditional constructs.

(2) For representing the actual rewriting of graphs, we consider as a second component a graph *instance*, representing the concrete graph state during rewriting. Thus, the semantics of a RePro process is a transition system whose transitions correspond to rule applications, but where, in contrast to the standard semantics of graph-rewriting systems, the selection of available rules is constrained by the control component.

As for the basic concept and operating mechanism of RePro, RePro utilizes CCS-like process specifications for control and uses process algebra techniques to generate transitions (i.e., admitted actions) of control processes. In turn, traces of those actions are utilized to narrow graph-rewriting derivations according to the standard semantics of graph rewriting. This structure is schematically represented in Figure 4.1.

To summarize, RePro contributes to the following main research challenges (cf. Sect. 3.3):

- RePro provides a specification language, *control processes*, for controlled graph-rewriting processes with a canonical semantics based on labeled transition systems (cf. Definition 4.3), thus, addressing **C1**.
- RePro provides a means to specify and analyze *concurrent* processes, i.e., behavior of processes running in parallel and distributedly (**C2**). Particularly, RePro addresses process specifications with *reactive* behavior, i.e., intended non-termination with interaction of asynchronous environmental components (cf. the WSN example in Sect. 1.3).

- In RePro, existing techniques in process algebra are reused for analyzing process behavior. Furthermore, RePro facilitates the integration of graph-rewriting analysis techniques into the operational framework provided by process algebra. Particularly, equivalence of graph-rewriting processes and independence of rewriting actions can be addressed adequately (**C3b**).[1]

In the rest of this chapter, we define the syntax and semantics of RePro *control* processes along with the necessary background on labeled transition systems and process calculi.

First, we demonstrate control processes by means of our running example from the WSN domain (Sect. 4.1). Afterwards, we provide a formal, process-algebraic definition of control processes (Sect. 4.2).

4.1 WSN BEHAVIOR BY CONTROL PROCESSES

In Chapter 1, we described in an intuitive manner a graph-based model of a wireless sensor network (WSN) and, in particular, in Section 1.3.2, we gave an overview of the *dynamic* part of the model, i.e., of the WSN behavior captured and represented by the model. Now, we first demonstrate the form and functioning of our RePro *control processes* by concretizing the behavior description provided there: each behavioral component becomes a control process, i.e., a process-algebraic algorithm specification for the purpose of our WSN modeling scenario.

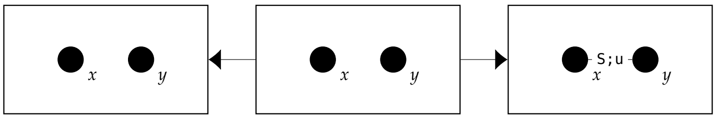
In this section, we proceed in concretizing our WSN model, introduced using a natural-language description in Chapter 1, by analyzing the required basic graph-rewriting operations and control constructs, and, thus, deriving expressiveness requirements for our control language. However, although we use a symbolic, term-based syntax which corresponds to the formal RePro syntax, we retain a certain degree of informality in this section, as we do not provide a rigorous language syntax nor a formal semantics (those are defined in the upcoming Sect. 4.2). Also, we consider DPO graph-rewriting rules as basic operations. When presenting the rules, we omit the graph names in the corresponding figures while assuming that a rule name p is associated with a span $(L_p \xleftarrow{l_p} K_p \xrightarrow{r_p} R_p)$.

In the following, we focus on underlay WSN behavior. As introduced in Sect. 1.3.2, underlay dynamics consists of three components: *node behavior*, *link behavior* and *topology control*.

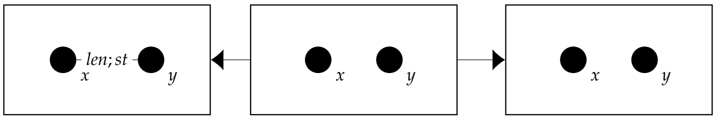
- **Node behavior:** The *Create Link* operation is succinctly representable by a DPO rule as shown in Figure 4.2a. Note that in p_{Cr}, we assume created links to be short and unclassified (i.e., new

---

[1] An adequate treatment of **C3a** is the subject of Chapter 7.



(a) p_{Cr}: Creating WSN Underlay Links



(b) $p_{Del}(len, st)$: Deleting WSN Underlay Links

Figure 4.2: DPO Rules for Link Creation and Deletion

physical links are created only within a short range, but their status is not known in the moment they appear).

Regarding the *Delete Link* operation (Figure 4.2b), in the rule name $p_{Del}(len, st)$, *len* and *st* are *name variables* that can be initialized to refer to a specific edge type as in the type graph for WSN (cf. Figure 2.1). Those variables appear also in the left-hand side of the rule. The use of name variables allows for a compact representation of families of similar rules only differing in typing, as is the case for deletion, where our specification needs a deletion rule for any link type. In any process specification containing this rule, any use of the rule name must be initialized using the value ranges $len \in \{\texttt{S},\texttt{L}\}$ and $st \in \{\texttt{a},\texttt{i},\texttt{u}\}$ (cf., e.g., P_{node} below). For example, the rule name $p_{Del}(L, a)$ belongs to a rule deleting a long active edge.[2]

We use *controlled graph-rewriting processes* to model node behavior based on those rules. As a very first example, we provide a simple process specification, using a yet informal process-algebraic syntax, for a process P_{cr} which performs a single *Create Link* operation and then terminates as indicated by the special process symbol **0**:

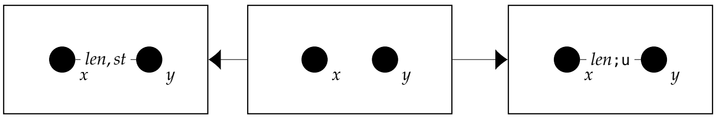
$$P_{cr} := p_{Cr}.\mathbf{0}$$

Next, we describe a more realistic process P_{node}, which applies (non-deterministically) one of the rules repeatedly. Using the same informal syntax as above, we specify this behavior by writing

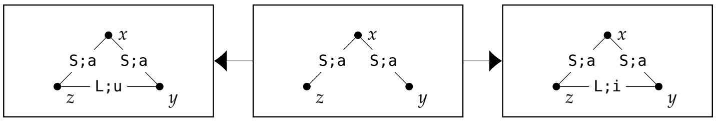
$$P_{node} := p_{Cr}.P_{node} + \sum_{len \in \{S,L\},\ st \in \{a,i,u\}} p_{Del}(len, st).P_{node}$$

---

[2] Formally, each instantiation is a separate rule name, i.e., they all appear in the rule name set $\mathcal{R}$ of their respective graph-rewriting system. Therefore, given the value ranges and the requirement that only instantiated rule names appear in processes, there is no ambiguity regarding the handling of rule names with variables.



(a) $p_{Uc}(len, st)$: Unclassifying Active or Inactive Links



(b) p_{kTC}: Resolving Unclassified Links by Inactivation

Figure 4.3: DPO Rules for Unclassifying and Resolving Links

In the above term, P_{node} is a *process name* which might appear again in process terms (as also seen here), expressing recursive behavior. The dot operator "." denotes sequentialization and the rule names appearing refer to the basic operations, i.e., graph-rewriting rule applications. The "+" operator denotes a non-deterministic choice: for example, P_{node} applies *either* p_{Cr} *or* one of the instantiations of p_{Del} whose choice is abbreviated by an indexed sum. Each of those applications in the different choice branches are followed by *recursion* as expressed by the process name P_{node} appearing in the specification: as a result, the execution of P_{node} is iterated in a loop.

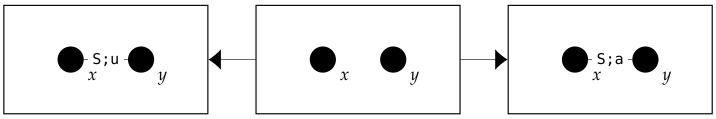
- **Link behavior:** The *Unclassify* operation is also easily representable by DPO rules as shown in Figure 4.3a, represented by a single rule span with variables (cf. Figure 4.2b); however, here we choose $st \in \{\mathtt{a}, \mathtt{i}\}$ in order to avoid identity rule applications (i.e., where the link is already unclassified beforehand).

  As in the case of node behavior, we specify link behavior with a process P_{link}, specified as

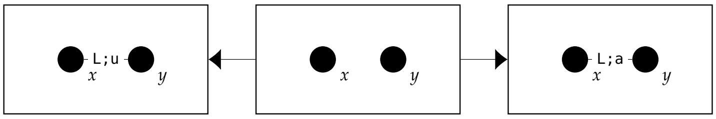
$$P_{link} := \sum_{len \in \{S,L\},\ st \in \{a,i\}} p_{Uc}(len, st).P_{link}$$

- **Topology control:** In Section 1.2, we have already presented a part of the *Resolve Unclassified Link* operation according to the kTC topology control algorithm: under what circumstances we want to *inactivate* an unclassified link. Now, we provide a proper DPO rule, p_{kTC}, for it (Figure 4.3b).

  However, topology control aims at classifying each unclassified link: if we had only p_{kTC} for this purpose, then unclassified links not being in such a triangle would remain unclassified. Therefore, we add rules p_{ActUS} and p_{ActUL} for activating (short and long) unclassified links (Figure 4.4). (Note that although we could present this behavior as a single rule with a length



(a) p_{ActUS}: Activating Short Unclassified Links



(b) p_{ActUL}: Activating Long Unclassified Links

Figure 4.4: DPO Rules for Activating Unclassified Links

variable, we present these rules separately for easier reference in later examples.)

Currently, our simple topology control mechanism is described by the process P_{TC}:

$$P_{TC} := p_{kTC}.P_{TC} + p_{ActUS}.P_{TC} + p_{ActUL}.P_{TC}$$

Note that this topology control specification is not yet satisfactory: it might non-deterministically activate an unclassified link even if this link is part of a triangle.

Finally, having provided processes for each dynamic underlay component, there is a straightforward way to specify the overall underlay behavior by letting the above processes act *in parallel*, using the *parallel composition* operator " $||$ ":

$$P_{underlay} := P_{node} \,||\, P_{link} \,||\, P_{TC}$$

In the following, we refine the simple preliminary specification above in the course of formalizing RePro and introducing new concepts in a stepwise manner.

4.2 FORMAL DEFINITION OF CONTROL PROCESSES

Control processes are terms of a *process calculus*, whose syntax is inductively given by a context-free grammar, whereas their semantics is often provided in the form of a *labeled transition system*.

SYNTAX. Regarding the syntax of RePro control processes, its basic ingredient is the set of DPO *rule names*, denoted by $\mathcal{R}$ for plain (i.e., non-parallel) rules and $\mathcal{R}^*$ for parallel rules, respectively (for any fixed GTS $\mathcal{G}$), where $\mathcal{R}$ is ranged over by p and $\mathcal{R}^*$ is ranged over by ρ (cf. also Definitions 2.2, 2.3). For rules appearing in a term, we assume that they are typed over the same type graph TG, whose identity we usually leave implicit. For instance, all the rules specified in Sect. 4.1 are typed over the WSN type graph shown in Section 2.2, Figure 2.1.

A control process term is composed of rule names as atomic actions within control structures: sequence, choice, recursion and parallel composition. We denote the set of process terms by $\mathcal{P}$, ranged over by P and Q.

Regarding the formal definition of recursion, there are several possibilities considered in process algebra literature; here, we use a solution (similar to that in [82]) based on a name set $\mathcal{K}$ of *process identifiers* (ranged over by A), which are mapped to process terms by means of *defining equations*.

The syntax of control processes $P, Q \in \mathcal{P}$ is provided by the following definition.

Definition 4.1 (Control Process Term Syntax). *Let $\mathcal{K}$ be the set of control process identifiers and $\mathcal{R}^*$ the set of parallel rules over rule set $\mathcal{R}$. The syntax of control process term $P, Q \in \mathcal{P}$ is inductively defined as*

$$P, Q ::= \mathbf{0} \mid \rho.P \mid A \mid P + Q \mid P \,||\, Q$$

where $\rho \in \mathcal{R}^*$ and a defining equation for $A \in \mathcal{K}$ is of the form $A := P$ with $P \in \mathcal{P}$. We assume that each $A \in \mathcal{K}$ has a unique defining equation.

The process $\mathbf{0}$ is the *inactive* process incapable of actions. Given a process P, $\rho.P$ represents an *action prefix*, meaning that this process can perform an action (i.e., an application of) ρ and then continue as P. As explained above, process identifiers are used to represent process terms through defining equations (of the form $A := P$), and thus might be used to describe recursive process behavior. $P + Q$ represents a process which non-deterministically behaves either as P or as Q. The *parallel composition* of P and Q, denoted as $P \,||\, Q$, is a process which might interleave the actions of P and Q or even execute them in parallel. We elaborate the meaning of this parallelism when we define the semantics of control processes in Definition 4.5. In the following, as usual, parentheses are also employed in process terms to explicitly denote operator precedence if relevant, even if parentheses are not part of the abstract syntax given above. Notice that each process specification provided in Sect. 4.1 is indeed a proper control process over those rules, according to the syntax just given.

With respect to the "standard" definition of CCS as proposed by Milner [79], in RePro, we do not consider the notions of *renaming* and *hiding*. As RePro terms are composed of rule names from a graph-rewriting system where names are mapped to concrete graph-rewriting rules, renaming as a syntactical ingredient would be counter-intuitive in our setting. As for hiding, that notion is directly connected to the concept of observability and silent actions, which we do not consider in RePro, as each graph-rewriting action has an observable effect on the graph being rewritten.

The notion of *structural congruence* (cf. also CCS [79]), denoted $\equiv$ is used to establish an equivalence of those process terms which differ syntactically only in semantically irrelevant details.[3] In particular, the structural congruence laws of RePro considers choice to be commutative and associative[4], and ensures global termination by reducing the parallel composition of inactive processes to a single **0**.

Definition 4.2 (Structural Congruence of Control Processes)**.** *Given P, Q, $R \in \mathcal{P}$, the relation $\equiv \subseteq \mathcal{P} \times \mathcal{P}$ is the least equivalence relation s.t.*

$$\mathbf{0} \,||\, \mathbf{0} \equiv \mathbf{0} \qquad P+Q \equiv Q+P \qquad (P+Q)+R \equiv P+(Q+R)$$

OPERATIONAL SEMANTICS. The operational semantics of a control process term is given by a system of SOS *inference rules*, defining a *labeled transition system* (LTS) having processes as states and labeled actions as transitions (cf. Definition 4.5 below). The LTS for control processes (abbreviated as CTS) has control process terms as states and the transitions represent (potential) rule applications, labelled by the corresponding rule name. In addition, we add to the set of action labels a special termination symbol $\checkmark$, indicating that a process reached its terminating **0**, and thus, no further (proper) actions are possible. Thus, $\checkmark$ in RePro is used to distinguish (traces of) successfully terminated processes from those which are not able to continue because of the non-applicability of rules on the actual graph instance. We first recall the standard definition of labeled transition systems and of their *traces* [1], and define the LTS of RePro control processes afterwards. We included a placeholder index X in the symbols for action sets and transition in our generic LTS definition, which we use later for distinguishing different transition systems and their equivalence relations.

Definition 4.3 (Labeled Transition System)**.** *A* labeled transition system *(LTS) is a tuple $(S, A_X, \rightarrow_X)$, where S is a set of* states, *A_X is a set of* actions *containing the distinguished element "$\checkmark$" representing successful termination, and $\rightarrow_X \subseteq S \times A_X \times S$ is a* transition relation. *As usual, we will write $s \xrightarrow{a}_X s'$ if $(s, a, s') \in \rightarrow_X$.*

Definition 4.4 (Trace, Trace Equivalence)**.** *A* trace *$t = a_1 a_2 \ldots a_n \ldots$ of a state $s \in S$ is a finite or infinite sequence of actions such that there exist states and transitions with $s \xrightarrow{a_1}_X s_1 \xrightarrow{a_2}_X \ldots$. A finite trace is* completed *if its last action a_n, and only it, is equal to $\checkmark$.*

States $s, s' \in S$ are trace equivalent *w.r.t. $\rightarrow_X$, denoted as $s \simeq^T_X s'$, if their sets of traces have the same elements.*

---

[3] Note that the choice of "laws" of structural congruence is arbitrary in the sense that each property captured might be equivalently captured on the transition level. However, our choice of those laws facilitates and simplifies the presentation of the transition relation while factoring out some technical details which are of less importance to the semantics.

[4] Although the abstract syntax above does not include parentheses, they are used in the concrete syntax here to denote sub-expressions.

$$\textsc{struct}\ \frac{P \equiv Q \qquad P \xrightarrow{\alpha} P'}{Q \xrightarrow{\alpha} P'} \qquad \textsc{pre}\ \frac{}{\rho.P \xrightarrow{\rho} P} \qquad \textsc{stop}\ \frac{}{\mathbf{0} \xrightarrow{\checkmark} \mathbf{0}}$$

$$\textsc{choice}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \textsc{rec}\ \frac{A := P \qquad P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}$$

$$\textsc{par0}\ \frac{P \xrightarrow{\rho} P'}{P \,||\, Q \xrightarrow{\rho} P' \,||\, Q} \qquad \textsc{par1}\ \frac{Q \xrightarrow{\rho} Q'}{P \,||\, Q \xrightarrow{\rho} P \,||\, Q'}$$

$$\textsc{sync}\ \frac{P \xrightarrow{\rho_1} P' \qquad Q \xrightarrow{\rho_2} Q'}{P \,||\, Q \xrightarrow{\rho_1|\rho_2} P' \,||\, Q'}$$

Figure 4.5: Inference Rules for Transitions of Control Processes

Definition 4.5 (Control Transition System). *Let $\mathcal{P}$ be the set of control process terms and $\mathcal{R}^*$ the set of parallel rule names over rule name set $\mathcal{R}$. The* control transition system *(CTS) is an LTS $(\mathcal{P}, \mathcal{R}^* \cup \{\checkmark\}, \rightarrow)$ with $\rightarrow$ being the least relation satisfying the rules in Fig. 4.5, where α ranges over $\mathcal{R}^* \cup \{\checkmark\}$ and $\rho, \rho_1, \rho_2 \in \mathcal{R}^*$.*

Now, we give an intuitive explanation for each rule in Figure 4.5:

- Rule PRE is the central rule for inferring the transition relation $\rightarrow$: any (potentially parallel) rule name $\rho \in \mathcal{R}^*$ appearing as a prefix induces a transition with that very rule name as label. Those transitions serve as the basis for the semantics of further control constructs.

- Rule STOP introduces the aforementioned special $\checkmark$-transition to denote termination, i.e., that the inactive process $\mathbf{0}$ has been reached.

- Rule STRUCT reflects semantically the intention of structural congruence (Definition 4.2): processes that are structurally congruent have exactly the same transitions.

- Rule REC provides the semantics of process identifiers: if a process identifier $A \in \mathcal{K}$ appears in a term, then the behavior is provided by the transitions of the process term P by which A is defined as in $A := P$. Importantly, this mechanism captures recursive behavior through defining equations (an alternative would be, e.g., to use a congruence notion).

- Rule CHOICE expresses the fact that process $P + Q$ can proceed as P or Q by firing any of their transitions (commutativity and associativity of $+$ is provided by STRUCT).

- Rule PAR0 and PAR1 deal with the *interleaving* behavior of parallel processes (the other behavioral aspect is *synchronization*, expressed by the next rule SYNC). In particular, in a parallel

Node behavior:

$$P_{cr} := p_{Cr}.\mathbf{0}$$

$$P_{node} := p_{Cr}.P_{node} + \sum_{len \in \{S,L\},\ st \in \{a,i,u\}} p_{Del}(len, st).P_{node}$$

Link behavior:

$$P_{link} := \sum_{len \in \{S,L\},\ st \in \{a,i\}} p_{Uc}(len, st).P_{link}$$

Topology control:

$$P_{TC} := p_{kTC}.P_{TC} + p_{ActUS}.P_{TC} + p_{ActUL}.P_{TC}$$

Overall underlay behavior:

$$P_{underlay} := P_{node} \,||\, P_{link} \,||\, P_{TC}$$

Figure 4.6: Control Process Specifications for WSN Underlay

process $P \,||\, Q$, any side (i.e., P as well as Q) might fire one of its available transitions and thus proceed, while the other side remains in place. (Although control processes are in this case symmetric and, thus, a commutative definition would be thinkable, the reason behind this specification is made clear by the notion of *asynchronicity* in Section 5.2. In the following, we might omit the number and refer to any or both of those rules as PAR if the distinction is immaterial.)

Note that these rules together with STOP induce a *global* termination, in the sense that parallel processes only have a ✓-transition if each of them has been terminated, as interleaving does not range over ✓-transitions. For instance, $P \,||\, \mathbf{0}$ has a ✓-transition only if $P \equiv \mathbf{0}$ due to the structural congruence law $\mathbf{0} \,||\, \mathbf{0} \equiv \mathbf{0}$ as in Definition 4.2.

- Rule SYNC, in contrast to PAR, induces a *synchronized* transition of P and Q in a process $P \,||\, Q$. If P and Q have some transitions labeled, e.g., by rules ρ_1 and ρ_2, respectively, then $P \,||\, Q$ has a transition labeled by the *parallel rule* $\rho_1 | \rho_2$ composed of ρ_1 and ρ_2.

As a source for concrete examples on how these rules work, we revisit the process specifications in Sect. 4.1 and recall them in Figure 4.6 for convenience.

First of all, each specification contains prefix actions (as any meaningful process would): e.g., P_{cr} has an outgoing transition $P_{cr} \xrightarrow{p_{Cr}} \mathbf{0}$ according to the rule PRE. Thereupon, according to rule STOP, this $\mathbf{0}$ has a loop transition ✓.

Our next example process, P_{node} addresses choice and recursion. Regarding the rule CHOICE, as $p_{Cr}.P_{node}$ and $p_{Del}.P_{node}$ have outgoing transitions labeled with p_{Cr} and p_{Del}, respectively, the choice in the specification of P_{node} results in a behavior where P_{node} has both of those outgoing transitions with their respective subsequent state. Here, in both cases, that subsequent state is P_{node} again, resulting in a recursive behavior; note that in our examples, we let the sets of process names (like P_{node}) and process identifiers (the elements of the set $\mathcal{K}$, cf. Definition 4.1) collapse for the sake of easier readability. As an example for structural congruence and the rule STRUCT, if we would write $p_{Del}.P_{node} + p_{Cr}.P_{node}$, this specification would have exactly the same behavior as the original one, as commutativity of choice is part of our structural congruence.

As for parallel processes, the overall specification $P_{underlay}$ has as outgoing transitions any transitions of its single processes, such as p_{Cr}, p_{kTC} and many others, according to rules PAR0 and PAR1. Furthermore, those actions might also synchronize. For example, there are transitions

$$P_{underlay} \xrightarrow{p_{Cr}|p_{UcS}} P_{underlay},$$

$$P_{underlay} \xrightarrow{p_{UcL}|p_{kTC}} P_{underlay},$$

and so on. Moreover, as the definition of parallel rules and processes admits associativity, there are also transitions with three rules composed, like

$$P_{underlay} \xrightarrow{p_{Del}|p_{UcL}|p_{ActUS}} P_{underlay}.$$

NON-APPLICABILITY CONDITIONS. Many control structures used in algorithm specifications (independently of the concrete language), such as conditional branchings (often termed *if-then-else* structures), rely on some kind of *negative conditions* for capturing if the condition on which the branching depends is violated (and, thus, the *else* branch is followed).

In our controlled setting, as we focus on algorithm descriptions over sets of graph-rewriting rules, we aim at defining a means on the *control* level for capturing the *non-applicability* of some rules. As an illustrative example, revisit the topology control specification

$$P_{TC} := p_{kTC}.P_{TC} + p_{ActUS}.P_{TC} + p_{ActUL}.P_{TC}$$

As already mentioned while introducing P_{TC}, this specification is not yet satisfactory: the actual intuition is that we should first try to eliminate triangles by applying p_{kTC}, and only if it is *not applicable* should we apply an appropriate activation rule. As a consequence, to achieve a more faithful representation of this intuition, we want to have some language ingredient to extend the specification with preconditions saying that p_{ActUS} and p_{ActUL} are applied only if p_{kTC} is not applicable.

Thus, prefix actions involving *non-applicability conditions* (NC) become pairs of a (positive) rule name to be applied and, as further condition(s), a set of further rule names which have to be non-applicable on the graph to be rewritten. Note, however, that on the control level which we consider in this section, this difference is only reflected in the shape of our actions and thus the definition of the action set, as pure control processes do not (yet) have a graph instance to work with.

We follow our above presentation pattern, i.e., we first provide the extended specification and then the formal definitions for introducing *non-applicability conditions*. The extended topology control specification P_{nTC} is given in RePro syntax as follows (extensions in bold):

$$P_{\mathbf{n}TC} := p_{kTC}.P_{\mathbf{n}TC} + (p_{ActUS}, \{\mathbf{p}_{\boldsymbol{kTC}}\}).P_{\mathbf{n}TC} + (p_{ActUL}, \{\mathbf{p}_{\boldsymbol{kTC}}\}).P_{\mathbf{n}TC}$$

Similarly, we might also extend the specification of P_{link} to $P_{\mathbf{n}link}$ by adding the requirement that unclassification should take place only if no kTC-inactivation action is possible:

$$P_{\mathbf{n}link} := (p_{UcS}, \{\mathbf{p}_{\boldsymbol{kTC}}\}).P_{\mathbf{n}link} + (p_{UcL}, \{\mathbf{p}_{\boldsymbol{kTC}}\}).P_{\mathbf{n}link}$$

The extended control transition system including non-applicability conditions, denoted nCTS, is functioning in the same way as CTS, and our definitions only require some slight adaptations. We proceed with giving those definitions, with highlighting the changes related to non-applicability conditions in bold.

An action $\gamma = (\rho, N)$ in nCTS intuitively corresponds to applying a rule $\rho \in \mathcal{R}^*$ while checking some *non-applicability conditions* $p \in N$. Formally, an action consists of a (positive) rule name (here, ρ) and a set $N = \{p_1, \ldots, p_k\}$ of (plain)[5] rule names, where for any $p_i \in N$, p_i should not be applicable in the current graph state in order to proceed. Note that while a parallel rule ρ might involve a single (plain or parallel) rule ρ_s multiple times as a component, it is sufficient for N to contain each rule name only once, as, in contrast to ρ, those names in N play the role of preconditions with a semantics invariant to multiple appearance.

First, we adapt the syntax of control processes to include non-applicability conditions, where, as an abuse of notation, we retain the same sets $\mathcal{P}$ and $\mathcal{K}$ for processes and process identifiers, respectively. (For the most part of the thesis, we consider processes with non-applicability conditions.)

Definition 4.6 (Control Process Term Syntax with NC)**.** *The syntax of a control process term $P \in \mathcal{P}$ with non-applicability conditions is inductively defined as follows, where $A \in \mathcal{K}$ and $\boldsymbol{\gamma} \in \boldsymbol{\mathcal{R}}^* \times \mathbf{2}^{\boldsymbol{\mathcal{R}}}$:*

$$P, Q ::= \mathbf{0} \;\mid\; \gamma.P \;\mid\; A \;\mid\; P + Q \;\mid\; P \,||\, Q$$

As seen in the above definition, the syntactical changes induced by non-applicability conditions modify only the syntax of prefixes and the

---

[5] It is sufficient to consider plain, i.e., non-parallel rules here, as any process specification is based on plain rules and non-applicability conditions do not get composed in parallel during parallel executions.

$$\textsc{struct}\,\frac{P \equiv Q \qquad P \xrightarrow{\alpha} P'}{Q \xrightarrow{\alpha} P'} \qquad \textsc{pre}\,\frac{}{\gamma.P \xrightarrow{\gamma} P} \qquad \textsc{stop}\,\frac{}{\mathbf{0} \xrightarrow{\checkmark} \mathbf{0}}$$

$$\textsc{choice}\,\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \textsc{rec}\,\frac{A := P \qquad P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}$$

$$\textsc{par0}\,\frac{P \xrightarrow{\gamma} P'}{P \,||\, Q \xrightarrow{\gamma} P' \,||\, Q} \qquad \textsc{par1}\,\frac{Q \xrightarrow{\gamma} Q'}{P \,||\, Q \xrightarrow{\gamma} P \,||\, Q'}$$

$$\textsc{sync}\,\frac{P \xrightarrow{(\rho_1,N_1)} P' \qquad Q \xrightarrow{(\rho_2,N_2)} Q'}{P \,||\, Q \xrightarrow{(\rho_1|\rho_2,N_1 \cup N_2)} P' \,||\, Q'}$$

Figure 4.7: Inference Rules for Transitions of Control Processes with NC

rest of the definition remains untouched. Also, the interpretation of further names and operators is the same as in Definition 4.1. Note that we still use prefixes without non-applicability sets in our examples, e.g., in the sub-process $p_{kTC}.P_{nTC}$ of P_{nTC}. In the following, p_{kTC} is a shorthand for $(p_{kTC}, \emptyset)$.

Regarding nCTS semantics, there are only slight changes needed. The only substantial concept to consider is how to extend synchronization to actions with NC. The following definition gives the LTS for nCTS (where we overload the transition symbol $\rightarrow$.

Definition 4.7 (Control Transition System with NC)**.** *The* control transition system with non-applicability conditions *(nCTS) of $\mathcal{G}$ is an LTS $(\mathcal{P}, (\boldsymbol{\mathcal{R}}^* \times \mathbf{2}^{\boldsymbol{\mathcal{R}}}) \cup \{\checkmark\}, \longrightarrow)$ with $\longrightarrow$ being the least relation satisfying the rules in Fig. 4.7, where α ranges over $(\mathcal{R}^* \times 2^{\mathcal{R}}) \cup \{\checkmark\}$, γ ranges over $\mathcal{R}^* \times 2^{\mathcal{R}}$, $\rho_1, \rho_2 \in \mathcal{R}^*$ and $N_1, N_2 \in 2^{\mathcal{R}}$.*

The rules STOP, STRUCT, REC, CHOICE, and PAR0-1 have exactly the same form and interpretation as in the definition of CTS (Definition 4.7). Their basis, rule PRE is functioning also in the same way as in CTS, with considered actions extended to include non-applicability conditions.

As for synchronization (rule SYNC), beyond building the corresponding parallel rule, we also synchronize the non-applicability conditions by taking as a new NC set the union of the sets on both sides. Consequently, the synchronized parallel rule $\rho_1|\rho_2$ can fire if none of the involved conditions are violated.

Note that synchronization in nCTS as above might result in actions for which there exists no graph to apply on; e.g., if a rule being in $\langle\rho_1\rangle \cup \langle\rho_2\rangle$ is also in $N_1 \cup N_2$, or even if there is a rule in $\langle\rho_1\rangle \cup \langle\rho_2\rangle$ whose applicability necessarily implies that of another rule which is $N_1 \cup N_2$. In our original paper [63], to capture at least the former case, which can be addressed on the level of rule names, we require a further premise for SYNC as follows:

$$((\langle\rho_1\rangle \cap N_2) \cup (\langle\rho_2\rangle \cap N_1) = \emptyset \qquad (*)$$

The inference rule which is identical to SYNC, but has in addition the above premise, is called SYNC*. This rule expresses that the positive rules should not be excluded by the non-applicability conditions of the other side. The variant of nCTS where SYNC is replaced by SYNC* is called nCTS*.

As an example for nCTS* terms, the process P_{nTC} has as outgoing transition labels not only the rule names; instead, we have transition labels like $(p_{UcS}, \{\mathbf{p}_{\boldsymbol{kTC}}\})$. Also, if we update the specification of the overall underlay behavior by replacing P_{TC} with P_{nTC} and P_{link} with P_{nlink}, then the additional condition for SYNC* disables some synchronized actions which are allowed in the original specification: there is no synchronized action over p_{kTC} on the one side and $(p_{UcS}, \{\mathbf{p}_{\boldsymbol{kTC}}\})$ or $(p_{UcL}, \{\mathbf{p}_{\boldsymbol{kTC}}\})$ on the other side, respectively, because the premise of SYNC* forbids a positive rule to appear in the NC set of the action it should synchronize with. In the following, however, we present our results using nCTS, and explicitly refer to the above distinction where it is necessary.

We remark already here that adding non-applicability conditions also take a direct influence on the expressiveness of our *control language* (as in contrast to enriching the rule application semantics with negative conditions, cf. also Sect. 8.2). In particular, having the ability to include non-applicability conditions on the control level gives us the expressive power to formulate branching and loop termination criteria, which makes our control language *computationally complete* according to the notion of Habel and Plump [49], who interpreted computational completeness for the specific setting of controlled graph rewriting, also providing a minimal reference language. We elaborate on this topic in Sect. 6.3. In this way, we will be able to relate our language both to the standard semantics of graph-rewriting systems and to the reference language of Habel and Plump by proposing an encoding.

Also, in the discussion following the complete presentation of REPRO (Sect. 8.2), we review the alternative notions to negative conditions from the graph-rewriting literature, along some observations on how they relate to our non-applicability conditions.

5

REPRO: A CALCULUS FOR CONTROLLED GRAPH REWRITING

In this chapter, we present the overall concept of RePro. First, recall our schematic representation in Figure 4.1: RePro utilizes process algebra techniques for generating control traces (cf. the previous Chapter 5), which traces in turn describe the allowed sequences for graph-rewriting derivations. Now, having the necessary underlying definitions, we concretize our RePro concept representation by providing pointers to the concrete definitions meant, as shown in Figure 5.1.

In particular, the upcoming definition of the RePro *transition system* (Definition 5.1) in this section is based on the previously defined control process transitions, combined with graph-rewriting rule-application semantics: whenever a transition is made available by a control process such that the corresponding rule is applicable on the actual graph, a RePro transition arises, where both components proceed: the control process continues and the graph gets rewritten by a rule application.

Again, we first demonstrate RePro processes using our WSN scenario, building on the example control processes provided in Sect. 4.1. Afterwards, we provide a formal definition of RePro.

5.1 WSN TOPOLOGY SIMULATION BY REPRO PROCESSES

Using control processes, we are able to describe the underlay behavior of our graph-based WSN model. Using RePro, we *simulate* the behavior of a topology involving both environmental events and proactive topology control: as graph instances, we consider typed graphs over the topology type graph TG_{Top} in Figure 2.1.

As a very first example, consider the simple control process $P_{cr} := p_{Cr}.\mathbf{0}$, which creates an unclassified underlay link between two existing nodes (i.e., the left-hand side of p_{Cr} consists of two nodes). Now, as a graph instance, consider the topology graph G_2 in Figure 5.2a, consisting of only two nodes without any links. The RePro process for link creation takes the following form:

$$(P_{cr}, G_2)$$

and in Figure 5.2c, we depict a part of the RePro LTS starting from that process.

