Analysis of Communication Topologies by Partner Abstraction

Dissertation

zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften (Dr.-Ing.) der
Naturwissenschaftlich-Technischen Fakultäten der
Universität des Saarlandes

von
Diplom-Informatiker

Jörg Bauer

Saarbrücken
September 2006

Tag des Kolloquiums:	13.12.2006
Dekan:	Prof. Dr.-Ing. Thorsten Herfet
Prüfungsausschuss:	
Vorsitzender:	Prof. Dr. Holger Hermanns Universität des Saarlandes, Saarbrücken
Gutachter:	Prof. Dr. Reinhard Wilhelm Universität des Saarlandes, Saarbrücken
	Prof. Dr. Werner Damm Carl von Ossietzky Universität, Oldenburg
	Prof. Dr. Arend Rensink University of Twente, Enschede
Akademischer Mitarbeiter:	Dr. Jan Schwinghammer

Abstract

Dynamic communication systems are hard to verify due to inherent unboundedness. Unbounded creation and destruction of objects and a dynamically evolving communication topology are characteristic features. Prominent examples include traffic control systems based on wireless communication and ad hoc networks. As dynamic communication systems have to meet safety-critical requirements, this thesis develops appropriate specification and verification techniques for them. It is shown that earlier attempts at doing so have failed.

Partner graph grammars are presented as an adequate specification formalism for dynamic communication systems. They form a novel variant of the single pushout approach to algebraic graph transformation equipped with a special kind of negative application conditions: Partner constraints that allow to reason about communication partners are specifically tailored to dynamic communication systems.

A novel verification technique based on abstract interpretation of partner graph grammars is proposed. It is based on a two-layered abstraction that keeps precise information about objects and the kinds of their communication partners. The analysis is formally proven sound. Some statically checkable cases are defined for which the analysis results are even complete.

The analysis has been implemented in the `hiralysis` tool. A complex case study – car platooning originally developed in the California PATH project – is modeled using partner graph grammars. An experimental evaluation using the tool discovered many flaws in the PATH specification of car platooning that had not been discovered earlier due to insufficient specification and verification methods. Many interesting properties can be automatically proven for a corrected implementation of car platooning using `hiralysis`.

Zusammenfassung

Aufgrund ihres unbeschränkten Verhaltens sind dynamisch kommunizierende Systeme schwierig zu verifizieren. Sie zeichnen sich durch unbegrenztes Erzeugen und Zerstören von Objekten sowie eine sich ständig ändernde Kommunikationstopologie aus. Funkbasierte Verkehrskontrollsysteme und drahtlose Ad-hoc Netzwerke sind bekannte Beispiele dynamisch kommunizierender Systeme. Da diese außerdem sicherheitskritischen Anforderungen genügen müssen, werden in dieser Arbeit Spezifikations- und Verifikationsmethoden für dynamisch kommunizierende Systeme entwickelt. Es wird gezeigt, dass frühere Versuche in dieser Richtung fehlgeschlagen sind.

Partner-Graphgrammatiken stellen einen geeigneten Formalismus zur Beschreibung solcher Systeme dar. Sie bilden eine neue Form des "single pushout" Ansatzes für algebraische Graphtransformationen erweitert um besondere negative Anwendungsbedingungen. "Partner constraints", die speziell für die Spezifikation dynamisch kommunizierender Systeme entwickelt wurden, erlauben, Nebenbedingungen an Objekte und ihre Kommunikationspartner zu formulieren.

Es wird eine neuartige Verifikationstechnik vorgeschlagen, die auf der abstrakten Interpretation von Partner-Graphgrammatiken beruht. Diese fußt auf einer Abstraktion, die präzise Informationen über Objekte und ihre Kommunikationspartner erhält. Die Analyse wird korrekt bewiesen, und es werden statisch erkennbare Fälle aufgezeigt, in denen die Analyse sogar vollständige Resultate liefert.

Die Analyse wurde in dem `hiralysis` Werkzeug implementiert. Eine komplexe Fallstudie – "car platooning", welche ursprünglich im Rahmen des California PATH Projektes entwickelt wurde – wird durch Partner-Graphgrammatiken modelliert. Eine experimentelle Auswertung mithilfe des Werkzeugs deckte zahlreiche Fehler in der ursprünglichen Modellierung auf, welche ihre Ursache in unzureichenden Spezifikations- und Verifikationsmethoden haben. Viele interessante Eigenschaften eines verbesserten Modells konnten mittels `hiralysis` automatisch bewiesen werden.

Extended Abstract

This thesis deals with the specification and verification of dynamic communication systems. Such systems are pervasive; traffic-control systems based on wireless communication and ad hoc networks are prominent examples of them. They have to meet safety-critical requirements. They are hard to verify, though, because they consist of an unbounded and constantly changing number of objects. Moreover, communication links among these objects are constantly changing, *i.e.*, there is an evolving communication topology. The goal of this thesis is thus to specify dynamically evolving communication topologies and to specify and verify properties of them.

A running example used throughout this thesis is based on car platooning maneuver protocols as they were defined in the context of the California PATH project. This project is concerned with maximizing traffic throughput on highways, while, at the same time, reducing energy costs by exploiting driving in slipstreams. Cars heading for the same direction are supposed to drive close to each other, in order to meet these goals. Such a set of closely spaced cars is called a platoon. Platoons can perform maneuvers like merging or splitting. Platoon maneuver protocols were designed within PATH. However, it will be shown, that both the specification and the verification of platoon maneuver protocols in the original PATH project are inappropriate and erroneous, because many features typical of a dynamic communication system were not considered.

A new formalism for the specification of dynamic communication systems, in particular car platoon maneuver protocols, is proposed in this thesis. It is called partner graph grammars, because

- It is a variant of the single pushout approach to algebraic graph transformation.
- The notion of partners is employed to develop a novel kind of negative application conditions: partner constraints.

The partners of an object are those objects to which it has communication links. It is conjectured that the possible behavior of an object is determined

by the state of the object itself and the states of its partners. This observation is exploited by the newly developed concept of partner constraints. As a part of a partner graph grammar, a partner constraint restricts the applicability of a graph transformation rule by constraining the possible partners of objects. This feature proves essential for modeling dynamic communication systems. Compared to general negative application conditions, it is less expressive. On the other hand, it is tailored to the application domain and still amenable to formal verification. The usefulness of partner graph grammars is demonstrated by augmenting the standard platoon maneuver protocols by features like message queues or unreliable communication links. In order to formally express properties of partner graph grammars, a new logic, $\mathcal{GL}$, is developed. All interesting properties of partner graph grammars in general and of the platoon example in particular are expressible in $\mathcal{GL}$.

Apart from the specification of dynamic communication systems and their properties, this thesis is also concerned with their verification. Technically, the verification amounts to an abstract interpretation of partner graph grammars. It is based on the notion of partner abstraction. Partner abstraction is a novel, two-layered abstraction based on the notion of partner equivalence. Just like partner constraints, partner equivalence is motivated by the observation about the behavior of an object being determined by the object itself and its communication partners. Two objects are considered partner equivalent, if they are in the same state *and* if the sets of states of their communication partners are equal. This means that partner equivalent objects will show an equal behavior.

Partner abstraction based abstract interpretation allows to statically compute a sound over-approximation of all possible communication topologies that can be generated by a partner graph grammar. Furthermore, statically checkable criteria are given, when an analysis result is even complete. Completeness results can be established for at least some of the platoon maneuver protocol implementations. Finally, strong property preservation results are obtained. They make statements about which properties of a partner graph grammar – specified in $\mathcal{GL}$ – are preserved in the abstract interpretation of the partner graph grammar.

Partner abstraction based abstract interpretation of partner graph grammars has been implemented in the `hiralysis` tool. It is used to perform an experimental evaluation of the platoon maneuver protocols. With the help of the tool, a number of flaws in the PATH specification are discovered, correct versions are developed and proven correct.

Ausführliche Zusammenfassung

Diese Arbeit behandelt die Spezifikation und Verifikation dynamisch kommunizierender Systeme. Diese Systeme sind allgegenwärtig. Funkbasierte Verkehrskontrollsysteme und drahtlose Ad-hoc Netzwerke sind nur zwei Beispiele. Häufig müssen sie sicherheitsrelevanten Anforderungen genügen. Dynamisch kommunizierende Systeme sind jedoch schwierig zu verifizieren, da sich sowohl die Anzahl der Objekte innerhalb eines Systems als auch die Kommunikationsverbindungen zwischen Objekten ständig ändern und von unbeschränkter Größe sind. Ziel dieser Arbeit ist deshalb, sich dynamisch ändernde Kommunikationstopologien und ihre Eigenschaften formal zu spezifizieren und zu verifizieren.

Protokolle für Platoon-Manöver, wie sie im Rahmen des California PATH Projektes definiert wurden, bilden die wesentliche Fallstudie dieser Arbeit. Das PATH Projekt strebt eine Optimierung des Verkehrsflusses auf Autobahnen bei gleichzeitig geringerem Energieverbrauch durch Windschattenfahren an. Dies wird durch Kolonnenbildung erreicht, wobei Kolonnen als Platoons bezeichnet werden. Platoons können sich vereinen und teilen. Protokolle für diese Abläufe wurden im Rahmen von PATH entwickelt. Es zeigt sich jedoch, dass sowohl die Spezifikation als auch die Verifikation dieser Protokolle unangemessen und fehlerbehaftet ist, da viele typische Merkmale dynamisch kommunizierender Systeme außer Acht gelassen werden.

In dieser Arbeit wird ein neuer Ansatz zur Spezifikation und Verifikation dynamisch kommunizierender Systeme vorgeschlagen. Er heißt Partner-Graphgrammatik, weil

- er eine besondere Form einer Graphgrammatik, sprich des "single push-out" Ansatzes für algebraische Graphtransformationen, ist.
- der Begriff des Partners genutzt wir, um eine neue Form negativer Anwendungsbedingungen, "partner constraints", zu entwickeln.

Die Partner eines Objektes sind diejenigen, mit denen es kommuniziert. Das Verhalten eines Objektes wird von seinem eigenen und dem Zustand seiner

Partner bestimmt. Auf Spezifikationsseite wird dies durch "partner constraints" ausgenutzt. Als Teil einer Partner-Graphgrammatik beschränken solche Nebenbedingungen die Anwendbarkeit von Transformationsregeln, was durch Anforderungen an die Partner von Objekten erreicht wird. Dies ist von herausragender Bedeutung für die Benutzbarkeit von Partner-Graphgrammatiken. Trotz geringerer Ausdrucksstärke im Vergleich zu allgemeinen negativen Anwendungsbedingungen, sind "partner constraints" ideal zur Spezifikation dynamisch kommunizierender Systeme und gleichzeitig zugänglich für formale Verifikation. Die Nützlichkeit von Partner-Graphgrammatiken wird durch die Erweiterung der Platoon Fallstudie um explizite Nachrichtenwarteschlangen und fehleranfällige Kommunikationsverbindungen demonstriert. Eine neue Logik, $\mathcal{GL}$, wird entwickelt, um Eigenschaften von Partner-Graphgrammatiken formal aufzuschreiben. Alle interessanten Eigenschaften von Partner-Graphgrammatiken im Allgemeinen und von Platoons im Besonderen sind in $\mathcal{GL}$ ausdrückbar.

Neben der Spezifikation dynamisch kommunizierender Systeme befasst sich diese Arbeit auch mit ihrer Verifikation. Technisch läuft dies auf abstrakte Interpretation von Partner-Graphgrammatiken hinaus. Diese basiert auf Partnerabstraktion, welche eine neuartige, zweistufige Abstraktion definiert. Diese beruht ihrerseits auf dem Begriff der Partneräquivalenz. Genau wie bei "partner constraints" macht man sich das Prinzip zunutze, dass das Verhalten eines Objekts von seinem eigenen und dem Zustand seiner Partner bestimmt wird. Zwei Objekte heißen partneräquivalent, wenn sie im selben Zustand sind, und wenn die Mengen der Zustände ihrer Partner übereinstimmen.

Mithilfe abstrakter Interpretation beruhend auf Partnerabstraktion kann eine korrekte Überapproximation aller möglichen Kommunikationstopologien einer Partner-Graphgrammatik statisch berechnet werden. Zudem werden statisch überprüfbare Bedingungen angegeben, unter denen eine abstrakte Interpretation sogar vollständig ist. Vollständigkeitsresultate können zumindest für manche der Platoonimplementierungen nachgewiesen werden. Schließlich werden in dieser Arbeit Sätze über die Erhaltung von in $\mathcal{GL}$ aufgeschriebenen Eigenschaften bewiesen. Diese Sätze machen Aussagen darüber welche Eigenschaften konkreter Partner-Graphgrammatiken von der abstrakten Interpretation erhalten werden.

Die abstrakte Interpretation von Partner-Graphgrammatiken ist in dem `hiralysis` Werkzeug realisiert worden. Mithilfe dieses Werkzeugs wird eine experimentelle Auswertung der Platoon Protokolle vorgenommen. Dadurch wurden zahlreiche Fehler in der ursprünglichen PATH Spezifikation aufgedeckt, sowie korrekte Protokolle entwickelt und als korrekt bewiesen.

Acknowledgement

First of all, I need to thank my advisor, Reinhard Wilhelm, for giving me the opportunity to do research under his guidance. This thesis would not have been possible without him. In particular, I want to thank him for his support during the hard times, for getting introduced to many interesting people, and for letting me participate in interesting Dagstuhl seminars.

I want to thank Mooly Sagiv for inviting me to Tel Aviv and for joined research that used to be as intense as running with him. Among Mooly's students, I would like to thank Noam and Greta.

My research did not really kick-off, until I got involved in the AVACS project that supported me with grants. There, I met Bernd and Tobe who have been my closest collaborators for almost three years. Also, we discovered the much appreciated platoon case study. Ina used to support us in the S2 project, but decided to leave it too early. Also, I want to thank the remaining crew of S2.

I want to express my gratitude to Hanne Riis Nielson and Flemming Nielson for taking care of my scientific life before and after the time at Reinhard Wilhelm's chair.

Finally, my thanks go to the reviewers of this thesis for reading the result of my efforts.

There is a life beside research, so I thank my family for lifelong support. Many people made my time at this chair worthwhile. Christian Probst gave me a hearty welcome and remained a friend throughout. Thanks to Dante, DaWall, Eiermatz, FunkyBunch, Magic, and Mari I enjoyed hilarious lunch breaks for most of the time. Sewi and Taker deserve my thanks for demanding badminton session, whereas most of the BB crew has been mentioned.

Last but not least, I want to express my utmost gratitude to Eva. She knows why.

Contents

1 Introduction **1**
- 1.1 A Problem . . . 2
- 1.2 . . . and its Solution . . . 4

2 Specification of Evolving Graphs **7**
- 2.1 Partner Graph Grammars . . . 7
  - 2.1.1 Multisets . . . 8
  - 2.1.2 Graph Preliminaries . . . 8
  - 2.1.3 Transformation Rules . . . 11
  - 2.1.4 Partner Constraints . . . 16
  - 2.1.5 Partner Graph Grammars . . . 18
  - 2.1.6 Special Cases and Shorthands . . . 20
- 2.2 $\mathcal{GL}$: Reasoning about Graph Grammars . . . 25
- 2.3 Relation to Algebraic Graph Transformation . . . 30

3 Case Studies **35**
- 3.1 Car Platooning . . . 36
  - 3.1.1 Idealized Platoons . . . 40
  - 3.1.2 Asynchronous Communication . . . 47
  - 3.1.3 Faulty Channels . . . 56
- 3.2 Further Applications . . . 59

4 Partner Abstraction **63**
- 4.1 Abstract Interpretation . . . 63
  - 4.1.1 Introduction to Abstract Interpretation . . . 64
  - 4.1.2 Partner Abstraction of Single Graphs . . . 66
  - 4.1.3 Materialization . . . 74
  - 4.1.4 Abstract Matches . . . 76
  - 4.1.5 Abstract Transformers . . . 82
- 4.2 Abstract Transition Systems . . . 85
  - 4.2.1 Cluster Evolution . . . 85

4.2.2 Node Evolution 87
4.3 Completeness Results 89
4.3.1 Completeness Notions 90
4.3.2 Friendly Systems 91
4.3.3 Cluster Completeness 93
4.3.4 Decidability of the Word Problem 103
4.4 Property Preservation 104
4.4.1 Property Preservation and Partner Abstraction 105
4.4.2 Invariants 106
4.4.3 Extensions 108
4.5 Evaluation of Case Studies 109
4.5.1 Implementation: `hiralysis` 110
4.5.2 Car Platooning 113
4.5.3 Experiences 118
4.6 Extensions 120
4.6.1 Counting Clusters 121
4.6.2 Generalized Clusters 124

5 Related Work **129**

6 Conclusion **135**
6.1 Contribution 135
6.2 Outlook 138

A Proofs **141**

B Tool Samples **159**

Chapter 1

Introduction

This thesis solves the problem of specification and verification of *dynamic communication systems*. In the beginning, it is elaborated on what dynamic communication systems are and why they and their properties are hard to specify. Once specified, properties of dynamic communication systems are very hard to verify, too. The application domain of dynamic communication systems is illustrated by a complex case study originally developed in the California PATH project. The achievements of this thesis are strengthened by the fact, that the PATH project could neither satisfactorily specify nor verify this particular case study. In contrast, this thesis presents a solution in terms of a novel specification formalism, a novel formalism for specifying properties of dynamic communication systems, and a novel verification technique. On top of that, the verification technique is implemented and proven useful by a number of experiments.

Dynamic communication systems are characterized by an unbounded number of dynamically created, stateful, linked objects. Other prominent examples of such systems are heap manipulating programs, distributed algorithms, and ad-hoc networks. The latter example is in fact an instance of a dynamic communication system, because it exhibits all the characterizing features:

1. Disappearance and unbounded creation of objects.
2. Wireless, unreliable communication among objects.
3. A dynamically evolving communication topology.

In contrast, heap-manipulating programs are mostly characterized by (1,3) and hardly by (2), whereas distributed algorithms are not so much dependent on (1). Moreover, state changes in heap-manipulating programs or

distributed algorithms happen in a much more disciplined manner compared to dynamic communication systems.

A *communication topology* of a dynamic communication system is a *global* state of the system. It describes all the objects currently present in the system and their interconnections. Each object itself has a *local* state. Changes to the communication topology happen frequently during a run of a dynamic communication system. They are caused by communication among objects, local computations by objects, appearances of new objects, disappearances of existing objects, or creation and destruction of communication links.

The *semantics* of a dynamic communication system is described as a transition system, where transitions occur between communication topologies. A more abstract semantics, later called the *graph semantics*, abstracts from the transitions and merely collects all communication topologies that may occur during any run of a dynamic communication system.

Properties (1), (2), and (3) inflict major problems on both the specification and the verification of dynamic communication systems. The semantics of a dynamic communication system will typically be a transition system of infinite size. Even worse, there will mostly be no a priori bound on the size of each communication topology within the transition system. This thesis presents a novel technique that addresses both the specification and the verification of dynamic communication systems.

1.1 A Problem ...

The running example used throughout this thesis presents a prototypical instance of a dynamic communication system. It is taken from the California PATH project [HESV91], the relevant part of which is concerned with cars driving on a highway. In order to make better use of the given space, cars heading for the same direction are supposed to drive very close to each other building *platoons*. Platoons can perform actions like merging or splitting.

Note that in the context of platoons, the general term communication link is often replaced by the term channel, because the latter is used in the original PATH specification of platoon maneuvers. A platoon consists of a *leader*, the foremost car, along with a number of *followers*. A leader without any followers is called *free agent* and is considered a special platoon. Within a platoon there are channels between the leader and each of its followers. Inter-platoon communication occurs only between leaders.

As an example of a maneuver involving two platoons, consider the platoon *merge maneuver*. It allows two approaching platoons to merge. The merge

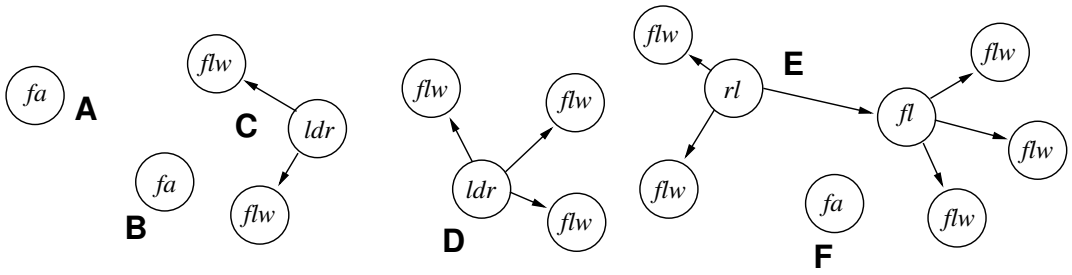


Fig. 1.1: The representation of a communication topology as directed, node-labeled graph, where the label of a node is written inside the node. The node label represents the local state of an object, here a car. The labels used here stand for free agent (*fa*), leader (*ldr*), follower (*flw*), rear leader (*rl*), and front leader (*fl*). Subgraphs **A**, **B**, and **F** represent platoons of one free agent each. **C** and **D** represent platoons of three and four cars, respectively, whereas **E** depicts a snapshot during a merge maneuver.

maneuver is initiated by opening a channel between two distinct platoon leaders, *i.e.*, leaders or free agents. Then, the rear leader passes its followers, if any, one by one to the front leader. Finally, when there are no followers left to the rear leader, it becomes itself a follower to the front leader.

A communication topology as it may occur during a run of the platoon dynamic communication system is depicted in Figure 1.1. Examples of interesting properties that could be investigated in the platoon case study are the following: Are there two cars that consider each other to be their leader? Will a merge maneuver always terminate? If a merge maneuver terminates, will it yield the right result? Do cars always have one unique leader, unless they are free agents? Does the merge protocol work in the presence of faulty channels? These properties will be addressed and verified later on.

In contrast, all the specification and verification methods developed in [HESV91] are inappropriate, because they consider static scenarios with a fixed number of cars and only limited concurrency. This will be dwelled on at each recurrence of the case study, which will be stated formally in Section 3.1. The failure of a big project like PATH to develop appropriate methods for dealing with dynamic communication systems stresses both the complexity of the problem and the achievements of this work.

1.2 . . . and its Solution

So far, the goal of this thesis has been formulated in terms of dynamic communication systems, *i.e.*, application driven. This section states the problem on a more technical level. At the same time, it indicates some solutions and presents an outline of the thesis.

Chapter 2 introduces a novel specification technique for dynamic communication systems. Communication topologies are naturally interpreted as node- and edge-labeled, directed graphs. Changes to the communication topology can thus be naturally described as the application of a *graph transformation rule*. Such a rule consists of a left graph and a right graph. If the left graph is found in some graph G the application of a rule replaces the occurrence of the left graph by the right graph. A set of such rules combined with an initial graph defines a *graph grammar*. Graph grammars have a rich underlying theory, and [Roz97] is a good starting point for further reading.

This thesis defines a new kind of graph grammar, a *partner graph grammar*. It is named after its distinguishing feature: *partner constraints*. Partner constraints are a special instance of negative application conditions used in standard graph grammars. They augment left graphs of transformation rules and restrict the applicability of the rule by explicitly stating conditions, when a rule shall *not* be applied. Partner constraints restrict the applicability of rules by putting constraints on the adjacent nodes of nodes occurring in left graphs of transformation rules. In other words, they constrain the possible communication partners of an object, making them an ideal tool for the application domain of dynamic communication systems.

Although negative application conditions are not novel, partner constraints present a restricted instance of them, that has great expressive power and is at the same time amenable to formal verification. Graph grammars become hardly usable, if they lack any kind of negative application conditions as is the case for the competing technique proposed in [RD06]. Other advantages of partner graph grammars include their constructive definition and their ease of use as compared to standard graph grammars. The relation of partner graph grammars and standard graph grammars is clarified in Section 2.3.

A new logic, $\mathcal{GL}$, is developed, in order to formally express sophisticated properties of partner graph grammars. All properties of the case studies are expressible and are in fact expressed in this logic. Technically, it is based on the computation tree logic CTL, which is augmented with novel features to conveniently reason about partner graph grammars.

Chapter 3 formally states the platoon case study in terms of partner

graph grammars. Additionally, the inappropriateness of the original platoon specification and verification within the PATH project is uncovered. The platoon scenario is extended by adding features like asynchronous communication based on message queues and the possibility of unreliable channels. Partner graph grammars lend themselves to the specification of applications from domains other than pure dynamic communication systems. This issue is elaborated on in Section 3.2, where also the limits of partner graph grammar based specification and verification are conceded.

Chapter 4 constitutes the technical core and the major contributions of this thesis. In the beginning, it defines a *static analysis* of partner graph grammars based on the technique of abstract interpretation [CC77, CC79]. Abstract interpretation has been applied in many domains. Originally developed for static program analysis, it has now been applied to such diverse fields as worst-case execution time determination [FH05] or systems biology [NNP04].

As partner graph grammars yield a semantics of unbounded size, abstraction needs to be applied to make provably correct statements about them. More precisely, the abstract interpretation proposed here statically computes an over-approximation of bounded size of the graph semantics of a given partner graph grammar. The graph semantics of a partner graph grammar is the set of all graphs that are generated by the grammar. The over-approximation is called the *abstract graph semantics* of the partner graph grammar. Recall that the graph semantics is the set of all graphs that can be generated by a partner graph grammar. The analysis is proven to be sound.

The abstract interpretation is based on a novel two-layered abstraction of graphs called *partner abstraction* – the abstraction twin of partner constraints that are used for specification. Partner abstraction itself is based on partner equivalence. From a dynamic communication system's point of view, two objects are considered equal, if they are in the same state *and* if the sets of states of their communication partners are equal to each other. This information determines the successor state of an object within a communication topology making partner equivalence the *ideal abstraction* for the analysis of dynamic communication systems. Technically, the first layer of abstraction is quotient graph building *wrt.* partner equivalence *per connected component* of a graph. The second layer of abstraction summarizes all connected components that are isomorphic after the first abstraction step. This is motivated by the fact, that objects that are not even indirectly connected by communication links cannot influence each others' behavior.

After the definition of abstract graphs, updates on abstract graphs need to be defined. This requires the notion of an *abstract match*, where a match

determines whether the left graph of a rule matches another graph. As nodes are summarized to obtain abstract graphs, abstract matches must be different from concrete matches. A central theorem of this work identifies cases, where abstract matches are in fact *equivalent* to concrete matches. More precisely, in these cases a rule matches an abstract graph $\hat{G}$, if and only if it matches all graphs G that have $\hat{G}$ as their abstraction.

Based on the matching theorem, the notion of completeness of an abstract graph semantics is defined in Section 4.3. Informally, an analysis is complete, if it does not lose any relevant information. The corresponding theorem is a major achievement. It provides statically checkable sufficient criteria for the completeness of an abstract graph semantics. These completeness results are surprising and could not have been expected before, considering the complexity of partner graph grammars.

Furthermore, the completeness results imply strong property preservation results discussed in Section 4.4. They are another essential contribution of this work. That section investigates which $\mathcal{GL}$ properties are preserved by the abstract interpretation of partner graph grammars and proves theorems about this preservation.

The abstract interpretation has been implemented in the `hiralysis` tool. It was applied to a significant set of experiments from the platoon case study. The evaluation of these experiments is reported in Section 4.5. Many interesting properties could be proven automatically using the `hiralysis` tool. All the results stated above present novel research applied to a case study not amenable to formal verification so far. Related approaches are discussed in Chapter 5.

Chapter 2

Specification of Evolving Graphs

Partner graph grammars are introduced as an adequate specification formalism for dynamic communication systems. They are a variant of standard, single-pushout based algebraic graph transformation systems. The relation to the latter will be clarified at the end of this chapter. Besides, the logic $\mathcal{GL}$ is introduced in this chapter. It allows to reason about properties of partner graph grammars.

2.1 Partner Graph Grammars

This section starts by introducing some multiset notation. Quite often, it will become necessary to reason about multisets in this thesis. Some preliminary notation for reasoning about graphs is introduced, before the crucial ingredient of any graph grammar is presented, transformation rules. Transformation rules are initially given in a simple form and are afterwards augmented with special negative application conditions called partner constraints. A set of transformation rules together with an initial graph will constitute a partner graph grammar. The abstract graph semantics of a partner graph grammar will be defined to be the set of all graphs generated by a partner graph grammar. If this set is equipped with the direct derivation relation between graphs, a graph transition system is obtained. This section concludes by introducing some syntactic sugar on top of partner graph grammars. This includes shorthand notations for transformation rules and for partner constraints.

2.1.1 Multisets

Formally, multisets are sets equipped with a mapping from elements to natural numbers denoting multiplicities. Although multisets are not used with that meaning in this work, some similar notation is introduced to ease and clarify formalizations. The additional notation centers around the notion of *disjoint set union*. For sets M and M', $M \dot{\cup} M'$ denotes the disjoint union of these sets. It is obtained by implicitly renaming elements in the intersection of M and M' such that they become distinguishable. The formal details of the renaming are left unspecified. Double braces are used to enumerate the elements of such "multisets", *e.g.*, $\{\!\{1, 1, 2\}\!\}$ denotes a set of three elements. Two of these were 1 originally, but are implicitly renamed to be distinguishable, *e.g.*, by indexing, while their original identity is maintained. Other operators used in this context resemble the standard set operators equipped with an index m. Examples include $\in_m$, $\subseteq_m$, $\cap_m$, or $\wp_m$ and have the obvious meaning. For brevity, the term multiset will be used at some points to denote these special notions.

2.1.2 Graph Preliminaries

A few basic notions and notations for reasoning about finite, directed, node and edge labeled graphs – for simplicity mostly called graphs – are introduced to begin with. Only graphs over a finite set of node labels and a finite set of edge labels are considered in this work.

Definition 2.1.1 (Graphs) *Let $n \geq 1$, $\mathcal{N}$ a finite set of* node labels *and $\mathcal{E} = \{\beta_1, \ldots, \beta_n\}$ a finite set of* edge labels. *A* graph over $\mathcal{N}$ and $\mathcal{E}$ *is a $n+2$-tuple $(V, E^{\beta_1}, \ldots, E^{\beta_n}, \ell)$, where*

- *V is a finite set of* nodes
- *$E^{\beta_i} \subseteq V \times V$ is a set of* β_i-labeled edges *for each $1 \leq i \leq n$, and*
- *$\ell : V \to \mathcal{N}$ is a* node labeling,

The set of all graphs over $\mathcal{N}$ and $\mathcal{E}$ is written $\mathcal{G}(\mathcal{N}, \mathcal{E})$. For a given graph G the sets V_G, E^{β}_G, and the mapping ℓ_G denote G's set of nodes, set of β-labeled edges, and node labeling, respectively. Let W be a set. The set of all graphs G over $\mathcal{N}$ and $\mathcal{E}$, where $V_G \subseteq W$ is written $\mathcal{G}(\mathcal{N}, \mathcal{E}, W)$.

For brevity, the node labeling of a graph is omitted, if the set of node labels is a singleton. If not stated otherwise, $\mathcal{N}$ and $\mathcal{E}$ are assumed to be

Metavariables	Typical Domains	Description
G, H, P, Q	$\mathcal{G}(\mathcal{N}, \mathcal{E})$	graphs
u, v, w, x, y, z	V_G, V_H	nodes
e, (u, v)	E_G^β, E_H^β	β-labeled edges
ν, μ	$\mathcal{N}$	node labels
β, δ, ϵ	$\mathcal{E}$	edge labels
f, g, h, m	$V_G \to V_H$	morphisms

Tab. 2.1: Frequently used metavariables reasoning about graphs. All of them may occur indexed or primed.The sets $\mathcal{N}$ and $\mathcal{E}$ can be arbitrary.

arbitrary finite label sets in the following. Metavariables ranging over graphs, node and edge labels, as well as nodes and edges are summarized in Table 2.1.

Most of the abstractions that will be defined later rely on the notion of quotient graph building. Quotient graphs are defined with respect to an equivalence relation on the nodes of the underlying graph. The nodes of the quotient graph are then the equivalence classes corresponding to the equivalence relation. In order to have meaningful node labels in the quotient graph, only a certain class of equivalence relations is considered. Two equivalent nodes must have the same labels. This requirement is crucial for the well-definedness of the node labeling in Definition 2.1.2.