We interpret the process (P_{cr}, G_2) as follows: one of the rules available in the control state P_{cr} is applied on the graph G_2, the subsequent state being the next control state composed with the rewritten graph state. Here, only p_{Cr} is available: consider the match

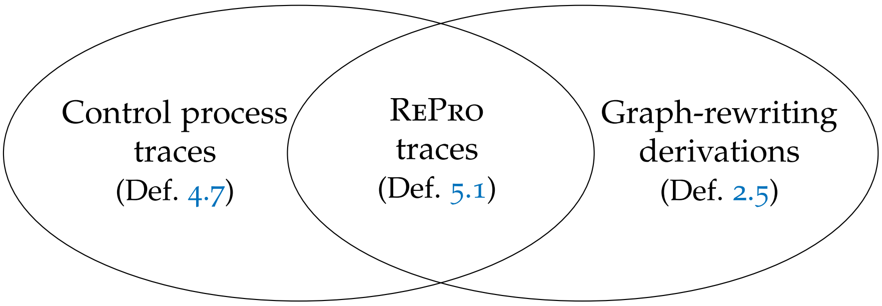


Figure 5.1: RePro Concept with Definition References

$m : L_{Cr} \to G_2$ which maps node x to node 1 and node y to node 2 (cf. Figures 4.2a and 5.2a); the corresponding DPO rule application is denoted by δ (cf. Definition 2.5). The resulting graph G_2' of the rule application $G_2 \overset{\delta}{\Rightarrow} G_2'$ is depicted in Figure 5.2b.

Note that p_{Cr} has four distinct matches and even if we restrict ourselves to injective matches (which is meaningful for p_{Cr} as we do not want to create loop links), there are two different ones: $m := (x \mapsto 1, y \mapsto 2)$ as above, and $m' := (x \mapsto 2, y \mapsto 1)$ with corresponding rule applications δ and δ'. The rule application $G_2 \overset{\delta'}{\Rightarrow} G_2''$ produces a graph G_2'' which is isomorphic to G_2', but not identical with it on the concrete graph level. Indeed, given the standard semantics of graph rewriting, this observation holds already for one match: as the DPO output graph construction is unique only up to isomorphism, there are infinitely many isomorphic result graphs (and thus, RePro transitions) for a single match on a given graph instance. Note that this phenomenon does not cause any problem in general, as process calculi allow for infinite branching.

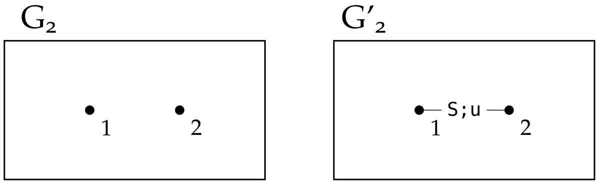
The transition label $(p_{Cr}, \emptyset)$ identifies the action performed during the transition from (P_{cr}, G_2) to $(\mathbf{0}, G_2')$. Here, we use actions as in nCTS and denote by $\emptyset$ that the action did not have non-applicability conditions. The process $(\mathbf{0}, G_2')$ has $\mathbf{0}$ as control process, therefore, it has an outgoing (loop) transition $\checkmark$.

Of course, having a topology graph G with more nodes, there are more outgoing transitions of (P_{cr}, G) leading to non-isomorphic results.

As a more elaborate example, let us consider the last specification of topology control, P_{nTC}, which already involves non-applicability conditions:

$$P_{nTC} := p_{kTC}.P_{nTC} + (p_{ActUS}, \{p_{kTC}\}).P_{nTC} + (p_{ActUL}, \{p_{kTC}\}).P_{nTC}$$

Furthermore, as a graph instance G_{ex}, consider the example topology presented in Chapter 1, depicted again in Figure 5.3a. Then, the RePro process (P_{nTC}, G_{ex}) has a large number of subsequent transitions, some of them depicted in Figure 5.3c.



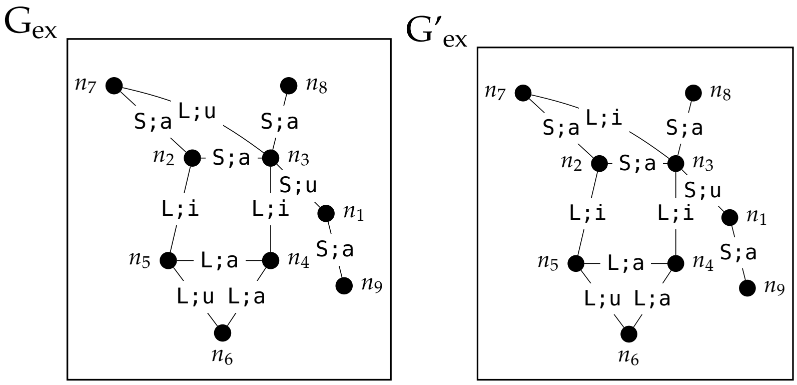
(a) Example Topology Graph G_2

(b) Example Topology Graph G_2'

$$\begin{tikzcd}
(P_{cr}, G_2) \arrow[r, "(p_{Cr}, \emptyset)"] & (\mathbf{0}, G_2') \arrow[loop below, "\checkmark"]
\end{tikzcd}$$

(c) RePro LTS Excerpt from (P_{cr}, G_2)

Figure 5.2: Graph Instances and LTS Excerpt for RePro Process (P_{cr}, G_2)



(a) Example Topology Graph G_{ex}

(b) Example Topology Graph G_{ex}'

$$\begin{tikzcd}
& (P_{nTC}, G_{ex}) \arrow[d, "(p_{kTC}, \emptyset)"] & \\
& (P_{nTC}, G_{ex}') \arrow[dl, "(p_{ActUS}, \{p_{kTC}\})"'] \arrow[dr, "(p_{ActUL}, \{p_{kTC}\})"] & \\
(P_{nTC}, G_{ex}'') & & (P_{nTC}, G_{ex}''')
\end{tikzcd}$$

(c) RePro LTS Excerpt from (P_{nTC}, G_{ex})

Figure 5.3: Graph Instances and LTS Excerpt for RePro Process (P_{nTC}, G_{ex})

Here, e.g., G'_{ex} is the graph depicted in Figure 5.3b after applying p_{kTC} at match $(x \mapsto n_2, y \mapsto n_3, z \mapsto n_7)$, G''_{ex} is the same as G'_{ex} with $n_1 n_3$ changed to active, while G'''_{ex} is the same as G'_{ex} with $n_5 n_6$ changed to active.

5.2 FORMAL DEFINITION OF REPRO

We start by defining a transition system for RePro processes based on the definition of (n)CTS (cf. Definition 4.5 and 4.7). Furthermore, for the graph component of a RePro process, RePro utilizes the rule application semantics as introduced in Chapter 2. We recall that $G \overset{\delta}{\Rightarrow} H$ if there is an application of rule ρ on input graph G at match m, producing H as output graph, where δ is the rule application diagram corresponding to ρ applied at match m. For further details, refer to Chapter 2.

A RePro transition represents the simultaneous transition of both the control process and the graph instance: a rule available by the control process specification is applied to the current graph instance (with a non-deterministically chosen match). Accordingly, our RePro transition system is derived from both the control process and the rule application semantics. However, regarding the labeling of RePro transitions, we have various choices depending on how much information do we include in labels. First, we define a variant which inherits the labeling of control processes. However, those labels do not identify uniquely the rewriting step performed; therefore, we also provide a semantics where the label is enriched by the complete DPO diagram uniquely representing the rule application.

Note that although we provide each RePro transition system definition on top of nCTS, any variant can be easily defined on top of CTS analogously. As for the notation of different transition relations, notice that the transition arrow of CTS does not have any distinguishing lower index, whereas, from now on, each transition arrow introduced has a distinguishing lower index.

Definition 5.1 (RePro Transition System)**.** *The* RePro *transition system (RTS) is an LTS* $(\mathcal{P} \times |\mathbf{Graph}_T|, (\mathcal{R}^* \times 2^{\mathcal{R}}), \rightarrow_R)$ *with* $\rightarrow_R$ *being the least relation satisfying the following rules, where* $\rho(\delta) = \rho$*:*

$$\textsc{mark}\,\frac{P \xrightarrow{(\rho,N)} P' \quad G \overset{\delta}{\Rightarrow} H \quad \forall p \in N : G \overset{p}{\not\Rightarrow}}{(P,G) \xrightarrow{(\rho,N)}_R (P',H)}$$

$$\textsc{stop}\,\frac{P \xrightarrow{\checkmark} P'}{(P,G) \xrightarrow{\checkmark}_R (P',G)}$$

Given an available transition of control process P with action (ρ, N) to subsequent state P', rule MARK verifies if (1) there is an application

δ with $\rho(\delta) = \delta$ on graph G at some match $m(\delta)$ with result graph H, such that (2) none of the rules in the NC set N are applicable on G. If those premises are fulfilled, then there is a RePro transition from (P, G) to (P', H). Rule STOP serves for lifting the nCTS ✓ termination symbol to the RePro transition system.

Now, we enrich the labeling by adding explicitly the DPO diagram belonging to the rule application performed. We assume a set $\mathcal{D}$ of DPO diagrams (accordingly typed), ranged over by δ.

Definition 5.2 (RePro DPO Transition System)**.** *The* RePro DPO transition system *(RDTS) is an LTS* $(\mathcal{P} \times |\mathbf{Graph}_T|, (\mathcal{R} \times \mathcal{D} \times 2^{\mathcal{R}}) \cup \{\checkmark\}, \rightarrow_D)$ *where* $\rightarrow_D$ *is the least relation satisfying the following rules, where* $\rho(\delta) = \rho$*:*

$$\textsc{mark}\,\frac{P \xrightarrow{(\rho,N)} P' \quad G \overset{\delta}{\Rightarrow} H \quad \forall p \in N : G \overset{p}{\not\Rightarrow}}{(P,G) \xrightarrow{(\rho,\delta,N)}_D (P',H)}$$

$$\textsc{stop}\,\frac{P \xrightarrow{\checkmark} P'}{(P,G) \xrightarrow{\checkmark}_D (P',G)}$$

6

PROPERTIES OF THE REPRO CALCULUS

In this chapter, after having defined RePro, we turn to its study: we examine the properties of RePro both as a process calculus and as a control language, in the light of the central questions of the respective theories. According to this program, we investigate the following facets of RePro:

- *Equivalence (Sect. 6.1):* It is among the aims of a process calculus to provide a minimalistic, abstract description of process behavior on a rigorous algebraic basis. As a consequence, some of the possible syntactic entities (i.e., terms) within a calculus might convey the same semantics under a particular interpretation. The topic of process *equivalence* is concerned with defining the most important such interpretations and the resulting process equivalence classes. As the structure of RePro largely resembles that of CCS [79] (cf. also Chapter 5), we focus here on notions primarily proposed and used in that context.

- *Independence (Sect. 6.2):* The independence of actions or events plays a central role in any operational theory, be it process algebra or rewriting. Roughly, independence means that a simultaneous occurrence of actions does not involve any causal correspondence or consequence. Here, we consider two complementary approaches to address independence in the context of RePro: (1) the so-called *direct approach* (Sect. 6.2.1) focuses on transferring independence notions of graph rewriting to the RePro setting, while (2) the *asynchronous approach* (Sect. 6.2.2) aims at the adoption of the so-called asynchronicity notion, originally proposed for CCS [83], for the purpose of RePro.

- *Expressiveness (Sect. 6.3):* Here, in slight contrast to the above points, RePro is primarily studied as a language: a syntax and semantics capable of describing computations, i.e., *input-output* functions. In a broader context, the notion of language expressiveness might have different interpretations. Complete expressiveness is often defined by a notion of *computational completeness*, widely reasoned about in terms of universal *Turing machines* [109]. In the specific setting of controlled graph rewriting, Annegret Habel and Detlef Plump have carefully adopted the notion of computational completeness [49]. Here, an additional consideration to make lies in the nature of control being specified *over* a given graph-rewriting system, i.e., a set of graph-rewriting rules. Consequently, completeness captures if a graph-rewriting control language is capable of expressing each computation of a fixed graph-rewriting system. We investigate the expressiveness of RePro using the reference framework of Habel and Plump [49].

6.1 EQUIVALENCE

First, we elaborate on how well-known equivalence notions of process algebra literature apply for RePro. In that context, one of the most usual ways of considering process equivalence is by observing and comparing traces (cf. Definition 4.4). We recall that a trace of a process state is a sequence of transition labels starting from that state, and two processes are trace equivalent if they have the same sets of traces.

First, we establish a connection between the two different RePro labelings RTS and RDTS as in Section 5.2 (Definitions 5.1 and 5.2). In particular, the following statement says that if two RePro processes have the same set of RDTS traces, then the same holds for their RTS traces.[1]

Proposition 6.1. *Given* $P, Q \in \mathcal{P}$ *and* $G, H \in |\mathbf{Graph}_T|$, *the following holds:* $(P, G) \simeq_D^T (Q, H)$ *implies* $(P, G) \simeq_R^T (Q, H)$.

Proof. There is an obvious forgetful function f from (sets of) RDTS traces $(\rho, \delta, N) \dots$ to (sets of) RTS traces $(\rho, N) \dots$ by dropping δ from each transition label in the trace (with $f(\checkmark) = \checkmark$. Thus, denoting the trace set of (P, G) by T_P^G and that of (Q, G) by T_Q^G, $T_P^G = T_Q^G$ implies $f(T_P^G) = f(T_Q^G)$ implies $(P, G) \simeq_R^T (Q, H)$.

□

Moreover, RDTS traces also prove to be useful in relating RePro to the standard semantics of graph rewriting, by allowing for a bijective correspondence between RDTS traces on the one hand, and derivations as in Definition 2.5 on the other hand.

In particular, RDTS traces correspond to the parallel derivations of Definition 2.5: Proposition 6.2.1 states that every RDTS trace naturally determines a(n underlying) parallel derivation; Proposition 6.2.2 provides a process definition by recursive choice, which has a successful trace for each linear derivation starting from a given graph, while Proposition 6.2.3 does the same for parallel (i.e., not necessarily linear) derivations by providing a recursive process allowing for arbitrary parallel composition of the rules as well. Note that the process specification $Q_{\mathcal{R}}$ in this last point also subsumes sequential composition, due to the semantics of $||$ which also allows for interleaving.

Proposition 6.2 (Traces and Derivations).

---

[1] The inverse direction does not hold: given (P, G) and (Q, H), even if both having just a single trace $\rho\checkmark$, the rule applications corresponding to that single ρ-steps are different, as they are on different graphs. However, the original statement implies $G = H$ if the traces are not empty—assuming that, the inverse implication also holds.

1. *Given a* RePro *process* (P, G)*, each of its traces uniquely identifies an* underlying parallel derivation *starting from G. In particular, if* $(\rho_1, \delta_1, N_1) \cdots (\rho_n, \delta_n, N_n)$ *is a trace of* (P, G)*, then* $\delta_1; \cdots ; \delta_n$ *is its underlying derivation.*

2. *Let* $P_{\mathcal{R}}$ *be the control process defined as follows:*
$$P_{\mathcal{R}} = \mathbf{0} + \textstyle\sum_{p \in \mathcal{R}} p.P_{\mathcal{R}}$$
Then for each graph G and for each linear derivation φ *starting from G there is a successful trace of* $(P_{\mathcal{R}}, G)$ *such that* φ *is its underlying derivation.*

3. *Let* $Q_{\mathcal{R}}$ *be the control process defined as follows:*
$$Q_{\mathcal{R}} = \mathbf{0} + \big((\textstyle\sum_{p \in \mathcal{R}} p.\mathbf{0} + \varepsilon.\mathbf{0}) \mid\mid Q_{\mathcal{R}}\big)$$
Then for each graph G and for each parallel derivation φ *starting from G there is a successful trace of* $(Q_{\mathcal{R}}, G)$ *such that* φ *is its underlying derivation.*

Proof.

1. Extracting $\delta_1; \cdots ; \delta_n$ from the trace provides a derivation as in Definition 2.5 indeed.

2. Let $\varphi = G \overset{\delta_1}{\Rightarrow} G_1 \cdots G_{n-1} \overset{\delta_n}{\Rightarrow} G_n$ be a linear derivation via $p_1, \ldots, p_n \in \mathcal{R}$. We proceed by induction on n. If $n = 0$, then transition $(P_{\mathcal{R}}, G) \xrightarrow{\checkmark}_D (\mathbf{0}, G)$ can be inferred using rules REC, CHOICE, and STOP. Therefore $(P_{\mathcal{R}}, G)$ has the successful trace $\checkmark$, which has an empty underlying derivation, as desired.

   If $n > 0$, it is sufficient to show that $(P_{\mathcal{R}}, G) \xrightarrow{(p_1, \delta_1, \emptyset)}_D (P_{\mathcal{R}}, G_1)$, and conclude by applying the induction hypothesis to $G_1 \overset{\delta_2}{\Rightarrow} G_2 \cdots G_{n-1} \overset{\delta_n}{\Rightarrow} G_n$, which has length $n - 1$. Indeed, $P_{\mathcal{R}} \xrightarrow{(p_1, \emptyset)} P_{\mathcal{R}}$ using REC and CHOICE (recall that p_1 is a shorthand for $(p_1, \emptyset)$), because $G \overset{\delta_1}{\Rightarrow} G_1$ holds by hypothesis. Thus, we obtain the desired transition by applying rule MARK (as $p_1 \notin \emptyset$).

3. Let $\varphi = G \overset{\delta_1}{\Rightarrow} G_1 \cdots G_{n-1} \overset{\delta_n}{\Rightarrow} G_n$ be a parallel derivation via $\rho_1, \ldots, \rho_n \in \mathcal{R}^*$. We proceed by induction on n. Case $n = 0$ is identical to the previous point. For case $n > 1$, by the induction hypothesis, there is a successful trace $(\rho_2, \delta_2, N_2) \ldots (\rho_n, \delta_n, N_n)\checkmark$ for $(Q_{\mathcal{R}}, G_1)$, and therefore there are control processes $P_2, \ldots,$ P_{n-1} such that
$$(Q_{\mathcal{R}}, G_1) \xrightarrow{(\rho_2, \delta_2, N_2)}_D (P_2, G_2) \ldots$$
$$\ldots (P_{n-1}, G_{n-1}) \xrightarrow{(\rho_n, \delta_n, N_n)}_D (P_n, G_n) \xrightarrow{\checkmark}_D (\mathbf{0}, G_n)$$
Furthermore, as shown below, the following holds:
$$(Q_{\mathcal{R}}, G) \xrightarrow{(\rho_1, \delta_1, \emptyset)}_D (\mathbf{0}^k(Q_{\mathcal{R}}), G_1) \qquad (\dagger)$$

for some $k > 0$, where process $\mathbf{0}^k(P)$ is defined as $\mathbf{0}^0(P) = P$ and $\mathbf{0}^{k+1}(P) = \mathbf{0} \,||\, (\mathbf{0}^k(P))$.

By exploiting rule PAR1 and the fact that $\mathbf{0}^k(\mathbf{0}) \equiv \mathbf{0}$ (by Definition 4.2), we infer that the marked process $(\mathbf{0}^k(Q_{\mathcal{R}}), G_1)$ has a successful trace identical to the one above of $(Q_{\mathcal{R}}, G_1)$, and therefore $(Q_{\mathcal{R}}, G)$ has a trace $(\rho_1, \delta_1, \emptyset)(\rho_2, \delta_2, N_2) \cdots (\rho_n, \delta_n, N_n)\checkmark$ with underlying derivation φ, as desired.

It remains to prove transition $(\dagger)$ above. We obtain it by rule MARK: the second precondition holds by hypothesis, because $G \overset{\delta_1}{\Rightarrow} G_1$ via ρ_1; the third one holds trivially; and for the first one, assuming that $\rho_1 = p_1 | \dots | p_m$ we proceed by induction on m to show that transition $\sigma_m : Q_{\mathcal{R}} \xrightarrow{(\rho_1, \emptyset)} \mathbf{0}^k(Q_{\mathcal{R}})$ holds for some $k > 0$.

By rule CHOICE we have a transition $\tau_q : \left(\sum_{p \in \mathcal{R}} p.\mathbf{0} + \varepsilon.\mathbf{0}\right) \xrightarrow{(q, \emptyset)} \mathbf{0}$ for each $q \in \mathcal{R} \cup \{\epsilon\}$, and thus also

$$Q_{\mathcal{R}} \xrightarrow{(q,\emptyset)} (\mathbf{0} \,||\, Q_{\mathcal{R}}) = \mathbf{0}^1(Q_{\mathcal{R}})$$

using rules REC, CHOICE and PAR0. This shows existence of σ_0 and σ_1. For $m > 1$, by induction hypothesis we have $\sigma_{m-1} : Q_{\mathcal{R}} \xrightarrow{(p_2 | \dots | p_m, \emptyset)} \mathbf{0}^k(Q_{\mathcal{R}})$. By applying rule SYNC to transitions τ_{p_1} and σ_{m-1} we obtain transition

$$\Big(\sum_{p \in \mathcal{R}} p.\mathbf{0} + \varepsilon.\mathbf{0} \Big) \,||\, Q_{\mathcal{R}} \xrightarrow{(p_1 | (p_2 | \dots | p_m)), \emptyset)} \mathbf{0} \,||\, \mathbf{0}^k(Q_{\mathcal{R}})$$

from which transition σ_m is easily obtained with rules REC and CHOICE.

□

Although comparing traces is a useful means to reason about process equivalence, traces do not represent faithfully the control structures of RePro processes: mere traces do not capture the *branching* structure of RePro processes, which is important in the context of concurrent and reactive systems (like our WSN example). *Bisimilarity* is a well-known branching-sensitive equivalence notion [79].

Definition 6.1 (Simulation, Bisimulation)**.** *Given an LTS* $(S, A_X, \rightarrow_X)$*. A* simulation *is a relation* $\mathsf{R} \subseteq S \times S$ *s.t. whenever* $s \,\mathsf{R}\, t$*, for each transition* $s \xrightarrow{\alpha}_X s'$ *(with* $\alpha \in A_X$*), there exists a transition* $t \xrightarrow{\alpha}_X t'$ *with* $s' \,\mathsf{R}\, t'$*. State* s *is* simulated by t *if there is a simulation relation* R *such that* $s \,\mathsf{R}\, t$*.*

A bisimulation *is a symmetric simulation. States* s *and* t *are* bisimilar*, denoted* $s \simeq_X^{BS} t$*, if there is a bisimulation* R *such that* $s \,\mathsf{R}\, t$*.*

As for the relation of control processes and full RePro processes w.r.t. equivalences, we show that control process equivalence and graph identity imply equivalence of the corresponding compound RePro processes, for both trace equivalence and bisimilarity.

Proposition 6.3. *For any* $P, Q \in \mathcal{P}$ *and* $G \in |\mathbf{Graph}_T|$,

1. $P \simeq^T Q$ *implies* $(P, G) \simeq^T_D (Q, G)$ *and*
2. $P \simeq^{BS} Q$ *implies* $(P, G) \simeq^{BS}_D (Q, G)$.

Proof.

1. We show by co-induction that any trace t of (P, G) is also a trace of (Q, G): by symmetry, also the converse holds, and thus, $(P, G) \simeq^T_D (Q, G)$. We have three cases:
   a) If $t = \varepsilon$, then t is trivially a trace of (Q, G) as well.
   b) If $t = (\rho, \delta, N) \cdot t'$, then

   $$(P, G) \xrightarrow{(\rho,\delta,N)}_D (P', G')$$

   and t' is a trace of (P', G'). By rule MARK of Definition 5.2, we know that $P \xrightarrow{(\rho,N)} P'$ and $G \overset{\delta}{\Rightarrow} G'$ via ρ. Since $P \simeq^T Q$, we also know that $Q \xrightarrow{(\rho,N)} Q'$ for a control process $Q' \simeq^T P'$. By co-induction, we assume that t' is a trace of (Q', G') and, thus, t is a trace of (Q, G).
   c) If $t = \checkmark \cdot t'$, then the argument is identical as before by rule STOP.
2. First, we show that $P \simeq^{BS} Q$, $(P, G) \xrightarrow{(\rho,\delta,N)}_D (P', G')$ implies $(Q, G) \xrightarrow{(\rho,\delta,N)}_D (Q', G')$.

   By rule MARK, $(P, G) \xrightarrow{(\rho,\delta,N)}_D (P', G')$ implies that $P \xrightarrow{(\rho,N)} P'$, $G \overset{\delta}{\Rightarrow} G'$ over ρ and $\forall p \in N : G \overset{p}{\not\Rightarrow}$. Then, again by rule MARK, $(Q, G) \xrightarrow{(\rho,\delta,N)}_D (Q', G')$ if $Q \xrightarrow{(\rho,N)} Q'$, which holds due to $P \simeq^{BS} Q$.

   By the symmetry of the above argument, this leads to a bisimulation if $(P', G') \simeq^{BS}_D (Q', G')$. By $P \simeq^{BS} Q \Rightarrow P' \simeq^{BS} Q'$, this holds by co-induction.

□

It is important to note here that the inverses of those statements do not hold: P and Q might still have some completely different parts, which is, in turn, covered by G in the RDTS semantics. As a simple example, consider $P := \rho.\mathbf{0} + \rho_1.\mathbf{0}$ and $Q := \rho.\mathbf{0} + \rho_2.\mathbf{0}$. If only ρ is applicable to G of the three rules involved, then (P, G) and (Q, H)

are bisimilar (and also trace equivalent) but P and Q are clearly not equivalent.

Bisimilarity as an equivalence notion is distinctive enough to reason about replacing a process in some context with another equivalent one. Intuitively, a context is a process specification with a "hole" where arbitrary processes can be plugged in. It is a well-known property (and a major strength) of standard CCS that all the syntactical operators can be used as such contexts [79]. This property is captured by the formal notion of *congruence* in the literature: formally, an equivalence relation $\simeq$ is congruent w.r.t. an operator f iff $a_i \simeq a'_i, i \in \{1, \ldots, n\}$ implies $f(a_1, \ldots, a_n) \simeq f(a'_1, \ldots, a'_n)$.[2]

In the following, we show that our bisimilarity for RePro processes has this desired congruence property: bisimilarity is retained if the control processes are expanded by a further context. We start by formulating a corresponding result for control processes.[3]

Theorem 6.1. *property of a relation between processes following from a particular semantics Given $P, Q, R \in \mathcal{P}$ with $P \simeq^{BS} Q$, the following propositions hold:*

1. $P + R \simeq^{BS} Q + R$,
2. $P \,||\, R \simeq^{BS} Q \,||\, R$ *and* $R \,||\, P \simeq^{BS} R \,||\, Q$, *and*
3. $\gamma.P \simeq^{BS} \gamma.Q$ *for any* $\gamma \in \mathcal{R}^* \times 2^{\mathcal{R}}$.

Proof.

1. $P + R \simeq^{BS} Q + R$: $P + R \xrightarrow{\alpha} P' + R$ implies $Q + R \xrightarrow{\alpha} Q' + R$ by $P \simeq^{BS} Q$ and rule CHOICE. $P + R \xrightarrow{\alpha} P + R'$ implies $Q + R \xrightarrow{\alpha} Q + R'$ by rule CHOICE. By coinduction and symmetry, $P + R \simeq^{BS} Q + R$ follows from $P' \simeq^{BS} Q'$.
2. $P \,||\, R \simeq^{BS} Q \,||\, R$:
   a) $P \,||\, R \xrightarrow{\alpha} P' \,||\, R$ implies $Q \,||\, R \xrightarrow{\alpha} Q' \,||\, R$ by $P \simeq^{BS} Q$ and rule PAR0.
   b) $P \,||\, R \xrightarrow{\alpha} P \,||\, R'$ implies $Q \,||\, R \xrightarrow{\alpha} Q \,||\, R'$ by rule PAR1.
   c) $P \,||\, R \xrightarrow{\alpha} P' \,||\, R'$ implies $Q \,||\, R \xrightarrow{\alpha} Q' \,||\, R'$ by $P \simeq^{BS} Q$ and rule SYNC.

---

[2] Note that this congruence notion is different from structural congruence (cf. Definition 4.2), and they are distinguished by always explicitly saying *structural* if the latter is meant. However, while structural congruence is an axiomatic description of syntactic properties, congruence is a property of an equivalence relation.

[3] As recursion in RePro is formulated by defining equations, there is no corresponding congruence statement; in process algebra literature, there often appears an equivalent formulation using a recursion *operator*, in which case congruence can be shown accordingly.

By coinduction and symmetry, $P \,||\, R \simeq^{BS} Q \,||\, R$ follows from $P' \simeq^{BS} Q'$.

The other part of the statement (R composed from the left) can be proved analogously by simply swapping PAR0 and PAR1.

3. $\gamma.P \simeq^{BS} \gamma.Q$: There are symmetrically matching transitions $\gamma.P \xrightarrow{\gamma} P$ and $\gamma.Q \xrightarrow{\gamma} Q$ by PRE, and those are the only transitions of $\gamma.P$ and $\gamma.Q$, respectively, as no further rule of Definition 4.7 matches. Thus, $\gamma.P \simeq^{BS} \gamma.Q$ follows from $P \simeq^{BS} Q$.

□

Based on the above theorem, we observe that bisimilarity of full RePro processes is also a congruence.

Corollary 6.1. *Given* $P, Q, R \in \mathcal{P}$ *with* $P \simeq^{BS} Q$, $\gamma = (\rho, N)$ *with* $\gamma \in \mathcal{R}^* \times 2^{\mathcal{R}}$ *and graph* $G \in |\mathbf{Graph}_T|$*. Then, the following propositions hold:*

1. $(P + R, G) \simeq_D^{BS} (Q + R, G)$,
2. $(P \,||\, R, G) \simeq_D^{BS} (Q \,||\, R, G)$ *and* $(R \,||\, P, G) \simeq_D^{BS} (R \,||\, Q, G)$, *and*
3. $(\gamma.P, G) \simeq_D^{BS} (\gamma.Q, G)$.

Proof. The statement is a direct consequence of Theorem 6.1 and Proposition 6.3. □

As a further observation, the congruence property is preserved for choice even if we do not require bisimilarity of control processes, but only that of the corresponding full RePro processes.

Proposition 6.4. *Given* $P, Q, R \in \mathcal{P}$ *and graph* $G \in |\mathbf{Graph}_T|$ *such that* $(P, G) \simeq_D^{BS} (Q, G)$,

$$(P + R, G) \simeq_D^{BS} (Q + R, G).$$

Proof. $(P + R, G) \xrightarrow{\alpha} (P' + R, G)$ implies $(Q + R, G) \xrightarrow{\alpha} (Q' + R, G)$ by $(P, G) \simeq_D^{BS} (Q, G)$ and rules CHOICE and MARK. $(P + R, G) \xrightarrow{\alpha} (P + R', G)$ implies $(Q + R, G) \xrightarrow{\alpha} (Q + R', G)$ by rules CHOICE and MARK. By coinduction and symmetry, $(P + R, G) \simeq_D^{BS} (Q + R, G)$ follows from $(P', G) \simeq_D^{BS} (Q', G)$.

□

Note, however, that a similar statement is not true in general for the further operators, namely prefix and parallel composition.

As a counterexample in the parallel composition case, first, consider the processes (P_{link}, G_2) and (P_{TC}, G_2) (refer to Figure 4.6 for the process specifications and to Figure 5.2a for the graph G_2). These processes are indeed bisimilar, both missing any outgoing transitions (i.e., both being stuck), as each of the available rules requires the presence of at least one link, which is not the case in G_2, consisting of only two nodes. However, composing both control processes with P_{cr} results in processes with different behavior: after p_{Cr} has fired on both sides, the subsequent processes (P_{link}, G_2') and (P_{TC}, G_2') (cf. Figure 5.2a for the graph G_2') are not bisimilar anymore, as P_{link} is still unable to fire, as the only link of G_2' is unclassified, whereas P_{TC} can apply p_{ActUS} to that one link.

We can construct a similar counterexample for the prefix case: taking the same pair of bisimilar processes as above, putting the rule p_{Cr} as a prefix results in the same deviating behavior as described above.

Summarizing, we conclude that regarding the well-known equivalence notions, trace equivalence and bisimulation, RePro control processes faithfully reflect the central properties of CCS. We focus on RDTS in the following and might refer to it simply as RePro transition system in the following.

6.2 INDEPENDENCE

The twofold advances in this section are motivated by the simultaneous presence of established independence notions in both theories considered: that of graph rewriting as well as that of process algebra. First, in Sect. 6.2.1, we follow a *direct approach*, meaning that we directly re-interpret *parallel independence* of graph-rewriting actions [38] for their occurrences as in the RePro semantics.

Second (Sect. 6.2.2), in parallel to a similar proposal for the CCS calculus due to Mukund and Nielsen [83], we investigate the adoption of *asynchronous transition systems* (ATS), originally proposed by Bednarczyk [8]. The originally proposed semantics for CCS [79], which, in turn, serves as a guideline for RePro, is an *interleaving* semantics, where the parallel composition of two actions cannot be distinguished from a corresponding choice over their possible sequences. Although such a semantic design might well be on purpose (as in the case of the original use of CCS), sometimes, e.g., when using process calculi for abstract specifications of concurrent systems (as in the case of RePro), a *non-interleaving* semantics is desirable. A common way of introducing a non-interleaving semantics is to incorporate a means of *asynchronous* action behavior into the semantics. Here, asynchronicity means that the semantics reflects the concurrency present in the specification by identifying independent parallel components as such.

Inspired by the literature and most importantly by the work of Mukund and Nielsen [83], we particularly consider the following motivations when defining an ATS for RePro:

- ATS allow for an extended bisimulation concept based on *independent events* rather than just matching actions. To retain the equivalence results from the previous Sect. 6.1 in an asynchronous semantic setting, we have to find an appropriate equivalence relation, which, in turn, transforms the actions of the underlying calculus into events, whose equivalence is then appropriately addressed in an asynchronous setting (cf. [83] for an analogous line of development). As for the asynchronous semantics of RePro, concluding Sect. 6.2.2, such an equivalence notion is presented in Definition 6.12 and the corresponding bisimulation result is obtained in Theorem 6.4.

- Mukund and Nielsen present a translation of ATS defined for CCS into a specific class of Petri nets, the so-called *1-safe* Petri nets, being particularly well-suited for analysis. This result is obtained by verifying a property of *elementarity* for their transition systems. Although we cannot directly reproduce an analogous result, we discuss this point later in a broader context, considering the place of RePro within concurrency theory, in Sect. 8.1, as part of a discussion chapter.

6.2.1 *Direct Approach*

We proceed by examining the notion of *independence* (of actions or steps), which plays a central role in both process algebra and graph rewriting. In particular, our goal is to elaborate and establish a formal correspondence between two central notions from the different underlying theories: *transition independence* on the one hand and *parallel independence* of graph-rewriting rule applications (cf. Chapter 2, Definition 2.6) on the other hand.

Conceptually, we consider two different approaches to capture the relationship between parallel independence and transition independence:

1. We lift the definition of parallel independence to transitions, i.e., we let transitions to be parallel independent exactly if they are available in parallel and the underlying rule applications are parallel independent. We call this the *direct approach*.

2. In our *asynchronous approach*, on the contrary, we depart from process algebra theory and build on an existing framework, called *asynchronous transition systems* [83] (ATS), where labeled transition systems are extended with an inherent transition independence relation, allowing for a finer characterization of control behavior. Thus, the asynchronous approach takes the process specification form more intensively into account.

We start by describing the direct approach. Recall that two rule applications

$$\begin{tikzcd}
& & G \arrow[dll, Rightarrow, "\delta_1"'] \arrow[drr, Rightarrow, "\delta_2"] \arrow[dd, Rightarrow, "\delta"] & & \\
H_1 \arrow[dr, Rightarrow, "\delta'_2"'] & & & & H_2 \arrow[dl, Rightarrow, "\delta'_1"] \\
& H_{12} \arrow[r, phantom, "\simeq"] & H \arrow[r, phantom, "\simeq"] & H_{21} &
\end{tikzcd}$$

Figure 6.1: Local Church-Rosser and Parallelism Properties of Rule Applications $H_1 \overset{\delta_1}{\Leftarrow} G \overset{\delta_2}{\Rightarrow} H_2$

$$H_1 \overset{\delta_1}{\Leftarrow} G \overset{\delta_2}{\Rightarrow} H_2$$

available simultaneously (i.e., starting from the same graph G) are parallel independent if after performing any of the applications, the other rule is still applicable *on the same match image as in the original rule application* (cf. Proposition 2.1). In the following, we say that a 4-tuple of DPO diagrams $\delta_1, \delta_2, \delta'_1, \delta'_2$ as in Proposition 2.1 (also cf. Figure 2.5) *has the Local Church-Rosser property*. For convenience, we repeat here the visual representation of the Local Church-Rosser property (Figure 6.1), originally presented in Chapter 2.

The following central definition of the direct approach lifts parallel independence into RePro by re-interpreting parallel rule-application independence for transitions. This lifting essentially amounts to requiring that two transitions are indeed available in parallel by the process, beyond requiring the Local Church-Rosser property for their underlying rule applications. Interpreting confluence for transitions allows for a further slight relaxation: it is sufficient to require that the respective subsequent control processes after the different sequences are bisimilar, with isomorphic graph instances.

Definition 6.2 (Parallel Transition Independence)**.** *Given a* RePro *process* (P, G) *with outgoing transitions*

1. $(P, G) \xrightarrow{(\rho_1, \delta_1, N_1)}_D (P_1, H_1)$ *and*
2. $(P, G) \xrightarrow{(\rho_2, \delta_2, N_2)}_D (P_2, H_2)$,

these outgoing transitions *are* parallel independent *if there exist the following transitions:*

3. $(P_1, H_1) \xrightarrow{(\rho_2, \delta'_2, N_2)}_D (P_{12}, H_{12})$, *and*
4. $(P_2, H_2) \xrightarrow{(\rho_1, \delta'_1, N_1)}_D (P_{21}, H_{21})$

such that $\delta_1, \delta'_1, \delta_2, \delta'_2$ *have the Local Church-Rosser property,* $P_{12} \simeq^{BS} P_{21}$ *and* $H_{12} \simeq H_{21}$.

In the following, $(P_1, H_1) \approx_X (P_2, H_2)$ denotes that $P_1 \simeq_X^{BS} P_2$ and $H_1 \simeq H_2$ (e.g., in the above definition, $(P_{12}, H_{12}) \approx (P_{21}, H_{21})$).

Regarding the properties of this independence definition, we start by an essential soundness claim: parallel *transition* independence implies parallel independence of the involved underlying rule applications.

Proposition 6.5. *Given two parallel independent transitions*

$$(P, G) \xrightarrow{(\rho_1, \delta_1, N_1)}_D (P_1, H_1) \text{ and } (P, G) \xrightarrow{(\rho_2, \delta_2, N_2)}_D (P_2, H_2),$$

the rule applications δ_1 and δ_2 are parallel independent.

Proof. According to Definition 6.2, there exist subsequent matches and, therefore, rule applications $H_1 \overset{\delta_2'}{\Rightarrow} H_{12}$ and $H_2 \overset{\delta_1'}{\Rightarrow} H_{21}$ as in Proposition 2.1.

We have to show that in this case, the original rule applications $G \overset{\delta_1}{\Rightarrow} H_1$ and $G \overset{\delta_2}{\Rightarrow} H_2$ are parallel independent. Suppose they are not: then at least one of the morphisms d_1 and d_2 (let us choose d_1 w.l.o.g.) as in Definition 2.6 does not exist. But if d_1 does not exist, we are not able to choose the subsequent match m_1' as in Proposition 2.1, which contradicts our assumption. □

Note that the inverse claim does not hold; parallel rule-application independence does not always imply parallel transition independence, as the confluence of transitions might be blocked not only due to conflicting matches, but also due to non-applicability conditions becoming enabled. For instance, our example topology control process

$$P_{nTC} := p_{kTC}.P_{nTC} + (p_{ActUS}, \{p_{kTC}\}).P_{nTC} + (p_{ActUL}, \{p_{kTC}\}).P_{nTC}$$

with non-applicability conditions has parallel independent transitions for any pairs of matches of p_{ActUS} and p_{ActUL} in any graph, as those rules match different types of edges and, thus, deleting one of those edges never obstructs the match of the other rule. However, the corresponding actions $(p_{ActUS}, \{p_{kTC}\})$ and $(p_{ActUL}, \{p_{kTC}\})$ also have a non-applicability condition each, forbidding that p_{kTC} is applicable. Consider as input topology for those rule applications the graph G_{ex2}' depicted in Fig. 6.2, which only differs from G_{ex}' (Figure 5.3b) in having an additional `L;u`-edge between nodes n_3 and n_9.

If we take, for example, as matches for the parallel applications of p_{ActUS} and p_{ActUL} the edges n_1n_3 and n_3n_9 (being parallel independent), then the corresponding transitions are *not* parallel independent: after performing p_{ActUS} on n_1n_3, a match for p_{kTC} arises (the triangle $n_1n_3n_9$), and the non-applicability condition prevents the action $(p_{ActUL}, \{p_{kTC}\})$ to fire.

Note that the above statement did not take the shape of the process specification into account, i.e., if the simultaneously available transitions arise from a choice or from a parallel composition. Now, we

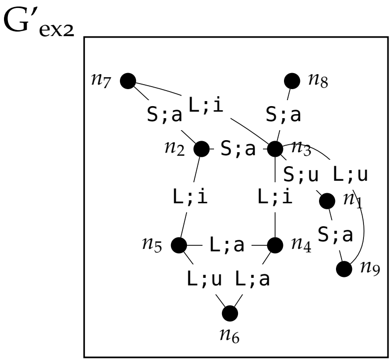


Figure 6.2: Example Topology Graph G'_{ex2}

conclude the presentation of the direct approach to independence by summarizing the correspondence between the relevant independent notions. In a way, the following theorem is the RePro counterpart of the Local-Church Rosser and Parallelism Theorems (cf. Proposition 2.1) known from graph-rewriting theory. In particular, we state that (i) as expected, equivalence of parallel independence and arbitrary sequentialization holds also for RePro transitions, and (ii) this Local Church-Rosser property also harmonizes with synchronization as a parallelism notion.

Theorem 6.2. *Given a RePro process (P, G), two rules ρ_1, ρ_2, and rule name sets N_1, N_2, the following statements are equivalent:*

1. *There are parallel independent transitions $(P, G) \xrightarrow{(\rho_1,\delta_1,N_1)}_D (P_1, H_1)$ and $(P, G) \xrightarrow{(\rho_2,\delta_2,N_2)}_D (P_2, H_2)$.*
2. *There are four transitions as in Figure 6.3 such that $\delta_1, \delta'_1, \delta_2, \delta'_2$ have the Local Church-Rosser property.*

Moreover, if there is a control transition $P \xrightarrow{(\rho_1|\rho_2,N_c)} P'$, with $N_c = N_1 \cup N_2$, then the following is also equivalent with the above two:

3. *There is a transition $(P, G) \xrightarrow{(\rho_1|\rho_2,\delta_c,N_c)}_D (P', H)$ such that (i) for the underlying matches m_1, m_2, m_c of $\delta_1, \delta_2, \delta_c$, respectively, it holds that $m_c = m_1 + m_2$ according to the coproduct construction, and (ii) H is isomorphic to H_{12} and H_{21} as in Figure 6.3.*

Proof.

(1) ⇒ (2) : This is a direct consequence of Definition 6.2.
(2) ⇒ (1) : First, let us observe that if there is a diamond of rule applications having the Local Church-Rosser property, then none of the upper rule applications δ_1 and δ_2 introduce graph structures which

$$\begin{tikzcd}
& (P,G) \arrow[dl, "(\rho_1,\delta_1,N_1)"'] \arrow[dr, "(\rho_2,\delta_2,N_2)"] & \\
(P_1,H_1) \arrow[dr, "(\rho_2,\delta'_2,N_2)"'] & & (P_2,H_2) \arrow[dl, "(\rho_1,\delta'_1,N_1)"] \\
& (P_{12},H_{12}) \approx (P_{21},H_{21}) &
\end{tikzcd}$$

Figure 6.3: Parallel Transition Independence

violate non-applicability conditions in N_1 or N_2. Thus, parallel independence of those transitions follows from the parallel independence of δ_1 and δ_2. Suppose they are not parallel independent: then at least one of the morphisms d_1 and d_2 (let us choose d_1 w.l.o.g.) as in Definition 2.6 does not exist. But if d_1 does not exist, we are not able to choose the subsequent match m'_1 as in Proposition 2.1, which contradicts our assumption.

(1) $\Rightarrow$ (3) : First note that due to $(1) \Rightarrow (2)$, isomorphic graphs H_{12} and H_{21} as in Figure 6.3 are given. Due to Proposition 6.5, there are parallel independent underlying rule applications $G \overset{\delta_1}{\Rightarrow} H_1$ and $G \overset{\delta_2}{\Rightarrow} H_2$. Thus, the statement is a consequence of Proposition 2.1.

(3) $\Rightarrow$ (1) : By the coproduct construction, there is a pair of unique monomorphisms $L_1 \xrightarrow{l_1^+} L_1 + L_2 \xleftarrow{l_2^+} L_2$ from the left-hand sides L_1 and L_2 of ρ_1 and ρ_2, respectively, to the left-hand side of the parallel rule $\rho_1|\rho_2$. Then, for a match $m_c : L_1 + L_2 \to G$, the matches m_1, m_2 of the single interleaved rule applications are given as $m_i = m_c \circ l_i^+$ for $i = 1,2$.

Then, those matches m_1, m_2 are parallel independent: assume by contradiction that, e.g., the application δ_2 would delete some element from $m_1(L_1)$ in G. But then, those m_1, m_2 cannot arise from a single m_c as above, as then, there would be no rule application δ_c; m_c would not be a valid match, as the DPO approach does not allow for matches where different elements are mapped onto a single input element, unless both rule elements are preserved by the rule. The parallel independence of the corresponding transitions follow from this and the fact that we know from the synchronized action that each non-applicability condition in both N_1 and N_2 are fulfilled in G.

□

6.2.2 *Asynchronous Approach*

Next, we present the *asynchronous approach* to independence in RePro.

In the previously described so-called direct approach, we have directly interpreted parallel rule-application independence (cf. Definition 2.6) in the context of our RePro transition system. Although this approach provides a natural correspondence between confluent transitions, the Local Church-Rosser Property and synchronization (Theorem 6.2), it is agnostic to the structure of the control process making the corresponding transitions available to fire in parallel. In turn, those structural considerations are a central subject in concurrency theory and, in particular, in process calculi: *asynchronicity* means, in this context, the causal independence of transitions. Causal independence, in turn, necessarily involves that those transitions come from separate parallel components of the process (and do not synchronize but rather proceed on their own, i.e., in an interleaved manner).

In particular, for adopting asynchronicity for RePro, consider as a first example the following processes:

$$P := \rho_1.\rho_2.\mathbf{0} + \rho_2.\rho_1.\mathbf{0}$$

$$Q := \rho_1.\mathbf{0} \,||\, \rho_2.\mathbf{0}$$

where ρ_1 and ρ_2 have parallel independent applications to a graph G. Figures 6.4a and 6.4b shows the relevant excerpts from RDTS, i.e., the transitions corresponding to those rule applications for both P and Q. (Note that for the sake of simplicity of demonstration, we do not include $\approx$ in the confluent state and simply take a single graph H reachable through both sequences.)

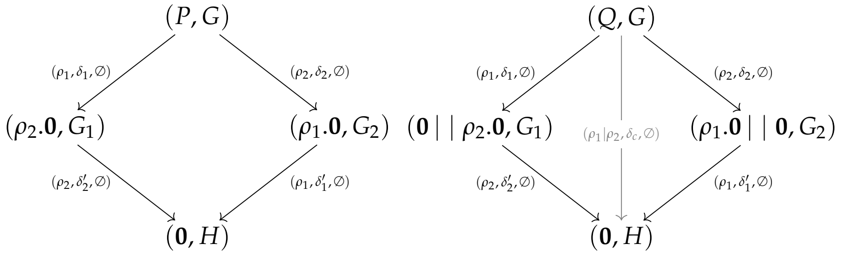
By Definition 6.2, both (P, G) and (Q, G) have parallel independent transitions in Figures 6.4a and 6.4b; however, control-level parallelism is present in the term structure only in the case of Q. Nevertheless, as a consequence of the presence of synchronized actions such as $(\rho_1|\rho_2, \delta_c, \emptyset)$ in Figure 6.4b, (P, G) and (Q, G) are *not* bisimilar. However, this side effect of RePro synchronization diminishes if we change the specification of P to

$$P' := \rho_1.\rho_2.\mathbf{0} + \rho_2.\rho_1.\mathbf{0} + (\rho_1|\rho_2).\mathbf{0}$$

Then, P' and Q are bisimilar, still, we are unable to distinguish sequential choice from parallel composition based on that relation.

In addition to the above artificial examples, recalling our considerations about distributedness (Sect. 1.4) and inherently concurrent systems like our WSN example (Sect. 1.3), asynchronicity might be an important indicator if an abstract model specification adheres to the distributed structure of the subject system. For example, although we summarized the node behavior (creation and deletion) in a single process P_{node} (cf. Figure 4.6), there might be external modeling constraints (e.g., a realistic model where node creation is human-triggered, whereas deletion happens spontaneously due to battery depletion) inducing a parallel structure

$$P_{cr} \,||\, P_{del} \text{ with } P_{cr} := p_{Cr}.P_{cr}, \quad P_{del} := p_{Del}.P_{del}$$


(P, G)
$(\rho_1, \delta_1, \emptyset)$
$(\rho_2, \delta_2, \emptyset)$
$(\rho_2.\mathbf{0}, G_1)$
$(\rho_1.\mathbf{0}, G_2)$
$(\rho_2, \delta_2', \emptyset)$
$(\rho_1, \delta_1', \emptyset)$
$(\mathbf{0}, H)$
(Q, G)
$(\rho_1, \delta_1, \emptyset)$
$(\rho_2, \delta_2, \emptyset)$
$(\mathbf{0} \,||\, \rho_2.\mathbf{0}, G_1)$
$(\rho_1|\rho_2, \delta_c, \emptyset)$
$(\rho_1.\mathbf{0} \,||\, \mathbf{0}, G_2)$
$(\rho_2, \delta_2', \emptyset)$
$(\rho_1, \delta_1', \emptyset)$
$(\mathbf{0}, H)$

(a) Interleaving (b) Parallel Composition

Figure 6.4: LTS Excerpts for Demonstrating Asynchronicity

The process $P_{cr} \,||\, P_{del}$ is bisimilar to P_{node}, despite the fundamental difference in their structure and principles. Asynchronicity addresses exactly such issues.

To address the issue of reflecting parallel composition structure in transition labels and, thus, of distinguishing such situations by an appropriate bisimulation relation, the idea of *asynchronous transition systems* (ATS) was first proposed in the PhD thesis of Marek Bednarczyk [8]. In the following, we build upon the work of Mukund and Nielsen [83] and address asynchronicity (i.e., specification-level parallelism) by extending the labels with *location tags*, i.e., identifiers revealing the originating parallel component of each transition.

To summarize, as demonstrated by the two examples above, we set a double goal while extending RePro by asynchronicity through locations:

(i) to define an independence notion aware of parallel composition structure along with a corresponding ATS for RePro processes, and

(ii) to define a bisimulation notion strong enough to distinguish synchronization but abstract enough to hide irrelevant rule application details.

First, we deal with independence and provide a proper ATS of graph-rewriting transitions; afterwards, we describe an appropriate bisimulation.

Intuitively, asynchronicity means that two transitions occur in a causally independent manner. In our setting, this boils down for two transitions to originate from different parallel components of a single process. Therefore, we adapt the procedure presented by Mukund and Nielsen [83] to our transitions and augment their labels with *location tags* uniquely identifying those parallel components. This serves as a basis for defining a corresponding independence notion.

In particular, for each parallel composition operator, its left-hand side is identified by a 0 whereas its right-hand side is identified by a 1 (which is also the reason for parallel composition not being

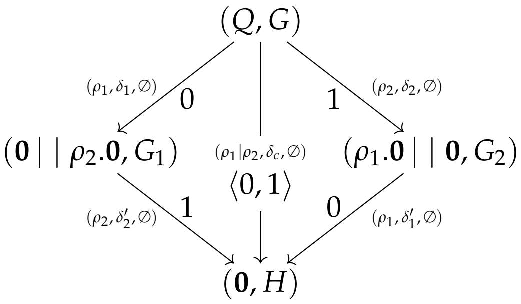


Figure 6.5: LTS Excerpt: Parallel Composition with Location Tags

commutative in our setting). For example, processes P and P' above (in the first example of the present section) are purely sequential and, thus, their transitions do not get location tags. In contrast, Figure 6.5 shows the LTS excerpt of Figure 6.4b with additional location tags for each transition.

Due to a potentially more complicated nested parallelism structure of terms, single numbers do not suffice for identification in general, but only strings of them. In addition, synchronization involves a synchronous dependence to each partaking component, therefore, tags are equipped with a hierarchical tree structure to reflect synchronization.

Formally, location tags (or often simply tags) are based on location strings, i.e., strings over the 2-element set $\{0,1\}$. The set of those strings is denoted as $\mathcal{S} = \{0,1\}^*$ (where $*$ is the usual string operation and,thus, a non-commutative one in contrast to $\mathcal{R}^*$). $\mathcal{S}$ is ranged over by s and also contains the empty string s_ε (often omitted as part of a concrete tag). Due to the possibly nested composition structure, tags are represented by a tree hierarchy of location strings.

Definition 6.3 (Location Tag)**.** *The syntax of* location tags u, v *is given inductively by the following grammar:*

$$u, v ::= \ s \ \mid \ s\langle u, v\rangle$$

with $s \in \mathcal{S}$.

The set of location tags is denoted Tag.

A *location* represented by a tag is a set of location strings comprising each parallel component involved in the transition which the tag belongs to. For concatenating strings, we simply write the strings after each other.

Definition 6.4 (Location)**.** *The* location $loc(u)$ *of a tag u is a set of location strings defined by the following function:*

$$loc(u) := \begin{cases} \{s\} \ \textit{if}\ \exists s \in \mathcal{S} : u = s \\ \{s_0 s_1 \mid s_1 \in (loc(v_1) \cup loc(v_2))\} \ \textit{if}\ u = s_0\langle v_1, v_2\rangle \end{cases}$$

$$\text{STRUCT}\ \frac{P \equiv Q \qquad P \xrightarrow[u]{\alpha}_A P'}{Q \xrightarrow[u]{\alpha}_A P'} \qquad \text{PRE}\ \frac{}{\gamma.P \xrightarrow{\gamma}_A P} \qquad \text{STOP}\ \frac{}{\mathbf{0} \xrightarrow{\checkmark}_A \mathbf{0}}$$

$$\text{CHOICE}\ \frac{P \xrightarrow[u]{\alpha}_A P'}{P + Q \xrightarrow[u]{\alpha}_A P'} \qquad \text{REC}\ \frac{A := P \qquad P \xrightarrow[u]{\alpha}_A P'}{A \xrightarrow[u]{\alpha}_A P'}$$

$$\text{PAR0}\ \frac{P \xrightarrow[u]{\alpha}_A P'}{P \,||\, Q \xrightarrow[0u]{\alpha}_A P' \,||\, Q} \qquad \text{PAR1}\ \frac{Q \xrightarrow[u]{\alpha}_A Q'}{P \,||\, Q \xrightarrow[1u]{\alpha}_A P \,||\, Q'}$$

$$\text{SYNC}\ \frac{P \xrightarrow[u]{(\rho_1,N_1)}_A P' \qquad Q \xrightarrow[v]{(\rho_2,N_2)}_A Q'}{P \,||\, Q \xrightarrow[\langle 0u,1v\rangle]{(\rho_1|\rho_2,N_1 \cup N_2)}_A P' \,||\, Q'}$$

Figure 6.6: Transition Rules of Asynchronous Control Processes

Now, we extend our nCTS labels by inductively derived tags. The resulting transition system, denoted ACTS, is a conservative extension of nCTS; thus, in the following inference rules, the graph-rewriting action part is the same as in Definition 4.5 (still including them here for convenience). The compound labeling set

$$(\mathcal{R}^* \times 2^{\mathcal{R}}) \times \mathit{Tag}$$

of ACTS is denoted by $\mathit{Act}_{\text{ACTS}}$.

Definition 6.5 (Asynchronous Control Transition System)**.** *The* asynchronous control transition system *(ACTS) of $\mathcal{G}$ is an LTS $(\mathcal{P}, \mathit{Act}_{ACTS} \cup \{\checkmark\}, \rightarrow_A)$ with $\rightarrow_A$ being the least relation satisfying the rules in Fig. 6.6, where*

- *α ranges over $(\mathcal{R}^* \times 2^{\mathcal{R}}) \cup \{\checkmark\}$,*
- *γ ranges over $\mathcal{R}^* \times 2^{\mathcal{R}}$,*
- *N ranges over $2^{\mathcal{R}}$, and*
- *u, v range over Tag.*

Sequential actions like PRE and STOP generate empty tags (omitted) and the rest of the rules not dealing with parallelism simply keeps tags. PAR0 and PAR1 add a 0 or a 1, respectively, to the front of a tag coming from a left-hand side or right-hand side parallel component. SYNC creates a compound tag by taking together those tags which would arise by interleaved actions.

As for examples regarding tags and locations induced by ACTS, first, consider the tags depicted in Figure 6.5: the single-digit tags of transitions at the edges of the diamond are the result of PAR0 and PAR1, respectively, representing interleaved transitions of parallel components, where 0 and 1 distinguishes the side of the composition where the respective transition originated. In contrast, the synchronized transition in the middle gets a compound tag, as additionally emphasized by the surrounding brackets: $\langle 0, 1\rangle$ represents a list of location strings, where the tags of the involved transitions (here, both of those are empty as they are derived by rule PRE) get prefixed by 0 and 1, respectively, due to rule SYNC. Note that the tag structure might even be nested, reflecting the similar structure of parallel composition; e.g., the process

$$\rho_1.\mathbf{0} \,||\, (\rho_2.\mathbf{0} \,||\, \rho_3.\mathbf{0})$$

has a tripartite synchronized transition

$$\xrightarrow[\langle 0,1\langle 0,1\rangle\rangle]{(\rho_1|\rho_2|\rho_3,\emptyset)}_A$$

Those structures get flattened by the notion of locations: while for single-digit and non-nested tags, the location function simply turns tags into sets (e.g., $loc(\langle 0, 1\rangle) = \{0, 1\}$), it reads in depth the nested tags:

$$loc(\langle 0, 1\langle 0, 1\rangle\rangle) = \{0, 10, 11\}$$

The asynchronous RePro transition system ARTS is defined analogously to RDTS (Definition 5.1) on top of ACTS. Its compound action set

$$(\mathcal{R}^* \times \mathcal{D} \times 2^{\mathcal{R}}) \times Tag$$

is denoted Act_{ARTS}, also ranged over by α if the identity of the respective transition system is clear from context. Moreover, the graph-rewriting component of an action (μ, u) is ranged over by μ and u denotes its location tag. Note that all the definitions for RDTS naturally apply to ARTS via a forgetful mapping which drops the tags.

Definition 6.6 (Asynchronous RePro Transition System)**.** *The* asynchronous RePro transition system *(ARTS) is an LTS* $(\mathcal{P} \times |\mathbf{Graph}_T|$, $Act_{ARTS} \cup \{\checkmark\}, \rightarrow_{AD})$ *where* $\rightarrow_{AD}$ *is the least relation satisfying the following rules:*

$$\text{MARK}\ \frac{P \xrightarrow[u]{(\rho,N)}_A P' \quad G \overset{\delta}{\Rightarrow} H \quad \forall p \in N : G \overset{p}{\not\Rightarrow}}{(P, G) \xrightarrow[u]{(\rho,\delta,N)}_{AD} (P', H)}$$

$$\text{STOP}\ \frac{P \xrightarrow{\checkmark}_A P'}{(P, G) \xrightarrow{\checkmark}_{AD} (P', G)}$$

Based on tags and locations, we can define what it means for transitions to be *asynchronous*. Intuitively, two transitions are asynchronous and, thus, causally independent if they originate in different parallel components. This is reflected by the following definition.

Definition 6.7 (Asynchronous Actions and Transitions)**.** *Two tags u_1 and u_2 are* asynchronous *if $\forall l_1 \in loc(u_1), l_2 \in loc(u_2) : l_1 \not\preceq l_2 \wedge l_2 \not\preceq l_1$, where $\preceq$ denotes a string prefix relation.*

Two transitions (of any LTS whose labels contain tags) are asynchronous if their labels contain asynchronous tags.

As a basic sanity check for the above definition of asynchronicity in the context of ARTS, we verify the claim that parallel independence *and* asynchronicity indeed means that those transitions come from separate parallel components and, thus, are able to synchronize.

Proposition 6.6. *If transitions*

$$(P, G) \xrightarrow[u_1]{(\rho_1,\delta_1,N_1)}_{AD} (P_1, H_1) \text{ and } (P, G) \xrightarrow[u_2]{(\rho_2,\delta_2,N_2)}_{AD} (P_2, H_2)$$

are parallel independent and asynchronous, then there is a synchronized transition

$$(P, G) \xrightarrow[\langle 0u_1,1u_2\rangle]{(\rho_1|\rho_2,\delta_c,N_1\cup N_2)}_{AD} (P', H)$$

Proof. The simultaneous availability of nCTS actions (ρ_1, N_1) and (ρ_2, N_2) means that P either contains a "top-level" choice or a parallel composition, i.e., either $P := P' + P''$ or $P := P' \,||\, P''$ (where P' and P'' might be defined by process identifiers, but this is of no relevance here). Assume w.l.o.g. that $(P', G) \xrightarrow[u_1]{(\rho_1,\delta_1,N_1)}_{AD} (P_1, H_1)$ and $(P'', G) \xrightarrow[u_2]{(\rho_2,\delta_2,N_2)}_{AD} (P_2, H_2)$ (i.e., that P' is capable of the first transition and P'' of the second one).

If $P := P' + P''$, those transitions cannot be asynchronous: according to Definition 6.5, rule CHOICE, both would get the same tag. Thus, $P := P' \,||\, P''$ and those transitions are always asynchronous due to the tag construction in Definition 6.5, rules PAR0 and PAR1: u_1 has a leading 0 while u_2 has a leading 1, preventing any of their location strings to be prefixes of each other, as each location of u_1 starts with a 0, while each location of u_2 starts with a 1.

Thus, the statement is a direct consequence of Theorem 6.2, as the transitions above can always synchronize in P.

□

Our final goal is to provide a proper asynchronous (labeled) transition system (ATS) for our controlled graph-rewriting processes. An ATS is an LTS equipped with an independence relation on their actions (i.e., transition labels) such that the well-known parallel and sequential shift properties hold, as recalled in the following definition [83].

Definition 6.8 (Asynchronous Labeled Transition System)**.** *An* asynchronous labeled transition system *(ATS) is a tuple* $(S, A, \rightarrow_X, I)$*, where* $(S, A, \rightarrow_X)$ *is an LTS and* $I \subseteq A \times A$ *is an* independence relation *on actions, satisfying the following conditions for any* $(\alpha_1, \alpha_2) \in I$ *and* $s, s_1, s_2, s_c \in S$*:*

(i) Whenever $s \xrightarrow{\alpha_1}_X s_1$ *and* $s \xrightarrow{\alpha_2}_X s_2$*, then* $\exists s_c$ *s.t.* $s_1 \xrightarrow{\alpha_2} s_c$ *and* $s_2 \xrightarrow{\alpha_1} s_c$*.*

(ii) Whenever $s \xrightarrow{\alpha_1}_X s_1$ *and* $s_1 \xrightarrow{\alpha_2}_X s_c$*, then* $\exists s_2$ *s.t.* $s_1 \xrightarrow{\alpha_2} s_2$ *and* $s_2 \xrightarrow{\alpha_1} s_c$*.*

For RePro actions, we observe that it is not straightforward to directly fit the above definition. In a way, DPO diagrams in our labels are too fine-grained for directly satisfying the above shift properties: whenever the graph becomes rewritten by a transition, even if some match of another rule was preserved by the rule application, the corresponding action in a new state results in a different DPO diagram—even if we are able to capture the connection of those "shifted" DPO diagrams along the lines of the Local Church-Rosser Theorem (Proposition 2.1). In the following, we make this notion precise by defining a *Church-Rosser equivalence* on DPO diagrams. Note that, as elsewhere in the paper, whenever the existence of a rule is required, we assume that rule coming from a given fixed GTS.

Definition 6.9 (Church-Rosser Equivalence, Classes)**.** *Let* δ_1 *and* δ_1' *be DPO diagrams with* δ_1 *being an application of rule* ρ_1 *on graph* G *with match* m_1*, and* δ_1' *being an application of* ρ_1 *as well, on graph* H_2 *with match* m_1'*.*

δ_1 *is* Church-Rosser-equivalent *(CRE) to* δ_1'*, denoted* $\delta_1 \equiv_{CR} \delta_1'$*, if there is a DPO diagram* δ_2 *representing an application of rule* ρ_2 *on* G *with match* m_2*, s.t. matches* m_1 *and* m_1' *are as in Proposition 2.1. The transitive closure of* $\equiv_{CR}$ *is denoted as* $\equiv_{CR}^*$*.*

A CRE class $E = \{\delta_1, \delta_2, \dots\}$ *is the largest set of DPO diagrams such that* $\forall \delta_i, \delta_j \in E : \delta_i \equiv_{CR}^* \delta_j$*. The set of CRE classes is denoted* $\mathcal{E}$*. The unique CRE class containing a DPO diagram* δ *is denoted* $E(\delta)$*.*

Note that uniqueness of $E(\delta)$ is due to each class containing all the equivalents along each possible shifts (i.e., CRE classes being *largest* sets).

We derive from ARTS a transition system which abstracts DPO transitions up to Church Rosser equivalence, thus, harmonizing with the general concept of ATS (Definition 6.8). The corresponding compound action set is denoted

$$Act_{\text{ACR-RTS}} := (\mathcal{R}^* \times \mathcal{E} \times 2^{\mathcal{R}}) \times Tag,$$

also ranged over by α if LTS identity is clear from context.

Definition 6.10 (Asynchronous Church-Rosser RePro Transition System)**.** *The* asynchronous Church-Rosser RePro transition system*, abbreviated as ACR-RTS, is an LTS* $(\mathcal{P} \times |\mathbf{Graph}_T|, Act_{\text{ACR-RTS}} \cup \{\checkmark\}, \rightarrow_{CR})$*,* *derived from ARTS s.t.* $\rightarrow_{CR}$ *is the least transition relation satisfying the following rule:*

$$\text{LCR}\ \frac{(P,G) \xrightarrow[u]{(\rho,\delta,N)}_{AD} (P',H)}{(P,G) \xrightarrow[u]{(\rho,E(\delta),N)}_{CR} (P',H)}$$

Having ACR-RTS with an appropriate abstraction of actions, we are now ready to define an independence relation which will make ACR-RTS a proper ATS according to Definition 6.8.

Definition 6.11 (Asynchronous Independence)**.** *Two ACR-RTS transitions*

1. $(P,G) \xrightarrow[u_1]{(\rho_1,E_1,N_1)}_{CR} (P_1,H_1)$ *and*
2. $(P,G) \xrightarrow[u_2]{(\rho_2,E_2,N_2)}_{CR} (P_2,H_2)$

are asynchronously independent *if* u_1, u_2 *are asynchronous and there are subsequent transitions*

3. $(P_1,H_1) \xrightarrow[u_2]{(\rho_2,E_2,N_2)}_{CR} (P',H)$ *and*
4. $(P_2,H_2) \xrightarrow[u_1]{(\rho_1,E_1,N_1)}_{CR} (P',H)$.

The asynchronous independence relation $I_{CR} \subseteq Act_{ACR\text{-}RTS} \times Act_{ACR\text{-}RTS}$ *is defined as follows:* $(\alpha_1, \alpha_2) \in I_{CR}$ *iff* α_1 *and* α_2 *are asynchronously independent.*

Theorem 6.3. $(ACR\text{-}RTS, I_{CR})$ *is an ATS.*

Proof. We have to verify Properties (i) and (ii) from Definition 6.8.

(i) This is a direct consequence of Definition 6.11.

(ii) For transitions $(P,G) \xrightarrow[u_1]{(\rho_1,E_1,N_1)}_{CR} (P_1,H_1) \xrightarrow[u_2]{(\rho_2,E_2,N_2)}_{CR} (P',H)$ with u_1, u_2 being asynchronous, there should be transitions in reverse order: $(P,G) \xrightarrow[u_2]{(\rho_2,E_2,N_2)}_{CR} (P_2,H_2) \xrightarrow[u_1]{(\rho_1,E_1,N_1)}_{CR} (P',H)$.

The existence of those subsequent transitions (3. and 4.) as in Definition 6.11 shows that there are parallel independent rule applications δ_1 and δ_2 of ρ_1 and ρ_2, respectively, on graph G: if this would not be the case, at least one of the corresponding $E_i = E(\delta_i), i \in \{1,2\}$ could not arise as subsequent transition, as there would be no Church-Rosser-equivalent DPO diagram to δ_i in E_i.

> But then, any Church-Rosser-equivalent of δ_2, shifted along δ_1, is in E_2, and thus, E_2 is also available as an outgoing transition form (P, G). Using the same argument, E_1 then also includes a Chruch-Rosser-equivalent of δ_1 shifted along δ_2, thus providing the two transitions required for our statement.

□

As we have seen on our second pair of example processes

$$P' := \rho_1.\rho_2.\mathbf{0} + \rho_2.\rho_1.\mathbf{0} + (\rho_1|\rho_2).\mathbf{0}$$

$$Q := \rho_1.\mathbf{0} \,||\, \rho_2.\mathbf{0}$$

in the beginning, given a graph G they cannot be distinguished by bisimulation over "plain" RePro transitions; nevertheless, location tags allow for a distinction, independently if a pair of simultaneously available transitions have parallel independent rule applications are not. Moreover, tags help to yield a proper ATS for RePro.