Definition 2.1.2 (Quotient Graphs) *Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph. If $\mathcal{R} \subseteq V \times V$ is an equivalence relation, and $u\ \mathcal{R}\ v$ implies $\ell_G(u) = \ell_G(v)$, then the set of nodes equivalent to u is called the* equivalence class of *u wrt. $\mathcal{R}$ – written $[u]_\mathcal{R}$. The graph H with*

- $V_H = \{[v]_\mathcal{R} \mid v \in V_G\}$,
- $E_H^\beta = \{([v]_\mathcal{R}, [v']_\mathcal{R}) \mid (v, v') \in E_G^\beta\}$ *for each $\beta \in \mathcal{E}$, and*
- $\ell_H = \lambda[v]_\mathcal{R}.\ell_G(v)$

is called the quotient graph of G with respect to $\mathcal{R}$ *and written $G/\mathcal{R}$.*

Consider Figure 2.1. The graph P in (a) displays an invalid (because of two adjacent red nodes) red-black tree. Formally, the sets of node and edge labels of P are $\{red, bl\}$ and $\{left, right\}$, respectively, representing a pointer to the left (right) child of a node. Furthermore,

$$V_P = \{u_1, u_2, u_3, u_4, u_5\}$$
$$E_P^{left} = \{(u_1, u_2), (u_2, u_4)\} \quad E_P^{right} = \{(u_1, u_3), (u_2, u_5)\}$$
$$\ell_P = [u_1 \mapsto red, u_2 \mapsto red, u_3 \mapsto bl, u_4 \mapsto bl, u_5 \mapsto bl]$$

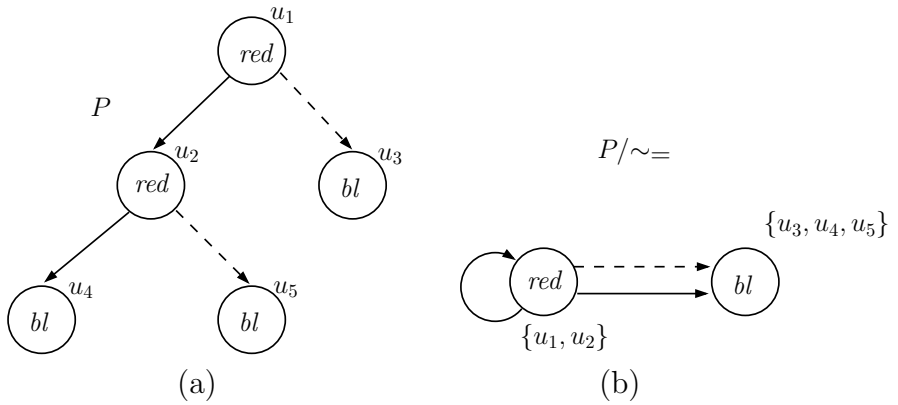


Fig. 2.1: A graph and its quotient graph. Nodes are drawn as circles with their labels inside. The graph P in (a) displays an invalid red-black tree and features two node labels, *red* and *bl*, and two edge labels, *left* (represented as solid edges) and *right* (dashed). In figures, edge labels are mostly distinguished by line styles. Node identities are written next to circles. The quotient graph in (b) is built *wrt.* the equivalence relation $\sim_=$ containing exactly the pairs of equally labeled nodes.

In part (b), the quotient graph $P/\sim_=$ is shown, where $u \sim_= v \Leftrightarrow \ell_P(u) = \ell_P(v)$. Note that nodes are equivalence classes, *i.e.* identified sets of elements of V_P.

The notion of connected graphs to be defined next is standard. It is worth noting, however, that Definition 2.1.3 defines connectedness regardless of the direction of edges – as opposed to strong connectedness.

Definition 2.1.3 (Connected Graphs) *Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph. Two nodes $u, v \in V_G$ are* connected*, written $u \sim_c v$, if there exist $u_1, \ldots, u_n \in V_G$ such that $u = u_1$, $v = u_n$, and for all $1 \leq i < n$ exists a $\beta_i \in \mathcal{E}$ such that $(u_i, u_{i+1}) \in E_G^{\beta_i}$ or $(u_{i+1}, u_i) \in E_G^{\beta_i}$.*

The graph G is connected*, if all its nodes are pairwise connected,* i.e. *if $|V_{G/\sim_c}| = 1$.*

The set $\mathcal{G}_c(\mathcal{N}, \mathcal{E}) \subset \mathcal{G}(\mathcal{N}, \mathcal{E})$ is the set of all connected graphs *over $\mathcal{N}$ and $\mathcal{E}$.*

Morphisms are label- and structure-preserving mappings between the node sets of graphs. They are crucial for defining graph grammars. Injective morphisms are used to define the subgraph relation.

Definition 2.1.4 (Graph Morphism) *Let $G, H \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be graphs. A mapping $h : V_G \to V_H$ is a* morphism from G to H*, iff the following conditions*

hold for all $\beta \in \mathcal{E}$

$$\forall w \in V_G.\ell_G(w) = \ell_H(h(w)) \tag{2.1}$$
$$\{(h(u), h(v)) \mid (u, v) \in E_G^\beta\} \subseteq E_H^\beta \tag{2.2}$$

Apart from the subgraph relation, Definition 2.1.5 provides notions for induced subgraphs and the set of connected components of a graph.

Definition 2.1.5 (Subgraphs) *Let* $G, H \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ *be graphs.*

1. *The graph* G *is a* subgraph *of* H*, written* $G \leq H$*, iff there exists an injective morphism from* G *to* H*.*

2. G *and* H *are* isomorphic*, written* $G \cong H$ *if there exists a bijective morphism from* G *to* H*.*

3. *Let* $V \subseteq V_G$*. The subgraph of* G induced by V*, written* $G\,|_V$ *is the graph* $(V, E^{\beta_1}, \ldots, E^{\beta_n}, \ell)$*, where* $\ell = \ell_G\,|_V$ *and* $E^\beta = E_G^\beta \cap (V \times V)$ *for all* $\beta \in \mathcal{E}$*.*

4. *A subgraph of* G *induced by an element of* $V_{G/\sim_c}$ *is called a* connected component *of* G*. The set of all connected components of* G *is written* $cc(G)$*.*

2.1.3 Transformation Rules

Having introduced basic graph notions and notations, a novel specification formalism for evolving graphs is presented: *partner graph grammars*. As before, if not stated otherwise, $\mathcal{N}$ and $\mathcal{E}$ are assumed to be arbitrary finite sets of node and edge labels, respectively. The first ingredient of a partner graph grammar is a *graph transformation rule*.

First, *simple* rules are presented. They will be augmented with *partner constraints* (as explained in Chapter 1) in Section 2.1.4. Simple rules are triples of the form (L, h, R), where L and R are graphs and where h is an injective mapping – not necessarily a morphism – from the nodes of L to the nodes of R. The intended meaning of a rule (L, h, R) is to transform a graph G into a graph H as follows. The meaning deviates slightly from other notions of graph transformation rules.

- The rule *matches* G, iff L is a subgraph of G due to injective morphism m.

- The mapping h relates the nodes in the left and in the right graph. Informally speaking, it defines anchor nodes of a transformation rule. Nodes in the left graph that are not in the domain of h will be removed by a rule application whereas nodes in the right graph of a rule that are not in the codomain of h are newly created by the rule application.

- All edges specified in the left graph are removed, and all edges in the right graph are added.

- A node $v \in V_G$ may change its label, if there exists a node $v' \in V_L$ such that $m(v') = v$, and v' and $h(v')$ have different labels.

Figure 2.2 gives an example of a rule and an application of this rule. The formal definition of simple transformation rules is given in Definition 2.1.6. The $\rightharpoonup$ arrow denotes partial mappings. Given a partial mapping $f : A \rightharpoonup B$, $dom(f)$ and $codom(f)$ refer to the domain and the codomain of f, whereas the overlined versions $\overline{dom(f)}$ and $\overline{codom(f)}$ stand for $A \setminus dom(f)$ and $B \setminus codom(f)$, respectively. Application of a mapping $f : A \to B$ to a subset $M \subseteq A$ of its domain is defined pointwise: $f(M) = \{f(a) \mid a \in M\}$.

Definition 2.1.6 (Simple Transformation Rules)

1. *A triple (L, h, R), where $L, R \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ and $h : V_L \rightharpoonup V_R$ is injective, is called a* simple transformation rule.

2. *A simple transformation rule (L, h, R)* matches *a graph $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$, iff $L \leq G$ due to morphism m, which is called a* match.

3. *If $r = (L, h, R)$ matches G due to match m, the result of the application of r to G is the graph H, where*

   - $V_H = (V_G \setminus m(\overline{dom(h)})) \dot\cup \overline{codom(h)}$
   - *For each $\beta \in \mathcal{E}$*

$$\begin{aligned} E_H^\beta \;=\; & (E_G^\beta \cap (V_H \times V_H)) \setminus \\ & \{(m(u), m(v)) \mid (u, v) \in E_L^\beta\} \cup \\ & \{(m'(u), m'(v)) \mid (u, v) \in E_R^\beta\} \end{aligned}$$

   where $m' : V_R \to V_H$ is defined to be

$$m'(v) = \begin{cases} v & \text{if } v \in \overline{codom(h)} \\ m(h^{-1}(v)) & \text{if } v \in codom(h) \end{cases}$$

- $\ell_H = \lambda v. \begin{cases} \ell_G(v) & \text{if } v \in \overline{codom(m)} \\ \ell_R(h(m^{-1}(v))) & \text{if } v \in codom(m) \\ \ell_R(v) & \text{if } v \in \overline{codom(h)} \end{cases}$

If H is obtained by applying r to G, it is said to be a direct derivation *of G, written $G \rightsquigarrow_r H$.*

The first remark concerning Definition 2.1.6 is about well-definedness, which is not obvious when inverse mappings are used. It is guaranteed, however, by the injectivity requirements imposed on both the match m and the mapping h relating the left and the right graph of a rule. In detail, Definition 2.1.6 reads as follows.

- The set of nodes of graph H resulting from applying r to G consists of those nodes that are not hit by the match and those maintained by the rule application. In contrast, the nodes of H corresponding to nodes not in the domain of h disappear. Newly created nodes are those that are not in the codomain of h. They are disjointly added. Formally, $\dot\cup$ creates a multiset, if a node in V_R happens to have the same identity as a node in G. However, by using notation in a sloppy way, some renaming is assumed and the multiset is treated as a normal set.

- The edges of H are first of all restricted to be between nodes of H. This results in a removal of all edges adjacent to a disappearing node. The second clause in the definition of E_H removes all edges specified in the left graph. Eventually, all edges specified in the right graph are added. Certainly, edges specified in the rule r are related to G using match m.

- As for the node labeling of the resulting graph, there are three cases to be distinguished. A node not in the codomain of the match keeps its label. Nodes in the codomain of m receive the label specified in the right graph by inverting the match and applying h. Finally, newly created nodes simply get their label as given in the right graph of the rule.

Consider the simple transformation rule [LIST] and its application in Figure 2.2 (b,c). Graph H in (c) is computed by Definition 2.1.6 as follows, where G and [LIST] $= (L', h', R')$ are defined by the drawing. All involved graphs are over the set $\{x, \nu, \llcorner\}$ of node labels and the set $\{\beta_1, \beta_2\}$ of edge labels. The node label $\llcorner$ is not drawn in Figure 2.2, β_1-labeled edges are drawn as dashed arrows and β_2-labeled edges are drawn as solid edges.

- $h' = [u'_1 \mapsto v'_1, u'_2 \mapsto v'_2]$
- The match m justifying $L' \leq G$ is given by $m = [u'_1 \mapsto w_1, u'_2 \mapsto w_2, u'_3 \mapsto w_3]$
- Plugging in the definition of m' yields $m' = [v'_1 \mapsto w_1, v'_2 \mapsto w_2, v'_3 \mapsto v'_3]$
- Some computations on domains and codomains of the above mappings follow. After that, the four ingredients of H are computed in detail.

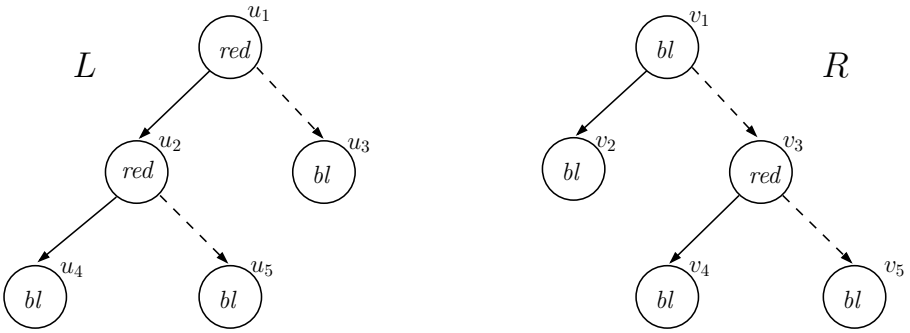
$$\begin{array}{rclcrcl} \overline{dom(h')} & = & \{u'_1, u'_2\} & & \overline{codom(h')} & = & \{v'_1, v'_2\} \\ \overline{dom(h')} & = & \{u'_3\} & & \overline{codom(h')} & = & \{v'_3\} \\ codom(m) & = & \{w_1, w_2, w_3\} & & \overline{codom(m)} & = & \{w_4\} \end{array}$$

$$\begin{array}{rcl} V_H & = & \{w_1, w_2, w_3, w_4\} \setminus m(\{u'_3\}) \dot\cup \{v'_3\} \\ & = & \{w_1, w_2, w_3, w_4\} \setminus \{w_3\} \dot\cup \{v'_3\} \\ & = & \{w_1, w_2, v'_3, w_4\} \end{array}$$

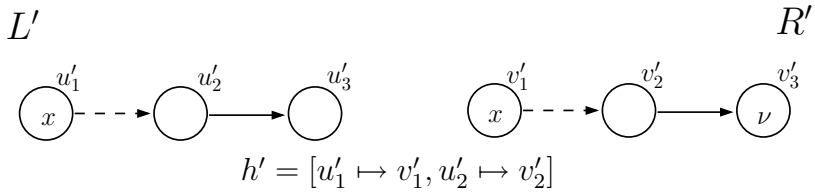
$$\begin{array}{rcl} E_H^{\beta_1} & = & (\{(w_1, w_2)\} \cap V_H \times V_H) \setminus \{(m(u'_1), m(u'_2)\} \cup \{(m'(v'_1), m'(v'_2))\} \\ & = & \{(w_1, w_2)\} \setminus \{(w_1, w_2)\} \cup \{(w_1, w_2)\} \\ & = & \{(w_1, w_2)\} \end{array}$$

$$\begin{array}{rcl} E_H^{\beta_2} & = & (\{(w_2, w_3), (w_3, w_4)\} \cap V_H \times V_H) \setminus \{(m(u'_2), m(u'_3))\} \cup \\ & & \{(m'(v'_2), m'(v'_3))\} \\ & = & \emptyset \setminus \{(w_2, w_3)\} \cup \{(w_2, v'_3)\} \\ & = & \{(w_2, v'_3)\} \end{array}$$

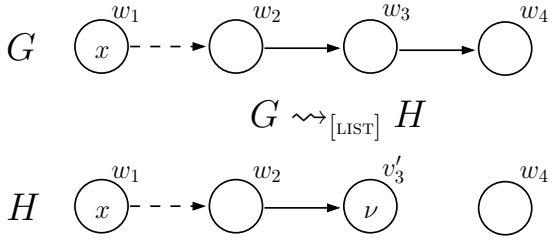
$$\begin{array}{rcl} \ell_H & = & [w_1 \mapsto \ell_{R'}(h'(m^{-1}(w_1))), w_2 \mapsto \ell_{R'}(h'(m^{-1}(w_2))), \\ & & v'_3 \mapsto \ell_{R'}(v'_3), w_4 \mapsto \ell_G(w_4)] \\ & = & [w_1 \mapsto x, w_2 \mapsto \lrcorner, v'_3 \mapsto \nu, w_4 \mapsto \lrcorner] \end{array}$$



(a) Transformation rule [ROTATE] = (L, h, R)



(b) Transformation rule [LIST] = (L', h', R')



(c) A sample application of the [LIST] rule

Fig. 2.2: Two examples of simple transformation rules: Rule [ROTATE] represents an implementation of a tree-balancing right rotation. This rule matches graph P of Figure 2.1. Rule [LIST] in (b) shows the effect of the C statement `free(x->n); x->n = malloc();`. Dashed arrows represent pointers from the stack into the heap, solid arrows stand for the next pointer `n` of a list. Node label ν is used for newly created heap cells.

2.1.4 Partner Constraints

Simple transformation rules provide a powerful specification mechanism for evolving graphs. However, for some cases, they are not quite expressive enough. Using simple transformation rules, it is not possible to express *negative application conditions*. In many situations, it is useful to reason when a rule shall not be applied. For example, consider the simple transformation rule [LIST] and its application in Figure 2.2. In this case, garbage is created by the unconstrained removal of node w_3. This rule would be more sensible, if its application was restricted to cases, where u'_3 matches the last node of a list only. *Partner constraints* are a way to express such conditions. In order to define them formally, the notion of *finite counting* is introduced.

Definition 2.1.7 (Finite Counting) *Let $\mathbb{N}$ be the set of natural numbers excluding 0. For any $k \in \mathbb{N}$ the set $\mathbb{N}_k$ is the set of all natural numbers smaller than or equal to k augmented with ∞,* i.e. $\mathbb{N}_k := \{1, 2, \ldots, k, \infty\}$.

On $\mathbb{N}_k$, the order $\sqsubseteq_k$ is defined by $n \sqsubseteq_k n'$, if and only if $n' = \infty$ or $n \leq n' \leq k$.

The binary operation $\oplus^k$ is defined on $\mathbb{N}_k$ as follows:

$$n \oplus^k n' := \begin{cases} \infty & \text{if } n = \infty,\ n' = \infty,\ \text{or } n + n' > k \\ n + n' & \text{otherwise} \end{cases}$$

For any $n \in \mathbb{N}$, $(n)_k$ is defined to be n, if $n \leq k$, and ∞ otherwise

Requirements on Adjacent Nodes Formally, a partner constraint is a subset of $\{in, out\} \times \mathcal{E} \times \mathcal{N} \times \mathbb{N}_k$. A partner constraint may be associated with a node in the left graph of a transformation rule. Assume (in, β, ν, n) is an element of a partner constraint associated with a node u in a left graph of a rule. Any node v that u matches must have *at least one* and *at most n* ν-labeled nodes connected to it by an incoming β-labeled edge. In this case, v is said to satisfy the four-tuple (in, β, ν, n).

If a partner constraint *pc* associated with a node u has more than one element, a node v matched by u must satisfy *exactly* all those elements. In particular, all nodes adjacent to v must correspond to one of the elements of the partner constraint. In this case, v is said to satisfy the partner constraint. A special instance of a partner constraint is the $\emptyset$ constraint. A node associated with this constraint can only match nodes that do not have adjacent nodes.

Partner constraints must be *consistent* with the left graph, in which they occur. For example, an $\emptyset$ constraint cannot be associated with a node u in a graph, where u already has an adjacent node. Another source of inconsistency

may be a graph, where u already has more adjacent nodes of the same kind than specified by a partner constraint.

Definition 2.1.8 (Partner Constraints) *Let $k \geq 1$. A* partner constraint *over the node labels $\mathcal{N}$ and the edge labels $\mathcal{E}$ is a subset of the set $\mathcal{PC}(\mathcal{N}, \mathcal{E}, k) := \{in, out\} \times \mathcal{E} \times \mathcal{N} \times \mathbb{N}_k$. Furthermore, for each $pc \in \mathcal{PC}(\mathcal{N}, \mathcal{E}, k)$, the notion $pc \searrow 3 := \{(io, \beta, \nu) \mid \exists n \in \mathbb{N}_k.(io, \beta, \nu, n) \in p\}$ is used.*

Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph, $u \in V_G$ a node of G, and $p \in \mathcal{PC}(\mathcal{N}, \mathcal{E}, k)$ a partner constraint. Node u satisfies *p* in *G, written $G, u \models p$ iff*

$$\begin{aligned} p \searrow 3 \;\; = \;\; & \{(in, \beta, \nu) \mid \exists v \in V_G.(v, u) \in E^\beta \wedge \ell_G(v) = \nu\} \;\cup \\ & \{(out, \beta, \nu) \mid \exists v \in V_G.(u, v) \in E^\beta \wedge \ell_G(v) = \nu\} \end{aligned}$$

and for each $(in, \beta, \nu, n) \in p$

$$(|\,\{(v, u) \in E^\beta \wedge \ell_G(v) = \nu\}\,|)_k \quad \sqsubseteq_k \quad n,$$

where the case for out is analogous. Let $L \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph, $u \in V_L$ be a node and $p \in \mathcal{PC}(\mathcal{N}, \mathcal{E}, k)$ be a partner constraint. Partner constraint p is consistent with L and u, if there exists a graph $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ such that L matches G by match m and $G, m(u) \models p$.

A partner constraint p is called simple*, if all "at most" components are ∞,* i.e. *if $p \subseteq \{in, out\} \times \mathcal{E} \times \mathcal{N} \times \{\infty\}$.*

The degree *of a partner constraint pc is its maximal non ∞ "at most" component,* i.e. $\max\{n \mid (io, \beta, \nu, n) \in pc, n \neq \infty\}$. *The degree of a simple partner constraint is defined to be 0.*

The definition of partner constraints does not forbid constraints with several element tuples only differing in the fourth, *i.e.*, the "at most", component. In this case, for a node of a graph to satisfy such a constraint, the most restrictive of these tuples must be satisfied. The most restrictive case is the one with the smallest "at most" component. The definition of a match is extended in a way to incorporate partner constraints. This is the only change to the definition of rule applications from Definition 2.1.6

Definition 2.1.9 (Transformation Rules) *Let $k \geq 1$.*

1. *A four-tuple $r = (L, h, p, R)$, where $L, R \in \mathcal{G}(\mathcal{N}, \mathcal{E})$, $h : V_L \to V_R$ is injective, and $p : V_L \rightharpoonup \mathcal{PC}(\mathcal{N}, \mathcal{E}, k)$ is a partial mapping, is called a* transformation rule*, if $p(u)$ is consistent with L and u for each $u \in dom(p)$.*

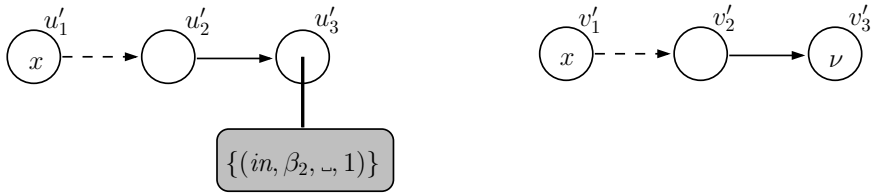


Fig. 2.3: Left and right graph of the transformation rule [LIST]′. This rule equals [LIST] up to the partner constraint $\{(in, \beta_2, \lrcorner, 1)\}$ attached to node u'_3. Partner constraints are depicted within lightly shaded boxes and connected to their associated node by a thick line to the center of the node. Due to the partner constraint [LIST]′ cannot be applied to graph G in Figure 2.2 any more.

2. *A transformation rule* $r = (L, h, p, R)$ matches *a graph* $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$, *if* $L \leq G$ *due to morphism* m *and* $G, m(u) \models p(u)$ *for all* $u \in dom(p)$. *In this case,* m *is called a* match.

3. *If* $r = (L, h, p, R)$ *matches* G, $r' = (L, h, R)$, *and* $G \rightsquigarrow_{r'} H$, *then* H *is called the* result of the application *of* r *to* G *or the* direct derivation *of* G *using* r. *Again, the notation* $G \rightsquigarrow_r H$ *is used.*

Figure 2.3 shows the general transformation rule [LIST]′, which is a slight extension of the simple rule [LIST] from Figure 2.2. Node u'_3 in this rule is equipped with the partner constraint $\{in, \beta_2, \lrcorner, 1\}$ enforcing every node w matching v'_3 to have exactly one adjacent node. This node must be connected to w by an incoming (seen from w) β_2 labeled edge and must itself be labeled $\lrcorner$. In particular, [LIST]′ cannot be applied to the graph G from Figure 2.2 on page 15. Moreover, the partner constraint ensures that no garbage can be created by an application of [LIST]′.

2.1.5 Partner Graph Grammars

So far, transformation rules have been defined and can be equipped with partner constraints for better usability. The latter restrict the applicability of transformation rules. Applying a transformation rule to a graph transforms this graph into another one.

In this section, sets of transformation rules are combined with an *initial graph* to form a *partner graph grammar*. A *graph transition system* is defined by taking the initial graph as the initial state and the application of one of the rules as transition between graphs. In other words, partner graph grammars are the syntax of a specification formalism, whose semantics are

graph transition systems. As usual, the reflexive, transitive closure of a relation $\mathcal{S}$ is written $\mathcal{S}^*$.

Definition 2.1.10 (Partner Graph Grammars) *Let $k \geq 1$, let $\mathcal{R}$ be a set of transformation rules over $\mathcal{N}$ and $\mathcal{E}$ and let $\mathcal{I} \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph. A pair $\mathfrak{G} = (\mathcal{R}, \mathcal{I})$ is called a* partner graph grammar.

Two graphs $G, H \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ are in a direct derivation relation $\rightsquigarrow_{\mathcal{R}}$*, if there exist a rule $r \in \mathcal{R}$ such that $G \rightsquigarrow_r H$.*

The graph semantics *of the partner graph grammar $\mathfrak{G}$ is the set $[\![\mathfrak{G}]\!] := \{G \in \mathcal{G}(\mathcal{N}, \mathcal{E}) \mid \mathcal{I} \rightsquigarrow_{\mathcal{R}}^* G\}$. The* graph transformation system induced by $\mathfrak{G}$ *is the pair $([\![\mathfrak{G}]\!], \rightsquigarrow_{\mathcal{R}})$.*

Verification of properties of such systems is difficult, because there is no a priori known bound on either

- the number of graphs in the graph semantics or
- the size of the individual graphs.

A Checkers Example The following is quoted from the free encyclopedia Wikipedia: "Checkers is played by two people, on opposite sides of a 16 by 16 playing board, alternating moves. One player has dark pieces, and the other has light pieces. The player with the light pieces makes the first move unless stated otherwise. Pieces move diagonally and pieces of the opponent are captured by jumping over them. The playable surface consists only of the dark squares. Capturing is mandatory. A piece that is captured is removed from the board. The player who has no pieces left or cannot move anymore has lost the game unless otherwise stated.

Uncrowned pieces ("men") move one step diagonally forwards and capture other pieces by making two steps in the same direction, jumping over the opponent's piece on the intermediate square. Multiple opposing pieces may be captured in a single turn provided this is done by successive jumps made by a single piece.

When men reach the kings' row – the farthest row forward – they become kings, marked by placing an additional piece on top of the first, and acquire additional powers including the ability to move backwards and capture backwards."

Checkers can be formalized beautifully as the partner graph grammar $\mathfrak{G}_C(\{[\textsc{MoveR}], [\textsc{CaptureR}], [\textsc{KingR}], \ldots\}, \textsc{Checkers})$. All components are given in Figure 2.4. Graphs in $\mathfrak{G}_C$ use node labels $\{w, ww, b, bb, \llcorner\}$ and edge labels $\{up, right\}$. Node labels model light men, light kings, dark men, dark kings, and unoccupied squares, respectively. Different square colors are

not modeled. Men are depicted with a thin rim, whereas kings get a thicker rim. Unoccupied squares are shown in white, occupied squares are shown in either gray or black corresponding to light and dark pieces. Edges are used to define the geometry of the board.

Rules [MoveR] and [CaptureR] cater for moving and capturing light men to the right. The remaining rule [KingR] is equipped with a partner constraint. Informally, this partner constraint ensures, that the row reached by the light man is the king row. This is enforced by saying that there is no outgoing *up* edge from node u_3. A shorthand notation is used to write down this partner constraint. All shorthands used here are explained in detail in Section 2.1.6.

All transformation rules have the implicit assumption that the mapping from the left to the right graph is of the form $u_i \mapsto u_i'$. More rules not stated here handle the many symmetric cases dealing with black men, kings, multiple jumps or the requirement of alternating moves from the light and the dark player. Some of the moves are easy to add, whereas cases like forced jumps are quite intricate to formalize in terms of transformation rules.

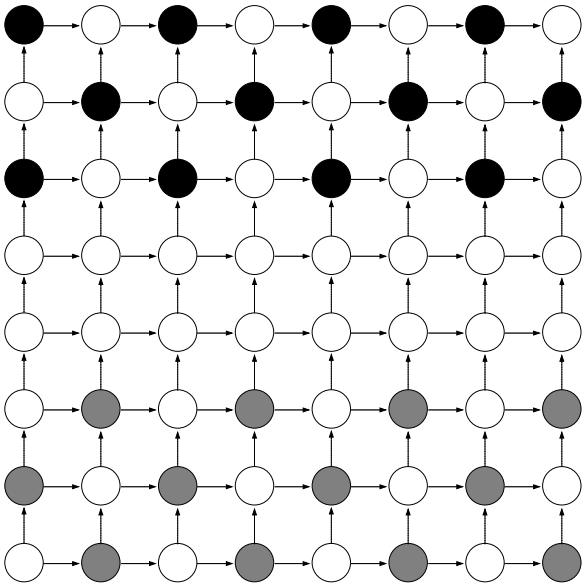
The graph semantics of $\mathfrak{G}_C$ is the set of all possible positions that may be reached from the starting position. The induced graph transformation system describes all possible games that can be played.

2.1.6 Special Cases and Shorthands

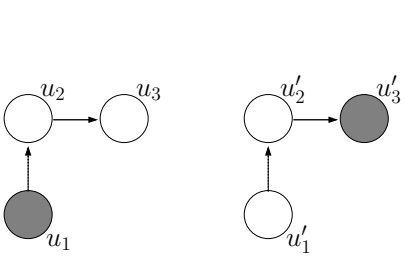
Writing down specifications using partner graph grammars may be tedious. In this section, some syntactic sugar both for writing down rules and for writing down partner constraints is thus introduced.

Label Variables Using *label variables* it is possible to write down an *augmented* transformation rule that stands for a finite set of transformation rules. A label variable is used as a special label in the left and/or right graph of a transformation rule. It can be bound to an arbitrary concrete label in a match. Several occurrences of the same label variable in one graph are possible.

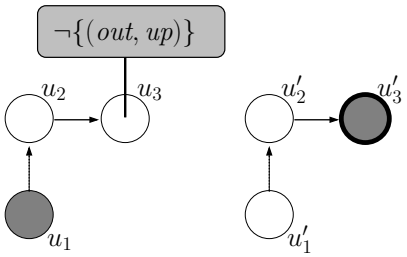
Also, it is possible to annotate a transformation rule with simple constraints on label variables, *label constraints*. A label constraint may require that a label variable can only be matched to a certain set of labels. Additionally, boolean combinations of these label constraints are allowed. Two examples are shown in Figure 2.5. In figures, label variables are written in bold mathematical font, whereas label constraints annotate rules and are drawn inside of boxes with thicker rims. Note that, in general, the usage of label variables is not restricted to node labels, even though it may appear



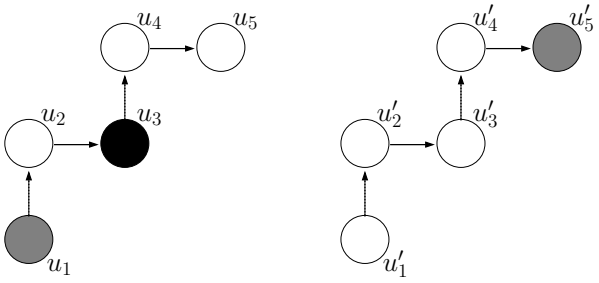
(a) The initial graph CHECKERS



(b) The rule [MOVER]

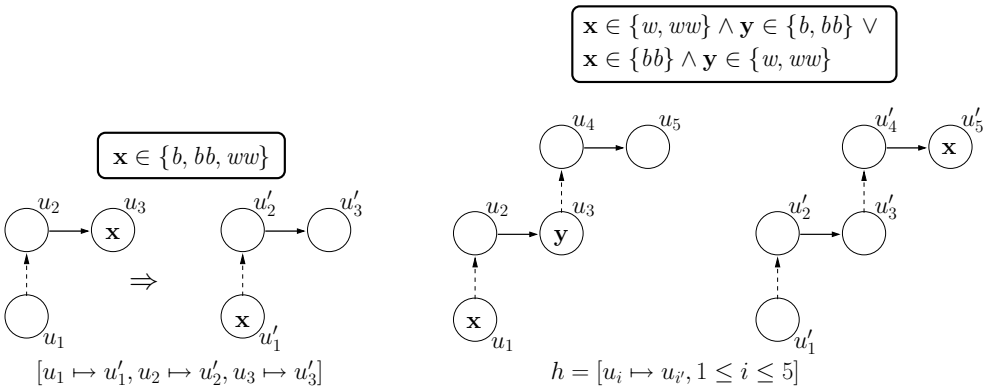


(c) The rule [KINGR]



(d) The rule [CAPTURER]

Fig. 2.4: Part of the partner graph grammar formalization of a game of checkers, *c.f.* the example in Section 2.1.5.



(a) Rule [Move] (b) Rule [Capture]

Fig. 2.5: Rule [Move] is a shorthand for three rules moving a piece backward and left. This piece can either be a dark man, a dark king, or a light king, as indicated by the label constraint on top. Label constraints for rules are shown in boxes. Bold font is used for label variables. Rule [Capture] models the forward right capturing of a piece. Either a dark piece can be captured by a light piece or a light piece by a dark king. A dark king is the only dark piece capable of moving forward.

like this in Figure 2.5. The formal definition in Definition 2.1.11 allows for labeling edges with label variables, too.

It would be possible to give a semantic definition to label variables and label constraints. However, this requires a re-definition of all the concepts concerning transformation rules and partner graph grammars. Rather, the other option is chosen, namely, to translate augmented transformation rules *syntactically* to a finite set of standard rules.

In the course of this translation, an *instantiation relation* between augmented and standard rules is defined, relating standard rules with the augmented ones of which they are an instance. To define the notion of an instance, an *assignment mapping* of label variables to labels is introduced. It is then checked whether this assignment satisfies the label constraints, if any. If this is the case, this assignment defines a valid instance of the augmented rule. Here is the formal definition of augmented transformation rules and their denotation as finite sets of standard transformation rules.

Definition 2.1.11 (Label Variables) *Let* $\mathbf{X}$ *be a finite set of* label variables. Label constraints LC *are defined by the following BNF, where* $\mathbf{x} \in \mathbf{X}$ *and* $M \subseteq \mathcal{N} \cup \mathcal{E}$

$$LC ::= \varepsilon \mid \mathbf{x} \in M \mid \neg LC \mid LC \wedge LC \mid LC \vee LC$$

An augmented transformation rule over *node labels* $\mathcal{N}$ *and edge labels* $\mathcal{E}$ *is a five-tuple* (L, h, p, R, lc)*, where* (L, h, p, R) *is a transformation rule with* $L, R \in \mathcal{G}(\mathcal{N} \dot\cup \mathbf{X}, \mathcal{E} \dot\cup \mathbf{X})$*, and where* $lc \in LC$*.*

A graph $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ *is an* instance *of a graph* $H \in \mathcal{G}(\mathcal{N} \dot\cup \mathbf{X}, \mathcal{E} \dot\cup \mathbf{X})$*, if and only if there exists an* assignment $\sigma : \mathbf{X} \to \mathcal{N} \cup \mathcal{E}$ *such that*

- $V_G = V_H$
- $E_G^\beta = E_H^\beta \cup \{e \in E_H^{\mathbf{x}} \mid \sigma(\mathbf{x}) = \beta\}$ *for all* $\beta \in \mathcal{E}$*.*
- $\ell_G(v) = \begin{cases} \ell_H(v) & \text{if } \ell_H(v) \in \mathcal{N} \\ \sigma(\ell_H(v)) & \text{otherwise} \end{cases}$

In this case, the notation $G \; \mathcal{R}_\sigma \; H$ *is used.*

An assignment $\sigma : \mathbf{X} \to \mathcal{N} \cup \mathcal{E}$ *is defined to* satisfy a label constraints *lc, written* $\sigma \models lc$*, by*

σ	$\models$	ε	*always*	
σ	$\models$	$\mathbf{x} \in M$	*if and only if*	$\sigma(\mathbf{x}) \in M$
σ	$\models$	$\neg lc$	*if and only if not*	$\sigma \models lc$
σ	$\models$	$lc_1 \wedge lc_2$	*if and only if*	$\sigma \models lc_1$ *and* $\sigma \models lc_2$
σ	$\models$	$lc_1 \vee lc_2$	*if and only if*	$\sigma \models lc_1$ *or* $\sigma \models lc_2$

A transformation rule $r' = (L', h', p', R')$ *is an* instance *of the augmented rule* $r = (L, h, p, R, lc)$*, iff there exists an assignment* σ *such that* $L' \; \mathcal{R}_\sigma \; L$*,* $R' \; \mathcal{R}_\sigma \; R$*, and* $\sigma \models lc$*.*

Later on, augmented rules are used wherever suitable without further notice. An augmented rule is implicitly identified with the set of its instantiations. In formal reasonings, however, it suffices to consider standard rules, because the augmentation with label constraints does not generate more expressive power (but more convenience).

Consider the transformation rule [CaptureR] in Figure 2.4 on page 21. It is an instance of the more general, augmented rule [Capture] given in Figure 2.5 on page 22. More formally, [CaptureR] $\mathcal{R}_\sigma$ [Capture] holds for the assignment $\sigma = [\mathbf{x} \mapsto w, \mathbf{y} \mapsto b]$ over $\mathbf{X} = \{\mathbf{x}, \mathbf{y}\}$. It is easy to comprehend that σ indeed satisfies the label constraint specified for [Capture].

Partner Constraint Shorthands The same trick that was applied to transformation rules will be applied to partner constraints again. *Augmented partner constraints* allow to conveniently write down a finite set of rules that merely differ with respect to partner constraints. Formally and similar to Definition 2.1.11, *instances* of augmented partner constraints are defined

and rules featuring augmented partner constraints are identified with the set of all their instances.

Remember that partner constraints are sets of four-tuples, where each four-tuple is an element of the set $\{in, out\} \times \mathcal{E} \times \mathcal{N} \times \mathbb{N}_k$. For example, a tuple $(in, \beta, \nu, 3)$ requires a node to have at least one and at most three ν-labeled adjacent nodes, to which it must be connected by incoming β-labeled edges. As a kind of syntactic sugar, it is allowed to leave out components of a four-tuple. Assume one of the following tuples is an element of a partner constraints associated with a node that is matched by a node u.

1. The tuple $(\beta, \nu, 3)$ requires u to have between one and three adjacent ν-labeled nodes to which it is connected by β-labeled edges — *regardless of their direction.*

2. The tuple $(in, \nu, 3)$ requires u to have between one and three ν-labeled adjacent nodes, to which it is connected by incoming edges — *regardless of the edge labels.*

3. The tuple $(in, \beta, 3)$ requires u to have between one and three adjacent nodes — *regardless of their labels* — to which it is connected by incoming β-labeled edges.

4. As pointed out earlier, the tuple (in, β, ν) is a shorthand for the tuple (in, β, ν, ∞).

Consider the first case above, *i.e.* a tuple (β, ν, n), where $n \in \mathbb{N}_k$ for some $k > 0$. First, assume $n \neq \infty$. In this case, a set $\{(io_1, \beta, \nu, n_1), \ldots, (io_q, \beta, \nu, n_q)\}$ is an instance of the the tuple (β, ν, n), if and only if $io_i \in \{in, out\}$ for all $1 \leq i \leq q$ and $n_1 \oplus^k \ldots \oplus^k n_1 = n$. If $n = \infty$, an instance is an arbitrary subset of $\{in, out\} \times \{\beta\} \times \{\nu\} \times \{\infty\}$. Analogously, instances are defined for the cases 2 and 3 above. The left out component can be chosen arbitrarily, if the arithmetic requirement is fulfilled. As for case 4, the only instance of a tuple (io, β, ν) is the set $\{(io, \beta, \nu, \infty)\}$. If a partner constraint with left out components consists of more than one tuple, any union of instances of the element tuples is an instance of this partner constraint. A transformation rule sporting partner constraints with left out components is then identified with the finite set of transformation rules, where partner constraints with left out components are replaced by any of their instances.

The second shorthand is the *negation* of a partner constraint pc, written $\neg pc$. An instance of $\neg pc$ may be an arbitrary subset of $\mathcal{PC}(\mathcal{N}, \mathcal{E}, k) \setminus p$. The

intuition behind this is, to allow for any combination of adjacent nodes *except* for those specified by p. Due to the subset construction, this sort of negated partner constraint may imply a large, but finite number of instances.

If a partner constraint, which already features left-out components, is negated, the final set of instances is obtained by first computing all the instances of the partner constraint with left-out components, and then by negating all these instances.

Certainly, it is possible to combine label constraints and augmented partner constraints, because, in effect, they are simply shorthand notations. An example of such a combination was given in rule [KingR] in Figure 2.4. The instances of the partner constraint under negation, *i.e.* of the partner constraint $\{(out, up\}$ are five singleton sets. Each set's unique element is an element of $\{out\} \times \{up\} \times \{b, bb, w, ww, \lrcorner\} \times \{\infty\}$. The negation is thus the set of all subsets of

$$(\{in\} \times \{up\} \cup \{in, out\} \times \{right\}) \times \{b, bb, w, ww, \lrcorner\} \times \mathbb{N}_k$$

for some $k > 0$.

2.2 $\mathcal{GL}$: Reasoning about Graph Grammars

The logic $\mathcal{GL}$ is designed to express properties of partner graph grammars, more precisely, properties of graph transition systems induced by graph grammars. It is based on the computation tree logic CTL and shares the temporal constructs of it. Whereas CTL features a next-operator, it is left out from $\mathcal{GL}$ (see final paragraph of this section). On the other hand, $\mathcal{GL}$ must be able to reason about graphs of a statically unknown size. First-order constructs are incorporated to facilitate this. However, unlike ETL [YRSW03] or ATL [Dis03], $\mathcal{GL}$ is unable to track node evolution over time, *i.e.*, first order quantification does not mix up with temporal quantification. Finally, temporal operators may be restricted by subsets of rules. Only the specified graph transformation rules may be applied in the scope of the operator for a formula to become true. The upcoming paragraph describes and defines the syntax of $\mathcal{GL}$ before giving some illustrating examples.

Syntax The logic $\mathcal{GL}$ is parameterized by syntactic categories. All of them are finite sets. First, there is the domain **LVar** of logical variables with metavariables $x, y, z, \ldots$ ranging over it. Let $(\mathcal{R}, \mathcal{I})$ be a partner graph grammar over the set $\mathcal{N}$ of node labels and the set $\mathcal{E}$ of edge labels. The domains $\mathcal{N}$, $\mathcal{E}$, and $\mathcal{R}$ serve as syntactic categories for $\mathcal{GL}$. Consider Table 2.1 for metavariables. Additionally, R ranges over subsets of $\mathcal{R}$.

Definition 2.2.1 ($\mathcal{GL}$ Syntax) *Let* **LVar**, $\mathcal{N}$, $\mathcal{E}$, *and* $\mathcal{R}$ *as aforementioned. A* $\mathcal{GL}$ *formula* ϕ *is defined by the following BNF.*

$$\begin{array}{rcl} \psi & ::= & \mathbf{0} \mid x = y \mid \nu(x) \mid \beta(x,y) \mid \neg\psi \mid \psi \wedge \psi \mid \forall x.\psi \mid \\ \phi & ::= & \psi \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{EF}_R\ \phi \mid \mathsf{AF}_R\ \phi \mid \mathsf{EG}_R\ \phi \mid \mathsf{AG}_R\ \phi \mid \\ & & \mathsf{E}[\phi\ \mathsf{U}_R\ \phi] \mid \mathsf{A}[\phi\ \mathsf{U}_R\ \phi] \end{array}$$

A $\mathcal{GL}$ formula is a *state formula*, *i.e.* it will be evaluated over a graph G and *wrt.* a given partner graph grammar. The informal description of the intended semantics is given for the first-order part and two temporal operators.

- The ψ constructs are standard in first-order logic. They are used to reason about single graphs. Graphs are easily encodable in first-order logic. The usual abbreviations for $\mathbf{1}$, $\vee$, $\rightarrow$, $\leftrightarrow$, and $\exists x.\phi$ will be used later.

- $\mathsf{EF}_R\ \phi$ evaluates to true on G, if there exists a sequence $(G_i r_i)_{i\geq 0}$ of direct derivation steps, such that $G_0 = G$, $r_i \in R$, and $G_i \rightsquigarrow_{r_i} G_{i+1}$ for all $i \geq 0$. Moreover, there must exist an $n \geq 0$ such that ϕ evaluates to true on G_n.

- $\mathsf{AG}_R\ \phi$ evaluates to true on G, if for all sequences $(G_i r_i)_{i\geq 0}$ of direct derivation steps – where $G_0 = G$, $r_i \in R$, and $G_i \rightsquigarrow_{r_i} G_{i+1}$ for all $i \geq 0$ – ϕ holds true of G_j for all $j \geq 0$.

- The intended *temporal* meaning of the remaining constructs corresponds to the temporal meaning of the respective operators in CTL. In addition to the temporal features of CTL, $\mathcal{GL}$ features the restriction of rule applications and the ability to reason about graphs of statically unknown size.

- The restriction R is dropped in case $R = \mathcal{R}$, because the application of rules is restricted to the set of all rules.

- Associativity, binding priorities, and the notions of free and bound variables are defined as usual for the first order part of $\mathcal{GL}$. All first order constructs bind more tightly than the temporal constructs, all of which bind equally strong.

Examples Reconsider the checkers graph grammar $\mathfrak{G}_C$ of Section 2.1.5 on page 18. In that example, nodes correspond to squares on a checkers board. Accordingly, existential or universal quantification over nodes picks squares on the board. The following formula over $\mathfrak{G}_C$

$$\phi_1 = \exists x_1.\exists x_2.\exists x_3.w(x_1) \ \wedge \ \llcorner(x_2) \ \wedge \ \llcorner(x_3) \ \wedge up(x_1, x_2) \ \wedge \ right(x_2, x_3)$$

evaluates to true on positions, where the rule [MoveR] is applicable meaning it is possible for a white man to move right forward. Remember that the missing restrictions stand for the case $R = \mathcal{R}$. For instance, Checkers, $[] \models \phi_1$, where Checkers is the graph modeling the start position of a checkers game. An example of a temporal construct is $\mathsf{AG}\,\phi_1$, expressing that a move right forward is always possible for the white player. As passionate checkers players know, this is not the case.

Formula ϕ_2 distributes the existential quantifier of ϕ_2 over the $\mathsf{E}[\cdot \ \mathsf{U} \ \cdot]$ construct.

$$\phi_2 = \mathsf{E}[\exists x.b(x) \ \mathsf{U} \ \exists x.ww(x)]$$

This formula states that there is a game starting from the current position, such that there exists a black piece in every position until a white king appears. This formula is certainly true if evaluated on the starting position Checkers. Notice that the squares can be newly chosen after every direct derivation step, because the existential quantifier is in the scope of the temporal one. If $\mathsf{E}[\cdot \ \mathsf{U} \ \cdot]$ is replaced by $\mathsf{A}[\cdot \ \mathsf{U} \ \cdot]$ in ϕ_3, it evaluates to false on Checkers, because there are games without any white king. The paragraph "Other Logics" below further illustrates the subtle interplay between temporal constructs and node quantifiers.

A concluding example digs into the role of restrictions to sets of rules. Let *Mov* be the set of all checkers transformation rules dealing with simple piece movement (no capturing or becoming king).

$$\phi_3 = \mathsf{AG}\,\mathsf{EF}_{Mov}\,\forall x.\neg b(x) \wedge \neg bb(x)$$

This formula expresses a classic liveness property. In all games it is always possible to reach a board with all black pieces captured. Evaluating ϕ_3 on the initial board yields false, since there are certainly games with a black winner. Even worse, ϕ_3 forces all black pieces to be captured by *moving only*. The possibility to restrict temporal operators may yield inconsistent formulas. For instance, $\mathsf{AG}_{R_1}\,\mathsf{EF}_{R_2}\ x = x$ for $R_1 \cap R_2 = \emptyset$ is unsatisfiable, as becomes clear from the semantics given in Definition 2.2.2. As a consequence, restrictions should be used carefully.

Semantics In order to ease the presentation of the $\mathcal{GL}$ semantics, a new notion is introduced. It denotes the set of all possible sequences of direct derivation steps starting from graph G while using only rules of set $R \subseteq \mathcal{R}$.

$$\vec{R}(G) := \{(G_i r_i)_{i\geq 0} \mid \forall j \geq 0. G_0 = G \ \wedge \ G_j \rightsquigarrow_{r_j} G_{j+1} \ \wedge \ r_j \in R\}$$

A $\mathcal{GL}$ formula is evaluated over some graph G. The CTL modalities A and E refer to *all* and to the *existence* of sequences of direct derivations starting from G, respectively. Evaluating a $\mathcal{GL}$ formula over a partner graph grammar $(\mathcal{R}, \mathcal{I})$ amounts to evaluating the formula over the initial graph $\mathcal{I}$.

Definition 2.2.2 ($\mathcal{GL}$ Semantics) *Let $\mathfrak{G} = (\mathcal{R}, \mathcal{I})$ be a partner graph grammar and let ϕ be a $\mathcal{GL}$ formula over $\mathfrak{G}$ and logical variables* **LVar**. *Let $\mathfrak{N}$ be the set of all nodes of the graphs in $[\![\mathfrak{G}]\!]$. An* assignment *ρ is a partial mapping* $\mathbf{LVar} \rightharpoonup \mathfrak{N}$.

The graph grammar $\mathfrak{G}$ models *ϕ, written $\mathfrak{G} \models \phi$, if and only if the initial graph with the empty assignment models ϕ.*

Graph G with assignment ρ models *formula ϕ, written $G, \rho \models \phi$ as defined in Table 2.2.*

Other Logics The CTL feature missing from $\mathcal{GL}$ is the *next* operator X. There are two reasons for that choice. First, the major application domain of the specification and verification techniques of this thesis are dynamic communication systems. They are rather heterogeneous and typically lack a global clock. From such a system's point of view a "next" step does not make much sense in the specification of its behavior. Many actions happen independently, simultaneously, or with varying, small delays. What is then the meaning of X? On the other hand, if X was included, the verification technique presented in Chapter 4, would not be able to verify such properties, because several direct derivation steps may be summarized in the abstraction.

Traditional temporal logics such as CTL, upon which $\mathcal{GL}$ builds, are *propositional*. Variants of propositional temporal logic like LTL, CTL, or variants thereof, are effectively used when reasoning about systems with a static number of components. In contrast, the main objective here is the verification of *dynamic systems*. Objects (nodes in terms of graph representations) are created at runtime and the task is to specify that each object, independent of its creation time or links to other objects, adheres to given requirements. The name *anonymous object* has been coined, for example in [YRSW03], for dynamically allocated (heap) objects that are not referable

G, ρ	$\models$	$\mathbf{0}$	**iff**	never
G, ρ	$\models$	$x = y$	**iff**	$\rho(x) = \rho(y)$
G, ρ	$\models$	$\nu(x)$	**iff**	$\ell_G(\rho(x)) = \nu$
G, ρ	$\models$	$\beta(x, y)$	**iff**	$(\rho(x), \rho(y)) \in E_G^\beta$
G, ρ	$\models$	$\neg\phi$	**iff**	not $G, \rho \models \phi$
G, ρ	$\models$	$\phi_1 \wedge \phi_2$	**iff**	$G, \rho \models \phi_1$ and $G, \rho \models \phi_2$
G, ρ	$\models$	$\forall x.\phi$	**iff**	$\bigwedge_{u \in V_G} G, \rho[x \mapsto u] \models \phi$
G, ρ	$\models$	$\mathsf{EF}_R\ \phi$	**iff**	there exists $(G_i r_i)_{i \geq 0} \in \vec{R}(G)$ and $n \geq 0$ such that $G_n, \rho \models \phi$
G, ρ	$\models$	$\mathsf{AF}_R\ \phi$	**iff**	for all $(G_i r_i)_{i \geq 0} \in \vec{R}(G)$ there exists a $n \geq 0$ such that $G_n, \rho \models \phi$
G, ρ	$\models$	$\mathsf{EG}_R\ \phi$	**iff**	there exists $(G_i r_i)_{i \geq 0} \in \vec{R}(G)$ such that for all $j \geq 0$ holds $G_j, \rho \models \phi$
G, ρ	$\models$	$\mathsf{AG}_R\ \phi$	**iff**	for all $(G_i r_i)_{i \geq 0} \in \vec{R}(G)$ and for all $j \geq 0$ holds $G_j, \rho \models \phi$
G, ρ	$\models$	$\mathsf{E}[\phi_1\ \mathsf{U}_R\ \phi_2]$	**iff**	there exists $(G_i r_i)_{i \geq 0} \in \vec{R}(G)$ and $n \geq 0$ such that $G_j, \rho \models \phi_1$ for all $j < n$ and $G_k, \rho \models \phi_2$
G, ρ	$\models$	$\mathsf{A}[\phi_1\ \mathsf{U}_R\ \phi_2]$	**iff**	for all $(G_i r_i)_{i \geq 0} \in \vec{R}(G)$ there exists $n \geq 0$ such that $G_j, \rho \models \phi_1$ for all $j < n$ and $G_k, \rho \models \phi_2$

Tab. 2.2: Definition of the $\mathcal{GL}$ semantics. See also Definition 2.2.2

by a fixed name, like a global variable, but have to be addressed symbolically. In contrast to [YRSW03], $\mathcal{GL}$ is not able to reason over the evolution of nodes over time, including their points of creation and disappearance. This is mainly due to the fact, that the verification of such properties is out of the scope of the work presented in this thesis.

Recent research produced a number of requirement specification languages for reasoning about anonymous objects, for example, [Dis03, YRSW03, DH01, BSTW06]. Actually, reasoning about the evolution of individuals already started in 1946 with the development of *quantified modal logics* (QML) [Car46]. Kripke established a so-called *possible world* semantics [Kri63] of QML, based on a set of worlds and a successor relation among them. In terms of partner graph grammars, a world corresponds to a graph and evolution between worlds corresponds to direct derivation steps. In the possible world semantics, identical nodes may occur in several worlds.

In 1968, Lewis argued against the fixed quantification domain of Kripke

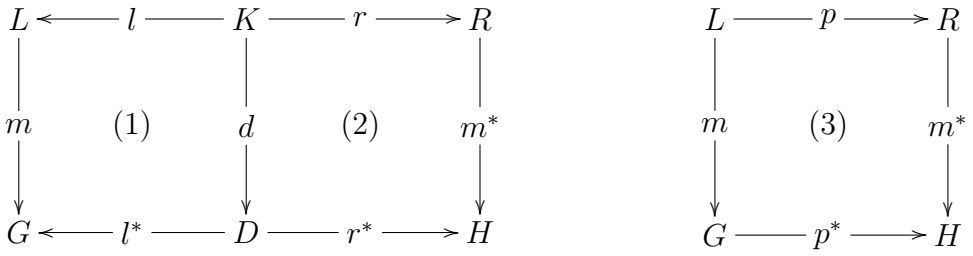
and developed his counterpart theory [Lew68] where each individual exists in exactly one world and corresponding individuals in different worlds are identified via a counterpart relation. This paradigm was, for example, chosen as the basis of ETL [YRSW03]. It avoids conflicts that may arise by objects that are destroyed and re-appear again. This is crucial for ETL, because, in contrast to $\mathcal{GL}$, it provides explicit constructs for reasoning about creation and destruction of objects. As argued for the X case, this is not in the scope of the verification techniques to be presented in Chapter 4. More importantly, as $\mathcal{GL}$ is not the main contribution of this thesis, a superfluous technical burden is avoided by not including this feature.

2.3 Relation to Algebraic Graph Transformation

A comprehensive survey on graph grammars of all shapes and colors is presented in [Roz97]. The approach that inspired the design of partner graph grammars belongs to the area of algebraic graph transformations as presented in Chapters 3 and 4 of [Roz97]. Other approaches include node- and hyperedge replacement (Chapters 1 and 2) grammars, monadic second order definable graph transductions (Chapter 5), and 2-structures (Chapter 6). They are not considered here.

In the theory of algebraic graph transformation, a graph is modeled as a two-sorted algebra, where one sort represents the nodes and the other the edges. The algebra is further equipped with two unary operations *source* and *target* each mapping an edge value to a node value with the obvious meaning. Note that this representation features multiple, equally labeled edges between the same pair of nodes. Graph transformations are formulated as pushout constructions in a categorical framework. Basic knowledge about category theory is helpful to understand the next paragraphs. Good introductory material can be found in [Pie91, BW95, ML71].

Depending on the details of the categorical construction, one distinguishes two main approaches in the theory of algebraic graph transformation: the single pushout (SPO) and the double pushout approach (DPO). Historically, DPO came first. Both approaches can be based on categories with graphs as objects. These are the graphs that can be represented by two-sorted algebras as described above. Many different variants of graphs are possible in both approaches. The arrows considered in the DPO approach are total morphisms, whereas the arrows considered in the SPO approach are partial morphisms.



DPO SPO

Fig. 2.6: Application of a transformation rule in the double and single pushout approach to algebraic graph transformation.

In the DPO approach a transformation rule consists of a pair

$$L \xleftarrow{l} K \xrightarrow{r} R$$

of total graph morphisms. Graph K is a common interface graph. An application of such a rule consists of a commuting diagram of two graphs and total graph morphisms as shown in (1) and (2) of Figure 2.6. The context graph D is computed from G by deleting all the elements from G that are not in $dom(l)$. In terms of categorical constructions, diagram (1) shows how D is computed as the *pushout complement* of match m and the l part of the transformation rule. The result H of the application is obtained by constructing the *pushout of* r and d as shown in diagram (2). Informally, all the elements of R are inserted into H that are not in $dom(r)$. Pushouts and pushout complements are known to exist in the categories under consideration. Note that matches are not required to be injective either in SPO or DPO.

In the SPO approach a transformation rule consists of a *partial* homomorphism p as given in the top line of diagram (3) in Figure 2.6. Obtaining the result of a rule application corresponds to the construction of the pushout of match and transformation rule – m and p in diagram (3). It is computed similar to the construction in Definition 2.1.6.

Gluing conditions are a notion of algebraic graph transformation that are needed to resolve two sorts of conflicts that may emerge in transformation rules. The first conflict is due to non-injective matches and is resolved by the *identification* condition. Consider Figure 2.7(a). A non-injective match maps two nodes to the same node that is to be deleted by the rule application. Is

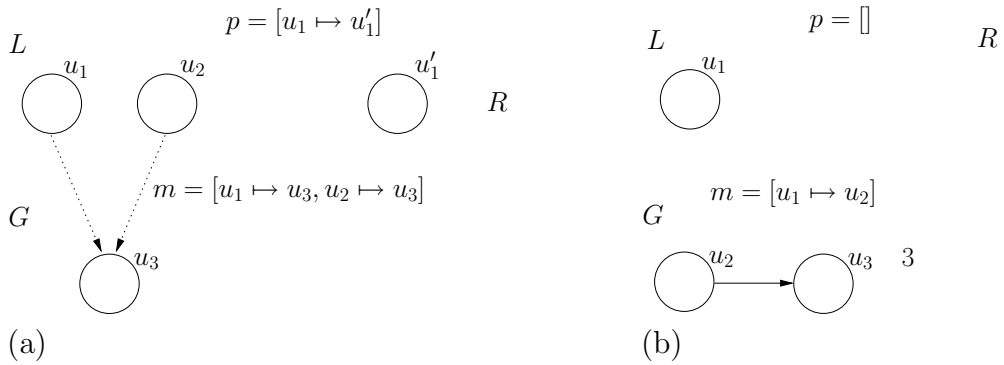


Fig. 2.7: Gluing conditions. Part (a) is a showcase application for which an identification condition is needed in the DPO approach. Part (b) demonstrates the need for a dangling condition.

one node to be deleted, in that case which one? Are both to be deleted? In the DPO approach these questions are answered by disallowing the rule application at all, whereas the SPO approach opts to delete both nodes. In partner graph grammars this conflict is excluded by non-injective matches.

Part (b) of Figure 2.7 shows a rule that is to delete a node with incident edges. Again, this is disallowed by a so-called *dangling condition* in the DPO approach. Both the SPO approach and partner graph grammars allow for such applications by deleting all incident edges implicitly.

If one looks at graph grammars as a programming language, one notices that they provide strong means of specifying updates, but hardly any means to specify control. *Negative application conditions* equip graph grammars with some notion of control by saying when a rule cannot be applied. Let L be the left side of a transformation rule in the SPO approach. A negative application condition is a set of constraints, where a constraint is a graph L' connected to L by a morphism $L \to L'$. A match $m : L \to G$ *satisfies* the constraint $L \to L'$, if there does not exist a total morphism from L' to G. Put differently, the constraint is satisfied, if the image of the match cannot be extended to contain the L' part of the constraint. General application conditions may contain several constraints, all of which must be satisfied in order for a match to satisfy the application condition.

Single Pushout versus Partner Graph Grammars This paragraph reviews the relation between the classic SPO approach to algebraic graph transformation and the notion of partner graph grammars. A detailed comparison of SPO and DPO is available in Chapter 4 of [Roz97]. Essentially, partner graph grammars are a variant of the SPO approach with a mild form

of negative application conditions. The differences are elaborated on below. They are necessary to keep partner graph grammars amenable to the formal verification technique presented in this work. In particular, general negative application conditions could not be handled.

1. Multigraphs, *i.e.*, graphs featuring multiple, equally labeled edges between the same pair of nodes, are handled in the SPO approach. Such edges are not possible in partner graph grammars.

2. Partial morphisms in SPO transformation rules are replaced by simple partial mappings in partner graph grammars. This allows easy node- and edge-relabeling in partner graph grammars. In SPO relabeling inflicts an additional burden [HP02].

3. Involved categorical constructions in all algebraic approaches to graph transformation are replaced by direct, constructive definitions in this thesis.

4. The problem of dangling edges created by rule applications is solved in favor of deletion both in SPO and partner graph grammars. The issue of identification raised by non-injective matches needs not to be addressed in partner graph grammars, because injective matches are required.

5. The concept of partner constraints in partner graph grammars, allows for a restricted form of negative application conditions. The restrictions lie in the *locality* of partner constraints, whereas general negative application conditions in the SPO approach are more expressive.

6. The SPO approach lends itself to the use of *attributed graphs* [LKW93], where additional labels are used for computational issues. The additional labels stem from an arbitrary algebra and operations on them may occur in a direct derivation step. Label constraints within partner graph grammars present a similar concept, even though it is only used for matching purposes.

In general, most of the work in the area of algebraic graph transformation is interested in advancing the state-of-the-art of the technique itself. Traditionally, not much work is dedicated to the *verification* of graph grammars. Exceptions include [Hec98, BKR04, RD06, KK06, BW06] and are discussed in Chapter 5. In the context of this thesis, partner graph grammars are mainly used as a means of specification of systems of evolving graphs. The

main focus, however, is on the verification of such systems. It is hence justified to design partner graph grammar as a trade-off of expressive power for analyzability.

Chapter 3

Case Studies

Partner graph grammars are tailored to specify dynamic communication systems. As matches can occur anywhere in a system, partner graph grammars can easily model highly concurrent systems without fixed control structures. Rather, if essential, control needs to be encoded (Section 3.2). While the previous remarks hold for graph grammars and dynamic communication systems in general, the strong ability of partner graph grammars to reason about the immediate neighborhood of a communicating object, *e.g.* by means of partner constraints, makes them a good candidate to model communication. Communication happens between adjacent objects. Moreover, partner graph grammars are still amenable to formal verification.

A complex case study from the domain of dynamic communication systems is presented in Section 3.1: car platooning, a system to automatically optimize traffic flow. It was originally studied in the PATH project [PAT03]. The rather simplistic specification and verification approach taken there is improved by a series of examples. They demonstrate the ability of partner graph grammars to specify many complex features inherent to dynamic communication systems: destruction and and unbounded creation of objects with a dynamically evolving communication topology (Section 3.1.1), unreliable communication due to faulty channels (Section 3.1.3), or asynchronous communication based on message queues (Section 3.1.2). Despite their complexity these features are still amenable to the verification methods proposed in later chapters.

This chapter concludes by naming more potential application domains and other examples to be specified by partner graph grammars: process calculi, heap-manipulating programs, and routing in ad-hoc networks.

3.1 Car Platooning

The main case study of this thesis is motivated by the "Program on Advanced Technology for the Highway" (PATH) [PAT03] conducted at the University of California, Berkeley. The relevant context for this work is the design and the verification of *car platooning maneuvers* [HESV91]. A good part of this introductory material is taken from this report.

Platoons are up to twenty closely spaced vehicles under automatic control and are used to organize highway traffic. This organization may ideally result in an increase of highway capacity while decreasing travel time at the same time. The work reported in [HESV91] focuses on the design of controllers for such platoons. The control tasks are arranged in a three layer hierarchy. The top layer is called *link layer*. There, a centralized controller assigns to each vehicle its target and a path through the highway. The other layers are implemented as controllers on each vehicle. A vehicle's *platoon layer* plans its trajectory to conform to its assigned path. This plan consists of a sequence of elementary maneuvers:

- *Merge* combines two platoons into one.
- *Split* separates a platoon in two.
- *Change lane* enables a single car to change lane.