However, introducing Church-Rosser equivalence classes to properly capture asynchronicity opens up a further problem: although transitions represent sets of DPO diagrams (up to Church-Rosser equivalence), the concrete graphs in the states prevent $\approx_A$ (cf. the notation after Definition 6.2) from being a bisimulation for ACR-RTS processes. Next, we deal with this problem and propose a further abstraction, namely, factoring out DPO diagrams up to isomorphism. Finally, we will show, that the combined abstraction of Church-Rosser equivalence and DPO diagram isomorphism leads to $\approx_A$ being an appropriate bisimulation relation for a corresponding variant of ACR-RTS.

Definition 6.12 (Abstract Church-Rosser Equivalence, Classes)**.** *Given two DPO diagrams δ_1 and δ_2 as in Definition 2.5, with each graph in them indexed by* 1 *and* 2*, respectively, they are* isomorphic*, denoted as $\delta_1 \cong \delta_2$, if there exist isomorphisms $L_1 \to L_2, K_1 \to K_2, R_1 \to R_2, G_1 \to G_2, D_1 \to D_2, H_1 \to H_2$, such that each arising square commutes.*

An abstract CRE *(ACRE)* class *$\tilde{E}$ is the largest set of DPO diagrams such that $\forall \delta_i, \delta_j \in \tilde{E}$, there is a δ_k (possibly being equal to δ_i or δ_j) such that $\delta_i \equiv^*_{CR} \delta_k \cong \delta_j$. The set of ACRE classes is denoted $\tilde{\mathcal{E}}$. The unique ACRE class containing a DPO diagram δ is denoted $\tilde{E}(\delta)$.*

As expected, there is a corresponding transition system where $\mathcal{E}$ is replaced by $\tilde{\mathcal{E}}$. The corresponding compound action set

$$(\mathcal{R}^* \times \tilde{\mathcal{E}} \times 2^{\mathcal{R}}) \times Tag$$

is denoted $Act_{\tilde{\text{A}}\text{CR-RTS}}$.

Definition 6.13 (Abstract Church-Rosser RePro Transition System)**.** *The* abstract Church-Rosser RePro transition system *($\tilde{\text{A}}$CR-RTS) is an LTS $(\mathcal{P} \times |\mathbf{Graph}_T|, Act_{\tilde{\text{A}}\text{CR-RTS}} \cup \{\checkmark\})$, derived from ARTS by the following rule:*

$$\textsc{alcr}\;\frac{(P,G) \xrightarrow[u]{(\rho,\delta,N)}_{AD} (P',H)}{(P,G) \xrightarrow[u]{(\rho,\tilde{E}(\delta),N)}_{ACR} (P',H)}$$

Now, we are ready to verify our bisimulation notion.

Theorem 6.4. *$\approx_A$ is a bisimulation on the set of ÃCR-RTS states.*

Proof. We verify if transitions of states $(P,G) \approx_A (Q,H)$ fulfill the requirements for a bisimulation (namely, that each transition is mimicked by the other side and the corresponding subsequent states are also bisimilar).

- $(P,G) \xrightarrow[u]{(\rho,\tilde{E}(\delta),N)}_{ACR} (P',G')$ implies $(Q,H) \xrightarrow[u]{(\rho,\tilde{E}(\delta),N)}_{ACR} (Q',H')$ due to $P \simeq^{BS}_{A} P'$ and rules MARK and ALCR, if we show that for each δ s.t. $(P,G) \xrightarrow[u]{(\rho,\delta,N)}_{AD} (P',G')$, there is a δ' with $(Q,H) \xrightarrow[u]{(\rho,\delta',N)}_{AD} (Q',H')$ s.t. $\tilde{E}(\delta) = \tilde{E}(\delta')$.

  Given isomorphism $i : G \to H$, taking $m' = i \circ m$ as the match for δ', δ and δ' are isomorphic: their span is the same (that of ρ), $G \simeq H$ by assumption, $D \simeq D'$ (the pushout complements of δ and δ', respectively) as the pushout complement is unique up to isomorphism, and $G' \simeq H'$ as the pushout object is also unique up to isomorphism. Then, $\tilde{E}(\delta) = \tilde{E}(\delta')$ due to Definition 6.12.

  The inverse implication can be shown analogously due to the symmetry of the relation.

- $(P',G') \approx_A (Q',H')$ due to $P \simeq^{BS}_{A} Q \Rightarrow P' \simeq^{BS}_{A} Q'$ and $G' \simeq H'$ as above; thus, bisimilarity holds by coinduction.

□

Summarizing, the above result reproduces that part of the work by Mukund and Nielsen [83] which is concerned with providing an independence-preserving bisimulation based on abstract events and a corresponding ATS. However, to verify the other result of their paper, i.e., if the resulting ATS is elementary and, thus, eligible for a 1-safe Petri net translation, would require the development of additional notions for RePro, e.g., that of *regions* and their *separations* (cf. Definition 4.9 and Lemma 4.12 in [83]). Thus, this remains as a major item of future work, but we discuss the relation of Petri nets and CCS-like processes in Sect. 8.1.

6.3 REPRO AS A CONTROL LANGUAGE: EXPRESSIVENESS

As RePro is motivated by the wish to increase the expressiveness of graph-rewriting systems by extending them with additional control, it is also important (besides the process-algebraic aspects) to evaluate if the control constructs provided by RePro are expressive enough, in the context of controlled graph rewriting scenarios. To this end, in the present section, we consider as a basis for comparison a reference control language called *Graph Programs* (GP) [49]. In particular, the following properties of GP make it appropriate to utilize it as reference:

- *Computational completeness:* A notion of completeness is central when reasoning about expressiveness. However, there are different interpretations of completeness for graph rewriting; in fact, it is straightforward to state that (DPO) graph rewriting itself is Turing-complete, as it is easy to simulate a Turing-machine using graph-rewriting rules [49]. Instead, our focus here is on the completeness of a control language *on top* of graph-rewriting rules. To that end, Habel and Plump interpret computational completeness in terms of partial functions on *labelled graphs*: their main result states that for every such function, there is a graph program computing it. (We refer to their paper [49] for motivations and details.) Nevertheless, we can construct a type graph corresponding to their family of labelled graphs, thus, GP can be meaningfully compared to RePro in terms of expressiveness.

- *Minimality:* Although there is a number of different control languages for graph rewriting, exhibiting a large scale in terms of richness in constructs; therefore, it is particularly appealing for expressiveness reasoning purposes that GP explicitly aims at employing as few constructs as possible to achieve completeness. Indeed, Habel and Plump show that their definition of GP, consisting of only three constructs, is minimal in the sense that each of those three constructs is necessary to retain completeness. Although establishing a correspondence between GP and RePro does not suffice to state that RePro can simulate every state-of-the-art controlled graph-rewriting approach (cf. also Chapter 3; a complete treatment of the topic is out of scope), still, it provides a neat baseline, also providing valuable insights for encoding more elaborate control structures (like, e.g., those of [102]).

- *Algorithms:* Besides the aforementioned formal aspects, GP also shares with RePro the aim of specifying algorithms whose basic operations are graph-rewriting steps. Moreover, both GP and RePro have a "slim" design, focusing on formal aspects and properties of such an approach, in contrast to richer, usability-centered languages (cf. Chapter 3). Note, however, that while RePro explicitly addresses concurrency, this is not the case for GP; as an expected consequence, parallel composition of RePro might be in fact omitted for establishing a correspondence to GP.

We start by recapitulating the three control constructs of graph programs, also stating under which circumstances the execution on a given graph G is considered to be successful, and what is the result graph of a successful execution:

1. *Non-deterministic choice* of a rule from a set of rules (also called an *elementary* graph program). A choice is successfully executed if at least one of the rules is applicable to G, and the (non-deterministic) result is the output graph of that rule application.

2. *Sequential composition* of arbitrary graph programs, where the result graph of a sequential component (i.e., sub-program) is passed as input graph to the subsequent one; consequently, the execution is successful if each component can be executed successfully.

3. *Maximal iteration* of the above sequences (potentially consisting of a single choice set), in the sense that the program is executed repeatedly as long as possible: after each successful iteration, the result graph of that single iteration is passed as input to the next iteration; the result graph of the overall execution is the graph on which the last iteration does not succeed. Accordingly, the overall execution of a maximal iteration construct is always successful.

In the following, we first recapitulate the formal definitions for GP; the syntax is iterative, based on the above constructs, and the semantics is a binary relation mapping input graphs to (sets of) output graphs, where an output graph is yielded if the program terminates. This semantic concept harmonizes with the aforementioned completeness notion of GP. Afterwards, we establish the correspondence between GP and RePro by providing an *encoding* of graph programs as RePro control processes.

Note that GP is based on graphs labeled over a label alphabet $\mathcal{C} = \langle \mathcal{C}_E, \mathcal{C}_N \rangle$ (for edge and node lables, respectively). Nevertheless, it is easy to construct a type graph $T_\mathcal{C}$ which is in a one-to-one correspondence to the labeling provided by $\mathcal{C}$; thus, $\mathcal{A}_\mathcal{C}$, the class of abstract graphs labeled over $\mathcal{C}$ (as introduced in [49]), is isomorphic to $\left[|\mathbf{Graph}_{T_\mathcal{C}}|\right]$. Nevertheless, we still use $\mathcal{A}_\mathcal{C}$ in definitions and results in the rest of this section, as the main result of completeness exploits constructions based on this assumption. However, due to the above observations, the representation of graphs does not influence our main goal in this section, namely reasoning about the expressive power of the control constructs in GP and RePro, respectively.

In the following, to avoid notational abundance, we simply use plain rule names (ranged over by p) in the syntax of GP; however, we always assume an underlying GTS with a mapping of those names to actual rule spans.

Definition 6.14 (Syntax of Graph Programs [49])**.** Graph programs *are inductively defined as follows (ranged over by GP):*

(1) A finite set of rule names $GP = \{p_1, \dots, p_n\}$ *is an* elementary *graph program.*

(2) If GP_1 *and* GP_2 *are graph programs, then* $GP_1; GP_2$ *is a graph program.*

(3) If GP is a graph program by (1) or (2), then $GP\downarrow$ *is a graph program.*

The set of graph programs is denoted as $\mathcal{GP}$*.*

Accordingly, the semantics of GP is also defined inductively. A graph program defines a binary relation on (labeled) graphs which maps input graphs to output graphs whenever the program terminates successfully. The underlying rule application relation $\Rightarrow_{GP}$ provides the input-output relation on (abstract labeled) graphs and, thus, the basis for the inductive semantics definition. Although $\Rightarrow_{GP}$ necessarily has to be distinguished (due to formal details) from the application relation $\Rightarrow$ used elsewhere in the thesis, this difference is immaterial to the following argumentation.

Due to the inherent non-determinism of rule applications as well as the GP choice construct, in general, a graph program maps a set of output graphs to a single input graph.

Definition 6.15 (Semantics of Graph Programs [49])**.** *Given a program GP over a label alphabet* $\mathcal{C}$*, the semantics of GP is a binary relation* $\rightarrow_{GP}$ *on* $\mathcal{A}_\mathcal{C}$*, which is inductively defined as follows:*

(1) $\rightarrow_{GP} = \Rightarrow_{GP}$ *if* $GP = \{p_1, \dots, p_n\}$ *is an elementary program (i.e., a rule name set).*

(2) $\rightarrow_{GP_1;GP_2} = \rightarrow_{GP_2} \circ \rightarrow_{GP_1}$*.*

(3) $\rightarrow_{GP\downarrow} = \{\langle G, H\rangle \mid G \rightarrow^*_{GP} H \text{ and } \nexists H' : H \rightarrow_{GP} H'\}$*.*

As for their respective semantic domains, RePro and GP do not directly correspond as RePro has an SOS semantics, whereas the semantics of GP is given by a relation on input-output graphs.

We proceed by defining such an *input-output* (IO) semantic relation for RePro.

Definition 6.16 (IO Semantics of RePro)**.** *The* IO semantics *of a control process P is given by the binary relation* $\Rightarrow^{IO}_P$ *over graphs in* $|\mathbf{Graph}_T|$ *defined as follows:* $G \Rightarrow^{IO}_P H$ *iff there exists a successful trace* (ρ_1, δ_1, N_1) $(\rho_2, \delta_2, N_2) \dots (\rho_n, \delta_n, N_n)\checkmark$ *from* (P, G) *s.t.* $G \overset{\delta_1}{\Rightarrow} \dots \overset{\delta_n}{\Rightarrow} H$*.*

Note that for technical convenience, the IO semantics of a graph process is defined over concrete graphs in $|\mathbf{Graph}_T|$, while in Definiton 6.15, we considered *abstract graphs* (as in [49]). However, the IO semantics can be easily shown to be invariant w.r.t. isomorphisms: if $G \Rightarrow^{IO}_P H$, then $G \Rightarrow^{IO}_P H$ for any $G \simeq G'$, $H \simeq H'$.

Now, we are ready to present the encoding of graph programs as RePro control processes. An encoding $[\![\_]\!] : \mathcal{GP} \to \mathcal{P}$ from graph programs to RePro control processes is a function; we provide a function definition which constructively synthesizes a RePro control process term from a given graph program. Afterwards, to verify the soundness of the encoding, we show that it preserves IO semantics (in the GP sense).

The encoding follows the inductive schema of GP and is based on the following ideas:

(1) For an elementary program, the corresponding control process is simply a choice over those rules. As an elementary program terminates after one rule application, each choice branch consists of a rule name followed by a **0**.

(2) For encoding sequences like $GP_1; GP_2$, we have to ensure that in exactly those states where GP_1 terminates successfully, the execution continues with GP_2. To that end, in the corresponding encoded RePro control processes, the closing **0**'s of the first process are replaced by a unique process identifier mapped to the second one by a defining equation.

(3) As for maximal iteration $GP\downarrow$ over a graph program GP, we again take a unique process identifier to capture iteration by recursion. In addition, we have to define a special control process (put as an alternative choice beside the recursive process) which acts as a *stopping criterion* for the iteration: this additional process should reach a **0** (i.e., terminate successfully) for exactly those graphs on which the program in the loop body does not succeed.

For providing those stopping processes as needed in the last case above, we follow the same inductive definition schema:

(1) If the loop body is an elementary program, then the stopping process consists of a single action with the empty rule ε as positive component and the set of all rules in the elementary program as non-applicability condition, followed by **0**. This **0** is reached exactly if the input graph is such that none of the rules is applicable on it.

(2) If the loop body contains a sequence like $GP_1; GP_2$, its failure means that either GP_1 fails or GP_1 is successful but GP_2 fails on the input GP_1 provides. This latter case is problematic as demonstrated by the following simple example:

Consider the simple program $GP = (\{p_1\} ; \{p_2\})\downarrow$. An encoding $[\![GP\downarrow]\!]$ applied to a graph G should have a successful derivation leading to G itself not only in the case p_1 cannot be applied to

G (encodable as $(\epsilon, \{p_1\}).\mathbf{0}$), but also when there is a "middle graph", e.g., H, such that $G \overset{p_1}{\Rightarrow} H$ and p_2 is not applicable to H.

As indicated by the example, there is a class of (iterated sequential) graph programs for which there is no general schema to define stopping processes, given the above IO semantics of RePro, well-aligned with GP semantics. To overcome this discrepancy between the encoding domain and the IO semantics, we identify a sub-language of so-called *backtracking-free programs* (Definition 6.18), which are still sufficient to obtain computational completeness of RePro, while avoiding the above issue.

(3) Although iterations cannot be iterated again directly, due to the above case, it is necessary to also provide a definition of a stopping process for maximal iterations. However, this should be an artificial process which cannot be executed successfully on any graph (as a maximal iteration is successful for any graph per definition); for this purpose, it is sufficient to specify a single-action process using any rule p, where p is used both as a positive component (rule to apply) and a non-applicability condition.

As a consequence of the requirements identified above, the formal definition of the encoding makes use of the following definitions: sequentialization of control processes (Definition 6.17) and the notion of *backtracking-free programs* (Definition 6.18) for delineating the domain of the encoding.

The sequentialization of two control processes P and Q is another process, written as $P \fatsemi Q$, essentially consists in correctly "plugging in" the definition of Q at those points of the execution of P where it would terminate, i.e., reach a $\mathbf{0}$ and fire a $\checkmark$.

Definition 6.17 (Sequentialization of Control Processes)**.** *Let $P, Q \in \mathcal{P}$ be two* sequential *processes (i.e., not containing $||$). Their* sequentialization *is the process $P \fatsemi Q := P[A_Q/\mathbf{0}]$ where $A_Q \in \mathcal{K}$ is a fresh identifier with defining equation $A_Q := Q$ and $P[X/\mathbf{0}]$ denotes the substitution of X for $\mathbf{0}$ in P, defined by induction on the structure of P as follows:*

- $\mathbf{0}[X/\mathbf{0}] = X$,
- $(\gamma.P)[X/\mathbf{0}] = \gamma.(P[X/\mathbf{0}])$,
- $(P + Q)[X/\mathbf{0}] = P[X/\mathbf{0}] + Q[X/\mathbf{0}]$, *and*
- $A[X/\mathbf{0}] = A_X$,

where A_X is a fresh process identifier with defining equation $A_X := P[X/\mathbf{0}]$ if $A := P$.

The notion of backtracking-free programs identifies that subset of graph programs which is handled (i.e., is within the domain) of our encoding. Intuitively, a program is backtracking-free if the applicability of the *whole* program to a given input graph can be "statically" decided by looking at the rules in the program term and the graph itself, *without the need to consider any other concrete graph*. Here, applicability of a program GP to a graph G means that there exists a graph H s.t. $G \rightarrow_{GP} H$. Formally, the backtracking-free property is captured by an inference system as presented in the following definition, where the single rules reflect the following intuitions:

- Every elementary graph program is backtracking-free, as we are able to check the applicability of each rule to G (cf. the axiom rule BF_{EL}).
- A graph program is backtracking-free if it is a maximal iteration, as such a program cannot fail (cf. the axiom rule BF_{ALAP}).
- A sequential graph program $GP_1; GP_2$ is backtracking-free if (1) GP_1 is backtracking-free and (2) GP_1 and GP_2 have a special relation such that GP_2 *always succeeds after* (ASA in short) GP_1 has been executed (cf. rule BF_{SEQ}).

Although the ASA relation is defined over intermediate graphs arising after the execution of GP_1, there are cases where the ASA relation of two programs might be shown through reasoning solely over program syntax and the constituting rules. As an example from our WSN scenario, consider the following simple programs (cf. Sect. 4.1):

$$GP_1 := \{p_{Cr}\}; \{p_{Cr}\}$$

$$GP_2 := \{p_{ActUS}\}; \{p_{ActUS}\}$$

Here, GP_2 will always succeed after GP_1: whenever we are successful in creating two (short unclassified) links (GP_1), we can be sure that afterwards, a program of activating two short unclassified links (GP_2) will also be successful, regardless of the concrete topology.

The following definition formalizes the foregoing considerations on backtracking-free programs.

Definition 6.18 (Backtracking-Free Programs). *A graph program GP is* backtracking-free (BF)*, denoted as $BF(GP)$, if this can be inferred by the following inference system:*

$$BF_{EL}\,\frac{}{BF(\{p_1, \ldots, p_n\})} \qquad BF_{ALAP}\,\frac{}{BF(GP\downarrow)}$$

$$BF_{SEQ}\,\frac{BF(GP_1) \quad ASA(GP_1, GP_2)}{BF(GP_1; GP_2)}$$

where the ASA predicate (for Always Succeeds After*) is defined as:*

$ASA(GP_1, GP_2)$ *iff for all* $G \rightarrow_{GP_1} X$ *there is an* H *such that* $X \rightarrow_{GP_2} H$.

The set of backtracking-free graph programs is denoted by $BF(\mathcal{GP})$.

A *failure process* $\widehat{GP}$ for a graph program GP is a control process which terminates successfully *exactly* on those input graphs where GP fails; moreover, the output of $\widehat{GP}$ is the unmodified input graph. Thus, failure processes can be used to test the applicability of GP on a given graph. Although failure processes cannot be specified constructively in general, this is possible for backtracking-free graph programs as shown in the following definition.

Definition 6.19 (Failure Processes)**.** *The* failure process $\widehat{GP}$ *of a backtracking-free graph program* $GP \in BF(\mathcal{GP})$ *is a* RePro *control process, defined inductively as follows:*

1. $\widehat{\{p_1, \dots, p_n\}} = (\varepsilon, \{p_1, \dots, p_n\}).\mathbf{0}$, *where* $\varepsilon\colon (\emptyset \leftarrow \emptyset \rightarrow \emptyset)$ *(see Definition 2.3);*
2. $\widehat{GP\downarrow} = (p, \{p\}).\mathbf{0}$, *where* p *is any rule;*
3. *if* $BF(GP_1)$ *and* $ASA(GP_1, GP_2)$, *then* $\widehat{GP_1 \,;\, GP_2} = \widehat{GP_1}$.

The following proposition ensures that failure processes indeed fulfill their purpose: (1) they do not modify the input graph (i.e., their IO semantics is an identity relation) and (2) they terminate successfully on every graph, and only on those, to which the original graph program is not applicable.

Proposition 6.7. *Let* $GP \in BF(\mathcal{GP})$ *be a backtracking-free program and let* $\widehat{GP}$ *be its failure process. Then for each graph* G,

$$(\widehat{GP}, G) \xrightarrow{\alpha}_D (P, H) \quad \text{iff} \quad P = \mathbf{0},\ H \cong G, \text{ and } \nexists G' \,.\, G \rightarrow_{GP} G'$$

Proof. We proceed by induction on the structure of the backtracking-free program GP.

1. If $GP = \{p_1, \dots, p_n\}$, there is a transition

   $$(\widehat{GP}, G) = ((\varepsilon, \{p_1, \dots, p_n\}).\mathbf{0}, G) \xrightarrow{(\varepsilon, \delta, \{p_1, \dots, p_n\})}_D (\mathbf{0}, H)$$

   iff (by rule MARK, Definition 5.2) none of the rules in GP is applicable to G (thus $\nexists G' \,.\, G \rightarrow_{GP} G'$) and $H \simeq G$, because pushouts preserve isomorphisms [38].
2. For a program $GP\downarrow$, given any graph G there is no transition from the graph process $(\widehat{GP\downarrow}, G) = ((p, \{p\}).\mathbf{0}, G)$, because the preconditions of rule MARK are inconsistent. In fact, $GP\downarrow$ can be applied successfully to any graph.

3. Given a program $GP = GP_1 \,;\, GP_2$, there is a transition $(\widehat{GP}, G) = (\widehat{GP_1}, G) \xrightarrow{\alpha}_D (P, H)$ iff (by inductive hypothesis, since $BF(GP_1)$) $P = \mathbf{0}$, $H \cong G$ and $\nexists G' \,.\, G \rightarrow_{GP_1} G'$, iff $P = \mathbf{0}$, $H \cong G$ and $\nexists G' \,.\, G \rightarrow_{GP_1;GP_2} G'$, because by hypothesis $ASA(GP_1, GP_2)$.

□

Definition 6.20 (Encoding Graph Programs as Processes). *The encoding function* $[\![\_]\!] : BF(\mathcal{GP}) \rightarrow \mathcal{P}$ *is defined as follows:*

- $[\![\{p_1, \dots, p_n\}]\!] := \sum_{i=1}^{n} p_i.\mathbf{0}$.
- $[\![GP_1 \,;\, GP_2]\!] := [\![GP_1]\!] \fatsemi [\![GP_2]\!]$.
- $[\![GP{\downarrow}]\!] := A_{GP\downarrow} \in \mathcal{K}$ *where* $A_{GP\downarrow} := [\![GP]\!] \fatsemi A_{GP\downarrow} + \widehat{GP}$.

First, note that the encoding generates only sequential processes. In order to prove the correctness of the encoding, we first present the following characterization of the IO semantics of a process obtained by sequentialization.

Proposition 6.8. *Let* $P, Q \in \mathcal{P}$ *be two* sequential *processes (i.e., not containing* $||$*), and* $P \fatsemi Q$ *be their sequentialization as in Definition 6.17. Then* $G \Rightarrow^{IO}_{P\fatsemi Q} H$ *if and only if there is a graph* X *such that* $G \Rightarrow^{IO}_{P} X$ *and* $X \Rightarrow^{IO}_{Q} H$.

Proof.

- **If:** By Definition 6.16, $G \Rightarrow^{IO}_{P} X$ iff there is a successful trace $(\rho_1, \delta_1, N_1) \dots (\rho_n, \delta_n, N_n)\checkmark$ such that $(P, G) \xrightarrow{(\rho_1,\delta_1,N_1)}_D (P_1, G_1) \dots \xrightarrow{(\rho_n,\delta_n,N_n)}_D (P_n, X) \xrightarrow{\checkmark}_D (\mathbf{0}, X)$. This is the case iff (by rule MARK) there is a derivation

$$G \overset{\delta_1}{\Rightarrow} G_1 \dots \overset{\delta_n}{\Rightarrow} X \qquad (\dagger)$$

and transitions $P \xrightarrow{(\rho_1,N_1)} P_1 \dots \xrightarrow{(\rho_n,N_n)} P_n \xrightarrow{\checkmark} \mathbf{0}$. Similarly, $X \Rightarrow^{IO}_{Q} H$ iff there is a derivation

$$X \overset{\delta'_1}{\Rightarrow} G'_1 \dots \overset{\delta'_m}{\Rightarrow} H \qquad (\ddagger)$$

and transitions $Q \xrightarrow{(\rho'_1,N'_1)} Q_1 \dots \xrightarrow{(\rho'_m,N'_m)} Q_m \xrightarrow{\checkmark} \mathbf{0}$.

Composing derivations $(\dagger)$ and $(\ddagger)$ we obtain a derivation $G \Rightarrow^{*} H$. By induction on the structure of P_n we can show that for each successful trace $\alpha_1 \dots \alpha_n \checkmark$ for P and for each transition $Q \xrightarrow{\alpha} Q'$ there is a trace $\alpha_1 \dots \alpha_n \alpha$ for $P \fatsemi Q = P[Q/\mathbf{0}]$ to Q'. This allows to compose the above traces for P and Q into a single successful trace for $P \fatsemi Q$, which together with the derivation $G \Rightarrow^{*} H$ witnesses the fact that $G \Rightarrow^{IO}_{P\fatsemi Q} H$.

- **Only if:** It can be shown by induction on the structure of the sequential process $P \fatsemi Q$, using Definition 6.17, that each successful trace for $P \fatsemi Q$ can be decomposed (possibly in more than one way) as $\sigma\tau\checkmark$, where $\sigma\checkmark$ is a successful trace for P and $\tau\checkmark$ is a successful trace for Q. This allows to split any successful trace from $(P \fatsemi Q, G)$ to $(\mathbf{0}, H)$ into a successful trace for (P, G) to $(\mathbf{0}, X)$ and one for (Q, X) to $(\mathbf{0}, H)$, as desired.

□

The following central proposition ensures that there is, as expected, a one-to-one correspondence between the semantics of backtracking-free graph programs and their encodings as RePro control processes.

Proposition 6.9. *For each backtracking-free graph program* $GP \in BF(\mathcal{GP})$ *and graph* $G \in |\mathbf{Graph}_{T_C}|$*, it holds that*

$$G \rightarrow_{GP} H \quad \textit{iff} \quad G \Rightarrow^{IO}_{[\![GP]\!]} H$$

Proof. We proceed by induction on the structure of GP.

(1) If $GP = \{p_1, \ldots, p_n\}$ is an elementary graph program, then $[G] \rightarrow_{GP} [H]$ iff $G \Rightarrow H$ with rule $p \in \{p_1, \ldots, p_n\}$ for some match m, iff $[\![GP]\!] = \sum_{i=1}^{n} p_i.\mathbf{0} \xrightarrow{(p,\emptyset)} \mathbf{0}$ and $G \overset{\delta}{\Rightarrow} H$ via p, iff by rule MARK we have $([\![GP]\!], G) \xrightarrow{(p,\delta,\emptyset)}_D (\mathbf{0}, H) \xrightarrow{\checkmark}_D (\mathbf{0}, H)$, that is $G \Rightarrow^{IO}_{[\![GP]\!]} H$.

(2) If $GP = GP_1; GP_2$, $[G] \rightarrow_{GP} [H]$ iff there is a graph X such that $[G] \rightarrow_{GP_1} [X]$ and $[X] \rightarrow_{GP_2} [H]$, iff (by induction hypothesis) $G \Rightarrow^{IO}_{[\![GP_1]\!]} X$ and $X \Rightarrow^{IO}_{[\![GP_2]\!]} H$ iff, by Proposition 6.8, $G \Rightarrow^{IO}_{[\![GP_1 \fatsemi GP_2]\!]} H$, iff $G \Rightarrow^{IO}_{[\![GP]\!]} H$.

(3) If $GP' = GP\downarrow$ then, by Definition 6.15, $[G] \rightarrow_{GP'} [H]$ iff there is a sequence of graphs $G = G_0, G_1, \ldots, G_n = H$, with $n \geq 0$, such that $[G_{i-1}] \rightarrow_{GP} [G_i]$ for all $i \in [1, n]$, and there is no graph X such that $[H] \rightarrow_{GP} [X]$. On the other hand by the definition of $[\![GP\downarrow]\!]$ one easily checks that $G \Rightarrow^{IO}_{[\![GP\downarrow]\!]} H$ iff there is a sequence of graphs $G = G'_1, G'_2, \ldots G'_n = H$ such that $G'_{i-1} \Rightarrow^{IO}_{[\![GP]\!]} G'_i$ for $i \in [1, n]$ and $(A_{GP\downarrow}, H) \xrightarrow{\alpha}_D (\mathbf{0}, H) \xrightarrow{\checkmark}_D (\mathbf{0}, H)$. Exploiting the induction hypothesis it only remains to show that

$$(A_{GP\downarrow}, H) \xrightarrow{\alpha}_D (\mathbf{0}, H) \qquad (\dagger)$$

if and only if $\nexists H' : H \rightarrow_{GP} H'$. First note that by the definition of $\fatsemi$, there cannot be any transition $[\![GP]\!] \fatsemi A_{GP\downarrow} \xrightarrow{\alpha} \mathbf{0}$, Therefore $(\dagger)$ must be obtained by a transition $\widehat{GP} \xrightarrow{\alpha} \mathbf{0}$. Thus we can conclude by showing, by induction on GP, that $(\widehat{GP}, H) \xrightarrow{\alpha}_D (\mathbf{0}, H)$ if and only if $\nexists H' : H \rightarrow_{GP} H'$, an immediate consequence of Proposition 6.7.

□

Finally, to conclude our considerations on expressiveness, we claim that the control constructs of RePro are indeed computationally complete in a precise sense based on the work of Habel and Plump [49]: as a consequence of Proposition 6.9 and Theorem 2 of the cited paper, control processes are also expressive enough to compute any partial function on graphs. This result can be obtained despite the fact that the above encoding is partial and is confined to backtracking-free programs: The proof of Theorem 2 of [49] is based on the concrete specification of a Turing machine [109], i.e., a universal graph program able to simulate any graph computations corresponding to input-output relations (cf. Definition 6.15). Thus, we can obtain an analogous result by showing that this Turing machine specification consists of graph programs which are backtracking-free (or might be directly encoded into RePro otherwise).

Corollary 6.2. *Given a label alphabet* $\mathcal{C}$ *and subalphabets* $\mathcal{C}_1$ *and* $\mathcal{C}_2$*, for every computable partial function* $f : \mathcal{A}_{\mathcal{C}_1} \to \mathcal{A}_{\mathcal{C}_2}$*, there exists a* RePro *control process that computes* f*.*

Proof. According to Theorem 2 in [49], there is a graph program GP which computes any such function, i.e., its semantics is a relation corresponding to f. Then the statement follows immediately from Proposition 6.9, if we are able to verify that the entire graph program presented in [49] for proving Theorem 2 can be encoded into a control process. In the following, we revisit the specification to see if it is eligible for such an encoding. In the rest of the proof, by 'paper', we specifically mean the work of Habel and Plump [49].

For most of the programs in the paper, we show that they are backtracking-free and, thus, encodable by the function in Definition 6.20.

In particular, most maximal iteration↓ operators in the paper are applied to elementary programs, which are BF by axiom BF_{EL} of Definition 6.18. An exception to this is the following definition, where `Choose`, `Inc`, `Relabel` and `Stop` are elementary:

$$\texttt{Prepare} = \langle \texttt{Choose}; \texttt{Inc}{\downarrow}; \texttt{Relabel}{\downarrow}; \texttt{Stop}\rangle{\downarrow}$$

For such a program, we infer the BF property in the following way using the inference rules in Definition 6.18 (where each subprogram name is replaced by its first letter), showing that the outermost↓ operator is applied to a BF program:

$$(2)\,\frac{(1)\,\dfrac{BF(\texttt{C}) \quad ASA(\texttt{C}, \texttt{I}{\downarrow})}{BF(\langle \texttt{C}; \texttt{I}{\downarrow}\rangle)} \qquad ASA(\langle \texttt{C}; \texttt{I}{\downarrow}\rangle, \texttt{R}{\downarrow})}{BF(\langle \texttt{C}; \texttt{I}{\downarrow}; \texttt{R}{\downarrow}\rangle)}$$

$$(3)\,\frac{BF(\langle \texttt{C}; \texttt{I}{\downarrow}; \texttt{R}{\downarrow}\rangle) \quad ASA(\langle \texttt{C}; \texttt{I}{\downarrow}; \texttt{R}{\downarrow}\rangle, \texttt{S})}{BF(\langle \texttt{C}; \texttt{I}{\downarrow}; \texttt{R}{\downarrow}; \texttt{S}\rangle)}$$

$ASA(GP_1, GP_2{\downarrow})$ holds for any programs GP_1 and GP_2, as $GP_2{\downarrow}$ never fails. This justifies the right premises of inferences (1) and (2). Instead establishing $ASA(\langle \mathtt{C}; \mathtt{I}{\downarrow}; \mathtt{R}{\downarrow}\rangle, \mathtt{S})$ requires an inspection of the rules of [49]. Specifically, each rule in `Choose` creates a loop; rules in `Inc` move the loop; rules in `Relabel` preserve the loop; and finally rules in `Stop` delete a loop. Therefore `Stop` can be applied successfully after applying `Choose` and any number of occurrences of `Inc` and `Relabel`.

All other programs of [49] to which the ALAP operator is applied can be shown to be backtracking-free with similar techniques, with the exception of the proposed implementation of the conditional program scheme, which is described as follows:

> $\mathtt{Ite}(K, P_1, P_2)$ checks whether the input graph equals K and executes P_1 or P_2 depending on whether the check is successful or not.

This scheme is used in the paper only for $K = \emptyset$ (the empty graph) and $K = \bullet$ (the discrete graph with one node). Assuming by induction hypothesis that we have encodings of P_1 and P_2, we can easily encode the scheme with $K = \emptyset$ as follows:

$$[\![\mathtt{Ite}(\emptyset, P_1, P_2)]\!] = (\varepsilon, \{p_\bullet\}).[\![P_1]\!] + p_\bullet.[\![P_2]\!]$$

where rule $p_\bullet = (\bullet \leftarrow \bullet \rightarrow \bullet)$ can be applied to a graph iff it is not empty. Similarly, we can encode the scheme with $K = \bullet$ using a set of rules that are applicable only if the graph contains at least one edge. This is correct because an inspection of the rules reveals that this program is never applied to a discrete graph with more than one node.