After the platoon layer has determined to execute a maneuver, it instructs the bottom layer, the *regulation layer*, to physically initiate the maneuver by, *e.g.*, accelerating or breaking. Information necessary to physically execute maneuvers comes from roadside monitors, sensors, and a communication infrastructure. Among the sensors, there are longitudinal and lateral range sensors. Communication means the exchange of messages by either broadcasting information to every vehicle in range or by addressing specific vehicles. All the physical issues are ignored for the purpose of [HESV91] and for this thesis, where ignoring means abstraction by non-determinism. [LL98] is a good starting point for readers who want to learn more about hybrid aspects of car platooning.

This case study as well as the report [HESV91] concentrate on design and verification of the platoon layer. In order to ensure the safety of a maneuver, vehicles need to negotiate according to *protocols*. A protocol is a structured sequence of messages. The protocols considered here, *i.e.*, their specification and verification in [HESV91], are described in the next paragraphs. It is shown that neither specification nor verification as described in [HESV91] are sufficiently close to reality. Even if physical behavior is abstracted by

non-determinism (as done both in [HESV91] and here), at least the following over-simplifications are found in the cited report.

- *Dynamics*: Some sort of artificial modularity is imposed on the protocols in [HESV91]. Therefore, only scenarios involving a fixed, static number of vehicles are considered. This neglects the highly dynamic and concurrent nature of the platoon scenario. Vehicles may appear and disappear (enter and leave the highway), for example. Many maneuvers may happen simultaneously.
- *Communication* is modeled by a simplistic and unrealistic shared memory approach. Synchronous communication via shared memory does not comply to the scenario of highly diverse vehicles on highways. Rather, asynchronous models with unreliable channels should be considered.
- *Failure resistance*: No attention is paid to unreliable communication due to faulty channels and to solutions to this issue.

One of these shortcomings is addressed in each Section 3.1.1, Section 3.1.3, and Section 3.1.2, respectively, where more refined partner graph grammar specifications of platoon maneuvers are presented. Certainly, verification of the refined models cannot be tackled by the methods of [HESV91].

Platoon Organization The lead car of a platoon is called its *leader*, the rest are called *followers*. Single vehicles are called *free agents*. Intra-platoon communication happens only between the leader and one or all of its followers. Inter-platoon communication consequently affects leaders only. The state of a vehicle, *i.e.* the information each vehicle possesses, include

- A hard-wired identity.
- Highway, lane, and section numbers available from roadside monitors.
- Optimal platoon size and speed provided by the link layer.
- Its leader's identity.
- The size of the platoon and its position within the platoon.
- A busy flag.

The last three items are updated after a maneuver has been negotiated. As stated before, the remaining information is out of scope of [HESV91] and this work, *i.e.*, behavior dependent on this information occurs non-deterministically.

Maneuvers All upcoming descriptions abstract a considerable amount of details from the physical or hybrid world, such as numerical information, geographical positions, velocities, highway geometry, and so on. In particular, these features will not be modeled in the partner graph grammar implementation of these maneuvers. The features having an influence on communication topologies are modeled non-deterministically.

Merge Assume two platoons on the same lane, where the rear platoon approaches the front platoon. A merge protocol is initiated by the rear leader after the rear leader has received a "car ahead" acknowledgement. Physically, a car ahead is sensor-triggered, whereas it may occur non-deterministically here. If size requirements are met and if the front leader is not busy (being involved in other maneuvers), it acknowledges the request and new leader information is distributed to the rear leader's followers.

Split A split may be initiated by a platoon leader or by a follower, because (i) the platoon size exceeds the optimal size, (ii) a car wants to move to an adjacent lane or (iii) become a free agent. If the leader wants to split, it informs its first follower and accelerates. The first follower broadcasts the new leader information. A follower that wants to split from a platoon sends a request to its leader. On acknowledgement it decelerates and confirms to its former leader.

Change lane This protocol may be executed by free agents only. Negotiations involve several cars depending on the following cases. Let A be the car willing to change lane. If both the lane A wants to move to – called the target lane – and the lane adjacent to the latter (not the car's current) – called far lane – are unoccupied, the car may just move. If the target lane is clear and the far lane is occupied by car C, A needs to check whether C wants to move to the same lane A wants. Finally, if the target lane is occupied by another platoon with leader B, then (i) B can ask A to decelerate and to move over behind B; (ii) B can decelerate to let A over in front of B; (iii) B can split to let A in. The option that is actually taken depends on information like the relative speeds and sizes of the involved cars and platoons.

COSPAN The formal specification and verification of platoon maneuvers in [HESV91] is conducted within the COSPAN framework [HK87]. A COSPAN specification consists of a finite set of *COSPAN automata* that synchronize via shared global memory. Each COSPAN automaton is a finite-state machine with output and guarded transitions. Each state of a COSPAN

automaton may be equipped with a set of possible outputs. In each step of the global system, one element of this set is chosen non-deterministically and written to the global memory that can be read and written by all automata in the system. COSPAN automata transitions may be guarded by boolean expressions over the global memory. The number of COSPAN automata in a system is fixed and static, building a closed system. The language generated by this system is the set of all possible sequences of global memory states.

The COSPAN specification of the merge maneuver, for instance, is made up of two *process automata* – modeling the front and the rear leader, respectively – and four *environment automata* modeling range sensors, busy flags and size conditions. They form a closed system.

COSPAN verification Verification of a COSPAN specification is tackled by *task monitors*. A task monitor is just another COSPAN automaton that does not write into the global memory thus not affecting system behavior. In order to define the acceptance behavior of a task monitor, one may specify cycle and recurrence constraints turning task monitors into a variant of Büchi automata. A word (sequence of global memory states) is accepted if the automaton either loops in a state specified to be a cycle state, or if it perpetually takes one of the transitions specified to be recurrent. The COSPAN model checker is able to check whether the language generated by a COSPAN system is included in the language accepted by a task monitor added to the system.

The task monitor of the merge protocol requires that either nothing happens or that a merge request by a rear leader is always followed by a positive or negative acknowledgment by the front leader.

COSPAN and [HESV91] Criticism Obviously, the platoon case study involves a big amount of concurrency, dynamics in the communication topology, as well as appearing and disappearing objects (cars). Hence, it is a prototypical instance of a dynamic communication system. Therefore, the static scenario of COSPAN is appropriate neither for specification nor verification of this particular case study. The major criticisms are as follows.

1. In a COSPAN specification, there is a fixed, a priori known, unchangeable number of processes building a closed system, whereas, in this case study, there are cars entering and leaving the highway while constantly changing their communication topology.

2. A synchronous model of communication using shared global memory is not exactly the first choice to model a dynamic system of heteroge-

nous objects with rapidly changing communication topology. In fact, the current communication topology is not even reflected in the global memory.

3. In a system like this case study, one cannot simply assume reliable communication, in particular, if the communication is wireless. No care is taken for unreliable, faulty channels in the COSPAN specification.

The authors of [HESV91] claim to be able to exclude these issues by enforcing a top layer, mutual exclusion policy driven by global "is busy" information. Again, this mutual exclusion policy is hardly verified. Moreover, one maneuver at a time without any interference seems like a severe restriction for a dynamic communication system. What happens if a car in a merging platoon suddenly needs to change lane? Another hint to the over-simplification is the number of states observed for the split maneuver in the COSPAN verification. 17. This appears to be clearly too few.

Apart from these general issues showing why COSPAN is not the first pick to verify dynamic communication systems, the task monitors simply impose certain sequences of messages to occur. Surprisingly, these sequences can be easily read from the specification automata as well. Verification becomes an almost trivial task if the properties are isomorphically read from the system specification.

The shortcomings listed above – no dynamics, synchronous communication without an explicit communication topology, and reliable channels – are addressed in the following three sections. The platoon scenario is specified in more detail using partner graph grammars underpinning their crucial role in the specification of dynamic communication systems. Verification of partner graph grammars and the platoon case study modeled by them is addressed in later chapters.

3.1.1 Idealized Platoons

The graph grammar $\mathfrak{G}_{\text{ideal}}$ is the first one modeling the platoon scenario. It is the simplest in a series of examples that become gradually more involved. As usual and as observed in [HESV91], all physical behavior is abstracted by non-determinism. On top of that, $\mathfrak{G}_{\text{ideal}}$ abstracts from explicit messages and from message queues. It is based on a synchronous model of communication. Also, no unreliable channels are considered in $\mathfrak{G}_{\text{ideal}}$.

However, it has the typical dynamic features. Nodes modeling cars can be created and destroyed dynamically, their number is not fixed and potentially unbounded. Moreover, the communication topology is modeled explicitly.

The graph grammar $\mathfrak{G}_{\text{ideal}}$ is defined in terms of the sets $\mathcal{N}_{\text{ideal}}$={*fa*, *ldr*, *flw*, *rl*, *fl*, *spl*, *acc*} of node labels and $\mathcal{E}_{\text{ideal}} = \{\_\}$ of edge labels. Here is a description of the role of the various node labels.

fa This label is used to model *free agents*, *i.e.*, cars that are not involved with others to build a platoon. It is also the state of cars that may appear or disappear.

ldr Platoon *leaders* are modeled using this label, unless they are engaged in a merge maneuver.

flw *Followers* assume this label regardless of maneuvers they may be involved with.

rl The *rear leader* of two merging platoons is modeled using this label, even if the rear platoon is a free agent. In that case, the free agent assumes this label, when the maneuver starts.

fl This label is analogous to the previous, except that it models the *front leader* of merging platoons.

spl The *spl* label marks *followers* that are willing to *split*. More precisely, it marks the follower that is to become a leader after the split.

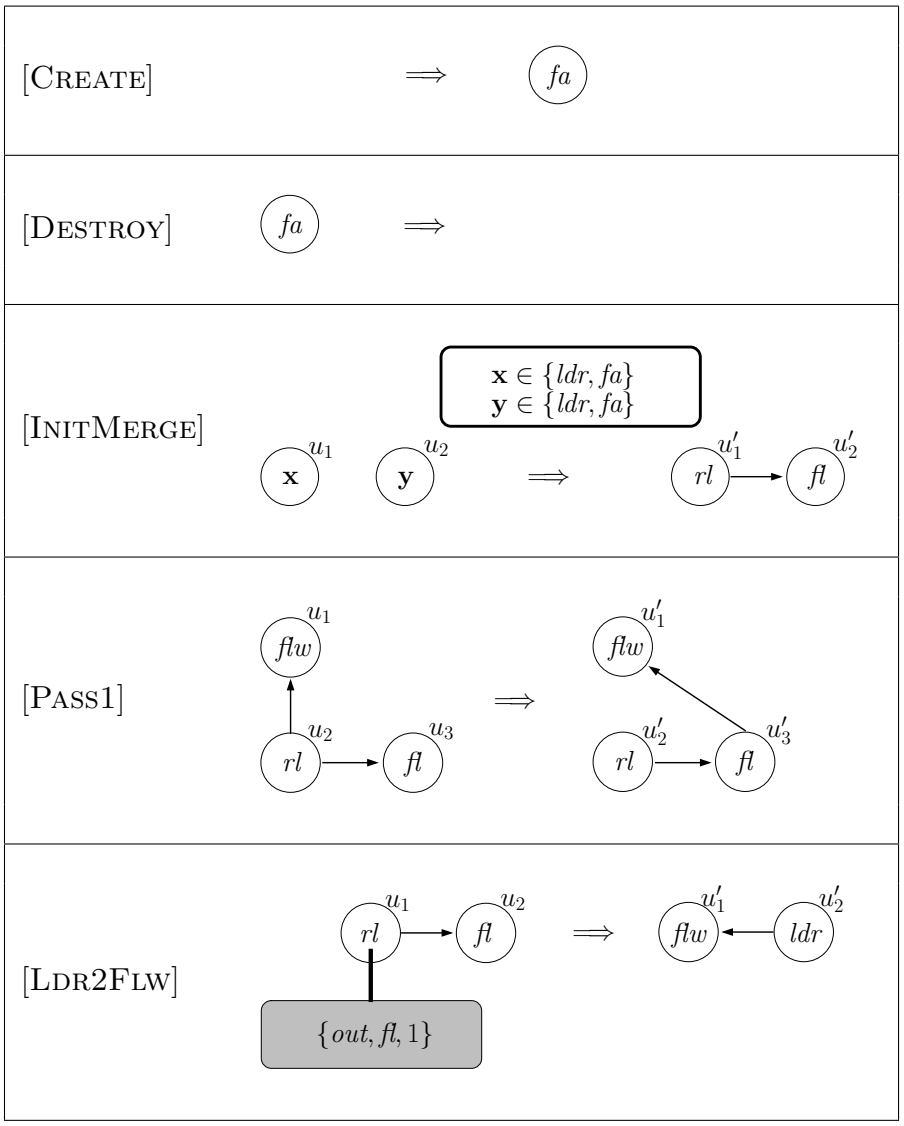
acc If the *leader* of a platoon needs to *split*, it can do so by accelerating which gave the name to the last label of $\mathcal{N}_{\text{ideal}}$.

Notice that the change lane maneuver is not in the scope of $\mathfrak{G}_{\text{ideal}}$ as it depends and affects merely physical parameters that are not in the scope of modeling by graph grammars – at least not at this point in time.

Merge Maneuver Table 3.1 presents the five transformation rules modeling a merge maneuver. They are collected in the rule set $\mathcal{R}_{\text{merge}}$. If not stated otherwise, the mapping between left and right graph of a rule is assumed to be $h = [u_i \mapsto u_i']$. It maps nodes to their primed versions. In these cases, as in Table 3.1, the h mapping is dropped from the pictorial notation of the rules.

[**Create**] As this rule always matches, a new free agent may enter the stage at any time...

[**Destroy**] ... and leave it again in an unconstrained manner.



Tab. 3.1: The merge maneuver rules $\mathcal{R}_{\text{merge}}$.

[**InitMerge**] This rule triggers the beginning of a merge maneuver and involves two cars that may either be leader or free agent. The label constraint on top denotes that an arbitrary (one of four) combination of this kind is possible. The ongoing merge maneuver is characterized by the rear leader/front leader connection. At the same time, this rule is a non-deterministic abstraction of sensor information triggering a merge maneuver in the real system.

[**Pass1**] During a merge maneuver the rear leader needs to hand over its followers to the front leader, because the latter will become the leader of the merged platoon.

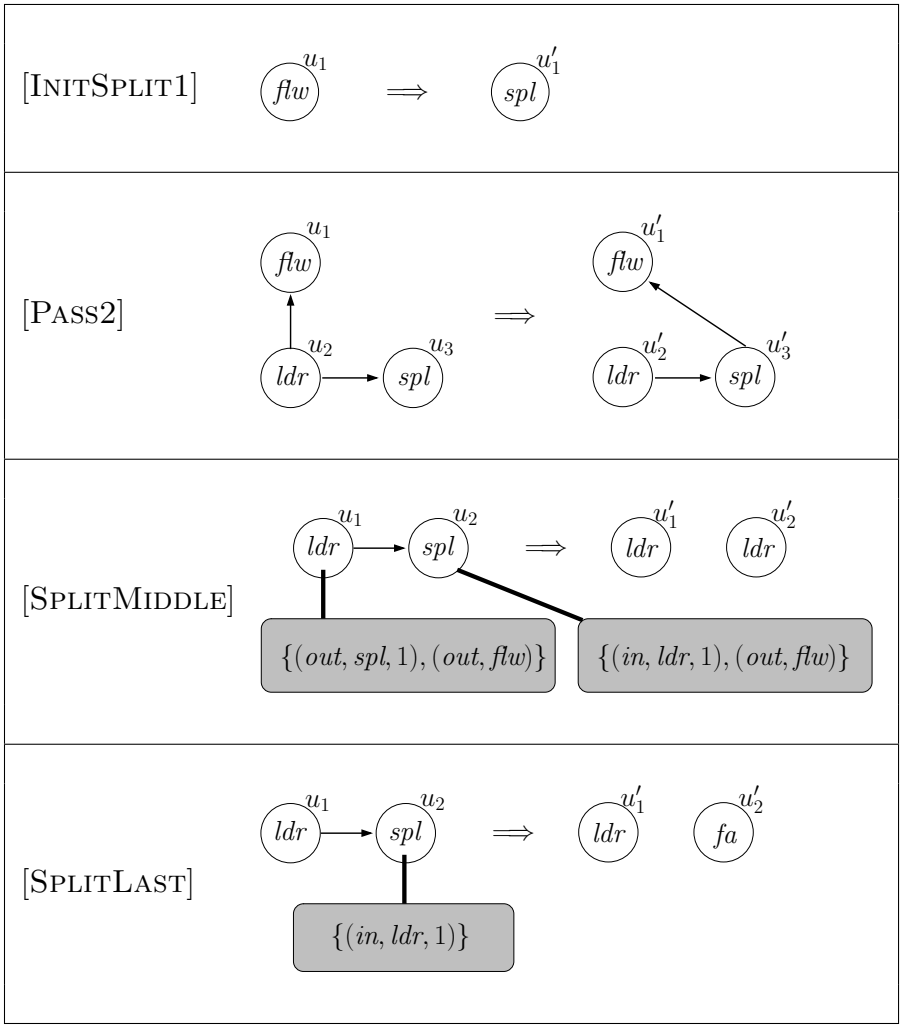
[**Ldr2Flw**] The partner constraint on the bottom is the formalization of the fact, that the rear leader has handed over all of its followers. In other words, it is only connected to a front leader. In that case, the rear leader may become a follower of the front leader, which in turn becomes the leader of the merged platoon. This rule concludes a merge maneuver.

Split in the Middle The split maneuver is two-fold. It needs to be distinguished whether a split is supposed to occur in the middle (or at the end) of a platoon or whether the leader needs to accelerate. The first set of rules, $\mathcal{R}_{\text{splitM}}$, presented in Table 3.2, deals with the first case.

[**InitSplit1**] This part of the split maneuver is triggered by a follower announcing a split. The rule may be applied at any time as long as there is a follower. Again, this amounts to a non-deterministic abstraction of the real system behavior.

[**Pass2**] This rule caters for re-structuring one platoon into two by assigning the split initiating car followers that originally belonged to the leader of the splitting platoon. How many cars there are eventually handed over – corresponding to the position of the split initiator – is not specified and abstracted by non-determinism.

[**SplitMiddle**] If non-determinism decided that the split initiator has a middle position within the splitting platoon, followers of the original leader are handed over to the initiator. If both the initiator in state *spl* and the original leader in state *ldr* have followers (as expressed by the partner constraints) their connecting channel is closed and both become regular leaders.



Tab. 3.2: Splitting a platoon in the middle or at the end: $\mathcal{R}_{\text{splitM}}$.

[**SplitLast**] In case the initiator was the last car within a platoon, it will not have any followers – as specified by the label constraint – and becomes a free agent, while the original leader stays what it is.

Split Leader As $\mathfrak{G}_{\text{ideal}}$ does not formalize the change lane maneuver, the remaining set of rules deals with splitting a leader from a platoon by accelerating. They are called $\mathcal{R}_{\text{splitL}}$ and given in Table 3.3.

[**InitSplit2**] A leader that wants to accelerate triggers the second variant of the split maneuver.

[**2Split**] This rule applies in cases of platoons of size two and splits the platoon into two free agents. The partner constraint expresses that there is exactly one follower. This constraint will be troublesome for the later verification.

[**Pass3**] The car in second position is chosen non-deterministically and assumes the leader state *ldr*. At the same time, a follower of the original leader is passed to the second position car. Due to injective matches, this rule applies only for platoons of size at least three.

[**Pass4**] As long as the leader of the splitting platoon in state *acc* still has followers left, it hands them over to the car in second position.

[**AccLeader**] If the original leader has handed over all its former followers to the new leader, *i.e.*, the car in second position within the platoon, it can safely accelerate and become a free agent – leaving behind a properly structured platoon.

Sample Run All the rules from Table 3.1, Table 3.2, and Table 3.3 make up the set $\mathcal{R}_{\text{ideal}} = \mathcal{R}_{\text{merge}} \cup \mathcal{R}_{\text{splitM}} \cup \mathcal{R}_{\text{splitL}}$ that is part of the graph grammar $\mathfrak{G}_{\text{ideal}} = (\mathcal{R}_{\text{ideal}}, \mathcal{I}_{\text{ideal}})$ defining an idealized merge/split platoon scenario. The initial graph $\mathcal{I}_{\text{ideal}}$ is chosen to be the empty graph E, *i.e.* a graph without nodes.

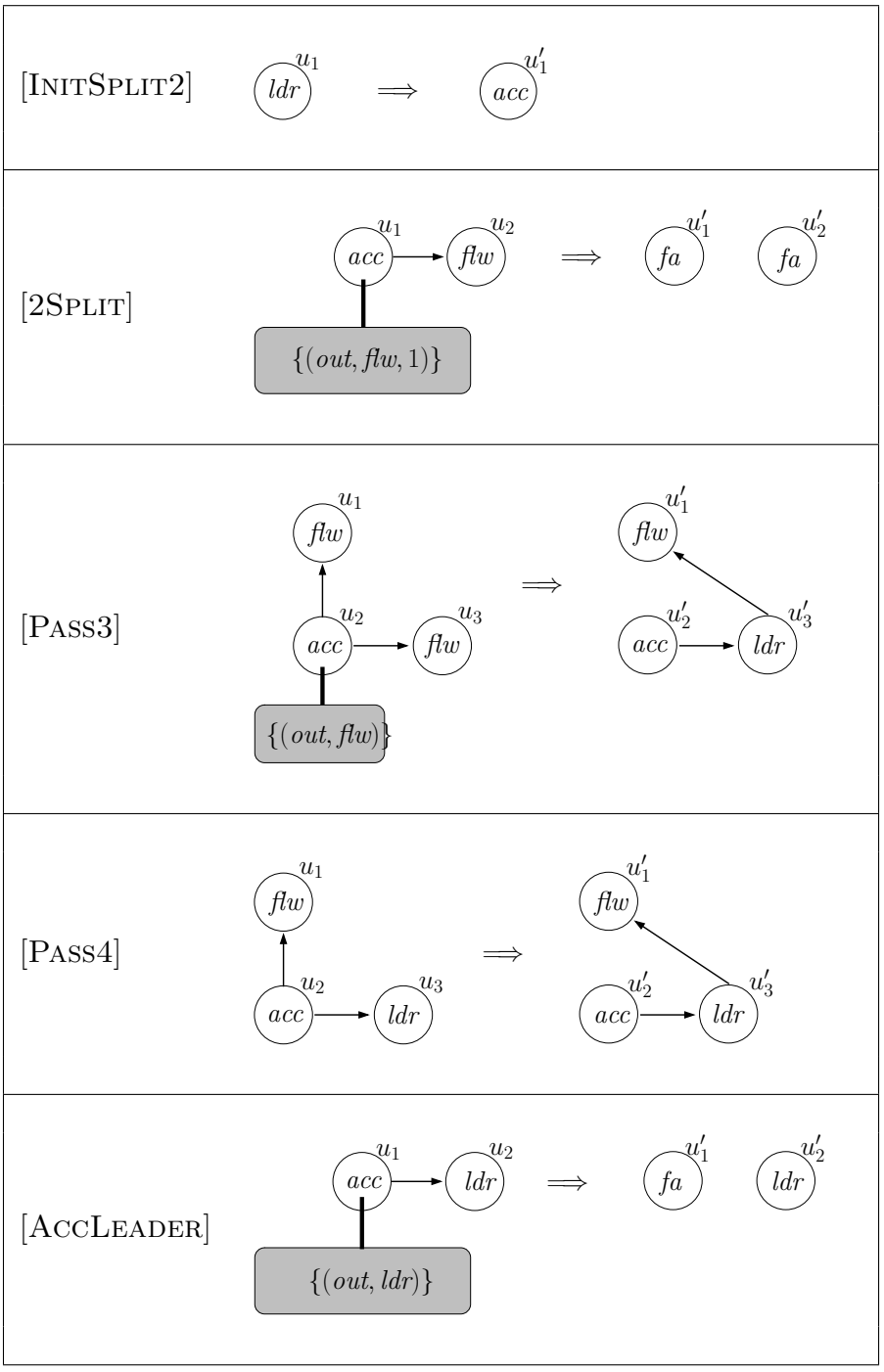
Figure 3.1 shows sample derivations of the partner graph grammar $\mathfrak{G}_{\text{ideal}}$. A few subtle mistakes were introduced in this graph grammar. For instance, consider the next to the last graph of Figure 3.1. Another matching rule for that graph is [InitSplit2]. Applying it, yields a situation, where the leader car wants to accelerate, while the last car is trying to perform a split. This is clearly an error situation. It demonstrates that putting two sets of rules – in this case, $\mathcal{R}_{\text{splitM}}$ and $\mathcal{R}_{\text{splitL}}$ – together without further thought may end up in erroneous behavior, although each single part on its own may be correct.

Another reason for introducing mistakes in this case study is, certainly, to explore the power of the verification techniques of the later chapters: Which mistakes can be discovered, and which cannot?

Sample Properties To conclude Section 3.1.1, some interesting properties of the case study are formalized using the $\mathcal{GL}$ logic of Section 2.2.

$$\mathsf{AG}\,\forall x_1 \forall x_2 . ldr(x_1) \wedge \lrcorner(x_1, x_2) \rightarrow \neg ldr(x_2) \tag{3.1}$$

$$\mathsf{AG}\,\forall x_1 . ldr(x_1) \rightarrow \exists x_2 . \lrcorner(x_1, x_2) \vee \lrcorner(x_2, x_1) \tag{3.2}$$



Tab. 3.3: Formalization of the leader leaving a platoon: $\mathcal{R}_{\text{splitL}}$.

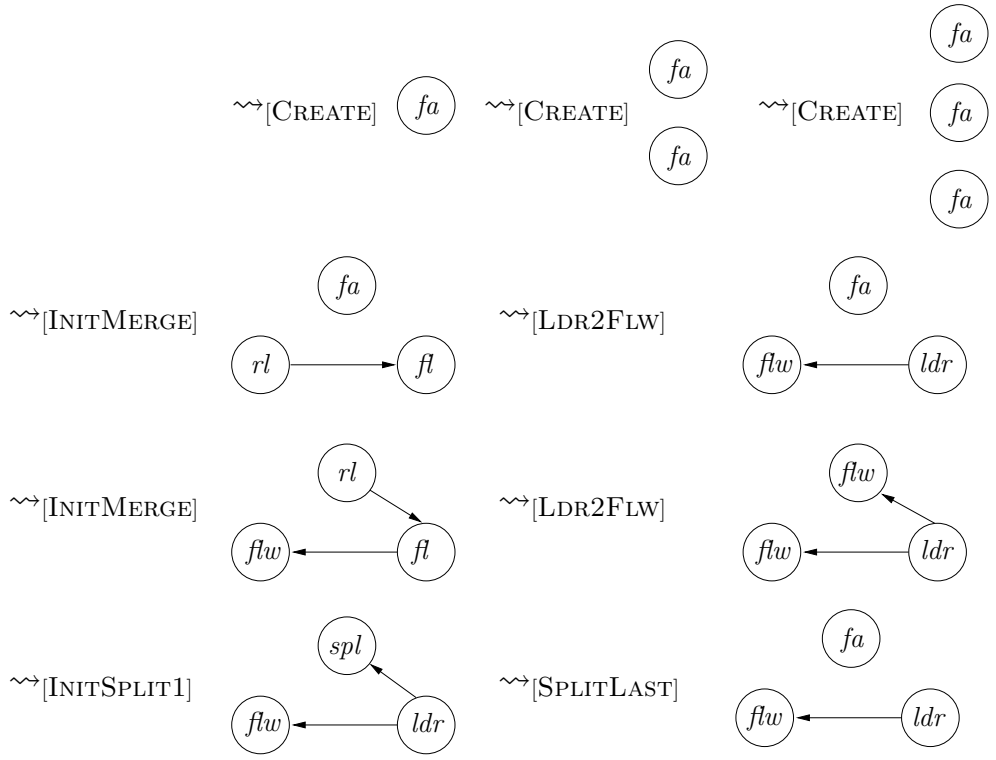


Fig. 3.1: Sample derivation of the partner graph grammar $\mathfrak{G}_{\text{ideal}}$. Three cars are created gradually building a platoon, from which the last car splits again.

Properties (3.1) and (3.2) are typical *safety* properties. The first formula requires that a leader is never connected to another leader. Property (3.2) states that a leader must always be connected to someone else. If it was alone, it should be a free agent. The verification of these and more properties is subject to the verification techniques to be presented in Chapter 4.

3.1.2 Asynchronous Communication

The second shortcoming of the platoon design within the PATH project is concerned with the communication model. In [HESV91], synchronous communication via shared memory is assumed. This, however, seems unrealistic given the heterogeneous nature of this scenario.

Again, partner graph grammars promise to formulate a more appropriate model with explicit and buffered messages. Dynamic features as specified in the previous section remain. Still, no faulty channels or other sources of

unreliability are modeled; neither are physical or hybrid features that remain abstracted by non-determinism.

In [BSTW06], a framework for specification and verification of dynamic communication systems was proposed. Within his context, the techniques presented in this thesis play a major role. Systems in [BSTW06] are specified using an automata-based approach. A lightweight version of it is given here. In particular, message *queues* are replaced with *sets* serving as buffers. In a scenario of queue length 1, these approaches are certainly identical. In general, the set representation is an abstraction of the queue representation. It is easy to see, that the set representation allows for any possible order of message receptions, whereas the queue representation has a well determined order. The formal details of this argument are not important, in order to prove the applicability of partner graph grammars to model asynchronous communication – in fact, queues may be modeled as well – and are thus left out.

An Asynchronous Partner Graph Grammar Template This paragraph introduces a *template* for generating a partner graph grammar modeling asynchronous, buffered, message-based communication. It consists of a number of rules with *parameters*. A valid partner graph grammar is obtained by instantiating them appropriately. To start with, the simplified version of the specification formalism of [BSTW06], *DCS protocols*, is described. After that, the grammar template is introduced and later on instantiated yielding a second, more involved platoon model.

In a slight simplification of [BSTW06] the semantics of a DCS protocol is defined to be a sequence of *topologies*. Each topology consists of a set of *processes* equipped with a unique identity. Each process is in a *local state*. Processes is just another name for cars, objects, or nodes. A process state corresponds to the local state of an object. Each process comes with a *message buffer*, that is a set of *messages*, each of which may be equipped with a *parameter* process identity linking the message to another process. Finally, each process comes with a finite set of named *channels*, where a channel may contain a set of process *identities* – spanning the communication structure of the topology. One topology may evolve into another one by one of the following actions.

Process Creation The set of states contains a non-empty set of *initial states*. A process in its initial state with an empty message buffer and empty channels may appear within a topology in an evolution step.

Process Destruction The set of states contains a (possibly empty) set of

fragile states. A process that is in one of those states may disappear in an evolution step regardless of the contents of its buffer and its channels. On destruction, the process' identity is collected from all other processes' channels and buffers.

Send Message A process in state q may *broadcast* a message m to all processes to which it is connected by channel c thus changing its state to q'. This is denoted by $s : (q, m, c, q')$. The process may attach its own identity or another known identity from channel c' to the message denoted $s : (q, m, id, c, q')$ or $s : (q, m, c', c, q')$, respectively. The message is added to *all* processes' buffers located in channel c of the sending process.

Receive Message A process in state q may receive a random message m from its message buffer, changing its state to q'. This is written $r : (q, m, q')$. If message m comes with a parameter identity, the process in state q adds this identity to its channel c, denoted $r : (q, m, c, q')$

Reset Channel A process in state q may erase the contents of its channel c thus changing its state to q', written $e : (q, c, q')$.

Environment Messages The set of messages contains a designated set of *environment messages*. The environment is a virtual, *i.e.* not existent in any topology, process that can send environment messages to any process at any time. The identity of any process in the current topology may be attached to such a message.

As may be guessed from the preceding description, the specification of a DCS protocol involves the following syntactic ingredients. By fixing these ingredients, a system is specified completely. The partner graph grammar template uses them as a parameter, yielding a proper partner graph grammar by instantiating them. The ingredients are:

1. A finite set Q of local states.
2. A non-empty set $I \subseteq Q$ of initial sets.
3. A set $F \subseteq Q$ of fragile states.
4. A set msg of messages.
5. A set $emsg \subseteq msg$ of environment messages.
6. A finite set χ of channel names.

7. A successor relation *succ* containing s, r, and e tuples as aforementioned.

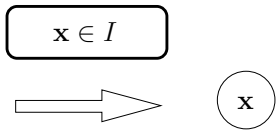
The partner graph grammar template for the specification of systems with asynchronous communications is instantiated by choosing a concrete set of local states $q \in Q$ including initial and fragile states, a concrete set of channels $c \in \chi$, and messages *msg* including environment messages. Given concrete values of Q, I, F, *msg*, *emsg*, χ, and *succ* the partner graph grammar template is instantiated to a partner graph grammar over the sets

$$\mathcal{N}_{\text{DCS}} = Q \ \cup \ Q \times \chi \times \mathbb{N} \ \cup \ msg$$
$$\mathcal{E}_{\text{DCS}} = \chi \dot{\cup} \{buf, par\}$$

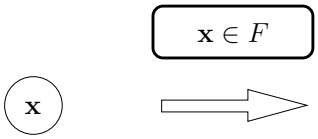
of node and edge labels, respectively. Processes as well as messages are thus modeled as nodes, channels are modeled as edges with one additional edge label used to link the message buffer to a process. The $Q \times \chi \times \mathbb{N}$ part is needed for technical reasons explained below.

Each rule of a partner graph grammar is obtained by instantiating one of the template rules that are now described in more detail. Having picked a set of initial states I and fragile states F, the first rules denote the spontaneous creation and destruction of processes in the initial or fragile states. As usual, the correspondence between left and right graph is implicitly given by using primed versions of node identities. Note that label constraints (Section 2.1.6) are employed as a shorthand notation.

[TCreate]

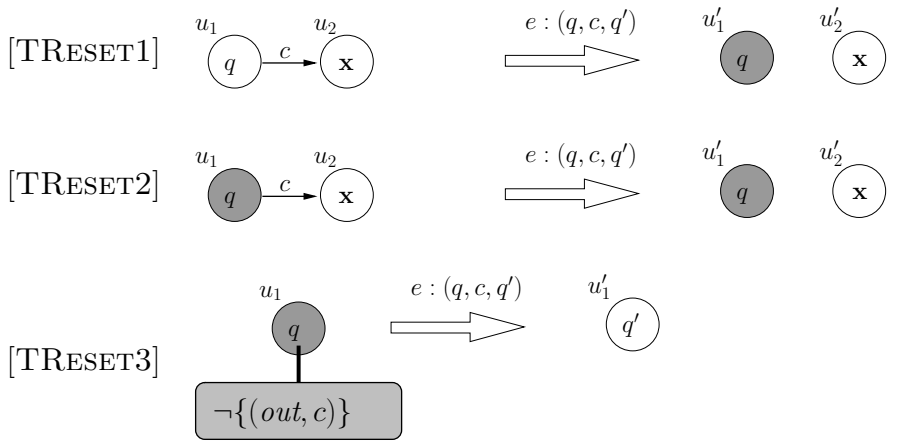


[TDestroy]



The following three more complicated rule templates are, at the same time, a showcase for what graph grammars are not very well suited to model. For each tuple $e : (\ldots) \in succ$ each of the following three templates must be instantiated. They serve to erase the contents of one channel by *iterating* over all the identities in the channel. This iteration cannot be coded naturally in terms of a partner graph grammar. Rather, an artificial label must be introduced. This role is played by the $Q \times \chi \times \mathbb{N}$ construct. The process in state q assumes the artificial state (q, c, n) (indicated by the shaded circle) in

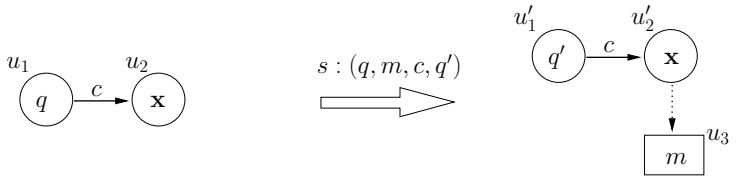
order to denote that it is resetting channel c. The natural number n is simply a flag. It is used to distinguish between several iterations that may happen at the same time – even in an interleaved fashion. (The c and the n do not occur in the figure for brevity.) As the artificial label is thus guaranteed not to occur anywhere else, it is ensured that the process in state q cannot do anything else but resetting channel c. After there is no identity left on this channel, as modeled by the partner constraint in template [TRESET3], the involved process may eventually assume the specified state q'.



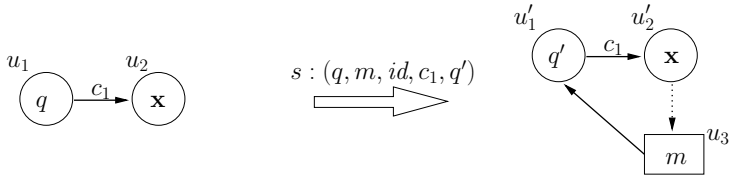
To summarize, partner graph grammars and graph grammars in general do not have a natural way of encoding complex control flow. Control-flow points need to be modeled explicitly by additional labels, which seems rather artificial. The loop encoding rules [TRESET1-3] may be abbreviated as *loop*$(c, [\text{TRESET1'}])$, where the latter rule equals [TRESET1] except for replacing the label of u'_1 by q' instead of q. The meaning of this construct is to expand rule [TRESET1'] in the same fashion as done above.

The *loop* construct shall be used in the following three templates as well. They are instantiated for each tuple $s : (\ldots) \in succ$ and model the sending of a message without and with an attached identity. In the latter case, there is one rule template for attaching the sender's identity and one for attaching an identity known to the sender. The dotted arrows denote links into the message buffer, *i.e.* *buf*-labeled edges. Solid arrows without an attached label link messages to their parameter, formally they are labeled *par*. Arrows denoting channels are drawn as solid arrows with their label, *i.e.* their channel name, attached. Remember that the bold variables like $\mathbf{x}$ or $\mathbf{y}$ are label variables and may denote any node label in $\mathcal{N}_{\text{DCS}}$. For the sake of readability nodes representing message are drawn as boxes.

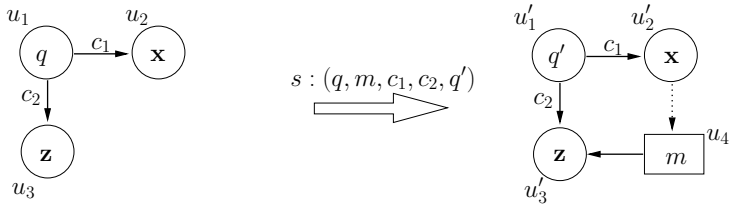
[TSEND1]



[TSEND2]



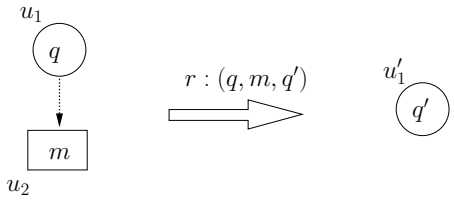
[TSEND3]



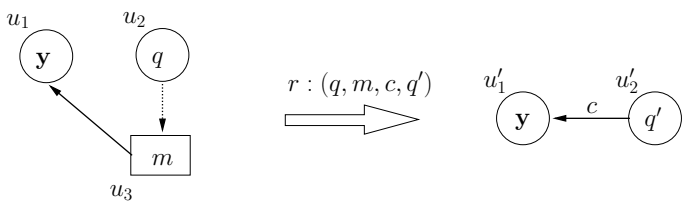
As stated, these rules would only send a message to *one random* process in a channel rather than to broadcast it. Therefore, the [TSEND] rules must be replaced by looping constructs: $loop(c, [\text{TSEND}\imath])$ for $\imath = 1, 2, 3$. Notice that whenever such a looping template is instantiated, a new flag must be introduced, in order not to make the application of these rules interfere.

Analogously to the send case but without loops, there are two rules managing the reception of messages. In contrast to the second case, no identity is attached to the received message in the first case. There is an instance of one of these rules for each occurrence of a $r : (\ldots)$ tuple in *succ*.

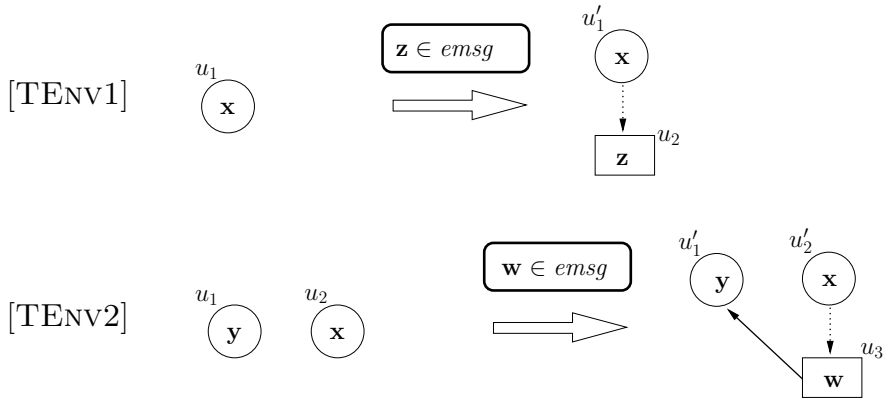
[TREC1]



[TREC2]



Another case of non-deterministic environment action to abstract from physical and hybrid influences in the real system is demonstrated by the next two rule templates. At any time, the environment may send one of the environment messages **z** to an arbitrary process **x**. It may also attach an arbitrary identity **y** to the message. In the platoon example below, this facility is employed to inform cars of cars ahead that are potentially willing to merge.



A Platoon Instance As an example of an instantiated partner graph grammar template, an example from [BSTW06] is chosen. Only the syntactic ingredients are mentioned. They define all necessary information for the instantiation. This instantiation is a platoon example that uses asynchronous, buffered communication via named channels, instead of shared global memory as presented in [HESV91]. The components are:

- States $Q = \{fa, ld, req, hnd, hnd', er, er', clr, fl, ann\}$.
- Initial state $A = \{fa\}$.
- Fragile states $F = \{fa, fl\}$.
- Channels $\chi = \{ldr, flws\}$, one is used to save the identity of the (potential) leader of the car, the other one to save identities of followers.
- Messages $msg = \{car\_ahead, request, new\_leader, new\_flwe\}$.
- Environment messages $emsg = \{car\_ahead\}$.

The only missing ingredient of the instantiation is the successor relation. It is presented below and comes with a detailed explanation of the meaning of the local states.

State *fa* This state models free agents driving on their own. The following transitions are possible from there.

- $r : (fa, car\_ahead, ldr, req)$: The environment has announced a car ahead to a free agent triggering the beginning of a merge maneuver. The rear free agent receives this message and changes its state to *req*. The identity attached to the message is saved in the leader channel *ldr*.
- $r : (fa, request, flws, ld)$: A free agent may receive a *request* message requesting a merge. In this case, the free agent drives ahead of another platoon or another free agent and it has been announced to the respective rear free agent or rear leader by the environment. It adds the identity attached to the message to its followers channel *flws*.

State *ld* The state that a leader of a platoon assumes.

- $r : (ld, car\_ahead, ldr, req)$: This tuple corresponds to the first tuple described for state *fa*. A car has been announced, its identity is saved in the *ldr* channel.
- $r : (ld, request, flws, ld)$: A leader in front may receive a request to merge from a car driving behind it. In such a case, it adds the attached identity to its follower channel *flws*.
- $r : (ld, new\_flwe, flws, ld)$: In order to finish a merge maneuver, the followers of the rear platoon announce themselves to their new leader by sending *new_flwe* messages. By receiving them, the latter adds all those identities to its follower channel *flws*.

State *req* This is the state of a process that was announced a car ahead by the environment. Originally, this process may have been a leader or a free agent. It uses this state to process the merge in the role of a rear leader. The only transition out of this state is to send a request message to the leader (or free agent) in front, that is stored in the *ldr* channel – $s : (req, request, id, ldr, hnd)$. The requesting car adds its own identity to the message.

States *hnd*,*hnd′* A process in this state hands over its followers to another leader. It is assumed by former leaders or free agents approaching another platoon from behind after requesting a merge from the car ahead. The hand over is handled by broadcasting a new leader announcement, first without the new leader identity stored in channel *ldr*

– $s : (hnd, new\_leader, flws, hnd')$. The subsequent state hnd' repeats this message once more, but this time the new leader identity information, *i.e.*, the identity of the leader or free agent ahead as stored in the *ldr* channel, is broadcasted to all followers, before the sending process assumes state $clr - s : (hnd', new\_leader, ldr, flws, clr)$

State *clr* In this state, the rear leader that was a leader or free agent originally clears its *flws* channel and becomes a follower itself: $e : (clr, flws, fl)$.

State *fl* The only thing a follower can do is receiving the announcement of a new leader. The new leader is announced by its former leader by means of a *new_leader* message. In that case the follower changes its state to *er* for further proceeding. Formally, the tuple looks like this: $r : (fl, new\_leader, er)$.

States *er*, *er′* Once in state *er*, the follower being about to get a new leader, deletes its old leader information from channel $ldr - e : (er, ldr, er')$. When in state er' it is able to receive the second broadcast of the new leader information that is equipped with the new leader identity. The *ldr* channel is then set to contain this freshly received identity – $r : (er', new\_leader, ldr, ann)$.

State *ann* The last thing to do for a follower that has just got to a new leader is to announce itself to this new leader, so that the latter can add it to its own follower channel *flws*. This issue is taken care of by a *new_flwe* message. After that, the follower can assume the proper follower state again. Formally, this yields the tuple $s : (ann, new\_flwe, id, ldr, fl)$.

Sample Run Plugging the detailed rules from the previous paragraph into the partner graph grammar template, the partner graph grammar $\mathfrak{G}_{\text{DCS}} = (\mathcal{R}_{\text{DCS}}, \mathcal{I}_{\text{DCS}})$ is obtained, where the initial graph is set to be the empty graph. A sample derivation is shown in Figure 3.2. Notice that the design of $\mathfrak{G}_{\text{DCS}}$ deviates slightly from $\mathfrak{G}_{\text{ideal}}$ due to the presence of named channels. Also, in the DCS specification language of [BSTW06] there is no means of making transitions dependent on objects (cars, processes) one is linked to. This is of course due to the asynchronous nature of this specification formalism, which is reflected in the partner graph grammar.

Sample Properties To conclude Section 3.1.2, two sample properties are stated in the $\mathcal{GL}$ logic of Section 2.2.

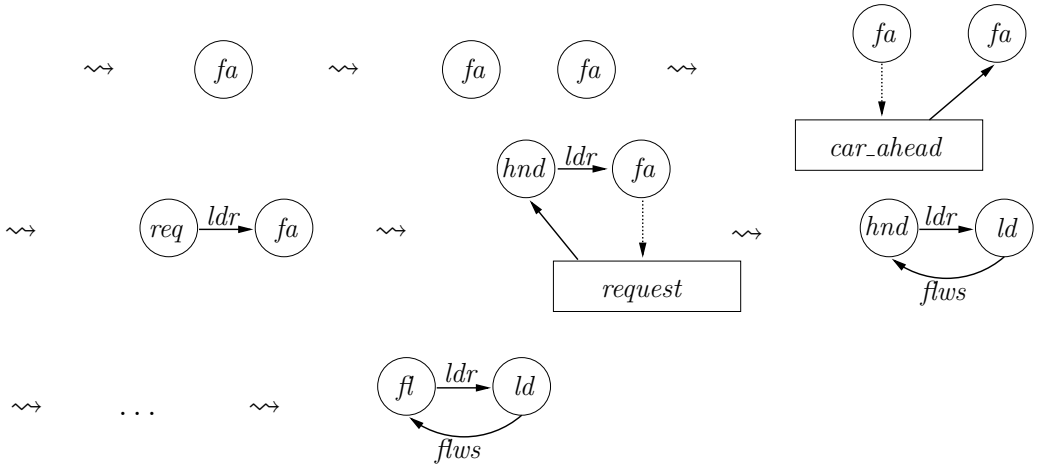


Fig. 3.2: Sample derivation of the DCS grammar $\mathfrak{G}_{\mathrm{DCS}}$. First two free agents are created. One is announced to be ahead of the other by a car ahead environment message featuring an identity. After that, the rear car request a merge adding the front car to its leader channel. The latter accepts and adds the rear car to its followers. The dots indicate the hand over of the non-existent followers that imply only state changes of the rear car. Eventually, a proper platoon of size two is established.

$$\mathsf{AG}\,\mathsf{EF}\,\forall x.\forall y.\neg(\mathit{buf}(x,y) \vee \mathit{par}(x,y)) \tag{3.3}$$

$$\mathsf{AG}\,\forall x.\mathit{ld}(x) \vee \mathit{fa}(x) \vee \exists y.(\mathit{ld}(x,y) \wedge \forall z.\mathit{ld}(x,z) \rightarrow y = z) \tag{3.4}$$

Property (3.3) expresses that each message buffer will eventually be emptied. In an unrestricted system, it may certainly be falsified by the environment swamping a process. It remains to be shown, whether the analysis techniques to be presented are able to find this out. It is certainly desired, that each car that is not a free agent or follower has a *unique* leader. This is formally expressed by property (3.4).

3.1.3 Faulty Channels

Until now, dynamic object creation, destruction, evolving communication topologies, and asynchronous, buffered, message-based communication were added to the static platoon scenario first described in [HESV91] and in detail in the beginning of Section 3.1. That already proved the usability and expressiveness of partner graph grammars and the specification logic $\mathcal{GL}$.

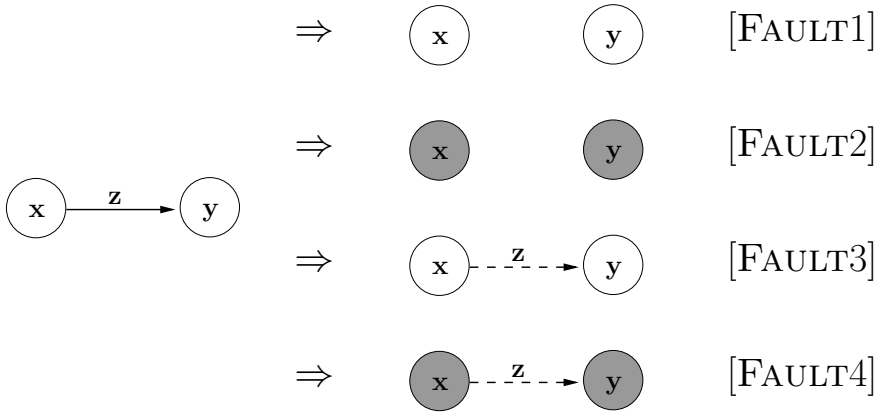


Fig. 3.3: Four ways to deal with broken communication (from top to bottom): (1) The link simply disappears. (2) The link disappears but the communication partners are notified. (3) Partners are not notified but a logical connection remains intact. (4) Object notification in combination with logical links.

In this section, the achievements are further strengthened by incorporating features that allow reasoning about *faultiness* of communication.

This section centers around the possibility of *faulty channels*. Again, channels are used as a synonym of the more general term links. The source of all faultiness thus lies in the spontaneous disappearance of edges. However, there should be means for a system to recover from such faults. Two dimensions for error recovery are proposed here. First, there may or not may be a *notification of objects* about losing a link. The second dimension introduces two sorts of channels: ordinary ones to model *physical* connections and *logical* links. The latter are mostly parallel to the communication links, but prevail after a communication breaks down. This corresponds to the intuition, that an object knows some sort of *address* of its communication partners modeled by logical links. This knowledge may not be lost, when the physical link collapses. Figure 3.3 illustrates four possible ways of introducing faulty channels into a graph grammar by adding one of the [Fault$1-4$] rules.

If one aims at investigating the fault resistance of a given graph grammar $\mathfrak{G} = (\mathcal{R}, \mathcal{I})$ over the sets $\mathcal{N}$ and $\mathcal{E}$ of node and edge labels, the following procedure is proposed depending on the choice of the recovery model. Equip $\mathfrak{G}$ with additional fault rules to create a faulty system *faulty*($\mathfrak{G}$). Consider the rules in Figure 3.3, where the correspondence between nodes in left and right graphs is given implicitly by positions. The domain of the label variables are

explained in detail below.

- If one opts for no notification and no logical links, the transformation rule [FAULT1] is added to $\mathcal{R}$. The sets of node and edge need not be adjusted. The label variables range over the complete set of labels, since an edge may disappear regardless of its label and its incident nodes.

- The notification/no logical links option is obtained by adding rule [FAULT2] to the rules of $\mathcal{R}$. Furthermore, the resulting partner graph grammar $\mathit{faulty}(\mathfrak{G})$ is over the set $\mathcal{N} \times \{\mathit{good}, \mathit{bad}\}$ to distinguish ordinary (good) states from those (bad) indicating the loss of a channel. In rule [FAULT2], the notion of label constraints is slightly abused, because $\mathbf{x}$ and $\mathbf{y}$ in the left graph range over $\mathcal{N} \times \{\mathit{good}\}$, whereas a shaded node labeled $\mathbf{x}$ in the right graph actually stands for $(\mathbf{x}, \mathit{bad})$. Certainly, $\mathbf{z}$ ranges over $\mathcal{E}$.

- The third option introduces *logical* links between objects that model the knowledge of another object even after the physical connection is lost. Logical links are introduced whenever a physical connection collapses. Since objects are not notified, the set of node labels prevails, whereas the set of edge labels is extended to be $\mathcal{E} \times \{\mathit{physic}, \mathit{logic}\}$. Rule [FAULT3] reads as follows – again abusing the notion of label constraints. The label variables for nodes range over $\mathcal{N}$. Label variable $\mathbf{z}$ ranges over $\mathcal{E} \times \{\mathit{physic}\}$ in the left graph and over $\mathcal{E} \times \{\mathit{logic}\}$ in the right graph – shown as a dashed edge labeled $\mathbf{z}$.

- The fourth option simply combines object notification and logical links. Hence the label sets are $\mathcal{N} \times \{\mathit{good}, \mathit{bad}\}$ and $\mathcal{E} \times \{\mathit{physic}, \mathit{logic}\}$, respectively. Rule [FAULT4] is the obvious mixture of [FAULT2] and [FAULT3].

Running the analysis proposed in later chapters reveals the behavior of the system specified by $\mathfrak{G}$ when faulty channels are introduced. An advantage of the method proposed in this section, is that it offers multiple models of error recovery to be picked by the user. Certainly, the designer of a fault tolerant system must add more rules to the original graph grammar to cope with error recovery. It must be said that the aforementioned transformation rules only introduce faulty channels and the *possibility* to overcome them. Concrete measure against faults must be taken by the system designer. However, much insight into the behavior of a faulty system can be gained by applying this method. It may thus be proposed as a general way to analyze faulty systems.

3.2 Further Applications

The car platooning scenario serves as a complex, prototypical instance of the main application domain of this work: dynamic communication systems. All the car platooning scenarios will be evaluated in detail in Section 4.5. In contrast, this section lists a number of other potential, promising application domains that are to be explored in the course of future work.

Heap Manipulating Programs The best-known application domain that involves evolving graphs is the domain of heap-manipulating programs, where heaps are represented as graphs. Similar to dynamic communication systems, destructive pointer updates can be characterized in a local fashion. However, the influence on shared nodes, *e.g.* the creation of dangling references, is not truthfully modeled by the single pushout approach (Section 2.3), where edges adjacent to disappearing nodes disappear, too, and do not remain dangling.

Figure 2.2 on page 15 shows examples of this application domain. Due to the locality of updates, small left graphs in rules are expected, which eases the effort of computing matches. Also the general advantage of using graph grammars applies to this application domain: Both specification of a program and an analysis result are easily visualizable.

The biggest problem with this application domain is modeling control flow in terms of graph grammars. As seen in the queue implementation, control flow constructs like iteration over the elements in a container or even procedures cannot be implemented naturally in graph grammars. Certainly, they can be encoded or imposed upon a rule-based core. At least for the verification techniques presented here, this yields a considerable loss of precision of the analysis results, as will be shown in the experimental evaluation in Section 4.5.

Another drawback – not of graph grammar based verification in general, but of partner abstraction as introduced in Chapter 4 – is the lack of features to reason about reachability within graphs. While this is not crucial for the application domain of dynamic communication systems, reachability is essential to be able to reason about in heap-manipulating programs. This is the reason why transitive closure logics are popular in this research area.

The three-valued logic based approach to verification of heap-manipulating programs as surveyed in [SRW02] has emerged as a de-facto standard in this application domain. It is capable of proving very advanced properties of heap-manipulating programs, such as the correctness of list-sorting algorithms [LARSW00] or safety properties of multi-threaded Java [Yah01]. Also interprocedural programs [RBR+05] or numeric analyses of array operations

[GRS05] are in the scope of this very general technique. It is thus very hard to compete with this technique in terms of general applicability.

However, improvements over this technique may be possible in terms of comprehensibility, ease of use, and scalability. The specification of an analysis in the [SRW02] framework is tedious and error-prone, because one needs to write involved predicate update formulas instead of simple graph transformation rules. The actual analysis using the TVLA [LAS00] tool is often very time-consuming.

Therefore, there is some hope to establish graph grammar based methods as a complementary technique besides TVLA based analyses. This work presents a step in this direction as well as analyses based on [BCK04] that, *e.g.*, allow for the analysis of red-black trees [BCE+05]. The approach closest to the one here is presented in [RD06]. More graph grammar based verification techniques are reviewed in Chapter 5.

Process Calculi One traditional way of specifying communication systems is the use of process calculi like CCS [Mil80], CSP [Hoa78], or variants of the π-calculus [SW01]. The main reason for not using them in this work is that communication topologies are to be represented explicitly using graphs. For case studies like the platoon scenario, graph transformation rules are a more natural and more high-level way of coding.

As process calculi occur so frequently and due to their close relationship to graph grammars, they present an obvious application domain for the verification techniques presented here. Many interesting applications modeled in a process calculus would become immediately amenable to the verification method of this work, if there was an automatic, semantic-preserving translation of process calculi into partner graph grammars. A hypergraph rewriting semantics of a π-calculus variant was given in [Kön00]. However, there is still a significant gap between that semantics and partner graph grammars. The major problem in this context may be the interpretation of π's replication operator. As the natural and obvious interpretation involves an unbounded number of transformation rules in the grammar, which is clearly not desirable.

Even though more work needs to be invested in the application domain of process calculi, it remains an interesting task to make the techniques of this work applicable to them and compare the results to related analyses of process calculi like control flow analyses in the style of [BDNN98].

Peer-to-Peer and Ad Hoc Networks The most promising and at the same time most sophisticated and most distant application deals with the

analysis of communication topologies in peer-to-peer or mobile ad hoc networks. In order to convey an impression of the application domain, a typical example is given. It originates in the Safari project at Rice University and presents a self-organizing hierarchical routing protocol for ad hoc networking. The protocol is formally described in [DKC+04]. It should become obvious from the description that the protocol can be modeled using partner graph grammars. However, the probabilistic features of it need to be abstracted by non-determinism. The same abstraction must be applied to concrete, physical information that is broadcasted over the network.

The hierarchy is formed as a recursive organization of nodes in the network into cells. The set of nodes is partitioned into cells, the set of cells is partitioned into super-cells, and so on. In general, a level k cell is partitioned into level $k-1$ cells. Within each level i cell there is a level i *drum*. Each level i drum, which is equipped with a unique identity, identifies a level i cell. At the same time, a level k drum is a level i drum for $i \leq k$, too. The *hierarchical address* of a node id is of the form $d_k.d_{k-1}.\ldots.id$, where d_i is the level i drum identifying the level i cell, in which node id is located.

Each drum disseminates periodic *beacon* packages containing information about locality, hierarchy, and routing. This information is forwarded by all nodes within a certain number of hops from the drum. A node stores all the beacons it forwards in a routing table. For sending a data packet, routing is performed according to the hierarchical address of the packet's source and destination. Routing happens towards the drum of the destination node's cell at each level of recursion.

This completes the description of the organization of the network. For brevity and because the protocol will not be analyzed formally, some details were left out in the description, *e.g.* the self-selecting algorithm of a level i drum among level $i-1$ drums. The purpose of this example is simply to convey the flavor of this particular application domain.

As with process calculi and heap-manipulating programs, left graphs of transformation rules are expected to be reasonably small. From a specification point of view, the problem consists rather of the numerous physical, *i.e.*, numerical, data broadcasted over the network. Also, failure models are typically described in terms of probability distributions. Both data and probability distributions may be abstracted using non-determinism. However, this may yield a very coarse abstraction.

There are two possible solutions to these modeling problems. First, stochastic graph grammars as proposed in [HLM04] may help to model stochastic failure behavior. This feature was used in the simulation of peer-to-peer networks based on graph grammars in [KL06]. The latter work does

not aim at verification, though. Rather, systems are simulated and evaluated statistically.

The problem of data may be addressed using attributed graphs [LKW93]. This should be rather straightforward when it comes to modeling a system like Safari routing. On the other hand, it will be more complex to analyze. There is some hope, however, that the tasks of analyzing the graphs and their attributes may be separated. This may also be a promising approach in the queue-based platoon scenario stated in Section 3.1.2, where queues will be shown to be a mischief-maker in Section 4.5.2.

Chapter 4

Partner Abstraction

Partner graph grammars have been introduced in Chapter 2 and proven useful in Chapter 3. The technical core of this work is presented in this chapter.

After a brief introduction to the underlying verification technique, abstract interpretation, a novel, two-layered abstraction of graphs is defined: partner abstraction. The application of transformation rules will be lifted from graphs to abstract graphs. This gives rise to the notion of an abstract graph semantics, which will be shown to be a sound over-approximation of the graph semantics of a partner graph grammar. Some statically checkable cases are identified, where the obtained information can be shown to be complete. For this purpose, three novel completeness notions are introduced. After that, the meaning of $\mathcal{GL}$ properties is defined for abstract graph semantics'. It is investigated, to what extent properties that hold of a partner graph grammar (more precisely, of its induced graph transition system) also hold of an abstract graph semantics. The abstract interpretation of partner graph grammars has been implemented in the `hiralysis` tool. It is used to implement and experimentally evaluate the platoon case studies of Section 3.1. The experiments and experiences gained from them are reported, before this chapter concludes by giving an outlook to further extensions to partner abstraction that may make the approach either more precise, more scalable, or more parameterized.

4.1 Abstract Interpretation

Initially, some foundations of abstract interpretation are presented. This technique was originally invented in two seminal papers by P. and R. Cousot [CC77, CC79]. A good survey reference is the book [NNH99], where Chapter

4 treats the theory of abstract interpretation. In this work, only the absolute necessities of the underlying theory are presented. Later on, the particular abstraction employed to analyze partner graph grammars is motivated and introduced. It is called *partner abstraction*. It maps *concrete* graphs to *abstract* graphs in two steps. Matching and application of transformation rules are lifted to work on abstract graphs. The key idea is to locally undo the abstraction by *materialization* and to then apply the same transformation rule as defined for the concrete case. In order to guarantee boundedness of the abstraction, another abstraction step follows the update.

One central theorem of Section 4.1 is called the *Matching Theorem*. It relates concrete and abstract matches. The second major theorem concerns the *soundness* of the proposed analysis. It is a trivial consequence of the embedding in the abstract interpretation framework. It needs to show that abstractions are preserved by *abstract transformers*. The soundness theorem implies that the computed abstract graph semantics is a true over-approximation of the concrete graph semantics, where the graph semantics is the set of all graphs generated by a partner graph grammar. As a consequence, certain undesired graphs may be proven not to occur in the concrete system by proving their absence in the abstract system.

4.1.1 Introduction to Abstract Interpretation

A static analysis finitely analyzes a program in advance of the program execution and extracts useful information from it. Abstract interpretation is a fundamental framework that can help to formulate and prove correct a static analysis. In terms of this work, programs are partner graph grammars and program executions are graph transition systems induced by partner graph grammars. It should be said, that this introduction to abstract interpretation is brief and not formal. The purpose is rather to set the stage for the later sections of this chapter. All material in them is self-explanatory. There is a huge amount of material on abstract interpretation. A good recommendation to start digging into it are the seminal papers [CC77, CC79] that laid the foundation of abstract interpretation and Chapter 4 of [NNH99] providing a solid introduction.

Abstract interpretation builds upon the notion of an *abstraction*. An abstraction is a value from some abstract domain and represents a set of concrete values. Concrete values are the semantic domain, on which actual computations of a program happen. In this work, an *abstract graph* will represent a set of concrete graphs. Abstract graphs are defined in Section 4.1.2. The dual of abstraction is concretization. The concretization of an abstract value is the set of all concrete values that are represented by this abstract

value. Abstractions and concretizations are formalized in terms of ordered sets and lattice theory. A good textbook on these subjects is [DP05]. The central notion relating abstraction and concretization is the notion of a *Galois connection* between two complete lattices, one representing the concrete values, the other representing the abstract values. As Galois connections will not occur explicitly in the remainder of this work, their precise definition is omitted.

Abstraction is only the first part of the story of abstract interpretation. The second part deals with *abstract transformers*. Abstract domains are typically chosen to be much simpler than concrete domains. This increases the chances that computations on the abstract domain may be tractable or at least decidable, whereas computations on the concrete domain are not. In the setting of partner graph grammars, computations on the concrete domain are essentially direct derivation steps between graphs.