□

Summarizing, in this section, we have argued that RePro as a control language is also computationally complete, just as the Graph Programs of Habel and Plump [49] used as reference. This can be shown by encoding the universal program provided in that paper as a RePro process.

Nevertheless, it is important to note that this result does not imply that every graph program operator can be directly reflected in RePro processes. In particular, as also seen above through the necessary partiality of the provided encoding, the maximal iteration operation cannot be simulated in RePro in general; thus, graph programs still have to be considered *more expressive* than RePro control processes.

7

ANALYSIS OF CONTROLLED GRAPH-REWRITING PROCESSES

In the previous chapters, we have discussed our graph-rewriting process calculus RePro from a process-algebraic and language-centric perspective, thus addressing most of the challenges **C1-3** as in Section 3.3. However, an important aspect of **C3**, viz. **C3a**, the requirement for *abstract reasoning*, has not been addressed so far.

In this context, we can observe a conceptual gap not covered by the literature so far:

- On the one hand, related to both the concurrent semantics of graph rewriting and the topic of graph language specification, various abstractions over sets of graphs (i.e., graph languages) and over complete graph-rewriting systems have been proposed to cope with the infinity arising in graph-based *state spaces* as, e.g., in transition systems or model checking scenarios [19, 20, 24, 95, 106]. However, none of those approaches considers the possibility that the graph-rewriting systems underlying those state spaces or transition systems might be externally controlled. Although RePro as proposed in the present thesis has both explicit control and a rich LTS as core components, it is also not exempt from the same problem, namely, infinite state space explosion, rendering the semantics intractable or at least inefficient for reasoning over its states.
- On the other hand, if explicit control is part of a language or formalism, it is
  - either proposed in a usability-focused pragmatic setting, with emphasis on the (visual) system specification capacities, thus, leaving reasoning out of consideration [42, 84];
  - or, if state-based reasoning such as model checking is offered [45, 91, 108], tractability issues such as state space explosion are not dealt with.

  Again, even though RePro has a well-founded LTS semantics, we have not considered any reasoning approaches available in this framework so far.

Thus, in the present chapter, we aim at proposing the outlines of a RePro*-based abstract reasoning* approach, along the following objectives:

- **Abstraction and control:** The proposed approach should indeed aim at the aforementioned gap: providing a state space abstraction technique in a controlled setting.

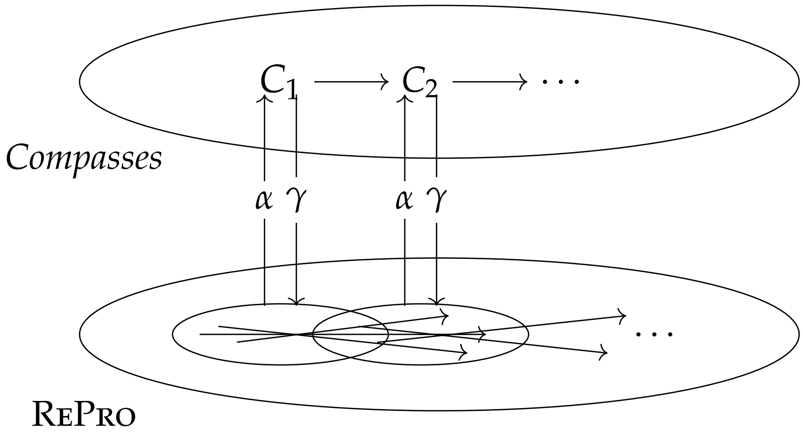


Figure 7.1: Abstraction Schema

- **Abstraction over operations:** Thereby, we should take into account the process-algebraic nature of the rich LTS semantics we have proposed for RePro in Chapters 4-5; i.e., we are looking for an abstraction harmonizing with this kind of operational semantics.
- **Abstraction over graphs:** To that end, we also develop an abstract representation of infinite graph families. This is necessitated by the presence of concrete graph instances in states: rendering the state space tractable, thus, requires an abstraction over graphs as well.
- **Abstract state predicates:** Finally, the proposed formalism should allow for verification on abstract states, based on model checking techniques already employed for graph rewriting, thus, we have to propose a formal model checking framework as part of our solution.

To meet all those needs, we aim at proposing a unifying concept of abstraction. Our approach is summarized as follows: given executions for some structures operating on concrete data (in our case, these are RePro processes rewriting graphs), an abstract semantic domain preserves those executions but represents data in an abstract way, potentially covering even an infinite number of concrete data instances in a single execution.

We demonstrate this setting schematically in Figure 7.1, calling the abstract domain *Compasses* and its sample elements C_1 and C_2, anticipating them to be defined in the next section. The functions α and γ intuitively represent the correspondences between elements of the abstract and the concrete domain in each direction.

In the following, first, we recapitulate and extend our motivating WSN scenario to illustrate our analysis approach (Sect. 7.1). Then,

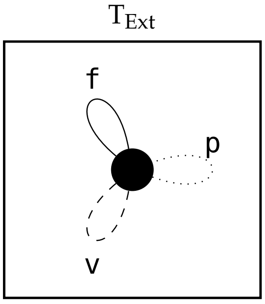


Figure 7.2: Type Graph T_{Ext} (with edges of T_{Top} omitted)

we propose *compasses* as an abstract domain (i.e., an abstraction over graph languages) and a corresponding *compass transition system*, i.e., an abstraction over the RePro LTS with compasses being components of states instead of graph instances (Sect. 7.2).

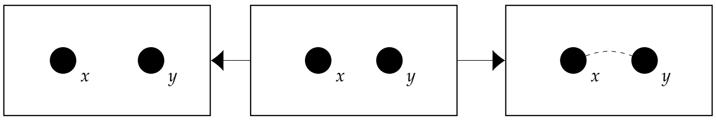
Afterwards, we connect the RePro transition system with the compass transition system by means of a *Galois connection* using the general ideas of *abstract interpretation* [27]. The aim of those developments is to conclude with a property preservation result, i.e., that if we are able to verify a model checking formula for a compass state, then the same formula holds for each RePro LTS state corresponding to that compass state (Sect. 7.3).

In this chapter, we focus on outlining the major formal ingredients towards a comprehensive RePro-based abstract verification. Thus, the proposal in its present state comes with some formal and conceptual limitations, while understanding the relation to some important approaches resembling ours remains a major work package in future investigations. We discuss some details of those limitations, potential future advancements and related work in the corresponding sections of Chapter 8.

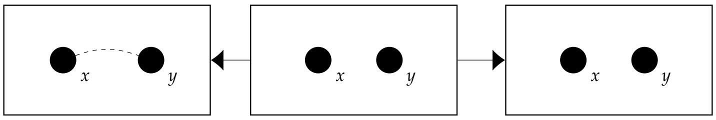
7.1 EXAMPLE: WSN OVERLAY VERIFICATION

In the previous chapter, we have focused on the *underlay* component of our example WSN model, and have not further specified the dynamics of its *overlay* so far. Recall that while the underlay consists of the physical links between sensor nodes, the overlay is a virtual network, accessible by an operator and defining constraints for improved routing functionality in the WSN (cf. also 1.3).

As a first step for providing our overlay model example, we have to extend the type graph T_{Top} (given in Figure 2.1) by further edge types for virtual links as well as some auxiliary constructs for our (simplified) routing constraint mechanism. As the overlay links reside within the same WSN as the underlay ones (and, thus, share the same node set), the extended type graph T_{Ext} has the same single node as T_{Top}; in Figure 7.2, we only show the new edge types and omit those of T_{Top} (Figure 2.1) for better readability. The label v, additionally



(a) p_{CrV}: Creating WSN Overlay Virtual Links



(b) p_{DelV}: Deleting WSN Overlay Virtual Links

Figure 7.3: DPO Rules for Administrator Interaction

indicated by a dashed line, refers to virtual links. Labels `f` (used only for loop edges) and `p` (additionally indicated by a dotted line) refer to auxiliary routing constructs expressed by edges in our model; their use is explained when we provide the behavior by DPO rules. (We omit labels `v`, `f` and `p` from rule specifications in the following, as dashing, dotting and looping unambiguously represent the respective type.)

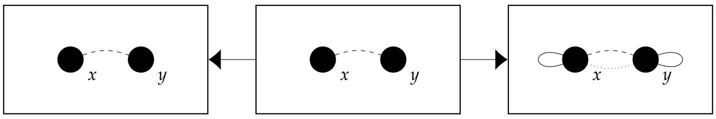
The first behavioral component of the overlay is that of *administrator interaction*, consisting of the simple creation and deletion of virtual links, thus, easily expressible by simple DPO rules p_{CrV} and p_{DelV} as shown in Figures 7.3a and 7.3b, respectively. The corresponding control process is specified as

$$P_{admin} := p_{CrV}.P_{admin} + p_{DelV}.P_{admin},$$

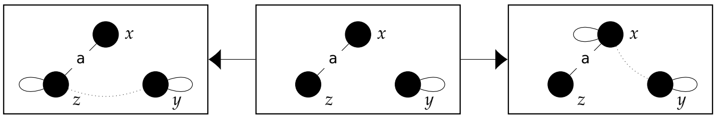
thus, we simulate administrator activity by simply allowing virtual links to appear or disappear at any time.

The most important part of our WSN overlay model (which we focus on in the rest of the chapter) is a simplified representation of routing constraints due to virtual links [65, 107], called *routing maintenance* in Section 1.3. We provide DPO rules to express this maintenance behavior, consisting in verifying and, occasionally, also repairing, underlay paths between end nodes of virtual links in order to optimize their activeness. (Note that we do not aim to reason about activeness guarantees and just provide a best-effort mechanism sufficient for our needs.) We split our behavioral model into a *search* and a *repair* part, specified by control processes P_{search} and P_{rep}, respectively. For the sake of simplicity, we omit underlay link lengths in the following (having no relevance here) and simply denote edge types by `a`, `i` and `u` to represent link statuses.

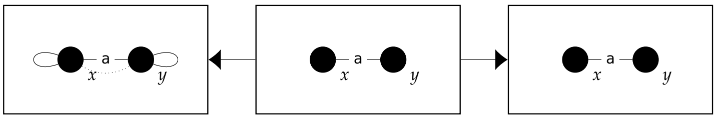
As for search (corresponding to operation *Search Active Path* in Sect. 1.3), we describe a mechanism which, upon noticing the appearance of an unhandled virtual link, initializes the corresponding auxiliary structures (rule p_{Aux}, Figure 7.4a), namely *flag* loop edges of type `f` for the end nodes of a virtual link as well as a *pairing* edge



(a) p_{Aux}: Initializing Auxiliary Routing Constructs



(b) p_{Search}: Moving Flags along Active Underlay Edges



(c) $p_{Success}$: Removing Auxiliary Constructs on Success

Figure 7.4: DPO Rules for Path Search

of type p to keep track of which flags belong together.[1] Thereupon, flags are able to navigate along active underlay links (rule p_{Search}, Figure 7.4b) and disappear on success, i.e., if a pair of flags get neighbors via an active edge (rule $p_{Success}$, Figure 7.4c). The process specification which matches the above algorithm is a simple interleaving choice:[2]

$$P_{search} := p_{Aux}.P_{search} + p_{Search}.P_{search} + p_{Success}.P_{search}$$

The other part of routing maintenance is a repair mechanism (operations *Activate Unclassified on Path* and *Unclassify on Path* in Sect. 1.3) which proactively activates or at least unclassifies (in hope for a further activation if it does not contradict underlay concerns) underlay links if the path search (by repeated applications of p_{Search}) gets blocked. Thus, although the repair process aims at optimizing towards overlay constraints, it also affects the underlay links thereby. The DPO rules corresponding to the above operations, p_{ActP} and p_{UcP} are shown in Figures 7.5a and 7.5b, respectively.

The specification of the repair process, thus, interleaves those operations if the search rule is not applicable in the current topology:

$$P_{rep} := (p_{ActP}, \{p_{Search}\}).P_{rep} + (p_{UcP}, \{p_{Search}\}).P_{rep}$$

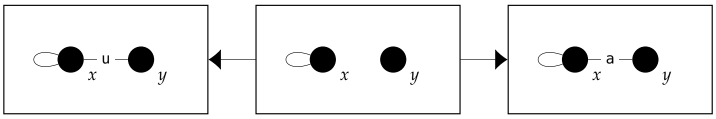
In turn, the overall overlay behavior is expressed by composing the search and the repair routines:

$$P_{overlay} := P_{search} \,||\, P_{rep}$$

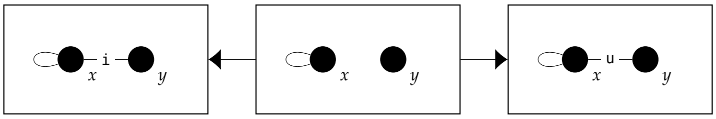
---

[1] Note that ensuring on a type graph level that f edges are loops, while v and p edges are non-loops is not treated.

[2] For the sake of neatness, we do not elaborate on preventing repeated initialization for the same virtual link; note that this can be easily implemented by a further "semaphore" edge.



(a) p_{ActP}: Activating Unclassified Links for Path Search



(b) p_{UcP}: Unclassifying Inactive Links for Path Search

Figure 7.5: DPO Rules for Path Repair

As mentioned before, we are aiming at verifying properties on an abstract level for executions of controlled graph-rewriting algorithms such as our WSN overlay specification. As a concrete example, given a property of single graph instances

"every virtual link is checked for an active path"

we are interested in a temporal property

"eventually, the above property holds on each execution path"

More formally, given a *predicate vpath* expressing the natural-language property above,[3] the μ-calculus property (for syntactic and semantic details, refer to [27, 60]) we want to verify is written as

$$\mu x.(vpath \vee (\Diamond true \wedge \Box x))$$

Motivated by the above example, we propose a general formal framework based on RePro allowing for abstract reasoning on such properties, thus, verifying properties for *graph algorithms* themselves, instead of just executions of given single input graphs.

7.2 COMPASSES: AN ABSTRACT DOMAIN FOR REPRO PROCESSES

We propose *compasses* as an abstract, finite representation of infinite graph languages. The underlying idea is the following: As a RePro control process proceeds, even if we do not have a concrete graph instance evolving through the execution, we still gather some information: by observing RePro control actions, i.e., which rules have been applied and which rules where not applicable, we are able to

---

[3] Note that we are not discussing the exact formulation of such predicates, as an abstract interpretation framework can be defined independently of those details. We remark that formulas for graph properties have been elaborated on different expressiveness levels in the literature, corresponding to different logical orders, e.g., in the context of *nested graph constraints* corresponding to first-order logics [96], as well as using *monadic second-order logic* [25]. A property as above, dealing with graph patterns of unbounded size, falls into the latter category.

approximate the graphs which *might* have arisen in each state, along observations such as: after applying a rule, its right-hand side definitely occurs in any resulting graph; the subsequent rule application is either disjoint to that occurrence, or overlaps with it; for some (specifically shaped) rules, non-applicability means that there is no occurrence of their left-hand side at all, etc. In this section, we deal with formally capturing this intuition in the setting of DPO graph rewriting. (Note that, as often in the thesis, typing is left implicit and we omit the identity of the type graph T from the definitions to avoid notational abundance.)

We start by giving the definition of a *compass*. According to the above intuition, a compass characterizes infinitely many graphs by *positive* patterns which *might* occur in those graphs, and *negative* patterns which *definitely* do not occur. Hence the name compass; it navigates us through the vast universe of graphs, with magnetic positive and negative *poles* showing the way, just as a needle of a compass does.

Recall the concept of canonical graphs (Definition 2.4); it is meaningful to utilize them in compasses, as we do not rely on graph identities, but only on their shapes in this setting, and, thus, we avoid unnecessary ambiguities within a compass that would arise due to graph isomorphism. In addition, to achieve a compact presentation of some formal advances below, we apply a mild restriction that elements in a negative pole are connected graphs. Note that this assumption is practically intuitive and, in particular, our examples adhere to it. Furthermore, we exclude by definition the absence of positive patterns, thus, ensuring that the language of a compass is not empty whenever the poles are not contradictory to each other.

Definition 7.1 (Compass)**.** *A* positive pole $\Pi \subseteq |\mathbf{CanGraph}|$ *is a finite,* non-empty *set of canonical graphs. A* negative pole $\bar{\Pi} \subseteq |\mathbf{CanGraph}|$ *is a finite set of* connected *canonical graphs.*

A compass $C = (\Pi, \bar{\Pi})$ *is a pair consisting of a positive pole* Π *and a negative pole* $\bar{\Pi}$*. The set of all compasses is denoted* $\mathcal{C}$*, ranged over by* C*. The positive and negative poles of* C *are also denoted* C_+ *and* C_-*, respectively.*

The language $\mathcal{L}(C) \subseteq |\mathbf{Graph}|$ *of compass* C *is a set of graphs s.t.* $G \in \mathcal{L}(C)$ *iff*

(i) $\exists\, X_+ \in C_+ : X_+ \rightarrowtail G$ *and*

(ii) $\forall\, X_- \in C_- : X_- \not\rightarrowtail G$*.*

A compass is consistent *if* $\mathcal{L}(C) \neq \emptyset$*.*

Note here that taking $\emptyset$ as a positive pole results in an empty compass language (i.e., *inconsistence*) due to the existential positive pole semantics.

For concrete examples of compasses, we revisit ideas from our motivating WSN model (and omit explicitly denoting canonicity of compass elements for notational clarity). First, consider a (canonical)

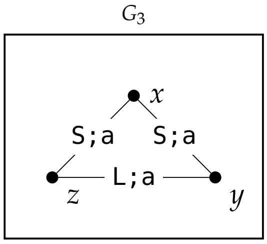


Figure 7.6: Graph G_3: A Pattern Avoided by kTC

topology graph G_3 (Figure 7.6) consisting of a single triangle of active links s.t. the link lengths correspond to the situation which the kTC algorithm aims at avoiding.

This graph can be used to compactly describe the language of all graphs having such a triangle using compasses (if we exclude looping of active edges on the type graph level or by forbidding those "folded" graphs through a negative pole):

$$C_3 := (\{G_3\}, \emptyset)$$

Similarly, the language of graphs *not* containing such triangles is also easily characterized:

$$C_{n3} := (\{G_\emptyset\}, \{G_3\})$$

where $G_\emptyset$ denotes the empty graph.

As a more elaborate example, with relevance to the foregoing WSN overlay model in Sect. 7.1, we describe each graph having an unfinished path search going on, such that the search will not terminate within two steps. Thereby, we might even reuse some rule components from $P_{overlay}$, writing

$$C_{search2} := (\{G_f\}, \{L_{Success}, L_{Success2}\})$$

where G_f is isomorphic to R_{Aux} without the virtual link, and $L_{Success2}$ looks like $L_{Success}$ with a 2-long active path between the flagged nodes.

A central observation is that there are specific compasses which actually play a role if used for language characterization, and whose languages subsume the language of a larger class of compasses. In general, a compass as above might contain *redundant* elements in their poles, whose removal would not modify the language we describe. Thus, it is useful to reason about *equivalence classes* of compasses instead, where the members of such a class describe the same language and, thus, are identical up to that redundancy.

For making precise this *minimality* of compasses, eliminating redundancy, we require two auxiliary graph notions: the well-known graph-theoretical notion of *cores* [51], as well as the *flattening* of a graph, which enumerates each graph epimorphic to the original one. Intuitively, a core of a graph is a subgraph of it such that the whole graph can be "folded" into that subgraph via a morphism. Note that each (finite) graph has a unique core up to isomorphisms.

Definition 7.2 (Core [51])**.** *A graph H is called a* retract *of a graph G if $H \hookrightarrow G$ and there exists a morphism $r : G \to H$ s.t. $r|_H = id_H$.*

A retract H of G is called a core *of G, denoted $H = core(G)$, if it has no proper retracts (i.e., retracts not isomorphic to H). For a set of graphs $\mathbb{G}$, $core(\mathbb{G})$ denotes $\{core(X) \mid X \in \mathbb{G}\}$.*

Definition 7.3 (Flattening)**.** *Given a graph G, the* flattening *$flat(G)$ of G is a set of graphs, defined as*

$$flat(G) := \{G' \mid G \xrightarrow{e} G', e \text{ is an epimorphism}\}$$

The flattening $flat(\mathbb{G})$ of a set of graphs $\mathbb{G}$ is defined as

$$\bigcup_{G \in \mathbb{G}} flat(G)$$

Now, we are ready to define minimality of compasses. First, we make precise which elements are considered redundant in a compass and define a function which eliminates them. This allows us to construct the aforementioned compass equivalence classes, each having a unique minimal member.

The minimalization of a compass is a constructive procedure, which, although formally captured as a function, can be thought of as happening in consecutive steps, as follows:

1. **Flattening:** We start with reducing the positive pole. First, the positive pole is flattened (Definition 7.3) as a preparation for the next stage: flattening effectively amounts to explicitly enumerating the potential graph patterns as they will appear in the graph language characterized by the positive pole, in an injective fashion, i.e., as the images of the respective pole element taken as a subgraph. For example, the graph G_3 in Figure 7.6 includes, in its flattened form, epimorphic images of it where two or even three nodes are conflated into a single one.

2. **Eliminating external redundancy:** Flattening allows for a fine-grained detection of the following situation: some epimorphic images might never appear in any graph of the language, if they are ruled out by an element of the negative pole. This happens if a negative pole element has an *injective* image in an element of the flattened positive pole: then, any graph potentially admitted by the latter would be at the same time ruled out by the former. Thus, we can safely remove such elements from the positive pole, not changing the language.

3. **Corification:** As a direct consequence of Definition 7.2, replacing a positive pole element with its canonical core does not change the set of graphs admitted by that element; moreover, doing so helps the next filtering step.

4. **Eliminating internal redundancy:** In the final step, we consider both poles. It holds that any pole element is redundant (i.e., can be omitted without changing the language) which has an incoming morphism (injective in the case of the negative pole) from another pole element. That this elimination step is deterministic also for the positive pole is due to the fact that at this point, we are working with cores; see the proof of Proposition 7.1.

The following definitions capture formally the above outline of our minimality function: the definition of a *core form* summarizes the first three preparatory steps, while the final definition of the minimalization function finishes the procedure. After the definitions, we verify claims in order to justify the definition; in particular, that the minimalization function provides a unique result as expected. Most importantly, we conclude the section with verifying our minimality notion: (1) Any compass has the same language as its minimal form. (2) Two compasses having the same language has a single unique minimal form.

Definition 7.4 (Pole Core Form)**.** *For a compass $C = (C_+, C_-)$, its* core form *is a compass $C^e = (C^e_+, C_-)$, defined as*

$$C^e_+ := core(flat(C_+) \setminus R^e)$$

with $R^e := \{X_+ \mid X_+ \in flat(C_+), \exists X_- \in C_- : X_- \hookrightarrow X_+\}$, leaving C_- unchanged.

Definition 7.5 (Minimal Compass)**.** *The function $min : \mathcal{C} \to \mathcal{C}$ takes each compass $C = (C_+, C_-)$ to its* minimal form

$$min(C) := (C^e_+ \setminus R^i_+, C_- \setminus R^i_-)$$

where

1. $R^i_+ := \{X^2_+ \mid X^2_+ \in C^e_+, \exists X^1_+ \neq X^2_+ \in C^e_+ : X^1_+ \to X^2_+\}$,
2. $R^i_- := \{X^2_- \mid X^2_- \in C_-, \exists X^1_- \neq X^2_- \in C_- : X^1_- \hookrightarrow X^2_-\}$.

Compasses C and C' are minimal-equivalent *if $min(C) = min(C')$. A set of minimal-equivalent compasses is a* compass equivalence class *(CEC), whose set is denoted as $[\mathcal{C}]$. The unique CEC for a given compass C is denoted $[C]$ s.t. $C \in [C]$. For any $[C] \in [\mathcal{C}]$, its unique minimal member is denoted $min([C])$.*

The language $\mathcal{L}([C])$ of a CEC is a graph language (i.e., a set of graphs) defined as $\mathcal{L}([C]) := \mathcal{L}(min([C]))$.

The relation $\preceq_\mathcal{L} \subseteq [\mathcal{C}] \times [\mathcal{C}]$ is defined as follows: $[C] \preceq_\mathcal{L} [C']$ if $\mathcal{L}([C]) \subseteq \mathcal{L}([C'])$.

For illustrating the minimality construction, revisit $C_{search2}$ as introduced above. Flattening G_f produces some different graphs, in

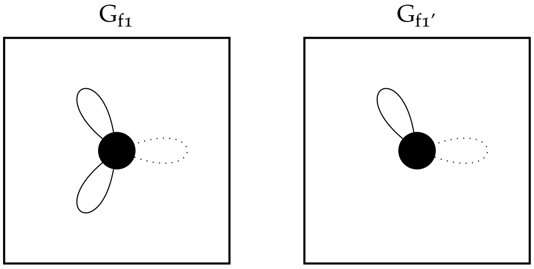


Figure 7.7: Graphs G_{f1} and $G_{f1'}$

particular, G_f itself, a graph G_{f1} with one node but two flag loops, and another one-node graph $G_{f1'}$ where the flag loops are also identified, both shown in Figure 7.7. None of those gets eliminated by R^e as in Definition 7.4, as particularly they do not contain underlay edges.

In the next step, we take the cores of each of those graphs. In particular, $G_{f1'}$ is the core of G_{f1}, thus, G_{f1} gets eliminated (its core is already in the pole). Next, due to $G_f \to G_{f1'}$, the latter gets eliminated by R^i_+ as in Definition 7.5.

The negative pole is minimal, as none of $L_{Success}$ and $L_{Success2}$ can be injectively embedded in the other graph.

Therefore,

$$min(C_{search2}) = C_{search2}$$

Note that minimalization would result in an empty positive pole for non-consistent compasses by the step which eliminates R_3. Therefore, as expected, we consider consistent compasses in the following (without changing the notation, for the sake of convenience).

Proposition 7.1. *For any $C \in \mathcal{C}$, $min(C)$ is unique.*

Proof. The statement follows from the uniqueness of the transformations involved in the definition of $min(C)$.

Cores and flattening are unique per definition.

R_3 is given uniquely for a fixed C_-.

For R_2, observe that monomorphisms within C_- yield a directed acyclic graph, in the sense that there exists no $X^1_-, X^2_- \in C_- : X^1_- \hookrightarrow X^2_- \hookrightarrow X^1_-$: isomorphism of distinct graphs would contradict the canonical setting.

For $R_1 \subseteq C^e_+$, observe that each element of C^e_+ is a core; again, there exists no $X^1_+, X^2_+ \in C^e_+ : X^1_+ \to X^2_+ \to X^1_+$: The image of $X^1_+ \to X^2_+ \to X^1_+$ is either identical with X^1_+, impossible because there exist no distinct isomorphic canonical graphs, or is a subgraph of X^1_+, but, then, X^1_+ would have a proper retract, contradicting that X^1_+ is a core. □

As the main property of minimalization, we show that it is indeed language-preserving.

Proposition 7.2. *Given a compass $C \in \mathcal{C}$, $\mathcal{L}(C) = \mathcal{L}(min(C))$.*

Proof.

- First, we show that *if* $G \in \mathcal{L}(C)$, *then* $G \in \mathcal{L}(min(C))$.

  In the following, graph set names correspond to Definition 7.5.

  We reach $min(C)$ from C in the following stages.

  1. *Flattening* C_+ *and removing* R_3: $\exists X_+ \in C_+ : X_+ \to G$ implies $\exists X^e_+ \in \mathit{flat}(C_+) : X^e_+ \hookrightarrow G$, as each (general) morphism can be decomposed into a sequence of an epi and a mono. Also, $\forall X_- \in C_- : X_- \not\hookrightarrow X^e_+$ because otherwise, $\exists X_- \in C_- : X_- \hookrightarrow X^e_+ \hookrightarrow G$, a contradiction. Therefore, $\exists X^e_+ \in (\mathit{flat}(C_+) \setminus R_3) : X^e_+ \to G$.
  2. *Taking cores*: For any X with $X \to G$, $core(X) \to G$ via $core(X) \hookrightarrow X$. Therefore, $\exists core(X^e_+) \in C^e_+ : core(X^e_+) \to G$.
  3. *Removing* R_1: Let us denote $core(X^e_+)$ as X^c_+. If $X^c_+ \in R_1$, then $\exists X'_+ \in C^e_+ : X'_+ \to X^c_+$ s.t. $\nexists X''_+ \neq X^c_+ : X''_+ \to X'_+$ (i.e., we take as X'_+ the "minimal" such graph). Thus, $X'_+ \in R_1$ iff $X^c_+ \to X'_+$, which leads to a contradiction: The image of $X^c_+ \to X'_+ \to X^c_+$ is either identical with X^c_+, impossible because there exist no distinct isomorphic canonical graphs, or is a subgraph of X^c_+, but, then, X^c_+ would have a proper retract, contradicting the previous stage.

     Therefore, $\exists X'_+ \in (C^e_+ \setminus R_1) : X'_+ \to G$ via $X'_+ \to X^c_+$.

  We have verified the positive criterion for $G \in \mathcal{L}([C])$, it remains to check the negative pole.

  4. *Removing* R_2: If $\forall X_- \in C_- : X_- \not\hookrightarrow G$, then it clearly holds also for $C_- \setminus R_2$.

- Second, we show that *if* $G \in \mathcal{L}(min(C))$, *then* $G \in \mathcal{L}(C)$.

  If there is $X^m_+ \in min(C)_+ : X^m_+ \xrightarrow{x} G$, then there is $X_+ \in C_+ : X_+ \xrightarrow{e} X^m_+$ s.t. e is epi. Therefore, also $X_+ \xrightarrow{x \circ e} G$.

  If for an $X_- \in (C_- \setminus min(C)_-) : X_- \hookrightarrow G$, then there would be a $X^m_- \in min(C)_- : X^m_- \hookrightarrow X_-$ (Definition 7.5) and, thus, $X^m_- \hookrightarrow G$, a contradiction. Therefore, $\forall X_- \in C_- : X_- \not\hookrightarrow G$.

□

As a direct important consequence, any compass has the same language as its CEC.

Proposition 7.3. *Given a compass* $C \in \mathcal{C}$, $\mathcal{L}(C) = \mathcal{L}([C])$.

Proof. The statement follows directly from Propositions 7.1 and 7.2. □

Another desired central property, shown in the following proposition, is that for minimal compasses, language equivalence implies actual identity.

Proposition 7.4. *Given* minimal *compasses* $C, C' \in \mathcal{C}$, $\mathcal{L}(C) = \mathcal{L}(C')$ *implies* $C = C'$.

Proof. We prove the statement by contradiction: let us show that for minimal compasses C and C', $C \neq C'$ implies $\mathcal{L}(C) \neq \mathcal{L}(C')$. Suppose first that $C_+ \neq C'_+$, and let us show that $\mathcal{L}(C) \neq \mathcal{L}(C')$. Without loss of generality, let $X \in C_+$ such that $X \notin C'_+$, and X is minimal in the partial order induced by $\rightarrow$,[4] i.e., there is no $X' \in C_+$ s.t. $X' \neq X$, $X' \rightarrow X$ and $X' \notin C'_+$. We show that (1) $X \in \mathcal{L}(C)$ and (2) $X \neq \mathcal{L}(C')$.

(1) There is an $X \in C_+$ such that $X \rightarrow X$. On the other hand, if there is a $Y \in C_-$ such that $Y \hookrightarrow X$, then $X \in R_3$ by Definition 7.5, and, thus, $X \notin C_+$ by minimality of C, which is a contradiction. Thus $X \in \mathcal{L}(C)$.

(2) Now suppose that there is $X' \in C'_+$ such that $X' \rightarrow X$. By the way we have chosen X, this implies that $X' \in C_+$ as well, but then $X \in R_1$ by Definition 7.5, and thus $X \notin C_+$ by minimality of C, which is a contradiction. Thus $X \notin \mathcal{L}(C')$.

Therefore, we have shown that $\mathcal{L}(C) = \mathcal{L}(C')$ implies $C_+ = C'_+$. Now suppose that $C_- \neq C'_-$, and, thus, without loss of generality, that $Y \in C_-$, $Y \notin C'_-$, and Y is of minimal size among such graphs. Since $C_+ (= C'_+)$ is not empty, let $X \in C_+$ be a graph of minimal size. We argue that $XY := X + Y$ (the disjoint union of graphs X and Y) is such that (1) $XY \notin \mathcal{L}(C)$ and (2) $XY \in \mathcal{L}(C')$, which concludes the proof.

(1) $Y \hookrightarrow XY$ and $Y \in C_-$, therefore, $XY \notin \mathcal{L}(C)$.

(2) Clearly, $X \rightarrow XY$ and $X \in C'_+$. It remains to show that there is no Y' in C'_- such that $Y' \hookrightarrow XY$. Since Y' is connected, either $Y' \hookrightarrow X$, but then $X \in R_3$ by Definition 7.5 and thus $X \notin C_+$, or $Y' \hookrightarrow Y$. In the last case, since Y' cannot be isomorphic to Y it has to be strictly smaller, but by the choice of Y this means that $Y' \in C_-$ as well, and therefore by $Y' \hookrightarrow Y$ we have that $Y \in R_2$ by Definition 7.5 and, thus, $Y \notin C_-$, a contradiction.

□

As an immediate corollary, the same identity holds for the respective CECs.

Corollary 7.1. *Given compasses* $C, C' \in \mathcal{C}$, $\mathcal{L}(C) = \mathcal{L}(C')$ *implies* $[C] = [C']$.

---

[4] Since C is minimal, C_+ contains only graphs that are cores and canonical. Therefore if $X \rightarrow Y$ and $Y \rightarrow X$, then $X = Y$; thus, $\rightarrow$ is a partial order—cf. also the proof of Proposition 7.1.

COMPASS TRANSITION SYSTEM. Just as RePro transitions represent a combination of control process steps on the one hand and rule applications, i.e., rewriting relations on graphs on the other hand, compass transitions are composed of the same control process steps and a corresponding relation on compasses. This relation, which we define in the following, naturally reflects DPO rule application semantics, and is constructed adequately such that a *compass step* captures all the *potential* changes to each potential input graph in a semantics-preserving manner. In particular, a compass step is, thus, deterministic for a single action.

Compasses allow for a separate, constructive step definition for positive and negative poles, respectively. Intuitively, the evolution of a positive pole for an action (ρ, N) should consider all the potential gluings of any existing positive pole element X with L, including, in particular, their disjoint union $X + L$ (expressing that in a potential graph already containing X, L might have a match disjoint to X), but also any other overlapping combinations on which ρ is applicable. Then, in turn, we apply ρ to those glued patterns and take the set of results as the next positive pole. Obviously, we do not have to consider those glued patterns on which some $p \in N$ is applicable or which contain some element from the negative pole.

In contrast, much more restrictive conditions hold for an element to become or remain a member of a negative pole during a step. Under certain (arguably rather strict) conditions for the rule ρ and a negative pole element $\bar{X}$, we are able to conclude that if $\bar{X}$ is not in an input graph for an application of ρ, then it will definitely not be present in the output graph. More precisely, we have to require that any gluing of R and $\bar{X}$ (which is injective on the freshly created parts of R) only overlaps for preserved and not for freshly created elements.

As we will show later (cf. Theorem 7.2), this requirement suffices to conclude that if a pattern is present in an output of an application of ρ, then it was so in the input as well—thus, this notion can be used to preserve negative pole elements along steps. In addition, the same conditions hold for *adding* left-hand sides of non-applicability conditions to the negative pole during a step, if we know that their left-hand side indeed never matches and not only further DPO conditions fail; this is guaranteed if a rule does not delete nodes (i.e., the *dangling condition* cannot be violated [38]). We call this requirement (generally, between a rule and a graph) *agnosticism*, and start by providing a separate definition for it for reading convenience.

Definition 7.6 (Agnosticism). *A rule ρ is* agnostic *to a graph X if*

1. *$r : K \to R$ is monic and*
2. *for each Y such that*
   a) *$R \to Y \hookleftarrow X$ are jointly epimorphic and*

b) *the following square is a pullback:*

$$\begin{tikzcd}
K \arrow[r, "id"] \arrow[d] \arrow[dr, phantom, "(PB)"] & K \arrow[d] \\
R \arrow[r] & Y
\end{tikzcd}$$

we have the following diagram:

$$\begin{tikzcd}
K \arrow[dr, hook, "r"'] & W \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(PB)"] & X \arrow[d, hook] \\
{} \arrow[ur, phantom, "(=)"] & R \arrow[r] & Y
\end{tikzcd}$$

Now, we are ready to formalize compass steps. In particular, starting from a given compass and a given control action, we provide separate constructions for the positive and the negative pole, respectively, which together yield the result of the compass step. In addition to agnosticism described above, we consider a further possibility for retaining a negative pole element during a step: if the application of a rule ρ could create that pole element by applying it at a structure forbidden by current non-applicability conditions, then we can conclude that it remains impossible for this structure to appear in the output graph. However, for similar reasons as for agnosticism, we have to restrict ρ to not creating nodes. (This time, creation is the relevant direction as we have to argue over inverse applications of ρ, cf. the proof of Theorem 7.2.)

Definition 7.7 (Compass Step)**.** *The relation* $\Rrightarrow \subseteq \mathcal{C} \times (\mathcal{R}^* \times 2^{\mathcal{R}}) \times \mathcal{C}$ *is a labeled relation on compasses s.t.*

$$(\Pi, \bar{\Pi}) \xRightarrow{(\rho,N)} (\Pi', \bar{\Pi}') \text{ if}$$

1. $X' \in \Pi'$ *iff there exists a diagram in* **CanGraph**

$$\begin{tikzcd}
X \arrow[dr, "x"] & Can(L) \arrow[d, dashed, "m"] \arrow[dr, phantom, "(PO)"] & Can(K) \arrow[l, dashed, "l"'] \arrow[r, dashed, "r"] \arrow[d, dashed, "k"] \arrow[dr, phantom, "(PO)"] & Can(R) \arrow[d, dashed, "n"] \\
 & G & D \arrow[l, "f"'] \arrow[r, "g"] & X' \\
 & \bar{X}_1 \cdots \bar{X}_{|\bar{\Pi}|} \arrow[u, hook, "/" marking, shift left] \arrow[u, hook, "/" marking, shift right] & &
\end{tikzcd}$$

where

a) $X \in \Pi$,

b) $\bar{X}_i \in \bar{\Pi}$ $(i = 1 \ldots |\bar{\Pi}|)$,

   c) $\forall p \in N : G \overset{p}{\not\Rightarrow}$ *and*

   d) *the pair* (x, m) *is jointly epimorphic.*

2. $\bar{X}' \in \bar{\Pi}'$ *iff* $\bar{X}'$ *is connected and*

   a) ρ *is agnostic to* $\bar{X}'$*, where*

      i. $\bar{X}' \in \bar{\Pi}$

      or

      ii. $\bar{X}' = Can(L_p)$ *for some* $p \in N$ *where* p *does not delete nodes*

   or

   b) $\bar{X}' \in \bar{\Pi}$, ρ *does not create nodes,* $Can(L_p) \overset{\delta}{\Rightarrow} \bar{X}'$ *for a rule* $p \in N$ *not deleting nodes, with* δ *in* **CanGraph** *s.t.* $\rho(\delta) = Can(\rho)$, $m(\delta) = Can(L) \overset{m}{\hookrightarrow} Can(L_p)$.

As an example, consider a step over p_{Search} (thus, with an empty N) on $min(C_{search2})$ as specified at the end of the previous section. As for the evolution of the positive pole, L_{Search} can be joined with G_f in various jointly epimorphic ways, in particular, by injectively embedding G_f into L_{Search}, yielding L_{Search} itself (as we are canonical in compasses). Thus, R_{Search} is in the result of the positive pole of the compass step. Note, however, that due to p_{Search} not being agnostic to $L_{Success}$ nor to $L_{Success2}$, as expected, we lose our negative compass elements as the assumption of $C_{search2}$ does not hold anymore. If, instead, we consider a step (from an adequate compass immaterial here) over the action $(p_{UcP}, \{p_{Search}\})$ as in P_{rep}, L_{Search} appears in the resulting negative pole, correctly expressing that the search rule does not match in any topology after that action, as there are no morphisms between the left- and right-hand sides of p_{UcP} and L_{Search} due to an edge type mismatch.

The compass transition system is defined analogously to the RePro transition system (Definition 5.2): whenever the control process component of a state is able to perform an action, the compass component also performs a step over the same action. (Note that, in contrast to the RePro transition system, this behavior is non-blocking: Definition 7.7 never results in an empty positive pole, as there is always at least one option, namely disjoint composition, for combining a previously existing element with the left-hand side of the rule.)

Definition 7.8 (Compass Transition System)**.** *The* compass transition system *is an LTS* $(\mathcal{P} \times [\mathcal{C}], (\mathcal{R}^* \times 2^{\mathcal{R}}), \rightarrow_C)$*, where* $\rightarrow_C$ *is the least relation satisfying the following rule:*

$$\text{COMPASS}\ \frac{P \xrightarrow{(\rho,N)} P' \qquad min([C]) \overset{(\rho,N)}{\Longrightarrow} C'}{(P,[C]) \xrightarrow{(\rho,N)}_C (P',[C'])}$$

COMPASS LANGUAGES. For providing an abstract verification approach through compasses and abstract interpretation, we do not intend to capture and involve any graph languages, but only those which actually arise during rule applications.

As a first simplification, compasses intentionally describe only infinite graph languages: Although the output set of a graph-rewriting system might be finite, such graph languages cannot be characterized by smaller (either in number or in size) positive or negative patterns in general. Having noted this, we identify the study of classes of finite graph languages admitting a "quantitative reduction" as future work.

As for infinite graph languages, languages which defy any "regular" (recurring) pattern characteristics arguably go beyond the expressive power of graph-rewriting systems and, thus, require completely different formalization and analysis approaches than those in the present thesis.

Therefore, in the thesis, we confine ourselves to graph languages having such regularities, i.e., exactly ones described by compasses. We argue that many relevant application examples fall into this category: in graph-based system modeling, one often formulates language characteristics by requiring the presence or absence of positive or negative patterns, respectively. For example, in the context of our WSN scenario, we might reason about classes of topologies such as: topologies *not containing an active triangle* or *containing an unexplored virtual link*. More precisely, *compass-definable languages* capture those graph languages which arise from applying a GTS to any arbitrary graph, as we will demonstrate below.

Definition 7.9 (Compass-Definable Graph Language). *A graph language* $\mathbb{G} \subseteq |\mathbf{Graph}|$ *is* compass-definable *(abbreviated as CGL) if* $\exists [C] \in [\mathcal{C}] : \mathcal{L}([C]) = \mathbb{G}$.

The set of all compass-definable graph languages is denoted $\mathcal{L}_{\mathcal{C}}$ *and also referred to as CGL.*

To conclude this section, we investigate some general properties of CGL, inspired by literature on abstract graph language specifications [20, 24, 106]. In particular, inspited by the general formal characterization criteria of Corradini et al. [20], we reason about some basic closure and decidability properties.

As for the latter, luckily, the definition of compasses (Definition 7.1) allows for an easy decision process for basic membership, as summarized below.

Proposition 7.5. *Given a compass C and a graph G (typed over the same type graph), the following statements are decidable:*

1. $G \in \mathcal{L}(C)$
2. $\mathcal{L}(C) = \emptyset$

Proof.

1. Given G, we can decide by a finite procedure if it fulfills the requirements of Definition 7.1, in which case it is in $\mathcal{L}(C_i)$, otherwise not.

   First, we enumerate the elements X_+ in C_+ and check, for each, if there is a morphism $X_+ \to G$ (known to be decidable [20]). If there was no such X_+, we can conclude that $G \notin \mathcal{L}(C)$.

   Then, if there is such an X_+, we enumerate the (finitely many) proper subgraphs of G and check if any of them is isomorphic to any of the elements of C_-, allowing for a final decision.

2. By Propositions 7.1 and 7.3, $\mathcal{L}(C)$ is empty iff the positive pole of $min(C)$ is empty, which can be easily verified by the construction in Definition 7.5.

□

Although, in the frame of the present thesis, we leave open the general question of language inclusion decidability, i.e., if we can always tell if $\mathcal{L}(C_1) \subseteq \mathcal{L}(C_2)$, we observe that in some specific cases, we directly have an answer (with $X^i_+ \in C^i_+$ and $X^i_- \in C^i_-$, $i \in \{1,2\}$):

- If $\exists X^2_- \in C^2_-$ such that $\exists X^1_+ : X^2_- \hookrightarrow X^1_+$ but $\nexists X^1_- : X^2_- \hookrightarrow X^1_-$, then $\mathcal{L}(C_1) \nsubseteq \mathcal{L}(C_2)$, as we can find a $G \in \mathcal{L}(C_1)$ which will be excluded in $\mathcal{L}(C_2)$ through X^2_-.

- If the negative poles are empty, then $\mathcal{L}(C_1) \subseteq \mathcal{L}(C_2)$ *if and only if* $\forall X^1_+ \exists X^2_+ : X^2_+ \to X^1_+$: In this case, if a graph G is admitted in $\mathcal{L}(C_1)$ by X^1_+, it will be also admitted in $\mathcal{L}(C_2)$, but if we have an X^1_+ *not* having a corresponding X^2_+, then $X^1_+ \in \mathcal{L}(C_1)$ but $X^1_+ \notin \mathcal{L}(C_2)$ as there is no element in C^2_- admitting it.

Unfortunately, due to the peculiar interplay between positive and negative poles, closure properties under standard set operations (e.g., union) are even harder to reason about and proving corresponding statements is left for future work. We might, however, obtain an intuitive impression of the problem at hand by the following intuition on extending compasses: The positive pole of a compass splits the set of all graphs into two parts, those admitted by it and those not. Independently, the negative pole introduces another splitting into graphs excluded and graphs not excluded by it. Now, the language of the compass turns out to be the intersection of that set admitted by the positive pole and that set not excluded by the negative pole. However, adding even a single graph to, e.g., the positive pole interferes with this scheme: while the set of admitted graphs grows monotonously, the overall growth of the compass language is derived from the intersection of the non-excluded graph set and the freshly admitted graphs, where the characterization of this interplay might largely depend on the morphism relations between the involved graphs.

Executing a formal analysis of decidability of CGL closure of compass union as well as similar properties of other set operators (intersection, complement) is left for future work.

7.3 ABSTRACT INTERPRETATION OF REPRO BY COMPASSES

In this section, we formally define our *abstract interpretation* framework for RePro based on compasses. Thereby, we follow the outlines of the work by Dams et al. [27] on abstract interpretation for reactive systems, based, in turn, on the original unifying program analysis framework proposed by Patrick and Radhia Cousot [26].

The main idea behind abstract interpretation is that each element of the abstract domain describes a (potentially infinite) number of concrete domain elements, ideally in a finitely representable way. In turn, there are *abstraction* and *concretization* functions (usually called α and γ, respectively), the former mapping concrete elements to the abstract one describing them, while the latter lists the concrete elements for a given abstract element.

In addition, both the concrete and the abstract domain might be embedded in states of a respective transition system [27]. This setting calls forth an abstract reasoning approach as required by challenge **C3a**, based on traditional *model checking* [5], i.e., the verification of temporal properties of transition systems with state predicates. If the abstract interpretation framework is used adequately, then properties holding for an abstract path are preserved for each concrete path conforming to it.

Note, however, that from a pragmatical perspective, we have a different take on the utilization of abstract interpretation. Usually, given a concrete domain, one derives an adequately suited, artificial abstract domain which then fits for verification purposes by construction. In contrast, we set off by providing an abstract transition system, i.e., a transition system whose states are composed of control processes and compasses. Afterwards, along the lines of [27], we connect the concrete and the abstract compass states by a *Galois connection*, enabling to prove our main theorem: that our abstract states indeed subsume each potential concrete RePro executions. In turn, we obtain the desired abstract verification framework through μ-calculus formulas as in [27].

A *Galois connection* makes precise and formal the intuition behind that "connection" we mentioned above, between concrete and abstract (sets of) elements, in turn embedded in states of their respective transition systems. Consequently, as abstract elements describe a set of concrete objects, we are working with such sets on the concrete side. Intuitively, a Galois connection consists of a pair of (total and monotonic) functions, mapping sets of concrete elements to abstract ones and vice versa, such that any concrete set is preserved after abstraction and re-concretization, and also, any abstract element remains at least as accurate after concretization and re-abstraction (according to an accuracy ordering on the abstract domain). Formally, given are two sets of objects: the set of *concrete* objects is denoted $\mathcal{B}$, while the set of *abstract* objects is denoted $\mathcal{A}$. Let $\mathcal{K} \subseteq 2^{\mathcal{B}}$ and $\mathcal{A}$ be the *concrete domain* and the *abstract domain*, respectively. Both domains are equipped with a partial order.

Definition 7.10 (Galois Connection). *Functions* $\alpha : \mathcal{K} \to \mathcal{A}$ *and* $\gamma : \mathcal{A} \to \mathcal{K}$ *are a* Galois connection *from poset* $(\mathcal{K}, \subseteq)$ *to poset* $(\mathcal{A}, \preceq)$ *if*

1. *both* α *and* γ *are total and monotonic,*
2. *for all* $K \in \mathcal{K}$, $K \subseteq \gamma \circ \alpha(K)$ *and*
3. *for all* $A \in \mathcal{A}$, $\alpha \circ \gamma(A) \preceq A$.

A Galois connection is a Galois insertion *if in addition,*

4. $A \preceq A' \Leftrightarrow \gamma(A) \subseteq \gamma(A')$.

As our main motivation is to avoid that each execution starts with a fixed input graph, we are aiming at defining an abstract domain which characterizes states of a controlled graph-rewriting process abstractly, but based on their respective control action history. Consequently, there is an *abstract transition system* whose states and transitions correspond to multiple states and transitions in RePro, respectively. In this context, correctness means that each abstract transition captures each concrete transition whose source state corresponds to the source of the abstract transition. For RePro, this means that for each abstract transition, each rule application from graphs described by its source leads to an output graph which is described by its target. Correctness is necessary for any domain pairs for a purposeful use of abstract interpretation.

Following the outline of the work of Dams et al. [27], on both sides of our connection, a *model* represents all ingredients together: the set of objects (which are sets again for the concrete side) with their respective partial orders as well as their underlying transition systems (implicit in the formal Galois connection definition). In particular, as a concrete model, we consider $\mathcal{L}_{\mathcal{C}}$ with subset inclusion as partial ordering and RePro as transition system. As an abstract model, we consider $\mathcal{C}$ with $\preceq_{\mathcal{L}}$ and the compass transition system. To justify this choice, we first have to check that $\preceq_{\mathcal{L}}$ is indeed a partial order.

Proposition 7.6. $\preceq_{\mathcal{L}} \subseteq [\mathcal{C}] \times [\mathcal{C}]$ *(Definition 7.5) is a partial order.*

Proof. Reflexivity and transitivity is obvious by the underlying subset inclusion. For antisymmetry, observe that $\mathcal{L}([C]) \subseteq \mathcal{L}([C'])$ and $\mathcal{L}([C']) \subseteq \mathcal{L}([C])$ imply $\mathcal{L}([C]) = \mathcal{L}([C'])$; by Proposition 7.3, this means $\mathcal{L}(C) = \mathcal{L}(C')$; then, by Corollary 7.1, $[C] = [C']$. □

We have now prepared the ground for defining the abstraction and concretization functions from CGL to compasses and vice versa, and verifying that our definitions indeed yield a Galois connection. In particular, although this is not strictly necessary for the further developments, due to the close relation of our domains, we get a Galois insertion.

Definition 7.11 (Compass Abstraction Function)**.** *The* compass abstraction function $\alpha : \mathcal{L}_\mathcal{C} \to [\mathcal{C}]$ *is a function mapping compass-definable graph languages to compass equivalence classes s.t. (i)* α *is total and (ii) for any* $\mathbb{G} \in \mathcal{L}_\mathcal{C}$*,* $\mathcal{L}(\alpha(\mathbb{G})) = \mathbb{G}$*.*

Note that such an α is provided over $\mathcal{L}_\mathcal{C}$ due to Definition 7.9.

Definition 7.12 (Compass Concretization Function)**.** *The* compass concretization function $\gamma : [\mathcal{C}] \to \mathcal{L}_\mathcal{C}$ *is a function from compass equivalence classes to compass-definable graph languages, defined as*

$$\gamma([C]) := \mathcal{L}([C])$$

From now on, α and γ denote the specific functions above.

Proposition 7.7 (Compass Galois Insertion)**.** (α, γ) *is a Galois insertion for* $(\mathcal{L}_\mathcal{C}, \subseteq)$ *and* $([\mathcal{C}], \preceq_\mathcal{L})$*. Moreover,* $\alpha \circ \gamma = id_{[\mathcal{C}]}$*.*

Proof. First, we check the properties in Definition 7.10 for (α, γ).

1. *α and γ are total and monotonic:* Totality is obvious by definition. For α, we have to show that $\mathbb{G} \subseteq \mathbb{G}' \Rightarrow \alpha(\mathbb{G}) \preceq_\mathcal{L} \alpha(\mathbb{G}')$ for $\mathbb{G}, \mathbb{G}' \in \mathcal{L}_\mathcal{C}$, which is a direct consequence of Definition 7.11. For γ, we have to show that $[C] \preceq_\mathcal{L} [C'] \Rightarrow \gamma[C] \subseteq \gamma[C']$, a direct consequence of Definitions 7.12 and 7.5.
2. $\mathbb{G} \subseteq \gamma(\alpha(\mathbb{G}))$*:* Follows directly from Definitions 7.11 and 7.12.
3. $\alpha(\gamma([C])) \preceq [C]$*:* Follows directly from Definitions 7.11 and 7.12.
4. $[C] \preceq [C']$ *if and only if* $\gamma[C] \subseteq \gamma[C']$*:* Follows directly from Definitions 7.5 and 7.12.

The claim that $\alpha \circ \gamma = id_{[\mathcal{C}]}$ also follows directly from Definitions 7.11 and 7.12. □

In accordance with the foregoing observations, the following main theorem of this section is based on our definition of compass concretization corresponding to the graph languages of compasses in abstract states: it verifies the desired *correctness* property by claiming that a compass step indeed reflects all the RePro steps being compatible with it. As a preparation, we recall two fundamental properties of DPO graph rewriting used in the proof of our main Theorem 7.2.

Proposition 7.8 (Pushouts Preserve Monomorphisms [38])**.** *If $B \xrightarrow{n} D \xleftarrow{g} C$ is a pushout of $B \xleftarrow{f} A \xrightarrow{m} C$ in* **Graph** *and m is a monomorphism, then n is a monomorphism too.*

Theorem 7.1 (Embedding Theorem [38])**.** *A morphism $G \xrightarrow{k} G'$ is* consistent *with a rule application $G \overset{\delta}{\Rightarrow} H$, if all the nodes and edges of G identified by k (*identification points*) and all the nodes in G whose images in G' have adjacent edges not in $k(G)$ (*dangling points*) are in $f(D)$ (cf. the diagram in Definition 2.5).*

Given a rule application $G \overset{\delta}{\Rightarrow} H$ and a morphism $G \xrightarrow{k} G'$, there exist $G' \overset{\delta'}{\Rightarrow} H'$ and $k' : H \to H'$ if k is consistent with δ (with $\rho(\delta) = \rho(\delta')$).

Theorem 7.2 (Abstraction Correctness)**.** *If $(P, [C]) \xrightarrow{(\rho,N)}_C (P', [C'])$, then* for each *$\delta$ with $G \in \gamma[C]$ as input graph and ρ as rule, s.t. $\forall p \in N : G \overset{p}{\not\Rightarrow}$, there exists a* RePro *transition $(P, G) \xrightarrow{(\rho,\delta,N)}_D (P', G')$ with $G' \in \gamma[C']$.*

Proof. We show that $(P, [C]) \xrightarrow{(\rho,N)}_C (P', [C'])$ indeed implies that $\gamma([C'])$ contains all graphs arising from rule applications from $\gamma([C])$.

More precisely, we show that if $G \in \gamma([C])$, then $(P, G) \xrightarrow{(\rho,\delta,N)}_D (P', G') \Rightarrow G' \in \gamma([C'])$.

First, we show that for each such G', $\exists X' \in C'_+ : X' \hookrightarrow G'$ and $\forall X'_- \in C'_- : X'_- \not\hookrightarrow G'$.

We start with demonstrating X'.

- If $\exists X_+ \in C_+ : X_+ \overset{x}{\hookrightarrow} G$, and δ has as match $m : L \to G$, then there is the following diagram, whose DPO part is denoted δ':

  $$\begin{tikzcd}
  X \arrow[dr, "x'"'] & Can(L) \arrow[d, "m'" description] \arrow[dr, phantom, "(PO)"] & Can(K) \arrow[l, "l" description] \arrow[r, "r" description] \arrow[d, "k" description] \arrow[dr, phantom, "(PO)"] & Can(R) \arrow[d, "n" description] \\
  & X_m & D \arrow[l, "f" description] \arrow[r, "g" description] & X' \\
  & \bar{X}_1 \cdots \bar{X}_{|\tilde{\Pi}|} \arrow[u, hook, shift left, "/" marking] \arrow[u, hook, shift right, "/" marking] & &
  \end{tikzcd}$$

  where X_m is the restriction of G to $x(X) \cup m(L)$, m' is the canonical match and x' is the monomorphism corresponding to m and x, respectively. Thus, (m', x') are jointly epimorphic.

- $X_m \hookrightarrow G$, thus, X_m fulfills the non-existence conditions of $\bar{\Pi}$ and the non-applicability conditions of N are fulfilled. Hence, $X' \in C'_+$. We still have to show that $G \overset{\delta}{\Rightarrow} G'$ implies $X' \hookrightarrow G'$.

- According to the Embedding Theorem (Theorem 7.1 [38]), $X_m \overset{\delta'}{\Rightarrow} X'$, $G \overset{\delta}{\Rightarrow} G'$ and $m_x : X_m \hookrightarrow G$ imply the existence of a morphism

$X' \to G'$ if δ' does not delete any identification or dangling points of m_x. As m_x is mono, there are no identification points. The existence of δ ensures that m' does not delete any dangling points. Hence, $X' \to G'$. Moreover, this morphism is mono due to Proposition 7.8 [38].

We still have to show that $\forall X'_{-} \in C'_{-} : X'_{-} \not\hookrightarrow G'$.

- Suppose there is $X''_{-} \in C'_{-} : X''_{-} \overset{x}{\hookrightarrow} G'$.
- If $X''_{-} \in \bar{\Pi}$ or $X''_{-} = Can(L_p)$ for some non-node-deleting p, $X''_{-} \not\hookrightarrow G$.
- The union Y of $n(R)$ (where $n : R \to G'$ is the co-match in δ) and $x(X''_{-})$ in G' takes the form of a gluing Y as required by Definition 7.6, because x is monic and any co-match is necessarily injective on the freshly created elements.
- Then, the pullback object W of $R \to Y \hookleftarrow X''_{-}$ (cf. Definition 7.6 for notation) with $W \to K \hookrightarrow R = W \to R$ guarantees that the overlapping of $n(R)$ and $x(X''_{-})$ (represented by W) in Y does not contain freshly created elements but only preserved ones (as expressed by the commutative arrows via K).
- The above fact, together with the monicness of x, implies that $x(X''_{-}) \hookrightarrow k(K)$ (with $k : K \to D$ being the middle vertical morphism of δ) and, therefore, that the inverse application $G' \overset{\delta^{-1}}{\Longrightarrow} G$ leaves X''_{-} in place and, thus, $X''_{-} \hookrightarrow G$, a contradiction.
- We only have to show yet that $L_p \overset{\delta_p}{\Rightarrow} X''_{-}$ with $\rho(\delta_p) = \rho$ and $m(\delta_p) = L \overset{m}{\hookrightarrow} L_p$ (as in Definition 7.7) also leads to a contradiction.
- $L_p \overset{\delta_p}{\Rightarrow} X''_{-}$ and $L \overset{m}{\hookrightarrow} L_p$ implies $R \hookrightarrow X''_{-}$ due to Proposition 7.8.
- Then, due to $X''_{-} \not\hookrightarrow G$ and $X''_{-} \hookrightarrow G$, $n(R)$ (n is the co-match of $G \overset{\delta}{\Rightarrow} G'$) is injectively embedded in $x(X''_{-})$.
- Then, the inverse of δ_p, $X''_{-} \overset{\delta'_p}{\Rightarrow} L_p$ can be extended to $G' \overset{\delta'}{\Rightarrow} G$ by the Embedding Theorem (Theorem 7.1) along x: x is mono and ρ cannot violate the dangling condition by definition.
- However, this means $L_p \hookrightarrow G$, contradicting the non-applicability of p.

Now, $G' \in \gamma[C]$ is a consequence of Proposition 7.3.

□

As a consequence, we observe that the above theorem justifies our intuition about abstract verification for RePro: For any RePro transition sequence, the corresponding compass transition sequence using the same actions and starting from the set of all graphs subsumes the concrete states in each state. We denote the compass describing all graphs as $C_\top = (\{G_\emptyset\}, \emptyset)$ with $G_\emptyset$ being the empty graph. First, observe that this compass in fact describes all graphs.

Proposition 7.9. $\gamma([C_\top]) = |\mathbf{Graph}|$.

Proof. First, observe that $min([C_\top]) = C_\top$ as $C_\top$ is minimal: the positive pole is a singleton, $G_\emptyset$ is a core and $flat(G_\emptyset) = \{G_\emptyset\}$ For any graph $G \in |\mathbf{Graph}|$, $G_\emptyset \to G$. As the negative pole is empty, its conditions are trivially fulfilled. $\square$

Corollary 7.2. *Let* $(P_0, [C_\top]) \xrightarrow{(\rho_1, N_1)}_C \dots \xrightarrow{(\rho_n, N_n)}_C (P_n, [C_n])$ *be a compass trace. Then, the following holds:*

If $(P_0, G_0) \xrightarrow{(\rho_1, \delta_1, N_1)}_D \dots \xrightarrow{(\rho_n, \delta_n, N_n)}_D (P_n, G_n)$, *then* $G_n \in \gamma([C_n])$.

Proof. For any G_0, $G_0 \in \gamma(C_\top)$ by Proposition 7.9.

We prove the statement by induction on the length k of the $\to_C$ sequence.

$k = 1$: For $(P_0, [C_\top]) \xrightarrow{(\rho_1, N_1)}_C (P_1, [C_1])$ and graph G_0, each G_1 with $(P_0, G_0) \xrightarrow{(\rho_1, \delta_1, N_1)}_D (P_1, G_1)$ is in $\gamma([C_1])$ due to Theorem 7.2.

$k > 1$: By hypothesis, for

$$(P_0, [C_\top]) \xrightarrow{(\rho_1, N_1)}_C \dots \xrightarrow{(\rho_{n-1}, N_{n-1})}_C (P_{n-1}, [C_{n-1}]),$$

$(P_0, G_0) \xrightarrow{(\rho_1, \delta_1, N_1)}_D \dots \xrightarrow{(\rho_{n-1}, \delta_{n-1}, N_{n-1})}_D (P_{n-1}, G_{n-1})$ implies $G_{n-1} \in \gamma([C_{n-1}])$. Then, for any $(P_{n-1}, G_{n-1}) \xrightarrow{(\rho_n, \delta_n, N_n)}_D (P_n, G_n)$, $G_n \in \gamma([C_n])$ due to Theorem 7.2. $\square$

EXAMPLES FOR ABSTRACT VERIFICATION. All the above developments had a leading principle to prepare grounds an abstract property verification for RePro. At this point, to conclude this section by justifying our approach, we shortly revisit the *μ-calculus*, employed in [27] as a general means for formulating model checking formulas; afterwards, using appropriately crafted examples, we are able to demonstrate the intentions and the potential behind our verification framework. However, a full formal treatment of predicates and model checking formulas for compasses opens up a different line of research and is, thus, left for future work.

The (modal) μ-calculus [60] consists of formulas for verifying properties on transition systems. Theoretically, widely used *temporal* logics such as LTL [93] and CTL* [39] can be encoded in the modal μ-calculus; here the term “temporal” means that formulas do not only express state properties, but also those of transition paths, in turn allowing for statements such as “eventually, there is a path where a given property holds" or "the property holds in all upcoming states". Thus, the modal μ-calculus represents a powerful foundation for verification approaches based on model checking [5]. For formal details, we refer to the sources cited above.

Without going into the formal details, μ-calculus formulas resemble predicate logic with variables, where the semantics maps a formula to those states where it holds; for example, a formula consisting of a simple predicate holds in those sates where that predicate holds, as expected. There are two distinguishing features: (1) the use of the so-called *smallest* and *greatest fix-point* operators μ and ν, used to declare if a variable in a formula has to be resolved in a minimal or maximal way, and the temporal operators $\Box$ and $\diamond$, intuitively denoting that the formula should hold in every or in at least one subsequent state, respectively.

As a simple example for applying μ-calculus in the context of RePro and our WSN example in particular, consider that we want to verify if, given a graph property *no3a* expressing that "there is no active triangle in a graph", it holds for a state (P, G) that "for all subsequent state, *no3a* holds" (a *universal safety* property). By writing

$$\nu x.(no3a \land \Box x),$$

we are able to formally conclude if this holds for (P, G) using the μ-calculus. Notice that νx ensures that x will be interpreted as any state set S' and verifies if $\phi := (no3a \land \Box \phi)$ holds in those states. By the semantics of $\land$ interpreted as conjunction, this means that *no3a* have to hold in all those states and, by the semantics of $\Box$, recursively also in each of their successors. Thus, (P, G) fulfills $\nu x.(no3a \land \Box x)$ indeed if and only if each state reachable from (P, G) fulfills *no3a*.

First, recall the compass

$$C_{n3} := (\{G_\emptyset\}, \{G_3\})$$

from Section 7.2. Arguably, we are able to conclude that $\mathcal{L}([C_{n3}])$ fulfills *no3a*, i.e., that no graph in the language of C_{n3} contains an active triangle. Furthermore, observe that C_{n3} describes *exactly* that set corresponding to *no3a*. In general, we observe that for any compass containing G_3 in its negative pole, its language does not contain any graphs with active triangles.

Concluding this example, *we can verify in general the correctness for a topology control algorithm given as a control process*. In particular, consider

$$P_{nTC} := p_{kTC}.P_{nTC} + (p_{ActUS}, \{p_{kTC}\}).P_{nTC} + (p_{ActUL}, \{p_{kTC}\}).P_{nTC}$$

from Sect. 4.2 as a topology control algorithm; the specifications of the rules involved are found in Figures 4.3 and 4.4.

$$(P_{nTC}, [C_{n3}])$$

fulfills

$$\nu x.(no3a \wedge \Box x)$$

holds exactly if no reachable compass language contains active triangles. Let us verify this by examining the possible transitions of $(P_{nTC}, [C_{n3}])$ based on Definition 7.8.

- $(P_{nTC}, [C_{n3}]) \xrightarrow{(p_{kTC}, \emptyset)}_C (P_{nTC}, [C'_{n3}])$ by $P_{nTC} \xrightarrow{(p_{kTC}, \emptyset)} P_{nTC}$ and $min([C_{n3}]) \overset{(p_{kTC}, \emptyset)}{\Longrightarrow} C'_{n3}$, where, in particular, the negative pole of C'_{n3} contains G_3 due to p_{kTC} being agnostic to G_3 (in particular, G_3 will never overlap with R_{kTC} on its freshly created `L;i`-link).

- Similarly, $(P_{nTC}, [C_{n3}]) \xrightarrow{(p_{ActUS}, \{p_{kTC}\})}_C (P_{nTC}, [C''_{n3}])$ and, in particular, the negative pole of C''_{n3} contains G_3 due to p_{ActUS} being agnostic to G_3.

- Most interestingly, for $(P_{nTC}, [C_{n3}]) \xrightarrow{(p_{ActUL}, \{p_{kTC}\})}_C (P_{nTC}, [C'''_{n3}])$, in particular, G_3 would get eliminated as p_{ActUL} is *not* agnostic to it: R_{ActUL} has an injective morphism to G_3 as the former is a single `L;a`-edge. However, G_3 gets reintroduced by the last construction in Definition 7.7: $L_{ActUL} \hookrightarrow L_{kTC}$, $p_{kTC} \in N$ (the set of non-applicability conditions), and $L_{kTC} \overset{p_{ActUL}}{\Longrightarrow} G_3$ (at the monic match above).

Thus, we can conclude by induction that as any step of P_{nTC} preserves G_3 in the negative pole, *no3a* holds in any subsequent state. This way, we can *formally* verify that P_{nTC} indeed prevents the creation of active triangles starting from any graph which does not contain any of them, as expected. Note that now, we can also reveal that the earlier, unripe version

$$P_{TC} := p_{kTC}.P_{TC} + p_{ActUS}.P_{TC} + p_{ActUL}.P_{TC}$$

of the same algorithm is incorrect w.r.t. this formula: although the first two cases above still hold, in the case of p_{ActUL}, we do not retain G_3 without p_{kTC} as a non-applicability condition and, thus, cannot guarantee that no active triangle is created.

As a more elaborate example, recall the formula $\mu x.(vpath \vee (\Diamond true \wedge \Box x))$ from Section 7.1. Here, $vpath \in Pred \subseteq Lit$ for our particular setting; *vpath*, expresses a property of a RePro state; more precisely, *vpath* is fulfilled in (P, G) if it holds for G that any virtual link has been checked by the search mechanism described in Section 7.1. This formula represents a *liveness* property. Strictly, this property is neither universal nor existential, but rather combines both in the expression $(\Diamond true \wedge \Box x)$: while $\Diamond true$ simply ensures that there *exists* at least one subsequent state, while $\Box x$ takes care of considering all of the subsequent paths. Now, considering the whole formula, $(vpath \vee$

$(\diamond true \wedge \Box x))$ can be read as follows: either *vpath* holds in the current state, making the property hold for the current path, or we have to (and are able to) continue, expressed by the other part of the formula as explained before. In turn, binding by μx ensures that the whole formula only holds in those states for which any path eventually reaches a state fulfilling *vpath*.