Abstract transformers correspond to computations on the abstract domain. An abstract transformer $f^\#$ is a *sound over-approximation* of a concrete transformer f, if and only if

$$\alpha \circ f \sqsubseteq f^\# \circ \alpha \quad \Leftrightarrow \quad f \circ \gamma \sqsubseteq \gamma \circ f^\# \tag{4.1}$$

assuming abstraction mapping α and concretization γ along with some orders on lattices of computations. This means that computing on the abstract domain is always a *conservative* approximation of the computation on the concrete domain. Additional behavior or properties may be introduced, but none are forgotten.

The *best abstract transformer* can be defined in terms of abstraction, concretization, and concrete transformer f to be $\alpha \circ f \circ \gamma$. Unfortunately, the best abstract transformer is not computable in general. Section 4.1.5 defines the notion of abstract transformers working on abstract graphs. They are proven to be sound.

Completeness is another notion to be investigated in the context of abstract interpretation. It means, that no information is lost at all by abstraction. It amounts to replacing the inequations in (4.1) by equations. An abstract transformer $f^\#$ is complete, if it does not matter, whether its corresponding concrete transformer f is applied to some value v and the result is then abstracted, or whether v is abstracted before the abstract transformer is applied. Various notions of soundness results for the abstract interpretation of partner graph grammars are studied in Section 4.3.

The theory of abstract interpretation is certainly much richer than presented in this brief appetizer. However, it should be sufficient to get in the right spirit to digest the remainder of this chapter. All of the material to

come is self-explanatory without using involved abstract interpretation details, although familiarity with the theory may be helpful. A mnemonic for abstract interpretation is just: Compute on the abstract domain using abstract transformers instead of computing on the concrete domain.

4.1.2 Partner Abstraction of Single Graphs

The abstraction of single graphs works in two steps. One of them is inspired by a rather *local* way of looking at graphs, whereas the other one takes the *global* picture into account. The local abstraction is based on quotient graph building, and is consequently defined in terms of an equivalence relation on nodes. The global abstraction first partitions a graph into subgraphs, applies the local abstraction subgraph-wise, and summarizes those subgraphs that are isomorphic after local abstraction.

The choice of the local abstraction is driven by the main application domain: *dynamic communication systems*. In such systems most of the actions depend on *two* objects communicating with each other. It is thus conjectured that left and right graphs of transformation rules modeling dynamic communication systems have a small diameter. More often than not they may consist of two nodes only. Therefore, it is crucial to keep precise information related to the *immediate neighborhood* of nodes, and the name *partner abstraction* will be used to denote the abstraction. As the local abstraction is based on quotient graph building, it comes with an equivalence relation on nodes, called *partner equivalence*. Besides a similar neighborhood, equivalent nodes should have the same label.

Two nodes are *partner equivalent*, iff they have the same label and the sets of labels of their adjacent nodes are equal respecting the labels and the directions of incident edges.

The formal notion is introduced in Definition 4.1.1. It will be demonstrated that it is even useful in the analysis and verification of other systems. Most probably, this is the case, because humans seem to prefer local views, when designing a system. For instance, destructive pointer updates in heap-manipulating programs have a local flavor, since only single edges are moved at a time, involving the source, the original target and the new target of the edge. Certainly, the proposed technique is generally applicable to any partner graph grammar. This is a benefit of using the abstract interpretation framework. However, the precision of the results is expected to be smaller, in case of very involved restructurings within one transformation rule.

The motivation of the second, global abstraction step is again driven by the dynamic communication systems application domain, although it is useful in more general settings, too. Obviously, only objects connected by communication links may influence each others behavior. It is hence natural to consider *connected components* of graphs (or communication topologies) as a natural granularity for abstractions. To underline their prominent role, the special term *cluster* is coined for connected components.

As objects within distinct connected components cannot (not even indirectly) communicate with each other, a connected component is the center of the focus for the global abstraction. Furthermore, it abstracts from the number of clusters showing similar behavior, where similar behavior means being isomorphic after applying the local abstraction. Put together, abstraction of graphs works as follows.

1. Quotient graph building *wrt.* partner equivalence *for each* cluster (connected component). This step is also called *cluster abstraction*. The result of it is called an *abstract cluster*.

2. Summarize clusters that are isomorphic after cluster abstraction. This step is also called *cluster summarization* and yields an *abstract graph* being a set of abstract clusters.

3. The result of steps 1 and 2 is called *partner abstraction*.

Example Recall the platoon case study of Chapter 3, in particular the simple model $\mathfrak{G}_{\text{ideal}}$ of Section 3.1.1. Consider a graph with a number of platoons (not currently merging or splitting) and free agents. Such a graph will be abstracted to one abstract platoon and one abstract free agent. Each free agent is a connected component of its own and partner abstraction of a single node without edges does not change anything. All free agents are hence isomorphic after abstraction and summarized to one abstract free agent. All followers within the same platoon are partner equivalent, because they have the same label and they all have one outgoing link to a leader. An abstract platoon consists of one follower – standing for an arbitrary number of them – and one leader. As all cluster abstracted platoons are isomorphic, they are all summarized to yield the eventual abstraction. This process is visualized in Figure 4.1, where a concrete graph $T_1 \in [\![\mathfrak{G}_{\text{ideal}}]\!]$ is given in part (a) and its abstraction in part (b) of the figure. Notice that T_1 additionally comprises two merging platoons, where the followers of the two original leaders remain distinct in the abstraction. Although they have the same label, their adjacent nodes do not.

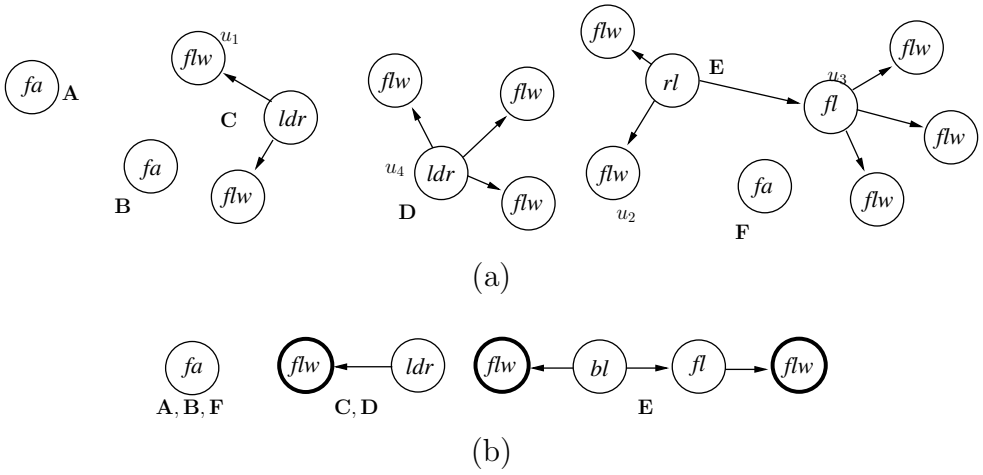


Fig. 4.1: Part (a) displays an element of the graph semantics of the idealized platoon case study as stated in Section 3.1.1. This graph will be called $T_1 \in [\![\mathfrak{G}_{\text{ideal}}]\!]$. Part (b) shows the partner abstraction of T_1. Connected components are denoted by bold capital letters. The graph is abstracted with $k = 1$. Summary nodes are drawn thickly rimmed.

Formalizations An essential notion used in the course of the introduction of partner abstraction is the notion of *partners*. Some notation is introduced to handily write down the set of adjacent nodes of node u of graph G while distinguishing whether connecting edges are *incoming* or *outgoing* and which labels these edges have. The operators $\lhd$ and $\rhd$ are used for this purpose. They are used in postfix notation together with a node. They are indexed by the current graph and superscripted by an edge label. The index is left out, when the graph is clear from the context. For instance, $u{\rhd}_G^\beta$ is the set of all nodes, to which u has an outgoing β-labeled edge in G. Formally, let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ and let $u \in V_G$. The sets of incoming and outgoing partners of u are defined as follows.

$$u{\lhd}_G^\beta = \{v \in V_G \mid (v, u) \in E_G^\beta\} \tag{4.2}$$

$$u{\rhd}_G^\beta = \{v \in V_G \mid (u, v) \in E_G^\beta\} \tag{4.3}$$

$$u{\lhd}_G = \bigcup_{\beta \in \mathcal{E}} u \lhd_G^\beta \tag{4.4}$$

$$u{\rhd}_G = \bigcup_{\beta \in \mathcal{E}} u{\rhd}_G^\beta \tag{4.5}$$

In the following definitions, as usual, application of a mapping $f : A \to B$

to a subset $M \subseteq A$ of its domain is defined pointwise: $f(M) = \{f(a) \mid a \in M\}$. As before, $\mathcal{N}$ and $\mathcal{E}$ denote arbitrary but finite sets of node and edge labels, respectively. Now, the notion of partner equivalence can be defined formally.

Definition 4.1.1 (Partner Equivalence) *Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$. Two nodes $u, v \in V_G$ are* partner equivalent*, written $u \bowtie_G v$, iff they have the same label and the sets of labels of their adjacent nodes are equal respecting the label and the direction of incident edges,*i.e.*, iff*

$$\ell_G(u) = \ell_G(v), \tag{4.6}$$
$$\forall \beta \in \mathcal{E}. \ell_G(u \triangleright_G^\beta) = \ell_G(v \triangleright_G^\beta), \text{ and} \tag{4.7}$$
$$\forall \beta \in \mathcal{E}. \ell_G(u \triangleleft_G^\beta) = \ell_G(v \triangleleft_G^\beta) \tag{4.8}$$

The graph index of $\bowtie$ is dropped, whenever it is clear from the context.

Recall that cluster abstraction, *i.e.*, quotient graph building *wrt.* partner equivalence, is applied *per cluster*, that is per connected component. If it was applied to a whole graph, there might be an undesired mix-up of nodes that are not even indirectly connected. If the result of the cluster abstraction *per cluster* were taken as the overall result, it might lead to an unbounded abstraction, because there might be an unbounded number of connected components in the graphs under consideration. These arguments motivate the subsequent step of summarizing clusters that are isomorphic after cluster abstraction.

If partner abstraction was implemented as indicated, a certain amount of isomorphism checks would become necessary. That is clearly undesirable. A trick from [SRW02] helps to overcome this problem: *canonical naming*. Every possible equivalence class *wrt.* partner equivalence may be identified by a pair of a node label – corresponding to requirement 4.6 of Definition 4.1.1 – and a *partner set* – corresponding to requirements (4.7) and (4.8). A partner set is a subset of $\{in, out\} \times \mathcal{E} \times \mathcal{N}$.

If two equivalence classes coincide on pairs of labels and partner sets, they may still differ on the identity and the number of nodes contained in them. If node identities are neglected for the moment, there is still the possibility of *finite counting* to distinguish the numbers of elements of equivalence classes up to some natural number k. The foundations of finite counting were introduced in connection with partner constraints in Definition 2.1.7 of Section 2.1. The number of elements in an equivalence class will also be called its *multiplicity*. An equivalence class with more than one element shall be called a *summary node* (reminiscent of [SRW02]).

Label, partner set, and multiplicity make up the notion of a *canonical name* (again reminiscent of [SRW02]). Canonical names are the key to gain a unique representation of abstract graphs by restricting the nodes of abstract graphs to be canonical names themselves. Recall the notion allowing to restrict the node set of a graph to subsets of W, $\mathcal{G}(\mathcal{N}, \mathcal{E}, W)$ defined in Definition 2.1.1. *Canonical graphs* are thus graphs, whose nodes are canonical names. After stating the formal definition of canonical names and graphs, it will be shown, that they yield a unique representation of abstract graphs.

Definition 4.1.2 (Canonical Names and Graphs) *Let $\mathcal{N}$, $\mathcal{E}$, and $k \geq 1$ be finite sets of node and edge labels, and a natural number, respectively. The set $\mathcal{N} \times \wp(\{in, out\} \times \mathcal{E} \times \mathcal{N}) \times \mathbb{N}_k$ is called the set of* k-canonical names over $\mathcal{N}$ and $\mathcal{E}$ *and is written canonicalNames$(\mathcal{N}, \mathcal{E}, k)$. An element of $\wp(\{in, out\} \times \mathcal{E} \times \mathcal{N})$ is called a* partner set.

The set $\mathcal{G}(\mathcal{N}, \mathcal{E}, canonicalNames(\mathcal{N}, \mathcal{E}, k))$ of graphs over $\mathcal{N}$ and $\mathcal{E}$ that have canonical names as nodes is called the set of canonical graphs *and is written $\mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k)$.*

The third component n of a node (ν, P, n) of a canonical graph is called the multiplicity *of the node. Nodes with multiplicities $n \neq 1$ are called n-*summary nodes.

Notice the close relation between partner constraints as defined in Definition 2.1.8 and partner sets. The abstraction will be designed as to precisely maintain the satisfaction of partner constraints and the applicability of rules.

The notion of canonical graphs opens up for a more concise and easier to implement notion of partner abstraction than before. Let G be a graph. First, for each cluster (connected component) C of G, replace each node with the triple $(\nu, P, 1)$ of its label ν, its partner set P, and its multiplicity 1. This replacement is handled by the function *partner*. In order for the node set to keep its cardinality, it must be made a multiset, because several nodes may be replaced with the same triple.

The function *collapse* makes this multiset a set by summarizing equal nodes, *i.e.*, equal canonical names. At the same time, it counts the number of nodes summarized up to some finite k. Effectively, this constitutes a graph homomorphism between two canonical graphs. The *collapse* function will be used again in the definition of abstract update. An abstract cluster $\hat{C}$ is obtained from C by first applying *partner* and then *collapse*. The abstract graph $\hat{G}$, being the partner abstraction of G, is then simply the set of all abstract clusters obtained from the clusters of G. In this scenario, equality under canonical naming does the job instead of isomorphism. Isomorphic abstract clusters are in fact equal as justified formally by Lemma 4.1.2 below.

The upcoming three definitions define the *partner* and *collapse* functions, as well as the notion of abstract clusters and graphs obtained by using these two functions. An illustrative example follows after the definitions. These functions are used both for the abstraction of concrete graphs and the renormalization of an already abstract graph.

Definition 4.1.3 (Partner) *Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph and $k \geq 1$ a natural number. The mapping*

$$partner_{G,k} : V_G \to canonicalNames(\mathcal{N}, \mathcal{E}, k)$$

is defined as

$$\begin{array}{l} partner_{G,k}(u) := \\ \quad (\ell_G(u),\ \bigcup_{\beta\in\mathcal{E}}\{out\} \times \{\beta\} \times \ell_G(u{\triangleright}^{\beta}_{G})\ \cup\ \bigcup_{\beta\in\mathcal{E}}\{in\} \times \{\beta\} \times \ell_G(u{\triangleleft}^{\beta}_{G}),\ n) \end{array}$$

where the multiplicity n is 1, if $u \notin canonicalNames(\mathcal{N}, \mathcal{E}, k)$, and q, if $u = (\nu, P, q)$.

The mapping $partner_k : \mathcal{G}(\mathcal{N}, \mathcal{E}) \to \mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k)$ lifts $partner_{G,k}$ to whole graphs as follows for all $\beta \in \mathcal{E}$, where $partner_k(G) = H$.

$$\begin{array}{rcl} V_H & = & \{\!\{partner_{G,k}(u) \mid u \in V_G\}\!\} \\ E^{\beta}_H & = & \{\!\{(partner_{G,k}(u), partner_{G,k}(v)) \mid (u,v) \in E^{\beta}_G\}\!\} \\ \ell_H & = & \lambda(\nu, P, n).\nu \end{array}$$

Notice that *partner* may be applied to nodes in a canonical graph. Typically, this will be the case during the application of a transformation rule to an abstract graph. In such a case, *partner* recomputes label and partner set of a node, while maintaining its multiplicity. The $\{\!\{\cdot\}\!\}$ notation in Definition 4.1.3 is used to do implicit renaming of equal nodes as elaborated on page 8. It is necessary to maintain the original graph structure under renaming by the *partner* function. The real quotient graph building is performed by the *collapse* mapping defined now:

Definition 4.1.4 (Collapse) *Let $k \geq 1$ be a natural number. The mapping $collapse : \mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k) \to \mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k)$ is defined as follows for all $\beta \in \mathcal{E}$, where $collapse_k(G) = H$.*

$$\begin{array}{rcl} V_H & = & \{(\nu, P, n) \mid \exists q \in \mathbb{N}_k.(\nu, P, q) \in V_G\ \wedge\ n = \bigoplus^{k}_{(\nu,P,n')\in V_G} n'\} \\ E^{\beta}_H & = & \{((\nu, P, n), (\nu', P', n')) \in V_H \times V_H \mid \\ & & \qquad \exists q, q' \in \mathbb{N}_k.((\nu, P, q), (\nu', P', q')) \in E^{\beta}_G\} \\ \ell_H & = & \lambda(\nu, P, n).\nu \end{array}$$

The mapping $Collapse_k$ lifts $collapse_k$ to the pointwise application to sets or multisets.

Let $\hat{G}$ and $\hat{H}$ be multisets of clusters such that $Collapse_k(\hat{G}) = \hat{H}$. The induced morphism $\zeta : V_{\hat{G}} \to V_{\hat{H}}$ is defined as follows, where $\hat{C} \in \hat{G}$, $(\nu, P, n) \in V_{\hat{C}}$ and $collapse_k(\hat{C}) = \hat{C}'$.

$$\zeta(\hat{C}, (\nu, P, n)) := (\hat{C}', (\nu, P, \oplus^k\{n' \mid (C, (\nu, P, n')) \in V_C\}$$

A canonical graph G is called ground*, iff for all $k \geq 1$ $collapse_k(G) = G$. The set of all connected, ground, canonical graphs is called the set of* abstract clusters *and written $\mathcal{C}(\mathcal{N}, \mathcal{E}, k)$.*

In order to distinguish equal nodes (canonical names) from different abstract clusters, the canonical name is often paired with the abstract cluster. Note that the morphism property of the induced mapping ζ is obvious by definition. Abstraction of graphs may be defined by simply concatenating *partner* and *collapse* and applying it cluster-wise, *i.e.* connected component-wise. The step of cluster summarization comes for free due to canonical naming. Still, abstraction is parameterized by the natural number k indicating up to which number the analysis shall count.

Definition 4.1.5 (Abstract Clusters and Graphs) *Let $k \geq 1$ be a natural number. The* k-abstraction *$\alpha_k : \mathcal{G}(\mathcal{N}, \mathcal{E}) \to \mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k)$ of a graph G is the set*

$$\alpha_k(G) := \{collapse_k \circ partner_k(C) \mid C \in cc(G)\}$$

The set $\alpha_k(G)$ is called an abstract graph*. Its elements are called* abstract clusters*.*

A node in an abstract graph is a pair of a cluster and a canonical name.

Note that an ordinary set (as opposed to a multiset) is used in the definition of α_k. Lemma 4.1.2 justifies this, because isomorphic abstract clusters are in fact equal and thus summarized, if put into a set. First the abstraction is shown at work using the example of Figure 4.1 on page 68. A concrete graph featuring clusters $\mathbf{A} - \mathbf{F}$ is displayed in part (a) of that figure. Some applications of *partner* and *collapse* illustrate the abstraction. Let $k \geq 1$ be arbitrary.

$$\begin{array}{rcl} partner_{\mathbf{C},k}(\mathbf{C}, u_1) & = & (\mathit{flw}, \{(\mathit{in}, \_, \mathit{ldr})\}, 1) \\ partner_{\mathbf{E},k}(\mathbf{E}, u_2) & = & (\mathit{flw}, \{(\mathit{in}, \_, \mathit{rl})\}, 1) \\ partner_{\mathbf{E},k}(\mathbf{E}, u_3) & = & (\mathit{fl}, \{(\mathit{in}, \_, \mathit{rl}), (\mathit{out}, \_, \mathit{flw}), \}, 1) \\ partner_{\mathbf{D},k}(\mathbf{D}, u_4) & = & (\mathit{ldr}, \{(\mathit{out}, \_, \mathit{flw})\}, 1) \end{array}$$

The example shows that u_2 and u_3 will not be summarized to the same equivalence class, because they are not partner equivalent. Notice the notation of nodes of abstract graphs as pairs of cluster and node within a cluster. This is necessary, because there may be equal nodes in distinct abstract clusters.

Two sample abstract clusters obtained from graph T_1 in Figure 4.1 demonstrate the effect of counting. Let $C_1 = \alpha_2(\mathbf{C})$ and $C_2 = \alpha_2(\mathbf{D})$. The node sets of C_1 and C_2 are:

$$\begin{aligned} V_{C_1} &= \{(ldr, \{(out, \sqcup, flw)\}, 1), (flw, \{(in, \sqcup, ldr)\}, 2)\} \\ V_{C_2} &= \{(ldr, \{(out, \sqcup, flw)\}, 1), (flw, \{(in, \sqcup, ldr)\}, \infty)\} \end{aligned}$$

These sets show that the abstraction of clusters **C** and **D** of graph T_1 in Figure 4.1 are in fact distinct. Counting to $k = 2$ enables the analysis to distinguish them. In contrast to this, the abstraction of Figure 4.1 was obtained for $k = 1$. In that case, no distinction between platoons with two and three followers is possible, meaning that such platoons are summarized by cluster summarization.

The following lemma states that there is an induced morphism between a graph and its abstraction, *i.e.*, the abstraction is *homomorphic* for any choice of k. This morphism will be called ξ. The proof of Lemma 4.1.1 is given in Appendix A. The morphism ξ will be used a lot in the constructions and proofs of later theorems. In particular, it is needed to prove the soundness of the analysis. The graphs in the lemma are supposed to be non-canonical, because this makes sure, which case in the definition of *partner* will be applied.

Lemma 4.1.1 (Homomorphic Abstraction) *Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a non-canonical graph and let $k \geq 1$ be a natural number. There exists a surjective graph morphism ξ from G to the disjoint graph union of $\alpha_k(G)$ defined to be $\xi(u) := (\hat{C}, (\nu, P, n))$, where*

1. *C is the connected component of G containing u and $\alpha_k(C) = \{\hat{C}\}$*
2. *$partner_{C,k}(u) = (\nu, P, 1)$*
3. *$n = (|\{v \in V_C \mid u \bowtie_c v\}|)_k$*

In the part (3) of the definition, it is important to require non-canonical graphs, because that allows for simply counting nodes, instead of adding up their multiplicities. Recall that the $(\cdot)_k$ notion casts a natural number to its k-bound version. It is computed by simply taking the union of all element

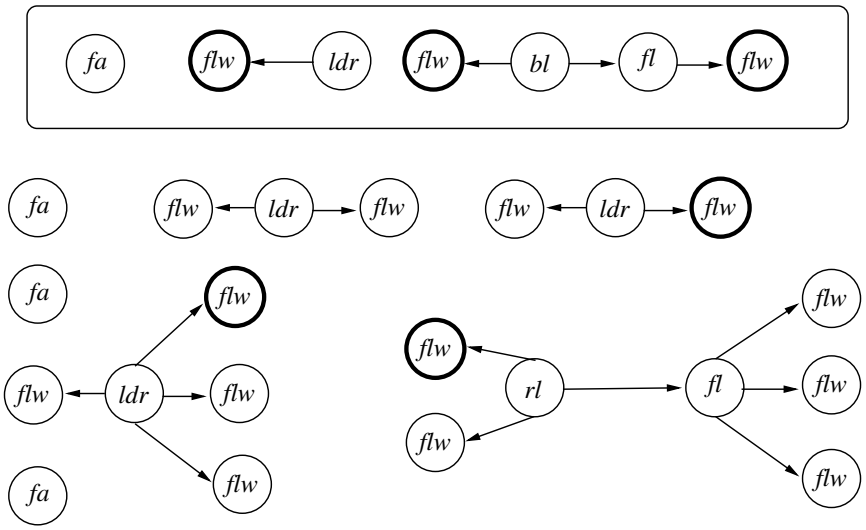


Fig. 4.2: An abstract graph and a sample materialization of it. Nodes with a thick rim denote ∞-summary nodes. The abstraction was computed with $k = 1$.

graphs. Nodes in distinct clusters were made distinct by pairing them with their respective cluster.

The final lemma of this section shows that isomorphic abstract clusters with identical multiplicities become equal because of canonical naming. This is the basis for cluster summarization and the termination proof of the analysis.

Lemma 4.1.2 (Unique Representation) *Let $k \geq 1$ be a natural number and let $G, H \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be connected, non-canonical graphs. Then, $\alpha_k(G) = \alpha_k(H)$, if and only if*

- *$G/\bowtie \cong H/\bowtie$ due to isomorphism ψ and*
- *$(|\, u \,|)_k = |\, \psi(u) \,|_k$ for all $u \in V_{G/\bowtie}$.*

4.1.3 Materialization

The next important concept in the abstract interpretation of partner graph grammars is *materialization*. It plays a crucial role in keeping *abstract rule application* as close to concrete applications as possible. In particular, the very same concrete transformation rules shall be applied to abstract graphs.

Consider the transformation rule [InitMerge] of Table 3.1 with the label assignments $\mathbf{x} = \mathbf{y} = ldr$. This rule is applicable to the graph T_1 of Figure 4.1. However, it is not applicable to the abstraction $\alpha_1(T_1)$, if injective matches are required. Therefore, either the injectivity requirement on matches may be relaxed, or one does not consider morphisms to abstract graphs. The second option is chosen here and later proven to be equivalent to the first option. In order to maintain injective matches, the abstraction of a graph is *locally undone by materialization*.

Materialization is not to be confused with the best abstract transformer defined in [CC79]. There, an abstraction is completely undone, before concrete transformers are applied. As in most cases, this cannot be done for partner abstraction, because there are typically infinitely many concretizations of an abstraction.

Materialization concretizes just enough to make concrete rules applicable. Moreover, how much to be materialized is driven by the transformation rule that is to be applied. Section 4.1.4 gives more details on matching abstract graphs. First, there is the formal definition of materialization.

Definition 4.1.6 (Materialization) *Let $k \geq 1$ and let $\hat{G} \in \wp(\mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k))$ be an abstract graph such that all its abstract clusters are ground. A multiset $\hat{H}$ of clusters is called a* materialization *of $\hat{G}$, iff*

- *$Collapse_k(\hat{H}) = \hat{G}$ and*
- *For each $\hat{C} \in_m \hat{H}$, $\nu \in \mathcal{N}$, and partner set P there is at most one element $(\nu, P, \infty) \in_m V_{\hat{C}}$, where $V_{\hat{C}}$ may be a multiset.*
- *For each $\hat{C} \in_m \hat{H}$ and for each $u \in_m V_{\hat{C}}$:*

$$(partner_{\hat{C},k}(u)) \downarrow 2 = (partner_{collapse_k(\hat{C}),k}(\zeta(u))) \downarrow 2$$

where ζ is the morphism induced by $collapse_k(\hat{C})$.

The $\downarrow n$ notation means projection onto the n-th component of a tuple. It is noteworthy to say, that materializations are still canonical graphs, they are just not ground anymore. This is important, because it allows to keep track of multiplicities, when it comes to updating materialized graphs in Section 4.1.5. In that case, the definition of *partner* for canonical graphs respects existing multiplicities (*c.f.* Definition 4.1.3).

Figure 4.2 shows the abstract graph $\alpha_1(T_1)$ of Figure 4.1 along with a sample materialization of it. Due to the multiset notion of Definition 4.1.6, there may be an arbitrary non-zero number of materializations of each abstract cluster of an abstract graph. As the abstraction was obtained for

$k = 1$, a thickly rimmed summary node stands for an arbitrary number greater than one of such nodes. At most one ∞ summary node may remain in the materialization.

Corresponding to the two levels of abstraction – cluster abstraction and cluster summarization – materialization may be seen as a two level process as well. The first step is *cluster materialization*, creating instances of abstract clusters, whereas the second step – *node materialization* – extracts non-summary nodes from summary nodes.

A first, straightforward property of materializations that is stated here and proven in Appendix A will become important in subsequent arguments. For any graph G and any $k \geq 1$ there is a materialization of the abstraction $\alpha_k(G)$ that is isomorphic to G and does not contain any summary nodes.

Lemma 4.1.3 (Materialization) *Let $k \geq 1$ and $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be arbitrary. There is a materialization $\hat{H}$ of $\alpha_k(G)$ such that $\dot{\cup}\hat{H} \cong G$ and $\hat{H}$ has no summary nodes.*

4.1.4 Abstract Matches

As indicated earlier, one goal in the formalization of abstract updates is to keep them as close as possible to concrete updates, *i.e.* to direct derivations. The first step in doing so is to define an appropriate notion of *abstract matches*. When does a concrete transformation rule match an abstract graph? In order for a rule to match a concrete graph two requirements need to be met (*c.f.* Definition 2.1.9).

1. The left graph of the rule must be a subgraph of the matched graph.
2. Possible partner constraints must be satisfied (Definition 2.1.8).

The concept of materialization presents a solution to first problem. A rule matches an abstract graph, if the left graph of the rule is a subgraph of *some materialization* of the matched graph. Additionally, it is required that only non-summary nodes are matched.

It remains to define, however, how the satisfaction of partner constraints is defined for abstract graphs. It is recommended to recall the definition in the concrete case: Definition 2.1.8. A partner constraint is a set of tuples (io, β, ν, n) associated with a node in the left graph of a transformation rule. It requires that a matched node has at least 1 and at most n adjacent, ν-labeled nodes to which it is connected by an incoming/outgoing β-labeled edge. Apart from the "at most" part, this is easily formalized for abstract graphs as well. As an extension to Definition 2.1.8 counting in Definition 4.1.7

takes multiplicities into account. The formal definition is as follows. The $\searrow n$ notation means projection onto the first n components of a tuple. It is familiar from the definition of partner constraint satisfaction in the concrete case and not to be confused with the $\downarrow n$ notation that projects onto the n-component of a tuple.

Definition 4.1.7 (Abstract Partner Constraint Satisfaction) *Let pc be a partner constraint over $\mathcal{N}$ and $\mathcal{E}$, and let $\hat{C} \in \mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k)$ be a connected canonical graph . Let $u = (\nu, P, n)$ be an element of $V_{\hat{C}}$. Node u in $\hat{C}$ satisfies pc, written $\hat{C}, u \models pc$, iff*

1. $pc \searrow 3 = (partner_{\hat{C},k}(u)) \downarrow 2$.
2. *For all $(in, \beta, \mu, q) \in pc$ holds $\bigoplus^k \{q' \mid ((\mu, \_, q'), u) \in E^{\beta}_{\hat{C}} \sqsubseteq_k q$.*
3. *For all $(out, \beta, \mu, q) \in pc$ holds $\bigoplus^k \{q' \mid (u, (\mu, \_, q')) \in E^{\beta}_{\hat{C}} \sqsubseteq_k q$.*

Having defined the notion of partner constraint satisfaction in the abstract world completes the ingredients needed to define an abstract match. The following definition formalizes these requirements.

1. There must be a materialization of the abstract graph, such that the left graph of a rule is a subgraph of this materialization.
2. Partner constraints must be satisfied.
3. The match morphism must not map a node to a summary node.

Definition 4.1.8 (Abstract Match) *Let $r = (L, h, p, R)$ be a transformation rule over $\mathcal{N}$ and $\mathcal{E}$, such that the maximal degree among partner constraints is k. Let $\hat{G}$ be an abstract graph with ground abstract clusters over $\mathcal{N}$, $\mathcal{E}$, and k. Rule r* matches *$\hat{G}$, iff*

1. *There exists a materialization $\hat{H}$ of $\hat{G}$ such that $L \leq \dot{\cup}\hat{H}$ due to match m.*
2. *For all $u \in dom(p)$ holds, that, if $m(u) \in V_{\hat{C}}$ for some abstract cluster $\hat{C}$, then $\hat{C}, m(u) \models p(u)$.*
3. *For all $u \in V_L$, $m(u) = (\_, (\_, \_, 1))$.*

Morphism m is called an abstract match.

Finding a matching materialization may not be an easy task, because there may be many tries necessary before it is found. Certainly, this formalization would be difficult to implement. Therefore, an equivalent characterization of abstract matching is given in Lemma 4.1.4 that is easier to check and to implement. It is based on matching a transformation rule to an abstract graph without taking the materialization detour.

Lemma 4.1.4 (Existence of Materializations) *Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph and $r = (L, h, p, R)$ a transformation rule over the same label sets. Let k be the maximal degree among the partner constraints in the image of p and $\hat{G} = \alpha_k(G)$ the abstraction of G. The following statements are equivalent.*

1. *Rule r matches $\hat{G}$.*
2. *There exists a morphism m from L to $\dot{\cup}\hat{G}$, such that $\dot{\cup}\hat{G}, m(u) \models p(u)$ for all $u \in dom(p)$, and for each $C \in cc(L)$ and each $u = (\nu, P, n) \in m(V_C)$ holds $\mid m^{-1}(u) \mid_k \sqsubseteq_k n$.*

As usual, the proof is given in Appendix A. The (2) $\rightarrow$ (1) part of it is constructive. It shows how, given a morphism to an abstract graph, a materialization is constructed. This construction is also used in the implementation of the analysis. It further clarifies the concepts and differences between cluster and node materialization.