Now, recall the process

$$P_{overlay} := P_{search} \,||\, P_{rep}$$

as well as the compass $C_{search2} := (\{G_f\}, \{L_{Success}, L_{Success2}\})$ (cf. 7.1 for further reference on the process specification). We know that $\mathcal{L}([C_{search2}])$ consists only of graphs containing search flags, thus, each of them violating *vpath*. Using our verification approach, we are able to verify generally the correctness of the repair capacities of P_{rep}, a component of $P_{overlay}$. In fact, we expect that

$$(P_{overlay}, [C_{search2}])$$

fulfills

$$\mu x.(vpath \vee (\diamond true \wedge \Box x)),$$

i.e., that for any execution of our overlay mechanism, even if the starting graph violates *vpath* (as the graphs in $\mathcal{L}([C_{search2}])$), the property will hold eventually. Due to the large number of potential actions in $P_{overlay}$, we do not elaborate on each possible step as above, but still delineate the reasoning process.

For actions in P_{search}, we just observe that they either keep (by search) or reduce the number of flags (by success), as now P_{admin} is inactive and no fresh virtual links are created. In particular, if there were a transition

$$(P_{overlay}, [C_{search2}]) \xrightarrow{(p_{Success})}_C (P_{overlay}, [C'_{search2}]),$$

G_f would get transformed to a graph of a single active edge by Definition 7.7, by taking $L_{Success}$ as a jointly epimorphic gluing. Obviously, the same argumentation holds for any starting compass with G_f in the positive pole. However, the presence of $L_{Success}$ in the negative pole results in the same graph appearing in the negative pole, leading to the disappearence of G_f by further minimalization (cf. $R3$ in Definition 7.5) and, thus, to inconsistence. We conclude that the compass construction prevents an immediate success action (as expected) and some repair action has to take place first.

In fact, by the transition

$$(P_{overlay}, [C_{search2}]) \xrightarrow{(p_{ActP})}_C (P_{overlay}, [C''_{search2}]),$$

each item in the positive pole of $C''_{search2}$ contains two flags as the construction extends G_f in different ways, and, in particular, some items would represent graphs where $p_{Success}$ is applicable now. Even more importantly, the above inconsistency is eliminated as p_{ActP} is *not* agnostic to $L_{Success}$ and, thus, the latter is not inherited in the negative pole of $C''_{search2}$.

Although promising, this conclusion is only partly satisfactory: it draws attention to an important line of future work by revealing two expressiveness leaks of compasses when compared to human intuition and reasoning: First, the positive part of Definition 7.7 does not prevent, e.g., p_{ActP} from repeatedly introducing fresh u-edges into the language of subsequent compass languages, also with flags not even being paired. Our intuitive reasoning on the correctness of the above verification formula is based on further implicit assumptions not captured by compasses: that each flag is paired and that the number of u-edges is decreasing. Second, p_{ActP} also immediately eliminates $L_{Success2}$ from a negative pole, although this intuitively represents a stronger condition. The present construction is not fine-granular enough to capture our intuition that $L_{Success}$ and $L_{Success2}$ together express the necessity of at least two subsequent repair steps.

8

DISCUSSION AND RELATED WORK

In this chapter, after having presented RePro and investigated its major characteristics as a process algebra and control language, we present a review of related literature. Thereby, we first explore the connections between RePro and further lines of work in the relevant theories (Sect. 8.1), and discuss some central design choices and the resulting limitations thereafter (Sect. 8.2).

8.1 RELATED WORK

REPRO AND CONCURRENCY THEORY. The notion of *concurrency* in computer science appeared as early as the first computing systems being capable of communication, and has been a major field of study ever since. As shortly delineated in Sect. 1.4, within the diverse takes and approaches on concurrency, *process algebra* is particularly concerned with the formal specification and analysis of communicating systems. Therein, a special attention is paid to issues related to dynamic, adaptive and reactive systems, constantly gaining on importance nowadays due to the rise of intelligent communication-based technologies as seen, e.g., in the IoT (Internet of Things) concept. As examples, the work of Mauw and Reniers [76], published in the comprehensive Handbook of Process Algebra, already considered a practical use of process-algebraic operational semantics to dynamic system interworking; more recently, according to the general trend of proposing specialized process calculi for application domains, an IoT calculus has been proposed both by Lanese et al. [67] and Lanotte and Merro [68].

As for the theoretical background, the design of RePro closely follows the principles of the CCS calculus, proposed by Robin Milner [79]. The choice of this calculus as a baseline has been motivated by the following factors: (1) its explicitly general purpose and high flexibility, (2) its minimal design and syntactic simplicity combined with expressiveness, and (3) the resemblance of its operators to the constructs usually employed in CGR practice. While these factors are thoroughly elaborated on in the thesis, here, we revisit some major process calculi other than CCS, in order to recall their distinguishing features and, most importantly, to comment on the effects those differences would exert on RePro if considered as an alternative baseline.

The CSP (Communicating Sequential Processes) calculus, first described in a 1978 paper by Tony Hoare [53], and substantially developed later on. In turn, formulating CSP as a process algebra, described by Brookes, Hoare and Roscoe [11], was influenced by the work of Milner on CCS, and these two calculi are still often handled as major competing alternatives for a minimalistic general-purpose communication calculus [12]. From a practical perspective, the main difference lies in the handling of *synchronization*: Standard CCS splits the domain of action names into complementary names and *co-names*, where synchronization only takes place upon corresponding name pairs firing in parallel—a mechanism we did not adapt for RePro, as in our special CGR setting, action names are rather derived from a fixed graph-rewriting system and, thus, introducing a complementary name set would feel unnatural. In contrast, CSP has a richer synchronization mechanism, where, instead of a universal parallel operator like our $||$, a parallel composition can be further parameterized by the names of actions *required* to be fired on both sides for the composed process to proceed. Thus, in contrast to RePro, this definition introduces a blocking synchronization behavior based on agreements between the two processes; however, the permissive RePro synchronization might be seen as a special case of CSP synchronization with an empty parameter list. Furthermore, RePro might be enriched with an adaptation of the CSP blocking synchronization mechanism if this would be, e.g., desired for a specific modeling domain: the synchronization parameter would range over the elements of the parallel rule monoid (cf. Definition 2.3) over the rule set from which the process is built.

Another highly influential process calculus is the π-calculus [81, 82], also proposed by Robin Milner, in collaboration with Joachim Parrow and David Walker. The main additional consideration, as the title of the original publication already suggests, is to capture the *mobility* of communicating agents, i.e., the changing system structure. Technically, this is most notably reflected by communication *channels* appearing as first-class syntactic and semantic ingredients, and the attached concept of *scopes*, representing those components (i.e., processes) where knowledge of a given name is shared among different processes, but concealed from other ones, i.e., the environment from the perspective of those processes. In turn, the scope structure might be thought of as a derivate of the channel structure and might change during execution by sending channel information along existing channels.

Directly adopting the π-calculus as a baseline for RePro would have probably shifted the emphasis from graph rewriting itself to the arising communication structures surrounding the graph and its evolution; moreover, such a communication model is rather rarely and superficially considered in graph-rewriting literature. However, the issues explicitly addressed by π-calculus are related to the most relevant challenges of the increasingly adaptive, autonomous and mobile computer systems of present-day practice. Therefore, investigating the adoption of π-calculus as a whole or some particular features of it in the context of RePro might have fruitful consequences for, e.g., the graph-based design of IoT or cyber-physical systems [69]. In this context, it is also important to mention that the communication structure derivable from a π-calculus specification is often termed topology and resembles, in fact, the real-life topologies of mobile ad-hoc networks just as in our WSN running example. Therefore, for the modeling of such dynamic communication systems, it would be interesting to investigate the π-calculus not only as a baseline, but also as a kind of meta-calculus, whose actions would, in turn, take effect on the topology term itself, integrating the control and the graph (data) aspect. In this context, we have to mention two further major calculi: (1) the *ambient calculus* [14], whose main goal is exactly to unify the communication and the transformation aspect of mobility: not only the communicating agents, but also the executables (e.g., rewriting rules or their sequences, even their processes) are mobile and might dynamically evolve or get transferred; and (2) the *fusion calculus* [87], designed with an intent to serve as a unifying process calculus and proven to be more general than the π-calculus, where the distinguishing *fusion* operation formally captures data sharing, which could also be adapted for graphs—although the interpretation of such operations is not straightforward within graph rewriting.

There is an abundance of further established process calculi, most of them designed to express the characteristics and fulfill the formal needs of specific system domains. Some of them, such as the *join calculus* [43] or the *reflective higher-order calculus* [77], aim at the support of formal reasoning over recent programming and system design paradigms and, thus, go into a different direction than the design principles behind RePro. Other, specialized calculi deliberately choose a narrower scope and rather serve as a formal modeling or specification vehicle for real-life systems, such as the *IoT calculus* to describe systems according to the Internet-of-Things principles [68]. Although such calculi do not harmonize with the intentions of RePro, i.e., providing a general process-algebraic control formalism for graph rewriting, specialized calculi would definitely be a valuable subject of study within the narrower scope of graph-based modeling within the specific domain.

Besides process calculi, *Petri nets* constitute another major formalism for the formal modeling of distributed and concurrent system behavior. Although the present thesis focuses on process algebra as a control mechanism for graph-rewriting processes, Petri nets could also serve the same purpose—in fact, we have described a preliminary study on employing Petri nets for controlling graph rewriting elsewhere [64].

Formally, the close connection between CCS and Petri nets has been noted and established early on; in particular, there is a semantics-preserving translation from the standard CCS of Robin Milner into Petri nets [47]. It is an often repeated claim that one of the major distinction between Petri nets and CCS (or even process algebra in general) is the *truly concurrent* or *asynchronous* nature of parallel transitions in Petri nets, i.e., that the Petri net model adequately captures the causal independence of transitions happening in parallel, in contrast to, e.g., the original CCS semantics which is fully interleaved: concurrency boils down to a number of equivalent sequentialization [46].

There are situations, however, where this property of CCS semantics is undesired, as discussed in detail by Mukund and Nielsen [83]. There is a number of different possibilities for providing CCS(-like terms) with a non-interleaving semantics, including

- the decoration of transitions, partly resembling our approach, as, e.g., in [10];
- or translating the term directly into a Petri net as, e.g., in [29].

Mukund and Nielsen [83] propose a comprehensible solution based on decorations, which, as a bonus, allows for a particularly favorable translation into the class of 1-safe Petri nets, ideal for reasoning purposes. This result is achieved through a categorical framework for models of concurrency, with this particular example being elaborated by Nielsen and Winskel [86]. The underlying intuitive observation is that during such a translation, semantically, one transposes from the domain of concrete process transitions to that of their underlying *events*, representing a more context-independent and abstract variant of system behavior. We might say that with the advancements in Sect. 6.2.2, we are halfway in adapting this framework: we provided an abstract event notion for graph-rewriting transitions by factoring out the non-conflicting differences in transition labels, but the notions and results for inferring the kind of Petri net translation applicable to RePro is part of future work.

Petri nets also play an important role in the concurrent semantics of graph rewriting proposed in the literature so far, which we will recapitulate in the following.

CONCURRENT SEMANTICS OF GRAPH REWRITING. The standard interpretation of (algebraic DPO) graph-rewriting systems usually understands the semantics (although it is even rarely termed so) for a given starting graph and a rule set (i.e., a GTS) as a transition system, where states are graphs and in each state, any rule of the GTS is available; transitions correspond to rule applications with their output graph as target, potentially labeled by their rule names, match morphisms or full DPO diagrams [38].

The notion of *traces* and *processes* have also been introduced already in the 90s in the context of graph rewriting, although with a different interpretation than that of RePro, influenced by Petri net theory and considering *event-structure* semantics instead of an operational one. Here, the graph structure is considered to represent the configuration of concurrent or distributed systems on an appropriate abstraction level, resembling our remarks above on a π-calculus term serving the same purpose.

In this context, Corradini et al. propose a *trace* notion and a corresponding trace-based equivalence relation for GTS. Here, a trace is an abstract rule application sequence lifting details of rule applications up to isomorphism, and the ordering of the single rule applications provided that they are equivalent [19]. This resembles the event abstraction technique of Sect. 6.2.2, however, the semantic domain of traces consists of equivalence classes of rule applications which get unified, focusing on their joint rewriting effects, instead of using the classes itself as events capturing the effects of single rule applications.

A related semantic construction, proposed by Corradini et al., is that of the so-called *graph processes* [18]. However, in contrast with RePro processes introduced above, these are static structures (in fact, a special kind of graph grammars) summarizing the potential conflicts and equivalences in the original GTS and, thus, they do not possess an operational interpretation.

A survey and comparison on the concurrent semantics of graph rewriting and the two aforementioned approaches in particular can be found in Baldan et al. [6].

From the operational perspective on the semantics of graph rewriting, both bisimulation and operational semantics has been studied before in the literature. As demonstrated in the context of RePro in Sect. 6.1 and especially in Theorem 6.1, one of the major power of bisimulation is that it allows for the mutual replacement of bisimilar processes in a larger context. However, to that end, it is required that bisimulation is a congruence. After establishing this for CCS, Robin Milner, together with James Leifer, established a general categorical framework for congruences in LTS-based bisimilarity [71]. Ehrig and König adopt this technique to *derive* (i.e., in contrast to RePro, not *a priori* provide) an LTS semantics of graph rewriting where, similarly to RePro, bisimilarity is a congruence [35]. However, their approach requires a special rule format and respective rule-application semantics, called *borrowed contexts*, which reflect the generic congruence notions of Leifer and Milner on the application level. Partly based on this technique, Dorman and Heindel propose a *structured operational semantics* (SOS) for graph rewriting [33]. Similar to RePro, the authors aim at combining notions from GTS and process calculi and especially focus on synchronization of parallel composition. Again, as in our above comments on π-calculus topologies and the motivation behind earlier trace and process notions, it is the process term itself which reflects the graph being rewritten, contrasting the RePro approach of employing a process term for external control.

NEGATIVE CONDITIONS IN GRAPH REWRITING. In connection to our proposed notion of *non-applicability conditions* (cf. Sect. 4.2), we discuss here further conceptual and technical alternatives, as non-applicability conditions are definitely not the only thinkable concept to handle forbidden graph structures or rule applications. Indeed, throughout the decades-long development of graph rewriting theory, a number of approaches have been proposed to deal with, vaguely speaking, expressing that some structural property of the graph (or of a graph and a match) should *not* hold—to all which we shortly refer here as negative conditions.

Although it is out of scope for the present thesis to elaborate on all the diverse notions proposed so far, still, we identify and discuss some major conceptual approaches to formalize negative conditions in the following.[1]

1. *Negative application conditions:* Introduced as early as 1986 by Ehrig and Habel [34], application conditions are arguably the most studied condition notion for graph rewriting.

   The basic principle of the application condition approach is to enrich rule specifications by *constraints* attached to their left- or right-hand sides. A constraint is an injective morphism from one of the rule sides, which thus extends the left- or right-hand side with some further structure. An application condition is a collection of such constraints equipped with polarities (i.e., if a constraint is positive or negative). In particular, a *negative left-hand (right-hand) constraint* in an application condition requires for a match (co-match) to yield a successful rule application that the occurrence of the left-hand side (right-hand side) is *not* extensible in a way that the forbidden structure also matches.

   The above principle is in stark contrast with the non-applicability conditions of RePro, even from two perspectives: First, while application conditions impose *local* graph structure constraints relative to a given match, non-applicability conditions result in *global* structural restrictions, which can in turn be derived from the non-applicability of a rule. (Note, however, that (i) non-applicability might still be captured by deriving synthetic application conditions, but this is contrary to the RePro principle of building control terms from existing rules; (ii) if using DPO rules, this is not equivalent with the non-occurrence of the left-hand side; even a valid occurrence might violate the so-called *gluing condition* of DPO graph rewriting [38].) Second, whereas application conditions are parts of rule specifications and are thus attached to specific rules, non-applicability conditions are part of the control terms and are in turn formulated *over* rules. This design choice has two main motivations: First, RePro has as explicit goal a strict separation of the control level and

---

[1] Note that several concrete approaches also encompass positive conditions within the same conceptual framework, but we do not address those here directly.

the underlying graph-rewriting system, preferring to use the elements of the latter without modification in control terms. Second, we want our control language to be computationally complete as demonstrated in Sect. 6.3; to this aim, it is necessary that the control language contains some notion of finite failure (basically for expressing termination of maximal iteration in the presented minimalistic setting).

Note that although the design choice above (namely, that negative conditions are on the control level) is justified in the conceptual framework of RePro, we do not intend to say that application conditions do not have a control effect and, thus, cannot be used for controlling a graph grammar; indeed, Habel et al. remarked already in 1995 that a wish for more precise control might be one of the motivations to enrich rules by application conditions [48].

2. *Consistency conditions:* Besides building on the aforementioned application conditions, Heckel and Wagner [50] introduce another kind of conditions called *consistency conditions*. In contrast to application conditions and more in the vein of RePro non-applicability conditions, consistency conditions are relative to graphs instead of rules and matches, and express *global* structural properties of those. Just as an application condition is a collection of application constraints, a consistency condition consists of *consistency constraints*.

   In turn, the concept of consistency constraints can be seen as an all-quantified variant of application constraints. In particular, they also come equipped with a notion of polarity: A *positive* consistency constraint is a morphism, holding for a graph if each occurrence of the morphism source therein can be extended to its target according to the constraint morphism itself. A *negative* consistency constraint is instead a single graph imposing global restrictions on other graphs in terms of forbidden subgraphs.

   Also, the restrictions imposed on graph-rewriting systems by consistency conditions might have a negative conditional (i.e., a forbidding) effect even if the constraint itself is not negative. Indeed, a large part of the original paper by Heckel and Wagner [50] is dedicated to translating global consistency conditions to (left-hand) application conditions of single rules from a given graph-rewriting system, based on the set of rules and the reachable graph states; even if the consistency conditions are not inherently negative, the generated application conditions might be of negative polarity (see above).

   A major difference between consistency conditions and RePro non-applicability conditions lies in the fact that while consistency conditions are formulated separately and over (graph) states, non-applicability conditions build on rule specifications already at hand, and utilize the rule-application semantics to formulate negative constraints for single actions of the transition system, instead of requiring their fulfillment for each graph instance arising during the execution.

3. The comprehensive approach of *nested conditions* as proposed by Arend Rensink [96] subsumes the above two notions both conceptually and technically. A nested condition is indeed a family of condition morphisms equipped with logical connectives, thus, representing a (first-order) logical formula over graphic conditions. Moreover, satisfaction of nested conditions is inherently interpreted both for graphs and morphisms, diminishing the above distinction between application conditions and constraints. Syntactically, a nested condition is a tree of morphisms connected by logical negation and conjunction; being sufficient to capture each operator in classical first-order logic. Indeed, Rensink shows that nested conditions achieve the expressiveness of first-order logic w.r.t. graph properties (seen as predicates).

A general remark about the relation of RePro non-applicability conditions and the condition notions above: there is no technical reason for RePro to exclude any "static" concept of negative conditions within the underlying rule-application semantics. Those and our non-applicability conditions simply belong to different conceptual levels: those negative condition notions are either attached to single rules or pose general restrictions on graph states in a graph-rewriting system, whereas our non-applicability conditions are part of control process terms and, thus, represent a fine-grained, action-level negative control mechanism.

Regarding that latter concept in the literature, the only prevalent state-of-the-art concept of control-level negative conditions lies implicitly in the *if-then-else* control construct appearing in most practical CGR approaches (cf. Chapter 3). Similarly to RePro non-applicability conditions, an if-then-else has a conditional rule evaluation which says that some control path (i.e., the *else* program branch) should be taken if a rule is not applicable. Nevertheless, if-then-else represents a special (even if fundamental) case for non-applicability conditions, restricting their use to conditional control decisions.

ABSTRACTIONS OF GRAPH LANGUAGES. In Chapter 7, we elaborated on the novel notion of compasses, representing a RePro-compatible solution for abstractly characterizing infinite graph languages. This topic of "taming" the infinity of graphs arising through graph-rewriting system execution has long been subject to research in the literature.

Before shortly surveying the related approaches, we remark that although compasses has been directly developed to fit the RePro setting, any of the following constructions could be investigated w.r.t. its capacities to serve as an abstract domain; however, they lack a labeled step notion for defining an abstract LTS in their current form.

Arend Rensink was among the first to explicitly raise the issue of handling abstract graphs for state space exploration [95], as part of his endeavors in developing the foundations for the Groove tool (cf. Sect. 3.2.6 and [45]), and inspired by similar advances in model checking [32]. He is inspired by *shape graphs*, where graph nodes might stand as placeholders for an unbounded number of other nodes potentially appearing, e.g., during graph rewriting. Shape graphs come equipped with constraints limiting their potential growth, a construction slightly resembling the role of negative poles in compasses.

Later on, directly based on the work of Rensink, Boneva et al. coined the term *abstract graph transformation*, describing a comprehensive shape-based framework for abstract graph verification [9]. Here, the central notion of *materialization* expresses the gradual concretizations of abstract graphs through rule applications. Materialization has remained an important notion for developments up to recent times, such as the work of Corradini et al. [24] proposing a unifying approach to abstract rewriting. Beyond the aforementioned line of research, another direct precursor of this paper is a general framework for recounting the properties of abstract graph language specifications, here, languages described by type graphs in particular [20].

Another paper aiming at defining abstract graph transformation based on shape graphs is the work of Steenken et al. [106]. Here, similarly to our compass transition system, the goal is to find abstract transitions being sound and complete approximations of their concrete counterparts.

We remark that, alternatively, also the different graph constraint notions, recapitulated above in the context of graph conditions, might be viewed as abstract graph language specificators and, thus, their use as such will be discussed in the following section in more detail, under the heading Compasses as Constraints.

8.2 DISCUSSION

In this section, we further discuss some of the central notions of RePro, where we think that the reasons behind our design choices or the borders of RePro compared to other viable alternatives should be made more explicit. In particular, we revisit the topics of synchronization and independence, as well as equivalence of graph-rewriting processes. In addition, we reason about the common features of a process-theoretic core theory of controlled graph rewriting, by means of comparing the SOS semantics of RePro as well as of state-of-the-art CGR tools Porgy and “full” GP as in Sect. 3.2.1.

PROCESS EQUIVALENCE. A major conceptual advantage of the design of RePro is the process-algebraic setting: it enables us to address equivalence of processes in an established and systematic way. In particular, none of the existing CGR approaches deals with the question if two executions (or even more generally, whole programs or processes) are considered to be *equivalent* under some relaxed notion of equivalence not requiring strict syntactic uniformity.

However, despite this unique feature, the direct adaptation of process-algebraic notions in a graph-rewriting context has some drawbacks.

1. *Too concrete states and transitions:* It is among the goals of process calculi to have some abstractional power over the system domain they primarily aim at specifying, just as CCS or the π-calculus capture the essential message-passing behavior of communication systems by introducing an appropriate abstraction over distributed communicating agents and their messages. Unfortunately, the same cannot be said of RePro: although the underlying theoretical constructs allow for defining equivalences (as we have seen with traces and bisimulation), the information carried by states and transition labels is way too concrete: graph instances signify concrete graphs while labels contain a full concrete DPO diagram, where even isomorphic objects are treated as separate entities. However, in Sect. 6.1, we propose (among others) an abstraction of the RePro transition system which takes isomorphism into consideration.
2. *Lack of hiding:* Despite the aforementioned issues, we were still able to define meaningful equivalence notions for RePro processes. But while trace equivalence suffers from the usual drawback of “forgetting” important control-relevant details (we referred to this as branching behavior before), RePro bisimulation notion appears to be a bit strict: as graph instances are inherent parts of their states, bisimulation is not able to abstract away from concrete graph identities. Even if we solve this issue by the aforementioned up-to-isomorphism factorization in Sect. 6.1, we still miss the real power of bisimulation, i.e., that it can be equipped with a notion of observability. In CCS, non-observable steps arise when paired actions (co-actions) synchronize, while the π-calculus has an even more sophisticated action hiding mechanism based on scopes. In contrast, in RePro, every synchronization leads to fresh graph-rewriting actions, which are observable under any circumstances. This is a deeper conceptual issue which cannot be solved by simply fine-tuning the definition

of bisimulation; therefore, the larger part of Chapter 7 deals with introducing an abstract domain of graph classes harmonizing with the RePro control semantics.

SYNCHRONIZATION AND PARALLEL INDEPENDENCE. Here, we first revisit and discuss the synchronization mechanism in RePro, being directly connected to parallel independence of rule applications and also that of transitions (cf. Definitions 2.6 and 6.2 as well as Proposition 2.1). Then, we examine the conceptual differences between our two definitions of transition independence, which we called *direct approach* and *asynchronous approach*, respectively (Sect. 5.2).

1. *Synchronization:* The RePro concept of synchronization formally arises by Definition 4.5: There, the rule SYNC allows for each two rule applications arising in parallel (i.e., on two sides of a parallel composition) to synchronize and thus perform those actions as a single parallel-rule application. Looking at only the control processes, this is always possible as there is nothing *blocking* the parallel action; whereas if we consider a graph instance as well, there might be single actions available in parallel for which the corresponding synchronized action does not have a valid match and is, thus, unavailable. Consequently, one might think of ways to semantically restrict synchronization already at the control level, at least for cases where the synchronized action will *never* become available. We have proposed such a variant in the original publication on RePro [63], presented above after Definition 4.7 as nCTS* (as in contrast to nCTS). In nCTS*, an additional premise of SYNC prevents the inference system to synthesize actions where one of the plain rules in the parallel rule is simultaneously a non-applicability condition.

   Unfortunately, this premise will not prevent the control semantics to produce infeasible actions: even if not formally the same rules, some plain component rule might require as match a smaller part of some forbidden structure. Implementing such a sophisticated mechanism, although technically feasible, would contradict the action-based spirit of CTS, as this would require a categorical analysis of the underlying rule spans. We argue that a completely liberal solution (i.e., one without any restrictions) as in the present nCTS (Definition 4.7) harmonizes the most with the design of RePro: at least on the control level, no blocking behavior is introduced, making control processes conceptually more aligned to CCS processes.

2. *Direct vs. asynchronous approach:* The direct approach to independence lifts the Local Church-Rosser property of DPO rule applications to the process-algebraic setting of RePro. Although this notion already considers RePro particularities not reflected

in the original notion (like the presence of non-applicability conditions), still, this approach might be considered naïve in terms of causal independence: if transitions arise in parallel, their originating state still has to be analyzed for its exact content to meet a conclusion about causality, which contradicts the usual assumptions about SOS semantics.

In contrast, the asynchronous approach is based on an established asynchronicity notion for CCS-like process calculi [83]. Here, asynchronicity means causal independence faithfully reflecting concurrency and distribution. This notion considers the structure of the process term and the asynchronous independence notion reflects the distribution of its parallel components. However, even this approach cannot be completely orthogonal to the Local Church-Rosser property: parallel independence have to be incorporated for reflecting not only control, but also graph-rewriting behavior (cf. Definition 6.11).

SOS CORE SEMANTICS: REPRO, GRAPH PROGRAMS, PORGY.

As also addressed by one of the central CGR challenges, **C1** (cf. Sect. 3.3), we aim at identifying a common semantic baseline for controlled graph rewriting. Although it cannot be our goal to establish a single ultimate semantic foundation in isolation, as argued before, an SOS formulation of core control constructs leads us towards properly addressing **C1**.

As seen in Chapter 3 and in the present chapter, there exist three major SOS semantics for controlled graph rewriting (including RePro). Instead of simply recalling the semantics of GP and Porgy, we proceed by elaborating on how each of the SOS semantics handles those control constructs common to them and, in turn, to each CGR approach. Thereby, we also present the corresponding inference rules of the other languages, while appropriately simplifying and adapting them up to the language-specific details irrelevant to present discussion. In particular, although GP and Porgy operate on abstract graphs (up to isomorphism), we do not explicitly represent that distinction as we focus on the shape of SOS semantic definitions for graph rewriting.

- *Single rule application:* Inferring basic graph-rewriting steps is the basis for any further constructs in each language. Recall that RePro addresses this by the SOS rules

$$\textsc{pre}\;\frac{}{\gamma.P \xrightarrow{\gamma} P} \qquad \textsc{mark}\;\frac{P \xrightarrow{(\rho,N)} P' \quad G \overset{\delta}{\Rightarrow} H \quad \forall p \in N : G \overset{p}{\not\Rightarrow}}{(P,G) \xrightarrow{(\rho,\delta,N)}_D (P',H)}$$

  where control transitions produced by PRE are in turn extended to proper rule applications by MARK. Whenever no rule application is possible, a RePro execution gets stuck. Other languages do not separate control semantics and formulate the corresponding behavior directly. For example, GP has the following rules, where GP_e is an elementary graph program:

$$\textsc{Call1}\,\frac{G \Rightarrow_{\{GP_e\}} H}{(GP_e, G) \rightarrow H} \qquad \textsc{Call2}\,\frac{G \not\Rightarrow_{\{GP_e\}}}{(GP_e, G) \rightarrow \mathit{fail}}$$

The rules Call1-2 are not only for rule applications, but they directly subsume also choice. Note that in contrast to RePro, GP transitions are unlabeled. In addition, GP does not get stuck but explicitly fails in the case of non-applicability (call2). A terminated program has no syntactic representation, i.e., GP terminal states are either a graph or the special *fail* state.

Porgy follows an almost identical schema, with r being a rule:

$$\frac{G \Rightarrow_{\{r\}} H_1, \ldots, H_n \qquad \forall H \text{ s.t. } G \Rightarrow_{\{r\}} H : H \in \{H_i\}}{(\mathrm{r}, G) \rightarrow \{(\mathit{Id}, H_1), \ldots, (\mathit{Id}, H_n)\}}$$

$$\frac{G \not\Rightarrow_{\{r\}}}{(\mathrm{r}, G) \rightarrow (\mathit{Fail}, G)}$$

This semantics is even more reduced, considering only the behavior of a single rule. Interestingly, instead of branching over different matches of r, Porgy creates as a single subsequent state a *multi-set* of control-graph pairs. Here, states are slightly richer than in GP, as the terminating process is signalized (*Id*, corresponding to **0** in RePro) and a *Fail* state also contains a graph, being the "cause" for the failure.

- *Choice:* Recall that RePro just as CCS, has a commutative $+$ operator for choice:

  $$\textsc{choice}\,\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

  Choice in GP is combined with rule-application semantics in the rules Call1-2 above.

  Porgy has a unique take also on choice behavior: if the choice is a probabilistic `ppick` function, then there is an additional transition reducing the choice set to a single strategy (without applying any rule). Thus, the design of Porgy is different from RePro and GP in that not every transition corresponds to a rule application.

- *Sequences:* For RePro, sequences are subsumed by pre above and, aligned to CCS, no general process-level sequentialization operator is considered.

  GP has the following rules (with GP and GQ being graph programs):

$$\textsc{Seq1}\ \frac{(GP, G) \to (GP', H)}{(GP;GQ, G) \to (GP';GQ, H)}$$

$$\textsc{Seq2}\ \frac{(GP, G) \to H}{(GP;GQ, G) \to (GQ, H)}$$

$$\textsc{Seq3}\ \frac{(GP, G) \to \mathit{fail}}{(GP;GQ, G) \to \mathit{fail}}$$

If GP is not yet successfully terminated, than the rest of the program GP' is also preserved in a sequence (Seq1). If GP terminates with graph H, then H is passed to the subsequent program GQ (Seq2), and failure is global (Seq3).

Porgy relies on the same mechanism—we omit the corresponding rule here due to their complicated and exotic syntax.

- *Maximal iteration:* RePro utilizes recursion and non-applicability conditions to encode maximal iteration—e.g., for a single rule p, the process P applying p as long as possible takes the following form:

  $$P := p.P + (\varepsilon, \{p\}).\mathbf{0}$$

  Refer to Section 6.3 and the encoding of maximal iteration of GP for details in the general case, including a discussion on the theoretical limitations.

  As for GP itself, the following rules express maximal iteration, with $\to^+$ denoting transitive closure of $\to$ and "(GP, G) finitely fails" meaning that there are no infinite transition sequences and each finite one terminates in *fail*.

  $$\textsc{Alap1}\ \frac{(GP, G) \to^+ H}{(GP!, G) \to (GP!, H)} \qquad \textsc{Alap2}\ \frac{(GP, G)\ \textit{finitely fails}}{(GP!, G) \to G}$$

  Thus, GP maximal iteration never fails, it rather continues whenever it is possible (Alap1) or terminates with the last graph state for which no iteration is possible (Alap2). (In general, this definition assumes termination of the iteration body.)

  In Porgy, maximal iteration is a special case of `while`:

  $$\texttt{while}(S)\,\texttt{do}(S)$$

  whose semantics is, in turn, given by a translational transition into an if-then-else:

  $$\texttt{if}\ S\ \texttt{then}\ (S;\texttt{while}(S)\,\texttt{do}(S))\ \texttt{else}\ \mathit{Id}$$

- *If-then-else:* In RePro, if-then-else is a special case of choice. As for GP, the definition is similar to maximal iteration: if the condition program successfully terminates, then the program in the *then* branch is chosen, if the condition program finitely fails, then the program in the *else* branch is chosen—we omit the rule specification as they are obvious. Again, PORGY follows the exact same pattern. (Our considerations on finite failure and termination also hold here.)

Summarizing, some design choices of PORGY in particular, non-rewriting transitions for transforming only the control strategy make it more exotic than the other two semantics. Regarding REPRO and GP, we argue that both languages represent a slim design, making them appropriate as baseline for an approach-independent core CGR semantics, REPRO more from a process-theoretic, GP more from a language-theoretic perspective. Depending on actual needs in a scenario, even combinations and cross-fertilizations are thinkable, e.g., process-level sequences in REPRO or general choice in GP. However, although parallelism is out of scope for the present discussion, we argue that the process-algebraic foundations of REPRO make it more adequate as a CGR baseline if concurrency is also considered.

COMPASSES AS CONSTRAINTS. We have surveyed the established notions for expressing graph constraints and conditions, at the beginning of this section; there, in the setting of negative conditions, relating them to the non-applicability conditions of REPRO.

We have seen that all the considered notions and, in particular, the most general *nested conditions* [96] consider negative as well as positive constraints in the form of graph patterns—just as compasses. Moreover, nested conditions allow essentially the repertoire of first-order logic to be used for composing those single graph patterns. Thus, from this perspective, we can justly claim that formally, compasses represent a special case of nested conditions.

However, we argue that the special form represented by compasses fits the novel context in which compasses are employed. In fact, earlier condition notions have been primarily utilized for either restricting the set of graphs permitted in a given situation, or, similarly, to reason about if a rule application adheres to a set of pre- and post-conditions. As for compass steps, we have the opposite conceptual outset: we allow any action to happen, and adapt the compasses (i.e., the constraints) themselves to reflect or at least to approximate the changes caused by that action. On a practical note, this allows for an appropriate formal handling of a frequently occurring scenario, where the verification goal is not the permanent preservation of a global invariant, but rather capturing a dynamically changing structural or behavioral pattern.