Having designed an abstraction it is interesting to know which properties are preserved by. Partner abstraction, as the name indicates aims at preserving as precise as possible information about the immediate neighborhood of nodes in graphs. It is therefore expected to preserve partner equivalence and partner constraint satisfaction. This fact is stated in the following lemma.

Lemma 4.1.5 (Partner Preservation) *Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph and $\hat{G} = \alpha_k(G)$ its abstraction for some $k \geq 1$. Let ξ be the induced morphism of the abstraction. Finally, let pc be a partner constraint. The following statements hold.*

1. *$partner_{\dot{\cup}\hat{G},k}(\hat{C}, (\nu, P, n)) = (\nu, P, n)$ for all $(\nu, P, n) \in V_{\hat{C}}$ and all $\hat{C} \in \hat{G}$.*
2. *$(partner_{G,k}(u)) \downarrow 2 = (partner_{\dot{\cup}\hat{G},k}(\xi(u))) \downarrow 2$ for all $u \in V_G$.*
3. *If pc is a simple partner constraint and $G, u \models pc$, then $\dot{\cup}\hat{G}, \xi(u) \models pc$.*
4. *If k is the degree of pc and $\dot{\cup}\hat{G}, \hat{u} \models pc$ for any $\hat{u} \in V_{\hat{G}}$, then $G, u \models p$ for all $u \in \xi^{-1}(\hat{u})$.*

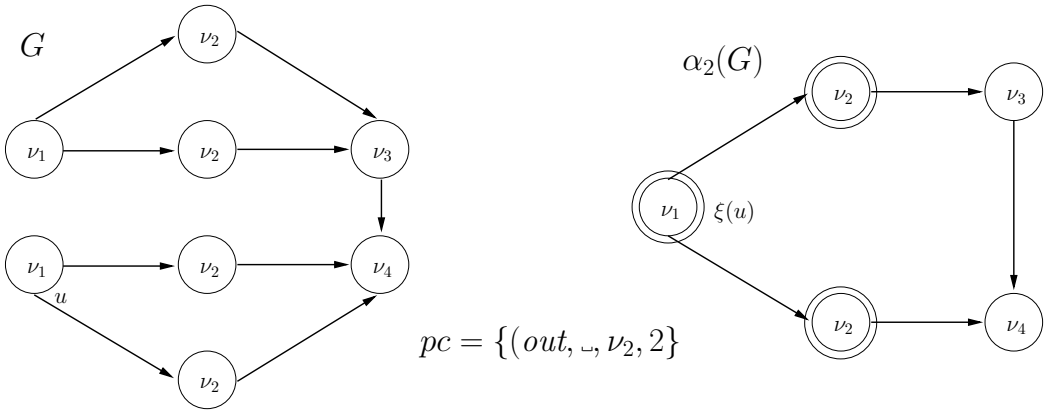
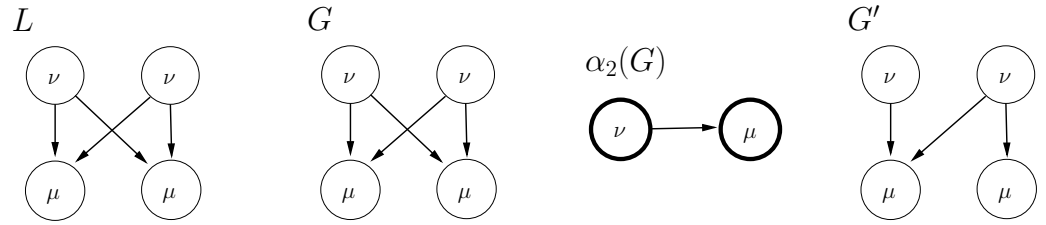


Fig. 4.3: Partner constraint preservation by abstraction. This figure shows an example of a concrete graph and its abstraction with $k = 2$, such that there exists a partner constraint satisfied by a node u in the concrete graph but not by its abstraction $\xi(u)$. The partner constraint pc requires, that a matched node has at most two ν_2 labeled partners, which is the case in the concrete but not in the abstract graph resulting in $G, u \models pc$ but $\alpha_2(G), \xi(u) \not\models pc$. Doubly-lined nodes represent 2-summary nodes.

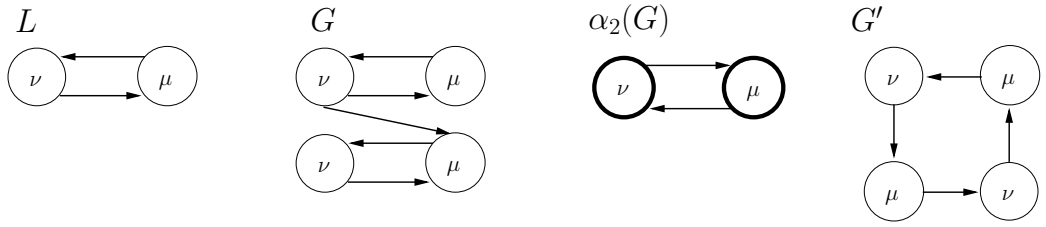
The first statement of Lemma 4.1.5 means, that the canonical name of a node in an abstract graph correctly describes its neighborhood. The second part effectively states the preservation of partner equivalence under abstraction. The third and fourth statements almost yield an equivalence between partner constraint satisfaction in a graph and in its abstraction. However, if a node u in a graph satisfies a non-simple partner constraint, it may be the case that $\xi(u)$ does *not* satisfy the partner constraint in the abstraction. An example of this is given in Figure 4.3. Recall that simple partner constraints are those that do not feature "at most" components other than ∞, *i.e.*, they have degree 0. In other words, they do not impose constraints on the maximal number of partners of a node.

It is crucial for the soundness proof of the analysis, that, whenever a rule matches a graph, it matches its abstraction. This is necessary in order not to *miss* a graph in the abstract graph semantics. This property is formalized in Lemma 4.1.6 and proven in Appendix A.

Lemma 4.1.6 (Match) *Let r be a transformation rule over $\mathcal{N}$ and $\mathcal{E}$ that features simple partner constraints only. Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph. For any $k \geq 1$ holds that, if r matches G, then r matches $\alpha_k(G)$.*



(a) Unique partners



(b) Summary Cycles

Fig. 4.4: Two examples of abstractions that are matched by a rule (L denotes the left graph of the rule). Additionally, one matching (G) and one non-matching (G') materialization are given. Part (a) and (b) illustrate the concepts of unique partners and summary cycles as defined in Definition 4.1.9. As usual, nodes with a thick rim denote ∞-summary nodes.

Lemma 4.1.6 will be the basis of the soundness proof of the analysis. If there was to be shown that the analysis might be complete, the inverse direction of the matching lemma would have to hold, too. Informally, this means, that whenever a rule matches an abstract graph, it matches *all* its concretizations.

In general, the inverse direction of the matching lemma cannot be expected to hold. In order to guarantee that there are only finitely many abstract graphs, the abstraction must mix up nodes, thereby discarding graph structure. If, on the other hand, the graph structure in left graphs of rules can be arbitrarily complex, there is no way to faithfully preserve such arbitrarily complex structure by only a finite number of canonical graphs. Figure 4.4 shows two examples of such behavior. In both parts, the left hand side L of some rule is given. A graph G matched by L is shown along with its abstraction. Furthermore, another graph G' with $\alpha_1(G') = \alpha_1(G)$ is given that is not matched by L.

Part (a) presents one of two sources of non-completeness, *unique partners*, more precisely, non-unique partners. The inverse of the matching lemma may not hold, if there is a node with two equally directed, equally labeled edges

to two distinct but equally labeled nodes. If a graph does not contain such a pattern, it is said to have the *unique partner property*.

The second source of non-completeness are *summary cycles* as illustrated in part (b) of Figure 4.4. If there is a (possibly singleton) set of summary nodes on an undirected cycle, abstract matching may not imply concrete matching. If an abstract graph features unique partners and no summary cycles, then all its materializations will have a subgraph isomorphic to the abstraction. This is the core of the proof of Theorem 4.1.7 given in Appendix A. In this theorem, it is shown that excluding this undesired behavior implies the inverse direction of the matching lemma, the *matching theorem*. The theorem is stated immediately after the definition of unique partners and summary cycles. Notice that the matching theorem is formulated in terms of connected components simplifying the proof.

Definition 4.1.9 (Unique Partners, Summary Cycles) *Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph. It has* unique partners*, iff for all $u \in V_G$, for all $\beta \in \mathcal{E}$, and for all $\nu \in \mathcal{N}$*

- $|\{v \mid (u,v) \in E_G^\beta, \ell_G(v) = \nu\}| \leq 1$ *and*
- $|\{v \mid (v,u) \in E_G^\beta \, \ell_G(v) = \nu\}| \leq 1$

If $G \in \mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k)$ is a canonical graph for any $k \geq 1$, then it is said to have a summary cycle*, iff there exist $n \geq 1$, $u_1, \ldots, u_n \in V_G$, $\beta_1, \ldots, \beta_n \in \mathcal{E}$, such that $u_1 = u_n$, u_i is a summary node, and pairwise distinct $(u_i, u_{i+1}) \in E^{\beta_i}$ or $(u_{i+1}, u_i) \in E^{\beta_i}$ for all $1 \leq i < n$.*

Theorem 4.1.7 (Match) *Let $C \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a connected graph and let $r = (L, h, p, R)$ be a transformation rule with simple partner constraints over the same set of labels such, where L is connected. Let $k \geq 1$ be arbitrary. If*

1. *r matches $\alpha_k(C)$, and the match morphism m according to Lemma 4.1.4 is injective.*
2. *$\alpha_k(C)$ has unique partners.*
3. *$\alpha_k(C)$ has no summary cycles.*

then r matches C.

In fact, the theorem holds for general partner constraints, too. But as the analysis is not even sound for general partner constraints, this case is omitted here.

4.1.5 Abstract Transformers

All the ingredients to define abstract transformers and the abstract graph semantics of a partner graph grammar are available at this point. The big picture of an abstract update is a three step process:

1. Find an abstract match and the corresponding materialization.
2. Apply the *concrete* transformer, *i.e.* a direct derivation step, to the materialization.
3. Abstract the result of the direct derivation.

The final step of this process is necessary in order to guarantee the termination of the abstract graph semantics computation (*c.f.* Theorem 4.1.8). In the remainder of this section, abstract updates and the abstract graph semantics of a partner graph grammar are defined and the termination and soundness of the analysis is proven.

Definition 4.1.10 (Abstract Graph Semantics) *Let $\mathfrak{G} = (\mathcal{R}, \mathcal{I})$ be a partner graph grammar over $\mathcal{N}$ and $\mathcal{E}$. Let $r \in \mathcal{R}$ be a transformation rule and let $\hat{G}, \hat{H} \in \mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k)$ be ground abstract graphs, where $k \geq 1$.*

The abstract graph $\hat{G}$ is in the abstract direct derivation relation with $\hat{H}$, written $\hat{G} \rightsquigarrow_r \hat{H}$, iff

1. *Rule r matches $\hat{G}$ due to materialization $\hat{M}$.*
2. *$\dot{\cup}\hat{M} \rightsquigarrow_r \hat{M}'$*
3. *$\hat{H} = \mathit{collapse}_k \circ \mathit{partner}_k(\hat{M}')$.*

One writes $\hat{G} \rightsquigarrow^k_{\mathcal{R}} \hat{H}$, if there exists an $r \in \mathcal{R}$ such that $\hat{G} \rightsquigarrow^k_r \hat{H}$. $\hat{M}$ is called the triggering materialization *of the update. The* abstract graph semantics *of $\mathfrak{G}$ is defined inductively:*

$$
\begin{array}{lcl}
[\![\mathfrak{G}]\!]^k_0 & := & \alpha_k(\mathcal{I}) \\
[\![\mathfrak{G}]\!]^k_i & := & [\![\mathfrak{G}]\!]^k_{i-1} \cup \bigcup\{\hat{H} \mid \exists \hat{G} \subseteq [\![\mathfrak{G}]\!]^k_{i-1}.\hat{G} \rightsquigarrow^k_r \hat{H}\} \quad \textit{for } i > 0 \\
[\![\mathfrak{G}]\!]^k & := & \bigcup_{i \geq 0} [\![\mathfrak{G}]\!]^k_i
\end{array}
$$

Two example abstract rule applications are presented in Figure 4.5 and Figure 4.6. The first figure shows the application of a transformation rule initiating a merge maneuver to an abstract graph representing arbitrary many platoons of size at least $k + 2$, because there is an ∞-summary node. The application shows an example of a cluster materialization. Notations are

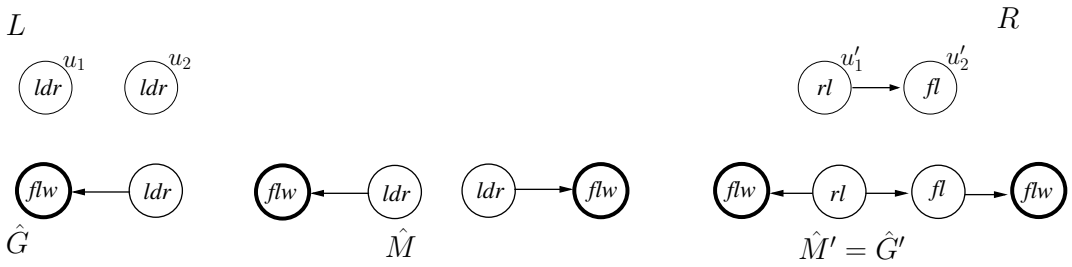


Fig. 4.5: An example of a rule application to an abstract graph such that $\hat{G} \rightsquigarrow \hat{G}'$. The required materialization is an example of a cluster materialization and is called $\hat{M}$. The concrete rule application to the materialization yields $\hat{M}'$, that, in this case, equals its abstraction $\hat{G}'$. Abstraction computed with $k = 1$.

chosen according to Definition 4.1.10, where $\hat{M}$ is a materialization and $\hat{M}'$ the updated materialization.

Figure 4.6 displays an example of node materialization. The transformation rule passing followers to another leader is applied to an abstract graph that represents an arbitrary number of merging platoons. As an abstract match morphism must not match to summary nodes, a non-summary follower is materialized. As before, the notation for materialization and updated materialization corresponds to the notation used in Definition 4.1.10.

Standard results in abstract interpretation also hold for the abstract interpretation of of partner graph grammars based on partner abstraction. The computation of the abstract graph semantics terminates and is sound. Soundness means that the abstract graph semantics is a conservative approximation of the concrete graph semantics. The soundness and termination results are presented in the following theorem and proven in Appendix A.

Theorem 4.1.8 (Termination and Soundness) *Let $\mathfrak{G}$ be a partner graph grammar and let $k \geq 1$ be arbitrary. There exists an $n \geq 0$ such that $[\![\mathfrak{G}]\!]^k_n = [\![\mathfrak{G}]\!]^k_i$ for all $i \geq n$. Furthermore,*

$$\bigcup_{G \in [\![\mathfrak{G}]\!]} \alpha_k(G) \subseteq [\![\mathfrak{G}]\!]^k$$

A Note on Complexity The worst case complexity of computing the abstract graph semantics of a partner graph grammar is rather large. Let n and e be the number of node and edge labels, respectively, and let k be the abstraction parameter. Then the number of possible canonical names is

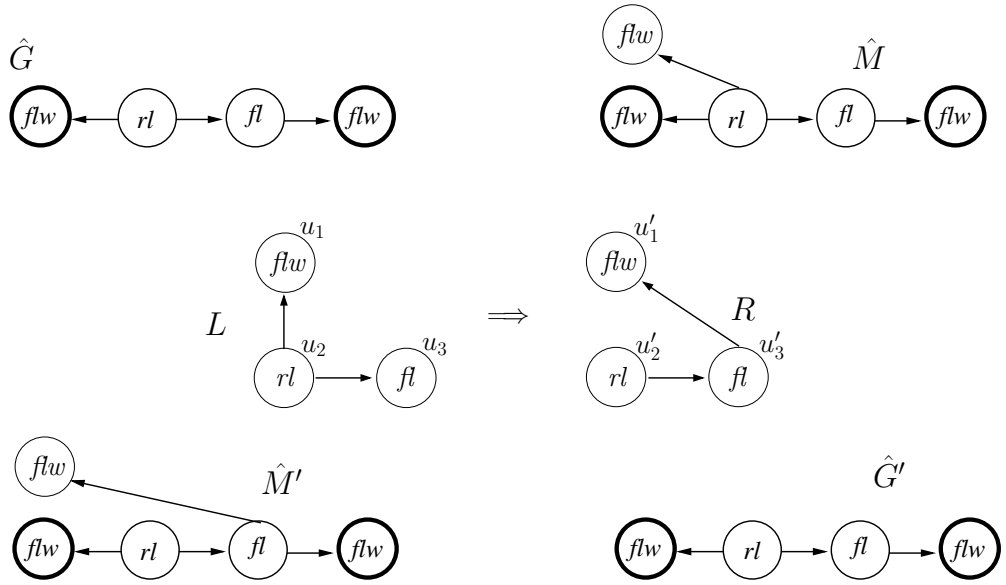


Fig. 4.6: An example of a rule application to an abstract graph such that $\hat{G} \rightsquigarrow \hat{G}'$. The required materialization employs node materialization. The materialization is called $\hat{M}$. The result of applying the rule to the materialization is called $\hat{M}'$. Abstraction computed with $k = 1$.

$c := n2^{2ne}k$. There are hence 2^c possibilities for the number of nodes in an abstract cluster. This number is multiplied by the possible distributions of e kinds of edges over the nodes to obtain the number of abstract clusters. The precise number is tedious to compute, because abstract clusters need to be connected. However, it is enough to know, that there is at least a number of abstract clusters double-exponential in n and e and exponential in k.

Another source of complexity is matching, which is exponential in the worst-case, too. This is not so problematic here, because left graphs in rules are expected to be reasonably small. Moreover, they are labeled which further simplifies the task. Despite the bad news about complexity, the implementation of the abstract interpretation of partner graph grammars described in Section 4.5 works for an example with $n = 18$ and $e = 4$.

It is conjectured, that correct protocols implementing dynamic communication systems are well-behaved in the number of possible communication topologies. However, experience showed that erroneous protocol implementations could blow up the size of the obtained results.

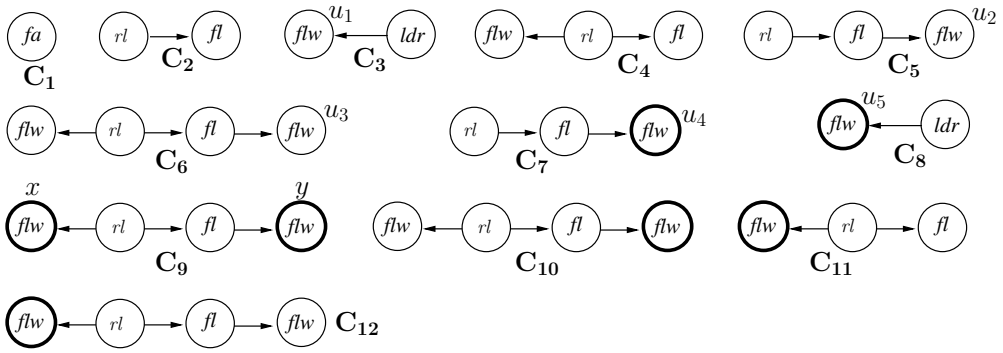


Fig. 4.7: The abstract graph semantics of the basic merge maneuver graph grammar, $[\![(\mathcal{R}_{\text{merge}}, E)]\!]^1$, where $\mathcal{R}_{\text{merge}}$ is the set of rules in Table 3.1 and where E is the empty graph. The single computation steps are as follows: $[\![(\mathcal{R}_{\text{merge}}, E)]\!]^1_0 = \emptyset$, $[\![(\mathcal{R}_{\text{merge}}, E)]\!]^1_1 = \{C_1\}$, $[\![(\mathcal{R}_{\text{merge}}, E)]\!]^1_2 = \{C_2\}$, $[\![(\mathcal{R}_{\text{merge}}, E)]\!]^1_3 = \{C_3\}$, $[\![(\mathcal{R}_{\text{merge}}, E)]\!]^1_4 = \{C_4, C_5, C_6\}$, $[\![(\mathcal{R}_{\text{merge}}, E)]\!]^1_5 = \{C_7, C_8\}$, $[\![(\mathcal{R}_{\text{merge}}, E)]\!]^1_6 = \{C_9, C_{10}, C_{11}, C_{12}\}$.

4.2 Abstract Transition Systems

Two semantic constructs exist for partner graph grammars, graph semantics and graph transition system (Definition 2.1.10). So far, only the abstract version of the graph semantics has been defined in Definition 4.1.10. The definition of an abstract graph transition system is more intricate, because the abstract graph semantics is a set of abstract clusters and several abstract clusters may be involved in the left and in the right side of a rule application. The $\mathcal{GL}$ logic introduced in Section 2.2 provides features to reason about the evolution of single nodes, which is even more difficult than the evolution of clusters. The additional difficulty originates in the summarization of nodes that is necessary to guarantee the boundedness of partner abstraction. The issues of node and cluster evolution are addressed in Section 4.2.1 and Section 4.2.2.

4.2.1 Cluster Evolution

As indicated in the introduction to Section 4.2, several abstract clusters may be involved in an abstract derivation step. This gives rise to a binary relation between *sets* of abstract clusters. On the other hand, binary relations over abstract clusters are certainly more convenient to reason about and to work with. Moreover, they are closer to transitions in ordi-

nary transition systems. Therefore, the cluster evolution relation $\mathfrak{E}$, where $\mathfrak{E} \subseteq \mathcal{C}(\mathcal{N}, \mathcal{E}, k) \times \mathcal{C}(\mathcal{N}, \mathcal{E}, k)$, does not capture all information that is available from the abstract graph semantics. Rather, given two abstract clusters, it captures whether these two occur on the left and right side of an abstract direct derivation, respectively. Finally, the cluster evolution relation will be parameterized by transformation rule names. This makes it one cluster evolution relation, $\mathfrak{E}(r)$, per transformation rule r of a given partner graph grammar.

Definition 4.2.1 (Cluster Evolution) *Let $\hat{G}, \hat{G}' \in \mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k)$ be abstract graphs, and let $\hat{C} \in \hat{G}$ and $\hat{C}' \in \hat{G}'$ be abstract clusters. Let $r = (L, h, p, R)$ be a transformation rule. The clusters $\hat{C}$ and $\hat{C}'$ are in the* cluster evolution relation*, written $\hat{C}\ \mathfrak{E}(r)\ \hat{C}'$, iff $\hat{G} \rightsquigarrow_r^k \hat{G}'$ due to materialization $\hat{M}$ and match morphism m from L to $\dot{\cup}\hat{M}$, such that for each $\hat{D} \in_m \hat{M}$ holds $m(V_L) \cap V_{\hat{D}} \neq \emptyset$.*

The abstract clusters in $\hat{G}$ are called the triggering clusters *of the cluster evolution, $\hat{M}$ is called the* triggering materialization*, and m is called the* triggering match*.*

Figure 4.7 shows the complete abstract graph semantics of the partner graph grammar $(\mathcal{R}_{\text{merge}}, E)$ abstracted with $k = 1$, where the set of rules is defined in Table 3.1 on page 42. It models the idealized platoon merge maneuver. The abstract clusters in the figure are named C_1 through C_{12}. In the caption of the figure, the respective iteration of the fixpoint computation of the abstract graph semantics, in which an abstract cluster first occurs, are given. Some sample cluster evolution relations:

$$C_1 \quad \mathfrak{E}([\textsc{InitMerge}]) \quad C_2 \tag{4.9}$$
$$C_3 \quad \mathfrak{E}([\textsc{InitMerge}]) \quad C_4 \tag{4.10}$$
$$C_3 \quad \mathfrak{E}([\textsc{InitMerge}]) \quad C_5 \tag{4.11}$$
$$C_9 \quad \mathfrak{E}([\textsc{Pass1}]) \quad C_9 \tag{4.12}$$
$$C_9 \quad \mathfrak{E}([\textsc{Pass1}]) \quad C_{10} \tag{4.13}$$

Relation (4.9) requires a cluster materialization, before [InitMerge] is applied. The cluster evolution relations may contain reflexive elements as indicated by (4.12). Relations (4.10) and (4.11) do not differ in the applied rule but in the triggering match leading to the rule application. The difference between (4.12) and (4.13) is the triggering node materialization. In the first case, an ∞-summary node for $k = 1$ is materialized into two non-summaries. In the second case, one ∞-summary node remains.

4.2.2 Node Evolution

Besides abstract clusters, nodes within an abstract cluster may evolve into each other, too. The formalization of node evolution is given in Definition 4.2.2. It allows for tracking nodes over time. In order to define node evolution, finite arithmetic is extended to include negative integers in the straightforward way. Furthermore, $-\infty$ is added with the additional requirement that the sum $-\infty \oplus^k \infty$ is undefined, *i.e.*, it may yield any number.

Definition 4.2.2 is rather involved technically. Whether node $(\hat{C}, \hat{u})$ in an abstract cluster $\hat{C}$ evolves into $(\hat{C}', \hat{u}')$ depends on a cluster evolution from $\hat{C}$ to $\hat{C}'$ and on a triggering materialization. First, a node u in the materialization is fixed, that is mapped to $\hat{u}$ by *collapse* (rather by the induced morphism ζ). After the update, u keeps its identity, because concrete updates are independent of node identities. In the updated graph, however, u must have the same label and partner set as $\hat{u}'$. This is formulated in requirement (4) of the definition. Certainly, u must be within a connected component that is abstracted to $\hat{C}'$.

Eventually, multiplicity information is tracked by node evolutions, too. It is said that $\hat{u}$ evolves into $\hat{u}'$ *by* $\Delta\ n$. The way to compute this Δ is specified in part (5) of the definition. This node is mapped to $\hat{u}$ by ζ as induced by *collapse*. The equation reads as follows. The sum of the multiplicities of all nodes mapped to $\hat{u}$ by ζ *except* u plus the unknown Δ yields the sum of the multiplicities of all nodes that become equivalent to u after the update.

Definition 4.2.2 (Node Evolution) *Let $\hat{C}, \hat{C}' \in [\![\mathfrak{G}]\!]^k$ be two abstract clusters, and let $\hat{u} \in V_{\hat{C}}$ and $\hat{u}' \in V_{\hat{C}'}$ be two nodes. Node $\hat{u}$* evolves into *$\hat{u}'$* by rule *$r = (L, h, p, R)$* and *$\Delta\ n$, written*

$$(\hat{C}, \hat{u})\ \mathfrak{E}(r, n)\ (\hat{C}', \hat{u}')$$

for $\hat{u} = (\nu, P, s)$ and $\hat{u}' = (\nu', P', s')$ iff

1. *$\hat{C}\ \mathfrak{E}(r)\ \hat{C}'$ with triggering materialization $\hat{M}$, such that $\dot{\cup}\hat{M} \rightsquigarrow_r \hat{M}'$.*
2. *There exist $C \in_m \hat{M}$ and $C' \in cc(\hat{M}')$ such that $collapse_k(C) = \hat{C}$ and $collapse_k \circ partner_k(C') = \hat{C}'$.*
3. *There exists $t \in \mathbb{N}_k$ and $u = (\nu, P, t) \in_m V_C \cap_m V_{C'}$ such that $u \notin_m m(V_L)$.*
4. *$partner_{C',k}(u) = (\nu, P', t)$.*

5. n satisfies the equation $s^- \oplus^k n = s^+$, where

$$\begin{aligned} s^- &= \bigoplus^k \{\!\{ q \mid \exists (\nu, P, q) \in V_C.u \neq (\nu, P, q) \}\!\} \\ s^+ &= \bigoplus^k \{\!\{ q \mid \exists v \in V_{C'}.u \neq v, \mathit{partner}_{k,C'}(v) = (\nu', P', q) \}\!\} \end{aligned}$$

The triggering clusters of the cluster evolution in (1) are called the affected (or triggering) clusters *of the node evolution. The materialization M is called the* triggering materialization.

Figure 4.7 provides a good source of examples of node evolution. The examples center around abstract cluster C_9 and the two ∞-summary nodes x and y of this cluster. Informally, by applying the transformation rule [Pass1], one node represented by x is passed over to y. Formally, the following relations hold.

$$(C_9, x) \quad \mathfrak{E}([\text{Pass1}], -1) \quad (C_9, x) \tag{4.14}$$
$$(C_9, y) \quad \mathfrak{E}([\text{Pass1}], 1) \quad (C_9, y) \tag{4.15}$$
$$(C_9, x) \quad \mathfrak{E}([\text{Pass1}], \infty) \quad (C_9, y) \tag{4.16}$$

Relation (4.14) means that the summary node representing the followers to the rear leader loses a node during the update, whereas relation (4.15) describes that the summary node representing followers to the front leader increases in size by 1. Applying [Pass1] to C_9 and obtaining C_9 is only possible, if x is split in a non-summary and an ∞-summary node in a materialization. The u in the definition is then chosen to be the remaining ∞-summary node rendering s^- to be 1 (the multiplicity of the materialized node). No other node becomes equivalent to u after the update, hence $s^+ = 0$ and $n = -1$.

In the same materialization, y may be considered. No node needs to be extracted from it, hence $s^- = 0$. On the other hand, one new node becomes equivalent to it after the update, yielding $s^+ = 1$ hence $n = 1$.

Relation (4.16) works for any Δ including ∞, because the only node u migrating from x to y by application of [Pass1] is the one materialized from x. As there is still multiplicity ∞ left in x, $s^- = \infty$ is obtained. After the update, y becomes equivalent to u, hence $s^+ = \infty$. The equation is then satisfied for any choice of n due to the undefinedness of computing with infinities.

To conclude this section, the transitive extension of a simple node evolution is defined: *node paths*. All Δ's along the evolution path are added in a finite manner up to some k. Furthermore, no ∞ or $-\infty$ may be involved in

the computation of the single Δ's and the sum. The finite sum then makes up the name of the path.

Definition 4.2.3 (d-Increment Path) *Let $\hat{C}, \hat{C}' \in [\![\mathfrak{G}]\!]^k$ be two abstract clusters, and let $\hat{u} \in V_{\hat{C}}$ and $\hat{u}' \in V_{\hat{C}'}$ be two nodes. A d-increment path from $(\hat{C}, \hat{u})$ to $(\hat{C}', \hat{u}')$ is a sequence of node evolutions*

$$(\hat{C}, \hat{u})\ \mathfrak{E}(r_1, n_1)\ \ldots\ \mathfrak{E}(r_t, n_t)\ (\hat{C}', \hat{u}')$$

where $\bigoplus_{1 \leq i \leq t}^{k} n_i = d$. Additionally, for all $q \leq i \leq t$, $n_i \neq \infty$, $n_i \neq -\infty$, and no ∞ or $-\infty$ is involved in the computation of n_i in the sense of (5) of Definition 4.2.2.

The number t is called the length *of the increment path. The set of* affected clusters *of the node path is the union of all affected clusters in the sequence of node evolutions.*

For example, there is a 2-increment path from (C_6, u_3) to (C_8, u_5) in Figure 4.7.

4.3 Completeness Results

This section presents results regarding completeness – some of which are strong and unexpected. Methods like partner abstraction must abstract much, losing a lot of information in order to remain bounded. As stated earlier in connection to the matching Theorem 4.1.7, in order to guarantee that there are only finitely many abstract graphs, the abstraction must mix up nodes discarding graph structure. On the other hand, the structure in left graphs of rules may be arbitrarily complex. Therefore, it is surprising that cases may be identified, where all information is maintained by the abstract interpretation based on partner abstraction. In such cases, the abstract graph semantics is said to be complete.

In the first part of this section, three different notions of completeness are defined. The strongest notion among them implies the decidability of the word problem for a precisely characterized class of partner graph grammars. Luckily, some of the case studies presented in Chapter 3 belong to that class.

However, such strong results cannot be expected for general partner graph grammars. This is illustrated using a series of examples leading to distinct features of partner graph grammars and/or abstract clusters that introduce spurious behavior. By excluding some or all of these sources of spuriousness, it is possible to obtain completeness theorems.

Deciding whether and according to which notion an abstract graph semantics is complete is not possible by looking at the partner graph grammar alone. Instead, the abstract graph semantics itself must be examined.

4.3.1 Completeness Notions

In a classical sense, an abstract interpretation is complete, if it maintains all the information available in the concrete world. In the case of partner abstraction, there are several possible layers of information, for which completeness may be investigated.