Nevertheless, on a technical note, we have to stress that compasses not only represent a special case of graph constraints, but, in turn, adhere to the (already mentioned) techniques proposed by Heckel and Wagner for shifting graph conditions [50]. In particular, the positive pole corresponds to a constraint with an empty premise and the possible positive patterns as conclusion, while the negative pole corresponds to the forbidden patterns as premises, with an empty conclusion. Thus, we have an easy construction for translating compasses into graph constraints expressing the same language.

In turn, for replicating the compass transition system through graph constraints, one might think of the constraints standing as preconditions before abstract rule applications, i.e., transitions. The shifting technique of Heckel and Wagner, then, allows for computing a strongest postcondition of the abstract rule application, thus, representing an alternative technique to express our intended abstraction. (The technique even allows for translations in the other direction, but this is not needed to replicate compasses.)

Concluding the connection of compass steps and translations of graph constraints, an accurate delineation is as follows: given a set of graph-rewriting rules, potentially equipped with application conditions, Heckel-Wagner-translations directly provide a precise characterization of subsequent abstract states through strongest postconditions. However, those application conditions cannot directly express the semantics of a RePro action: a rule is applicable, while some others are not applicable, i.e., a RePro action involves a negative check over the complete DPO applicability. This compound nature is the major source of imprecision during the calculation of subsequent abstract states.

In the previous chapter, we have also pointed out some expressiveness shortcomings of compasses, exemplified by our example compass $C_{search2}$; we remark that nested constraints are expressive enough to, at least, capture more complex graph properties such as "any flag is paired". It is a further investigation of high importance to consider a generalization of compasses inspired by nested constraint expressiveness. Note, however that it is not clear to what extent nested constraints would help in reasoning tasks involving, e.g., bounded element cardinalities and derived negative conditions.

In the context of the graph programming language GP, we have mentioned (Sect. 3.2.1) another verification approach for graph algorithms which considers properties of graph languages. The Hoare-style proof system proposed in [94], even if having a different philosophy than our model-checking approach, turns out to be rather powerful for sequential graph-rewriting processes: based on preservation properties of single rules in the system, it inductively derives properties for whole graph programs.

DECIDABLE LANGUAGE PROPERTIES. We argue above that given our transition-based concurrent setting, the reactive version of abstract interpretation as proposed by Dams et al. [27] provides an appropriate context for ensuring property preservation for concrete graphs.

Different ways have been proposed to reason about properties of single concrete graphs and to capture it in a logic-based setting. Most notably, the aforementioned nested conditions [96] correspond to first-order predicates on graphs, while the monadic second-order logic (MSOL) approach [25] goes even further and embeds graph structures themselves as well as rewriting rules into MSOL formulas. While MSOL gives the required expressive power to reason about complex graph properties comprising structures of unbounded size, it is not straightforward how to juxtapose the aforementioned approaches with process-algebraic transition systems for embracing concurrency. However, as a future direction for studying RePro, it would be worth considering the effects of RePro actions on other graph language characterizations than compasses (e.g., the above ones); there might be step concepts and rewriting approaches for other structures as well, representing potential alternatives to the compass transition system.

However, in the present chapter, we have left the evaluation of μ-calculus formulas for compass transition systems out of our scope. This topic is definitely a worthy field for further studies: the form of a compass might give specific clues under what circumstances we can *decide* if a property holds for each graph in the compass language. We do not intend to elaborate on this in the present thesis, but let us consider a simple example: if we can verify that each graph in a language contains some pattern, e.g., a triangle. If we have a compass having a single triangle in its positive pole, then the question is positively decidable. However, if we have further permitted structures, then we have to answer with no—but then again, also negative patterns can take influence on the outcome, and so forth. Thus, we conclude that decidability of compass properties is a proper field of study on its own, because it does not only involve the analysis of single elements in poles, but also how they are composed together to a compass.

Another related issue is the most general class(es) of languages which can be captured in our setting. We have demonstrated above that the languages arising from repeated rule applications from a fixed GTS are subsumed by our compass semantics. However, our characterization of them depends on that very semantics, which choice is justified for abstract interpretation—still, an interesting direction of further studies could be to analyze other means to characterize graph languages with recurring patterns and finitely expressible structural features independently of the notion of compasses. Alternatively, future work could also consider potential adaptations of the shape or the semantics of compasses, with an explicit goal of not having to restrict the powerset of graphs as a concrete domain.

9

CONCLUSION AND FUTURE WORK

With the growing complexity and autonomy of systems, abstract modeling to study and formally analyze those systems is gaining on importance. Graph rewriting is an established, theoretically founded formalism for modeling the structure and the behavior of complex systems by an appropriate graph-based abstraction, proposed as a generalization of classical string grammars [38, 98]. However, it is exactly those modern systems, often involving distributedness and, thus, concurrency and reactive behavior, which pose a challenge to the hidden assumption of global knowledge behind graph-based modeling approaches. Thus, abstractly speaking, the motivating challenge of the present thesis boils down to a conflict between the inherent closedness of graph-based abstract representations and the intention of using them for faithfully capturing the pluralistic nature of concurrent systems.

Formally, a graph-rewriting system consists of declarative rules, providing templates for potential changes in the modeled graph structures over time. In particular, a graph-rewriting rule specifies a pattern to be matched and then modified in potential input graphs, by deleting and creating sub-structures as specified by the rule. However, describing the dynamic behavior of complex systems by those rules often involves an additional control logic to represent not only the structural but also the algorithmic system aspects. To that end, following similar developments in the theory of string grammars and as an extension to the existing theory, controlled graph rewriting has been proposed. Here, some external control language guides the sequence in which rules are applied. However, approaches elaborating on this idea so far either have a practical, implementational focus, or are formalized in a denotational manner, thus focusing on the relations of input and output graphs instead of the operational behavior of controlled graph-rewriting systems.

In the thesis, we proposed an operational theory for controlled graph rewriting, based on well-established notions from concurrency theory and, in particular, process algebra [79, 83]. We argued that although state-of-the-art abstract relational semantics are useful for studying languages of graphs generated by such systems, completely omitting operational details obstructs a fine-grained analysis of graph-rewriting rule applications, which is essential to reason about graph-rewriting systems. Furthermore, the behavior of a system class of increasing importance, namely that of non-terminating reactive systems, cannot be captured justly by those semantic domains. In contrast, practical operational semantics proposed so far did not address concurrency and parallelism, which is of utmost importance for nowadays multi-agent systems of increasing autonomy and self-adaptivity. In addition, there exists even a strong correlation between concurrent system structure and reactive behavior.

In the first part of the thesis, we started with demonstrating the aforementioned fundamental phenomena by describing a simplified model of wireless sensor networks (WSN). After recapitulating the necessary background on DPO graph rewriting, the particular formal framework used throughout the thesis, we presented an extensive survey on state-of-the-art approaches to controlled graph rewriting, particularly for deriving and emphasizing the most important challenges we addressed:

1. We proposed a unified formal foundations to control graph rewriting, based on process calculi, especially CCS [79].

2. Thereby, we have been focusing on using graph-rewriting systems for modeling inherently concurrent and reactive systems, naturally invoking the fundamental phenomena above.

3. As a fundamental aspect of such approaches, we achieved a novel handling of *equivalence* as well as *independence* notions for controlled graph-rewriting processes. In particular, we proposed an independence notion combining structural and causal aspects of the notion, as well as a different approach based on asynchronous transition systems [83].

4. Building on process algebra also enables a founded handling of abstract reasoning, i.e., the verification of graph properties. In particular, abstract interpretation of reactive systems is defined over transition systems.

In the second part of the thesis, we elaborated our theoretical contributions. As a novel approach, we proposed a process calculus for controlled graph rewriting, called RePro, where DPO rule applications are controlled by control process terms closely resembling the process calculus CCS. This way, we achieved an appropriate approach to the aforementioned challenges by building on established results and techniques in process algebra:

1. Process calculi have rich and strong formal foundations.

2. Concurrency and reactiveness is inherently and naturally addressed by the syntax and semantics of processes.

3. Equivalence and independence notions are among the central study and design goals of process calculi, comprising different abstraction levels.

4. There are established extensions to their theory for supporting symbolic property verification; in particular, the modal μ-calculus represents an established, comprehensive formal technique for model checking on transition systems.

In particular, we showed that a central property of CCS, namely that bisimulation is a congruence, is preserved by RePro control processes and developed a notion for addressing both structural and process-algebraic aspects of action independence. We reason about the expressiveness of RePro control terms by comparing them to the state-of-the-art language of Graph Programs [49]. As for abstract reasoning, we used a version of the abstract interpretation framework, adapted for reactive systems [27], to propose an abstract domain of so-called compasses for reasoning about graph properties without relying on concrete graph instances, and even finitely capturing infinite graph languages potentially arising by RePro process executions.

Regarding future directions for further work, as already noticed at several points throughout the thesis, RePro has an immense potential for further extension due to the multifacetedness of the underlying theories. Here, to conclude the thesis, we mention some major directions.

- *Practical applicability:* The design of RePro as a control language and compasses as a verification approach explicitly focuses on laying formal foundations for some less explored phenomena in graph rewriting, rather than being directly transferable into practice, or even into concrete software implementations of controlled graph rewriting (cf. Chapter 3). (Just as CCS was not designed for actually implementing communication systems, but for abstractly studying their essence.) However, there are several aspects of RePro making it worth to consider even in a practical setting.

  First, the syntax of semantics of RePro might be used as a reference for practical dialects of controlled graph rewriting: even if a given concrete syntax involves constructs which increase usability, the constructs of RePro arguably represent a fundamental baseline and, thus, an encoding of any concrete syntax into RePro terms “under the hood” is a valuable sanity check as well as a starting point for further language extensions. In turn, such an encoding could give rise for designing the first tool supporting concurrent specifications of graph-rewriting systems: by the RePro principle, the specifier does not have to anticipate the possible interactions of concurrent components; on the contrary, it is encouraged to enforce a strict separation on specification level, facilitating the design of graph-based abstractions for such systems.

Second, such an endeavor might also be extended into the direction of practicable, tool-supported verification. We have already seen in Chapter 3 that some state-of-the-art tools offer model checking—for properties of single graphs in a state space. A first step towards abstract verification based on the present thesis could be the implementation of compasses as a description mechanism for graph classes. As for predicates, we are able to rely on existing model checking approaches; however, it would be crucial to understand the practical relation between graph predicates and compasses. In theory, we expect predicates to be semi-decidable, but this should not hamper the development of tool support: by identifying compasses with decidable properties, we would achieve a practically valuable verification approach, whose outcome would not be restricted to statements on single input graphs: we could claim correctness for controlled graph-rewriting algorithms analogously to well-known techniques for traditional programming languages.

On a related note, we have remarked that compasses still have to be thoroughly investigated for their graph language specification capacities, building on the preliminary observations in Section 7.2 and well-founded, yet practice-oriented related work [20, 24].

- *Connection to Petri net theory:* As commented on already in Section 1.4 on a rather philosophical note, there are insightful connections between Petri nets and process calculi in terms of their approach to concurrent phenomena. Consequently, RePro also touches on Petri net theory at several points.

  First, the relation of control processes to full RePro processes resembles that of *unmarked* to *marked* Petri nets: unmarked Petri nets just provide a static structure of potential transitions, while the appearance of markings induces transitions evolving the state (here, the marking itself). Along those lines, it would be interesting to reason about what influence on RePro would further restriction or extension notions take. As a restrictive example, *1-safe* Petri nets are conservative, but have many advantageous properties for analysis—is there an analogy for RePro? In contrast, as a potential proper extension, there is an obvious conceptual connection between *inhibitor* arcs and non-applicability or other negative graph-rewriting conditions. It would be interesting to see if that connection can be exploited theoretically in either direction.

  A repeated claim for Petri nets is that, in contrast to process calculi, they represent the right model for *true concurrency*: without an explicit parallelization, in any step, non-conflicting tokens might move in complete causal independence. In turn, the framework of asynchronous transition systems (ATS) we use to address independence in RePro also permits, under some mild restrictions for the underlying process algebra, for a translation into (a well-defined class of) Petri nets. Exploiting those theoretical connections, we could achieve having a bridge between RePro processes and truly concurrent actions.

Another issue related to true concurrence achievable by Petri nets is how RePro can be generalized to work on multiple graphs or, probably equivalently, on different parts of a partly unknown graph context in parallel. Having a data flow as that of Petri nets immediately allows to reason about such parallelizations. We have indeed promoted the use of parallel Petri net structures in the context of our *graph-rewriting Petri nets* [64], there to provide formal foundations for an omnipresent practical concept for controlled graph rewriting implementations: *subgraph binding* for constraining where rules are matching based on the co-matches of earlier rule applications. Letting such considerations influence RePro theory could lead to breaking with a strictly linear way of specifying processes, thus, improving on expressiveness in a yet unexplored dimension.

Finally, we mention an interesting constellation of abstract verification of graph rewriting using Petri nets: in a paper of König and Kozioura [59], an approach is presented which over-approximates a GTS by a Petri net, whose reachable markings, in turn, represent (more precisely, over-approximate) the graphs reachable from a given start graph. However, it requires further investigations to find out (i) if the translation technique can be extended to incorporate control over graph-rewriting systems and (ii) which property classes might be verified over a Petri net state space representation.

- *Independence from rewriting formalisms:* In the thesis, we have presented RePro theory built upon DPO graph rewriting as an underlying formalism. This choice was mainly motivated by the fact that DPO is arguably the single most studied (algebraic) approach to graph rewriting, at least in theoretical terms. However, the overall RePro concept (cf., e.g., Figure 5.1) is not directly depending on that formalism. In particular, control process terms are composed of rule names that might be attached to any declarative rewriting rules, with even potentially generalizing graphs to other mathematical structures. We have already presented shortly some examples of alternative formalisms in Chapter 2; also, in the seminal book of Ehrig et al. [38] and elsewhere in the literature, more general categorical properties have been studied which make the objects of a category a proper subject of DPO rewriting. (Their example are graphs with typed attributes, but also hypergraphs and other generalizations fit.)

Going even further, one could conceive of rules themselves as operationalizable, yet general categoric structures, thus RePro could evolve into a comprehensive platform to accommodate and combine any rewriting needs with a single fixed underlying semantics. We think that the categorical notion of *sketches* [7] might be the right vehicle for formalizing such a generic "operational category theory".

- *Embedding rewriting objects into terms:* Another direction of generalization would be to open up for more advanced concepts in process algebra. The generic framework of *nominal transition systems* [88] with an elaborate name binding notion is able to accommodate a wide range of calculi. In turn, using a categoric version of the generic calculus, one could incorporate the rewriting objects as above into the structure of the process term itself, thus, dissolving the compound nature of RePro states into a representation unifying the strengths of two of the most general mathematical frameworks known to date: category theory and nominal systems.

BIBLIOGRAPHY

[1] Luca Aceto, Willem Jan Fokkink, and Chris Verhoef. "Structural Operational Semantics." In: *BRICS Report Series* 6.30 (1999).

[2] J. Adamek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. On-line version (2004), last accessed 15 Jan 2019. Wiley Interscience, 1990. URL: `http://katmat.math.uni-bremen.de/acc/`.

[3] Vasco Amaral, Jordi Cabot, and Miguel Goulão. "Special issue on quality in Model-Driven Engineering." In: *Computer Languages, Systems & Structures* 54 (2018).

[4] Steve Awodey. *Category Theory*. New York: Oxford University Press, Inc., 2010.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[6] Paolo Baldan, Andrea Corradini, Ugo Montanari, Francesca Ros-si, Hartmut Ehrig, and Michael Löwe. "Concurrent Semantics of Algebraic Graph Transformation." In: *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3*. World Scientific, 1999, pp. 107–187.

[7] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall International Series in Computer Science, 1990.

[8] Marek Bednarczyk. "Categories of Asynchronous Systems." PhD thesis. University of Sussex, 1988.

[9] I.B. Boneva, Arend Rensink, M.E. Kurban, and J. Bauer. *Graph Abstraction and Abstract Graph Transformation*. CTIT Technical Report Series LNCS4549/TR-CTIT-07-50. Formal Methods and Tools (FMT), 2007.

[10] Gérard Boudol and Ilaria Castellani. "Three equivalent semantics for CCS." In: *Semantics of Systems of Concurrent Processes*. Ed. by Irène Guessarian. Springer, 1990, pp. 96–141.

[11] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. "A Theory of Communicating Sequential Processes." In: *Journal of ACM* 31.3 (1984), pp. 560–599.

[12] Stephen D. Brookes. "On the relationship of CCS and CSP." In: *Automata, Languages and Programming: 10th Colloquium*. Springer, 1983, pp. 83–96.

[13] Horst Bunke. "Programmed graph grammars." In: *Workshop on Graph Grammars and Their Application to Computer Science*. Vol. 73. LNCS. Springer. 1978, pp. 155–166.

[14] Luca Cardelli and Andrew D. Gordon. "Mobile ambients." In: *Theoretical Computer Science* 240.1 (2000), pp. 177 –213.

[15] Rudolf Carnap. *An Introduction to the Philosophy of Science*. Dover Publications, 1974.

[16] Noam Chomsky. "Three models for the description of language." In: *IRE Transactions on information theory* 2.3 (1956), pp. 113–124.

[17] Marco Conti and Silvia Giordano. "Mobile ad hoc networking: milestones, challenges, and new research directions." In: *IEEE Communications Magazine* 52.1 (2014), pp. 85–96.

[18] A. Corradini, U. Montanari, and F. Rossi. "Graph Processes." In: *Fundamenta Informaticae* 26.3/4 (1996), pp. 241–265.

[19] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. "Abstract graph derivations in the Double Pushout approach." In: *Dagstuhl Workshop on Graph Transformations in Computer Science, 1993*. Springer, 1994, pp. 86–103.

[20] Andrea Corradini, Barbara König, and Dennis Nolte. "Specifying graph languages with type graphs." In: *Journal of Logic and Algebraic Methods in Programming* 104 (2019), pp. 176–200.

[21] Andrea Corradini, Hartmut Ehrig, Michael Löwe, Ugo Montanari, and Francesca Rossi. "Note on standard representation of graphs and graph derivations." In: *Graph Transformations in Computer Science*. Springer, 1994, pp. 104–118.

[22] Andrea Corradini, Tobias Heindel, Frank Hermann, and Barbara König. "Sesqui-Pushout Rewriting." In: *Proc. of the International Conference on Graph Transformation*. Vol. 4178. LNCS. Sprin-ger, 2006, pp. 30–45.

[23] Andrea Corradini, Dominique Duval, Rachid Echahed, Frederic Prost, and Leila Ribeiro. "AGREE–algebraic graph rewriting with controlled embedding." In: *Proc. of the International Conference on Graph Transformation*. Vol. 9151. LNCS. Springer. 2015, pp. 35–51.

[24] Andrea Corradini, Tobias Heindel, Barbara König, Dennis Nolte, and Arend Rensink. "Rewriting Abstract Structures: Materialization Explained Categorically." In: *Proc. of the Internationcal Conference on Foundations of Software Science and Computation Structures*. Vol. 11425. Lecture Notes in Computer Science. Springer, 2019, pp. 169–188.

[25] Bruno Courcelle. "The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic." In: *Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 1*. 1997, pp. 313–400.

[26] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1977, pp. 238–252.

[27] Dennis Dams, Rob Gerth, and Orna Grumberg. "Abstract interpretation of reactive systems." In: *Transactions on Programming Languages and Systems* 19.2 (1997), pp. 253–291.

[28] Jürgen Dassow, Gheorghe Păun, and Arto Salomaa. "Grammars with Controlled Derivations." In: *Handbook of Formal Languages: Volume 2*. Springer, 1997, pp. 101–154.

[29] Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. "A distributed operational semantics for CCS based on condition/event systems." In: *Acta Informatica* 26.1 (1988), pp. 59–91.

[30] Isabel Dietrich and Falko Dressler. "On the lifetime of wireless sensor networks." In: *Trans. Sen. Netw.* 5.1 (2009), pp. 1–39.

[31] E. W. Dijkstra. "Solution of a problem in concurrent programming control." In: *Commun. ACM* 8.9 (Sept. 1965), p. 569.

[32] Dino Distefano, Arend Rensink, and Joost-Pieter Katoen. "Model Checking Birth and Death." In: *Proc. of the 2nd IFIP International Conference on Theoretical Computer Science*. Springer, 2002.

[33] Andrei Dorman and Tobias Heindel. "Structured Operational Semantics for Graph Rewriting." In: *ICE 2011*. Vol. 59. EPTCS. 2011, pp. 37–51.

[34] H. Ehrig and A. Habel. "Graph grammars with application conditions." In: *The Book of L*. Springer, 1986, pp. 87–100.

[35] Hartmut Ehrig and Barbara König. "Deriving Bisimulation Congruences in the DPO Approach to Graph Rewriting." In: *FOSSACS*. Springer, 2004, pp. 151–166.

[36] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. "Graph-grammars: An algebraic approach." In: *14th Annual Symposium on Switching and Automata Theory (swat 1973)*. IEEE. 1973, pp. 167–180.

[37] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, eds. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2*. World Scientific, 1999.

[38] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taen-tzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[39] E. Allen Emerson and Joseph Y. Halpern. "Sometimes and Not Never Revisited: On Branching Versus Linear Time Temporal Logic." In: *J. ACM* 33.1 (1986), pp. 151–178.

[40] Editors of Encyclopaedia Britannica. “Ashtadhyayi.” In: *Encyclopaedia Britannica*. On-line version, last accessed 15 Jan 2019. 2015. URL: `https://www.britannica.com/topic/Ashtadhyayi`.

[41] Maribel Fernández, Hélène Kirchner, and Bruno Pinaud. “Strategic port graph rewriting: an interactive modelling framework.” In: *Mathematical Structures in Computer Science* (2018), pp. 1–48.

[42] Maribel Fernández and Olivier Namet. “Strategic programming on graph rewriting systems.” In: *EPTCS* 44 (2010), pp. 1–20.

[43] Cédric Fournet and Georges Gonthier. “The Reflexive CHAM and the Join-calculus.” In: *Proc. of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1996, pp. 372–385.

[44] K Ruben Gabriel and Robert R Sokal. “A new statistical approach to geographic variation analysis.” In: *Systematic zoology* 18.3 (1969), pp. 259–278.

[45] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. “Modelling and Analysis using GROOVE.” In: *International Journal on Software Tools for Technology Transfer* 14 (2012), pp. 15–40.

[46] Ursula Goltz. “CCS and petri nets.” In: *Semantics of Systems of Concurrent Processes*. Springer, 1990, pp. 334–357.

[47] Ursula Goltz and Alan Mycroft. “On the Relationship of CCS and Petri Nets.” In: *Proc. of ICALP*. Springer, 1984, pp. 196–208.

[48] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. “Graph grammars with negative application conditions.” In: *Fundam. Inf.* 26.3,4 (1996), pp. 287–313.

[49] Annegret Habel and Detlef Plump. “Computational completeness of programming languages based on graph transformation.” In: *Proc. of the International Conference on Foundations of Software Science and Computation Structures*. Vol. 2030. LNCS. Springer, 2001, pp. 230–245.

[50] Reiko Heckel and Annika Wagner. “Ensuring consistency of conditional graph rewriting - a constructive approach.” In: *Electr. Notes Theor. Comput. Sci.* 2 (1995), pp. 118–126.

[51] Pavol Hell and Jaroslav Nesetril. “The core of a graph.” In: *Discrete Mathematics* 109.1-3 (1992), pp. 117–126.

[52] C. A. R. Hoare. “An axiomatic basis for computer programming.” In: *Commun. ACM* 12.10 (1969), pp. 576–580.

[53] C. A. R. Hoare. “Communicating Sequential Processes.” In: *Communications of the ACM* 21.8 (1978), pp. 666–677.

[54] Ghada Jaber, Rahim Kacimi, and Thierry Gayraud. "Data freshness aware content-centric networking in WSNs." In: *Wireless Days* (2017), pp. 238–240.

[55] William James. *A Pluralistic Universe*. Longmans, Green, and Co., 1909.

[56] Yichao Jin, Parag Kulkarni, Sedat Gormus, and Mahesh Sooriyabandara. "Content-centric and load-balancing aware dynamic data aggregation in multihop wireless networks." In: *Proc. of the International Conference on Wireless and Mobile Computing, Networking and Communications*. IEEE. 2012, pp. 179–186.

[57] Xaroula Charalampia Kerasidou. "Figuring ubicomp (out)." In: *Personal and Ubiquitous Computing* 21.3 (2017), pp. 593–605.

[58] Roland Kluge, Michael Stein, Gergely Varró, Andy Schürr, Matthias Hollick, and Max Mühlhäuser. "A Systematic Approach to Constructing Families of Incremental Topology Control Algorithms Using Graph Transformation." In: *Journal of Software and Systems Modeling* 18.1 (2019), pp. 279–319.

[59] Barbara König and Vitali Kozioura. "Counterexample-guided abstraction refinement for the analysis of graph transformation systems." In: *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3920. LNCS. Springer, 2006, pp. 197–211.

[60] Dexter Kozen. "Results on the propositional mu-calculus." In: *Theoretical Computer Science* 27.3 (1983), pp. 333 –354.

[61] Hans-Jörg Kreowski and Sabine Kuske. "On the interleaving semantics of transformation units - a step into GRACE." In: *Proc. of the 5th International Workshop on Graph Gramars and Their Application to Computer Science*. Vol. 1073. LNCS. 1994, pp. 89–106.

[62] Hans-Jörg Kreowski, Sabine Kuske, and Grzegorz Rozenberg. "Graph transformation units – an overview." In: *Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*. Vol. 5065. LNCS. Springer, 2008, pp. 57–75.

[63] Géza Kulcsár, Andrea Corradini, and Malte Lochau. "Equivalence and independence in controlled graph-rewriting systems." In: *Proc. of the International Conference on Graph Transformation*. Vol. 10887. LNCS. Springer, 2018, pp. 134–151.

[64] Géza Kulcsár, Malte Lochau, and Andy Schürr. "Graph-rewriting Petri nets." In: *Proc. of the International Conference on Graph Transformation*. Vol. 10887. LNCS. Springer, 2018, pp. 79–96.

[65] Géza Kulcsár, Michael Stein, Immanuel Schweizer, Gergely Varró, Max Mühlhäuser, and Andy Schürr. “Rapid prototyping of topology control algorithms by graph transformation.” In: *Electronic Communications of the EASST* 68 (2014).

[66] Leslie Lamport. “Turing lecture: The computer science of concurrency: the early years.” In: *Commun. ACM* 58.6 (2015), pp. 71–76.

[67] Ivan Lanese, Luca Bedogni, and Marco Di Felice. “Internet of things: a process calculus approach.” In: *28th Annual ACM Symposium on Applied Computing*. 2013, pp. 1339–1346.

[68] Ruggero Lanotte and Massimo Merro. “A semantic theory of the Internet of Things.” In: *Information and Computation* 259 (2018), pp. 72 –101.

[69] Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, Peter Fritzson, Jörg Brauer, Christian Kleijn, Thierry Lecomte, Markus Pfeil, Ole Green, Stylianos Basagiannis, et al. “Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project.” In: *2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*. 2016, pp. 1–6.

[70] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. “Developing eMoflon with eMoflon.” In: *Proc. of the International Conference on Theory and Practice of Model Transformations*. Vol. 8568. LNCS. Springer, 2014, pp. 138–145.

[71] James J Leifer and Robin Milner. “Deriving Bisimulation Congruences for Reactive Systems.” In: *CONCUR*. Springer. 2000, pp. 243–258.

[72] Mo Li, Zhenjiang Li, and Athanasios V Vasilakos. “A survey on topology control in wireless sensor networks: Taxonomy, comparative study, and open issues.” In: *Proceedings of the IEEE* 101.12 (2013), pp. 2538–2557.

[73] Michael Löwe. “Algebraic approach to single-pushout graph transformation.” In: *Theoretical Computer Science* 109.1 (1993), pp. 181 –224.

[74] Melanie Luderer. “Control Conditions for Transformation Units: Parallelism, As-long-as-possible, and Stepwise Control.” PhD thesis. University of Bremen, 2016.

[75] Andrea Maggiolo-Schettini and Józef Winkowski. “Dynamic graphs.” In: *Proc. of the International Symposium on Mathematical Foundations of Computer Science*. Vol. 1113. LNCS. 1996, pp. 431–442.

[76] S. Mauw and M. A. Reniers. “A Process Algebra for Interworkings.” In: *Handbook of Process Algebra*. Elsevier Science, 2001, pp. 1269 –1327.

[77] L.G. Meredith and Matthias Radestock. "A Reflective Higher-order Calculus." In: *Electronic Notes in Theoretical Computer Science* 141.5 (2005). Proc. of the Workshop on the Foundations of Interactive Computation (FInCo 2005), pp. 49 –67.

[78] Giorgio De Michelis. "The contribution of Carl Adam Petri to our understanding of 'computing'." In: *History and Philosophy of Computing - Third International Conference*. 2015, pp. 156–167.

[79] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[80] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[81] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, I." In: *Information and Computation* 100.1 (1992), pp. 1 –40.

[82] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, II." In: *Information and Computation* 100.1 (1992), pp. 41 –77.

[83] Madhavan Mukund and Mogens Nielsen. "CCS, locations and asynchronous transition systems." In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Vol. 652. LNCS. Springer. 1992, pp. 328–341.

[84] Manfred Nagl, ed. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. Vol. 1170. LNCS. Sprin-ger, 1996.

[85] Ulrich Nickel, Jörg Niere, and Albert Zündorf. "The FUJABA environment." In: *Proc. of the International Conference on Software Engineering*. ACM, 2000, pp. 742–745.

[86] Mogens Nielsen and Glynn Winskel. "Petri nets and bisimulation." In: *Theoretical Computer Science* 153.1 (1996), pp. 211 –244.

[87] Joachim Parrow and Björn Victor. "The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes." In: *LICS*. 1998, pp. 176–185.

[88] Joachim Parrow, Johannes Borgström, Lars-Henrik Eriksson, Ramunas Gutkovas, and Tjark Weber. "Modal Logics for Nominal Transition Systems." In: *Proc. of the International Conference on Concurrency Theory*. 2015, pp. 198–211.

[89] Carl Adam Petri. *Net modelling – fit for science?* Lecture Presented at the 24th European Conference on Application and Theory of Petri Nets, Eindhoven, 2003.

[90] Gordon D. Plotkin. "A Structural Approach to Operational Semantics." In: *J. Log. Algebr. Program.* 60-61 (2004), pp. 17–139.

[91] Detlef Plump. "The graph programming language GP." In: *Proc. of the International Conference on Algebraic Informatics*. Vol. 5725. LNCS. Springer, 2009, pp. 99–122.

[92] Detlef Plump. "The design of GP 2." In: *Proc. of the International Workshop on Reduction Strategies in Rewriting and Programming*. 2011.

[93] Amir Pnueli. "The Temporal Logic of Programs." In: *Proc. of the Annual Symposium on Foundations of Computer Science*. IEEE, 1977, pp. 46–57.

[94] Christopher M. Poskitt and Detlef Plump. "Hoare-style verification of graph programs." In: *Fundam. Inf.* 118.1-2 (2012), pp. 135–175.

[95] Arend Rensink. "Canonical Graph Shapes." In: *Proc. of the European Symposium on Programming*. Vol. 2986. Lecture Notes in Computer Science. Springer, 2004, pp. 401–415.

[96] Arend Rensink. "Representing first-order logic using graphs." In: *Proc. of the International Conference on Graph Transformation*. Vol. 4178. LNCS. Springer. 2004, pp. 319–335.

[97] Richard Rorty. *Philosophy and the Mirror of Nature*. Princeton University Press, 1980.

[98] Grzegorz Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1*. World Scientific, 1997.

[99] Andreas Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen - formale Definitionen, Anwendungsbeispiele und Werkzeugunterstützung*. DUV Informatik. Deutscher Universitätsverlag, 1991.

[100] Andy Schürr. "Logic based programmed structure rewriting systems." In: *Fundam. Inform.* 26.3/4 (1996), pp. 363–385.

[101] Andy Schürr. "Programmed Graph Replacement Systems." In: *Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 1*. 1997, pp. 479–546.

[102] Andy Schürr, A. J. Winter, and Albert Zündorf. "The PROGRES-approach: language and environment." In: *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2*. World Scientific, 1999, pp. 487–550.

[103] Immanuel Schweizer, Michael Wagner, Dirk Bradler, Max Mühlhäuser, and Thorsten Strufe. "kTC - Robust and adaptive wireless ad-hoc topology control." In: *Proc. of the International Conference on Computer Communications and Networks*. IEEE. 2012, pp. 1–9.

[104] Shankara and Ganganatha Jha. *The Chandogyopanishad*. Poona Oriental Book Agency, 1942.

[105] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[106] Dominik Steenken, Heike Wehrheim, and Daniel Wonisch. "Sound and Complete Abstract Graph Transformation." In: *Proc. of the Brazilian Conference on Formal Methods: Foundations and Applications*. Springer, 2011, pp. 92–107.

[107] Michael Stein, Tobias Petry, Immanuel Schweizer, Martina Brachmann, and Max Mühlhäuser. "Topology control in wireless sensor networks: What blocks the breakthrough?" In: *Conference on Local Computer Networks*. IEEE. 2016, pp. 389–397.

[108] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. "Henshin: A usability-focused framework for EMF model transformation development." In: *Proc. of the International Conference on Graph Transformation*. Vol. 10373. LNCS. Springer, 2017, 196–208.

[109] Alan M Turing. "On computable numbers, with an application to the Entscheidungsproblem." In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265.

[110] Jason Vallet, Hélène Kirchner, Bruno Pinaud, and Guy Melançon. "A visual analytics approach to compare propagation models in social networks." In: *Proc. of the International Workshop on Graphs as Models*. 2015.

[111] Roger Wattenhofer and Aaron Zollinger. "XTC: A practical topology control algorithm for ad-hoc networks." In: *Parallel and Distributed Processing Symposium*. IEEE. 2004, p. 216.

[112] Roger Wattenhofer, Li Li, Paramvir Bahl, and Y-M Wang. "Distributed topology control for power efficient operation in multihop wireless ad hoc networks." In: *Proc. of the International Conference on Computer Communications*. IEEE. 2001, pp. 1388–1397.

[113] Mark Weiser. "The computer for the 21st century." In: *Scientific American* 265.3 (1991), pp. 94–105.

CURRICULUM VITÆ

	Personal Details
Date of Birth	9th of Sep, 1988
Place of Birth	Cegléd, Hungary
Nationality	Hungarian

	Work Experience
2013 – 2019	Researcher and Ph.D. candidate, Real-Time Systems Lab, TU Darmstadt, Germany
2010 – 2013	Software developer, Ericsson Telecommunications, Budapest, Hungary

	Education
2013	Master of Science, Computational Engineering, TU Budapest, Hungary. Thesis title: Connections between Graphs and Matroids (in Hungarian)
2011	Bachelor of Science, Computational Engineering, TU Budapest, Hungary. Thesis title: Intelligent Automatic Parameterization of Texture Analysis Algorithms (in Hungarian)
2007	A-Levels, Kossuth Lajos Gimnázium, Cegléd, Hungary