The strongest possible completeness notion definable in terms of this work is *indistinguishability* wrt. $\mathcal{GL}$ *formulas*. This means that any $\mathcal{GL}$ formula is satisfied for a partner graph grammar, if and only if it is satisfied for the abstract partner graph grammar – the latter being the abstract graph semantics along with node and cluster evolution. Only in Section 4.4 this issue will be investigated. It is easy to guess, however, that completeness in this strongest sense cannot be expected to hold for general $\mathcal{GL}$ formulas and general partner graph grammars.

A slightly weaker notion of completeness is the *decidability of the word problem* for partner graph grammars. Does a given graph belong to the graph semantics of a partner graph grammar? A complete analysis answers this question by saying: if and only if its abstraction belongs to the abstract graph semantics. This notion is weaker than indistinguishability under $\mathcal{GL}$ formulas, because it neglects the way graphs are derived by the partner graph grammar. On the other hand, there is in fact a class of partner graph grammars for which the abstract interpretation based on partner abstraction can be shown complete in this sense.

Another interesting question to ask is whether *all* clusters represented by an abstract cluster in fact occur in the concrete graph semantics. If this holds for all abstract clusters in the abstract graph semantics, one may speak of *cluster completeness*. Notice the difference between cluster completeness and the decidability of the word problem. Cluster completeness does not make any statement about the *combinations* of concretizations of abstract clusters occurring in the graph semantics. Therefore it is a weaker notion and implied by word problem decidability.

Although it is the weakest completeness notion investigated in this work, *cluster spuriousness* is still of practical relevance. An abstract cluster is spurious, if there is no graph in the graph semantics whose abstraction contains it. A clearly defined class of partner graph grammars will be given whose

abstract graph semantics are guaranteed not to contain any spurious abstract cluster.

Four notions of completeness were elaborated on, that are ordered by logical implication. Three of them are defined formally in Definition 4.3.1, whereas the evaluation of $\mathcal{GL}$ formulas in the abstract world is postponed to Section 4.4. Classes of partner graph grammars complete in the weaker senses are gradually derived in the remainder of this section.

Definition 4.3.1 (Completeness) *Let $\mathfrak{G}$ be a partner graph grammar, $k \geq 1$, $\mathcal{N}$ and $\mathcal{E}$ label sets. The abstract graph semantics $[\![\mathfrak{G}]\!]^k$ is called*

- word decidable*, iff*

$$[\![\mathfrak{G}]\!] = \{G \in \mathcal{G}(\mathcal{N}, \mathcal{E}) \mid \alpha_k(G) \subseteq [\![\mathfrak{G}]\!]^k\}$$

- cluster complete*, iff for each $\hat{C} \in [\![\mathfrak{G}]\!]^k$ and for each connected $C \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ holds: If $\alpha_k(C) = \{\hat{C}\}$, then there exists a $G \in [\![\mathfrak{G}]\!]$ such that $C \in cc(G)$.*

- free of spurious clusters*, iff*

$$\bigcup_{G \in [\![\mathfrak{G}]\!]} \alpha_k(G) = [\![\mathfrak{G}]\!]^k$$

As stated above, these completeness notions are to be understood up to isomorphism only, because, *e.g.*in the word decidable case, only one particular graph is derived in the concrete graph semantics, whereas the definition requires the whole isomorphism class of this graph to be in the concrete graph semantics as well. To conclude the introduction of the various completeness notions, notice the similarity between lack of spurious clusters and soundness as stated in Theorem 4.1.8.

4.3.2 Friendly Systems

Partner abstraction forgets about the number of clusters summarized to one abstract cluster. This yields at the same time a strong reduction of size of the abstraction and a potential source of spurious behavior introduced by cluster materialization. Reconsider Figure 4.5 on page 83 in Section 4.1.5 showing an instance of cluster materialization. There, the transformation rule [InitMerge] is applied to a single abstract cluster representing a platoon. In the materialization, however, two platoons need to be present for the

rule to be applicable. The existence of two platoons cannot be guaranteed in the general case.

The previous paragraph suggests, that, in general, it is desirable that the existence of one instance of an abstract cluster justifies the existence of an arbitrary number of such instances. This property of a partner graph grammar is called *cluster multiplicity*. It shows that there is very simple and easily checkable sufficient condition for cluster multiplicity. A partner graph grammar has the cluster multiplicity property, if it has an empty initial graph. If the graph semantics is not empty, there must be at least one rule with an empty left graph – a *create rule*. Otherwise, the empty initial graph cannot be matched by any rule. This means, that the derivation of any graph starts by applying create rules. This derivation may then be repeated arbitrarily often, because every graph is matched by a create rule. Moreover, the recurring derivation cannot be prevented by partner constraints, because they cannot express the absence of another cluster.

As seen earlier, simple partner constraints are necessary for the soundness of the abstract interpretation based on partner abstraction. Hence they should be added to the completely static notion of a *friendly grammar*.

Definition 4.3.2 (Friendly Grammars) *A partner graph grammar* $(\mathcal{R}, \mathcal{I})$ *is* friendly*, iff it features simple partner constraints only and if* $\mathcal{I}$ *is the empty graph.*

A rule (L, h, p, R) *is called a* create rule*, iff* L *is the empty graph.*

The remainder of this section assumes friendly grammars, even if not explicitly stated. The formalization of cluster multiplicity is presented in Lemma 4.3.1. It is given in an even stronger form than needed for reasoning about completeness. Whenever two graphs are in the graph semantics of a partner graph grammar, then also their disjoint graph union is in the graph semantics. The formal proof is omitted, because all the necessary arguments were given in the paragraphs before Definition 4.3.2. Essentially, every derivation of a graph in the graph semantics may be repeated arbitrarily often.

Lemma 4.3.1 (Cluster Multiplicity) *Let* $\mathfrak{G}$ *be a friendly partner graph grammar. If* $G_1, G_2 \in [\![\mathfrak{G}]\!]$*, then* $G_1 \dot{\cup} G_2 \in [\![\mathfrak{G}]\!]$.

Notice that friendliness is a sufficient but not a necessary condition for cluster multiplicity. An example of a non-friendly partner graph grammar that has the cluster multiplicity property may be obtained as follows. There is only one node label ν, the initial graph is a singleton node labeled ν, and there is only the following transformation rule.

4.3.3 Cluster Completeness

Among the three notions of completeness defined in Definition 4.3.1, cluster completeness is the first to be investigated. It states that for each abstract cluster $\hat{C}$ of an abstract graph semantics, all its concretizations occur as some subgraph of a graph in the concrete graph semantics. In order to better understand this notion, it is sensible to look at examples of partner graph grammars that have spurious clusters. By understanding the spurious examples, conditions preventing them may be identified. It helps to recall some sources of information loss by partner abstraction.

1. Information about the number of isomorphic abstract clusters is lost.
2. Information about the number of nodes summarized in a summary node is only maintained up to some finite k.
3. Partner abstraction is based on partners. It records the *kind* of partners, *i.e.* their labels and the kind of connection. It does not record the *number* and the *identity* of the partners.

The first source of information loss has already been addressed by introducing friendly graph grammars. The influence of the second and third source of information loss on the spuriousness of abstract clusters is illustrated in Figure 4.8. Both friendly graph grammars of this figure are over a trivial set of edge labels (consisting of one label only) and the set $\{\nu, \nu', \mu, \mu'\}$ of node labels.

Consider the left column: A friendly partner graph grammar is specified in the first row and its graph semantics is shown in the second row. When the abstract graph semantics is computed for $k = 1$, the right-hand side of the create rule is abstracted to a ν-labeled node connected to a μ-labeled ∞-summary. The second rule of this grammar may be applied to this abstraction due to *two* possible materializations. The μ-labeled ∞-summary node may either be split into two non-summaries or in one non-summary and

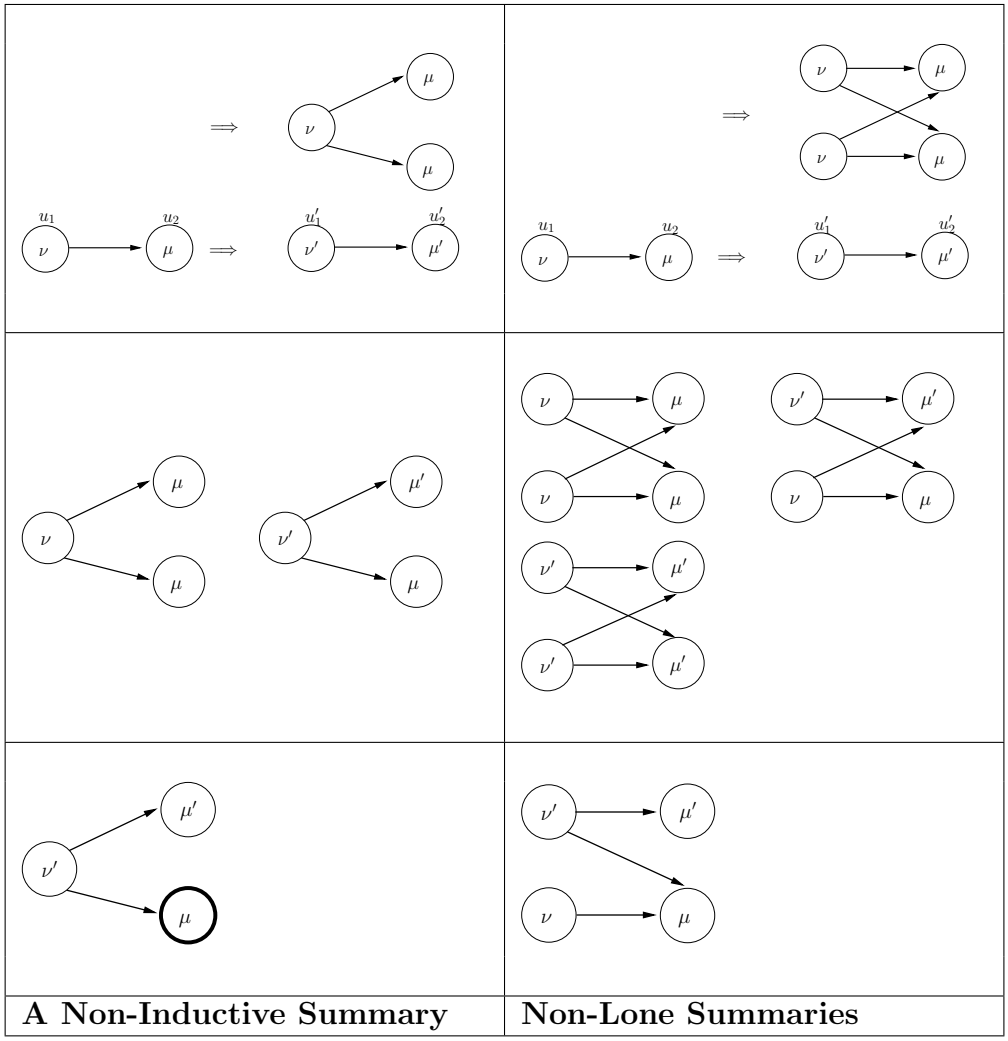


Fig. 4.8: Partner graph grammars with spurious abstract clusters. Each column presents a friendly partner graph grammar in the first row, the complete graph semantics in the second row, and a spurious abstract cluster of the abstract graph semantics in the third row. The abstractions are computed using $k = 1$.

one ∞-summary. The second case yields the spurious abstract cluster in the bottom row of the left column. It is spurious, because every concretization has one μ'-labeled and at least two μ-labeled nodes. This cannot happen in the concrete graph semantics. However, for soundness, the analysis must assume that a cluster with an ∞-summary node has concretizations with every possible number $n > k$ of copies of this summary node. An ∞-summary node

that actually does represent all possible number of nodes will be called an *inductive summary*. Later on, a statically checkable criterion for inductiveness of summary nodes is presented.

Consider the right column of Figure 4.8. Again, the grammar is presented in the first row, then its concrete graph semantics, and finally an example of a spurious abstract cluster within the abstract graph semantics. When the latter is computed for $k = 1$, the abstraction of the right-hand side of the create rule is a graph of two ∞-summaries connected by an edge. The spurious abstract cluster results from a materialization that looks like the first (top, left) graph of the graph semantics but *without* one of the diagonal edges. However, this materialization has no concrete counterpart in the graph semantics. This problem is a very general one, occurring whenever two summary nodes are adjacent. It may be excluded by disallowing connected summary nodes. A summary node that does not have an edge to another summary node (including itself) is called a *lone summary node*.

It shows that inductive and lone summary nodes are the key to prove cluster completeness. The proof will be conducted analogously to the proof of the soundness theorem (Theorem 4.1.8): Given a derivation of an abstract cluster, find a corresponding derivation of concrete graphs. The matching Theorem 4.1.7 was a first step towards that goal. However, it shows that the conditions of the matching theorem are not strong enough to guarantee the absence of spurious clusters.

First, the concept of lone summary nodes is introduced formally. It is then proven, that abstract clusters featuring only lone summary nodes have a very restricted form of materializations only. Effectively, all materializations are equal up to the number of nodes materialized from ∞-summary nodes.

Lone Summary Nodes and S-Materializations Recall the right column of Figure 4.8. The spurious abstract cluster was due to spurious materializations of connected summary nodes. Lone summary nodes are a way to prevent such materializations.

Definition 4.3.3 (Lone Summary Nodes) *Let $\hat{C} \in \mathcal{C}(\mathcal{N}, \mathcal{E}, k)$ be a connected, ground canonical graph. A summary node $\hat{u} \in V_{\hat{C}}$ is* lone*, iff for all $\nu \in \mathcal{N}$ and for all $\beta \in \mathcal{E}$*

- $\bigoplus^k \{n \mid \exists P.((\nu, P, n), \hat{u}) \in E^{\beta}_{\hat{C}}\} \sqsubseteq_k 1$*, and*
- $\bigoplus^k \{n \mid \exists P.(\hat{u}, (\nu, P, n)) \in E^{\beta}_{\hat{C}}\} \sqsubseteq_k 1$

The graph $\hat{C}$ has lone summary nodes*, iff all of its summary nodes are lone.*

The big advantage of lone summary nodes is that they allow for very restricted and disciplined materializations only. Such materializations are completely characterized by the way summary nodes are split, whereas there is no choice for the edges. All possible edges must be there. All materializations are otherwise equal and faithfully mimic the abstract cluster they are collapsed to. The S parameter given in the upcoming Definition 4.3.4 specifies the way a node is split in the materialization.

Definition 4.3.4 (S-Materialization) *Let* $\hat{C} \in \mathcal{C}(\mathcal{N}, \mathcal{E}, k)$ *be a connected, ground canonical graph. Let* $S : V_{\hat{C}} \to \wp_m(\mathbb{N}_k)$ *be a mapping, such that* $S(\nu, P, n) = M$ *if and only if* $\bigoplus^k M = n$*. Additionally, let* $S(\hat{u})$ *contain at most one* ∞ *for each* $\hat{u} \in V_{\hat{C}}$*. The graph* $\hat{M}$ *where*

$$\begin{array}{rcl} V_{\hat{M}} & = & \{\!\{(\nu, P, s) \mid (\nu, P, n) \in V_{\hat{C}}, s \in S(\nu, P, n)\}\!\} \\ E^{\beta}_{\hat{M}} & = & \{\!\{((\nu_1, P_1, s_1), (\nu_2, P_2, s_2)) \mid u_i = (\nu_i, P_i, n_i) \in V_{\hat{C}}, s_i \in S(u_i), i = 1, 2\}\!\} \\ \ell_{\hat{M}} & = & \lambda(\nu, P, n).\nu \end{array}$$

is called the S-materialization of $\hat{C}$*.*

It is straightforward to see that an S-materialization indeed adheres to the requirements of a materialization as defined in Definition 4.1.6. The S parameter specifies in how many nodes of which multiplicity a summary node has to be split in a materialization. In some sense, an S-materialization is a full materialization, because all possible edges exist in it. Figure 4.9 illustrates the concept of S-materializations. In this case all S-materializations are complete bipartite graphs. Certainly, for the example of Figure 4.9, one may think of materializations that are not complete bipartite. In the case of lone summary nodes, however, this is enforced. A graph $\hat{C}$ is a materialization of an abstract cluster with lone summary nodes, if and only if $\hat{C}$ is an S-materialization of the abstract cluster. This is formalized in Lemma 4.3.2 and proven in Appendix A.

Lemma 4.3.2 (S-Materialization) *Let* $\hat{C} \in \mathcal{C}(\mathcal{N}, \mathcal{E}, k)$ *be a connected, ground canonical graph with lone summary nodes.* $\hat{M}$ *is a materialization of* $\hat{C}$*, if and only if it is an* S*-materialization of* $\hat{C}$ *for some* S*.*

Lemma 4.3.2 has a strong consequence for abstract clusters that do not contain ∞-summary nodes. If such a cluster has lone summary nodes and no ∞-summary node, it has exactly (up to isomorphism) one concretization, *i.e.* one concrete graph C, such that $\alpha_k(C) = \hat{C}$. As usual the proof is given in Appendix A.

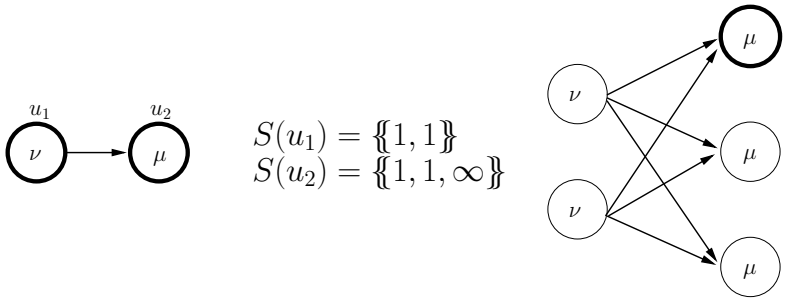


Fig. 4.9: Example of an S-materialization; abstraction based on $k = 1$.

Lemma 4.3.3 (Unique Concretization) *Let $\hat{C} \in \mathcal{C}(\mathcal{N}, \mathcal{E}, k)$ be a connected, ground canonical graph with lone summary nodes. All graphs C such that $\alpha_k(C) = \hat{C}$ are isomorphic, if $\hat{C}$ does not contain an ∞-summary node.*

Generating Orders A small remark on the relation between concretizations and materializations initiates this paragraph. Each materialization is also a concretization but not vice versa. Materializations require nodes to be canonical names. Hence, given a concretization G, a materialization is obtained by computing $\mathit{partner}_k(G)$, which replaces nodes by their canonical names. The so obtained materialization does not have summary nodes.

By now, a sufficient condition has been explored, guaranteeing that all materializations of an abstract cluster are S-materializations. The only remaining ingredient for cluster completeness is to find a condition, that *for each* possible S-materializations there exists a graph in the concrete graph semantics that is isomorphic to the materialization. This would mean that each abstract derivation step is mimicked in the concrete graph transition system. Note that the only critical part for S-materializations are ∞-summary nodes, because non-∞-summary nodes are precisely tracked up to the parameter k of the abstraction.

A close manual inspection of the abstract graph semantics of the simple merge maneuver given in Figure 4.7 on page 85 reveals that it is even word decidable, in particular, it is cluster complete. How does a computer scientist convince herself of that fact? She considers an arbitrary abstract cluster and an arbitrary concretization of it. Then she uses her intelligence to derive a derivation in the partner graph grammar of a graph that has this concretization as a subgraph. Moreover, she uses her knowledge about the system implemented by the partner graph grammar. Consider examples of clusters and concretizations as given in Figure 4.7 and how to derive them

in the concrete world.

1. C_8: This abstract cluster represents all platoons of at least two followers. It is possible to derive any such platoon with $t > 1$ followers in the partner graph grammar $\mathfrak{G}_{\text{ideal}}$. Generate $t + 1$ free agents by application of the [CREATE] rule. Let two of them merge to build a platoon of two cars. Then let each of the remaining free agents successively merge with that platoon. In the abstract graph semantics, one proceeds from C_1 to C_2 to C_3, which represents a platoon of two cars. Another car is merged to it, by traversing C_7 and then C_8. From there on, the loop $C_8 \to C_7 \to C_8$ is repeated as often as needed.

2. C_9: This abstract cluster represents two merging platoons, where both have at least two followers. There is a very simple way to create an arbitrary concretization of this cluster. First create two platoons with the desired number of followers, then start a merge.

The following two examples give a hint to the remaining ingredients of a cluster completeness proof besides lone summary nodes. The crucial role is played by the summary node (C_8, u_5) in Figure 4.7. Example 1 above showed that it represents an arbitrary number of followers in the concrete graph semantics. A summary node, for which all possible concretizations may occur, is called *inductive*. The inductiveness of (C_8, u_5) was derived by finding two 1-increment paths (*c.f.*, Definition 4.2.3):

$$(C_3, u_1)\ \mathfrak{E}([\text{INITMERGE}], 0)\ (C_5, u_2)\ \mathfrak{E}([\text{LDR2FLW}], 1)\ (C_8, u_5) \quad (4.17)$$

$$(C_8, u_5)\ \mathfrak{E}([\text{INITMERGE}], 0)\ (C_7, u_4)\ \mathfrak{E}([\text{LDR2FLW}], 1)\ (C_8, u_5) \quad (4.18)$$

Notice that C_3 and C_8 are equal up to the summary status of one node. Since friendly partner graph grammars have simple partner constraints, the existence of (4.17) already implies the existence of path (4.18). Remember that simple partner constraints cannot express "at most" conditions. Hence a rule applicable to C_3 is also applicable to C_8.

Another ingredient making the 1-increment argument feasible is cluster multiplicity as implied by the friendliness of the partner graph grammar. There is an infinite supply of all clusters that are necessary to execute the paths (4.17) and (4.18), in this case an infinite supply of free agents.

A summary node such as (C_8, u_5) serves as a *generator* for other summary nodes. In fact, all other summary nodes in Figure 4.7 may be obtained in one abstract derivation step by 0-increment paths from (C_8, u_5). This proves their inductiveness. In general abstract graph semantics, there may be an arbitrary number of such generators.

The previous arguments reveal a sufficient condition for cluster completeness: a *generating order*. If the clusters of an abstract graph semantics can be ordered in the following manner, then the abstract graph semantics is cluster complete. All involved abstract clusters must have lone summary nodes as elaborated on earlier. Clusters that have an ∞-summary node are called ∞-clusters.

1. Start with create rules. The abstractions of their right graphs should not contain ∞-summary nodes and become the minimal elements of the generating order.

2. Find a number of generator ∞-summary nodes. They are proven to be inductive by 1-increment paths in the smaller part of the abstract graph semantics. No ∞-summaries may occur in this smaller part. The clusters containing such summary nodes are minimal among ∞-clusters.

3. All other ∞-summaries must evolve, potentially transitively, from a generator summary node by 0-increment paths.

These three requirements are formalized in Definition 4.3.5. They are satisfied, if there exists a *strict order* on the clusters of an abstract graph semantics. This order must adhere to the requirements 1, 2, and 3 above. Due to the finiteness of the abstract graph semantics such a strict order will also be well-founded. Accordingly, the proof of the cluster completeness theorem will be by well-founded induction.

Here is a quick reminder of some of the notions used in the following definition. A more detailed introduction to the underlying theory may be found in various textbooks, *e.g.* in [DP05]. A *strict order on a set* A is a binary relation $<$ that is transitive and irreflexive. It is hence also antisymmetric. An element $a \in A$ is *minimal*, if for all $b \in A$ $b < a$ implies $b = a$. Because of $<$'s irreflexive ness, this is equivalent to saying, that a is minimal, if there is no $b \in A$ such that $b < a$. The definition of maximal elements is analogous. A strict order on A is well-founded, if each subset of A has at least one minimal element. This is equivalent to the absence of infinite-descending chains in A *wrt.* this given order. Given a well-founded strict order on A one may prove properties of all elements of A by *well-founded induction*. In a well-founded induction prove a property is shown for all minimal elements first. Then it is shown for an element a by assuming the property for all $b < a$.

Definition 4.3.5 (Generating Order) *Let $\mathfrak{G} = (\mathcal{R}, \mathcal{I})$ be a friendly partner graph grammar, and let $[\![\mathfrak{G}]\!]^k$ be its abstract graph semantics. Let $<:$ be*

a strict order on $[\![\mathfrak{G}]\!]^k$*, and let* $\mathcal{S}$ *be the set of abstract clusters with at least one* ∞*-summary node, called the set of* ∞-clusters*. The strict order* $<:$ *is called* generating*, iff*

1. *An abstract cluster* $\hat{C}$ *is minimal* wrt. $<:$*, if and only if there exists a create rule* $(L, h, p, R) \in \mathcal{R}$ *such that* $\hat{C} \in \alpha_k(R)$*.*

2. $\hat{C} <: \hat{D}$ *implies* $\hat{C}\ \mathfrak{E}(\mathcal{R})^+\ \hat{D}$*, where* $\mathfrak{E}(\mathcal{R})^+$ *is the transitive closure of the union of* $\mathfrak{E}(r)$ *over all* $r \in \mathcal{R}$*.*

3. *Each* $\hat{C} \in \mathcal{S}$ *has exactly one* ∞*-summary node, unless it is maximal and not minimal.*

4. *If* $\hat{C}$ *is minimal in* $\mathcal{S}$ *with the only summary node* $\hat{u} = (\nu, P, \infty)$*, then there exists an abstract cluster* $\hat{C}' <: \hat{C}$ *that is equal to* $\hat{C}$ *except that* $\hat{u}$ *is replaced by* $\hat{u}' = (\nu, P, k)$*. Moreover, there exists a 1-increment path from* $(\hat{C}', \hat{u}')$ *to* $(\hat{C}, \hat{u})$ *such that all affected clusters are smaller than* $\hat{C}$*.*

5. *For each* ∞*-summary node* $(\hat{C}, \hat{u})$ *that is not minimal, there exists a 0-increment path of length 1 from another* ∞*-summary node such that all affected clusters are smaller than* $\hat{C}$*.*

6. *If* $\hat{C}$ *is maximal and has more than one* ∞*-summary node, then all* ∞*-summary nodes of* $\hat{C}$ *are due to node evolutions using the same rule and the same triggering materialization.*

Remember the aforementioned comments on the cluster completeness of the abstract graph semantics of the simple merge maneuver grammar as given in Figure 4.7. This abstract graph semantics has indeed a generating order:

$$C_1 <: C_2 <: C_3 <: C_4, C_5, C_6 <: C_8 <: C_7, C_9, C_{10}, C_{11}, C_{12} \qquad (4.19)$$

In equation (4.19), incomparable arguments are separated by commas. Among the clusters with ∞-summary nodes, C_8 is the only minimal one. As argued earlier and as witnessed by equations (4.17) and (4.18), there is a 1-increment path according to (4) of Definition 4.3.5 and 0-increment paths according to (5). The only minimal element is cluster C_1, which is the only one corresponding to the create rule [Create]. The only element featuring two ∞-summary nodes is C_9, which is maximal.

Generating orders are the exact formalization of a sufficient condition for cluster completeness. Unfortunately, no efficient algorithmic solution to the problem of the existence of a generating order is known so far. The designer

of a protocol, however, may have a good intuition to find a generating order, once she is provided with the abstract graph semantics of her partner graph grammar. She may then use generating orders to easily prove the cluster completeness or even the decidability of her abstract graph semantics.

Theorem 4.3.4 (Cluster Completeness) *Let $\mathfrak{G} = (\mathcal{R}, \mathcal{I})$ be a friendly partner graph grammar and let*

$$k \geq \max\{|\,\ell_L(\nu)^{-1}\,| \mid (L, \_, \_, \_) \in \mathcal{R}, \nu \in \mathcal{N}\}$$

The abstract graph semantics $[\![\mathfrak{G}]\!]^k$ is cluster complete, if it has a generating order and all $\hat{C} \in [\![\mathfrak{G}]\!]^k$ have lone summary nodes and unique partners.

The cluster completeness result is established using the following observations made precise in the proof in Appendix A.

A first observation relates to the subtle differences between materializations and concretizations and introduces special kinds of S-materializations. For the abstract graph semantics under concern, Lemma 4.3.2 shows that all materializations are S-materializations. On the other hand, each concretization C is obviously isomorphic to the materialization $\mathit{partner}_k(C)$. In the latter, all nodes have the multiplicity 1. Such a materialization is called an S_1-materialization. Hence the set of concretizations equals the set of S_1 materializations for a graph with lone summary nodes (up to isomorphism). An important property of S-materializations is, that partner equivalent nodes in an S-materialization have in fact *equal* partners, whereas partner equivalence only requires the labels of partners to be equal.

This special notion of materialization facilitates the proof that d-increment paths – a purely static notion – can be obtained in the concrete graph semantics as well. In other words, if $(\hat{C}, \hat{u})\ \mathfrak{E}(r, n)\ (\hat{C}', \hat{u}')$ and $\hat{C}$ has a concretization, in which $\hat{u}$ has t instances, then $\hat{C}'$ has a concretization, in which $\hat{u}'$ has $n + t$ instances. Note that in this case the standard plus on integers is used – in contrast to the finite plus that counts up to k. In order to prove this statement, the restriction on the analysis parameter k becomes crucial. The statement holds only, if the number of nodes mapped to an ∞-summary node $\hat{u}$ by some match is smaller than k. Hence k is safely chosen to be greater than this number by exceeding the maximal number of equally labeled nodes in any left graph of any transformation rule.

Given the previous reasoning,the proof of cluster completeness works by well-founded induction over the generating order. In the smallest part of the order, there are no ∞-summary nodes at all. This makes the abstraction precise, because there are unique S_1-materializations. Secondly, there are the

∞-summary nodes in cluster that are minimal among the ∞-clusters. They are assumed to have 1-increment paths in the smaller part. By induction hypothesis and the observation made above, the existence of such a path in the concrete graph semantics is guaranteed. By another inductive argument it follows, that these summary nodes are inductive. The final step uses this result to show that all the remaining summary nodes have all their concretization. A case distinction is necessary distinguishing maximal and non-maximal abstract clusters. Only the latter may have more than one ∞-summary node.

Remember that cluster completeness was not the weakest of the completeness notions. Rather there was also the notion of absence of spurious clusters. A sufficient condition for this notion may certainly be expected to be weaker than that of generating orders. It is called an *almost generating order*. The definition becomes obvious when reading the proof of the cluster completeness theorem in Appendix A. The requirements imposed on clusters in a generating order are also imposed on all clusters in an almost generating order *except for the maximal ones*. This implies that for all except for the maximal clusters in the order cluster completeness holds. For the maximal ones, however, the existence of one concretization in the concrete graph semantics can be shown.

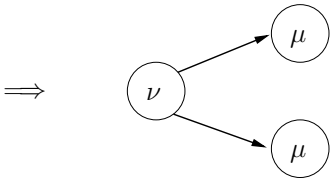
Definition 4.3.6 (Almost Generating Order) *Let $\mathfrak{G}$ be a friendly partner graph grammar, and let $[\![\mathfrak{G}]\!]^k$ be its abstract graph semantics. Let $<:$ be a strict order on $[\![\mathfrak{G}]\!]^k$, and let $\mathcal{S}$ be the set ∞-clusters. The strict order $<:$ is called* almost generating*, if and only if requirements (1-3) of Definition 4.3.5 hold for all clusters, and requirements 4 and 5 hold for all but the maximal clusters.*

Theorem 4.3.5 (Absence of Spurious Clusters) *Let $\mathfrak{G}$ be a friendly partner graph grammar and let*

$$k \geq \max\{|\,\ell_L(\nu)^{-1}\,| \mid (L, \_, \_, \_) \in \mathcal{R}, \nu \in \mathcal{N}\}$$

The abstract graph semantics $[\![\mathfrak{G}]\!]^k$ is free of spurious clusters, if it has an almost generating order, and lone summary nodes and unique partners for all clusters except the maximal ones.

Before the decidability of the word problem for partner graph grammars is handled, two concluding remarks regarding cluster completeness and the absence of spurious clusters are given. These concepts seem very similar, but they are in fact not equal. Consider this friendly partner graph grammar consisting of only one create rule:

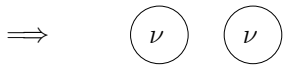


The abstract graph semantics of this partner graph grammar consists of only one trivial strict order, which is an almost generating but not a generating order. The only element of the abstract graph semantics is both a minimal and maximal element *wrt.* the trivial order. Hence requirements (4) and (5) of the definition of generating orders (Definition 4.3.5) do not need to hold. This means that there does not need to be a 1-increment path for the abstract cluster. The abstract graph semantics is still free of spurious clusters.

One concluding remark concerning generating orders and relating to the previous create rules: If the abstraction of the right graph of a create rule has an ∞-summary node, it must be maximal in any *almost* generating order. There is no generating order in that case. Such an abstract graph semantics cannot be proven cluster complete by using the notion of generating orders.

4.3.4 Decidability of the Word Problem

Cluster completeness is still not enough to decide the word problem for partner graph grammars. Consider the following friendly partner graph grammar that consists of one create rule:



It is obviously cluster complete, but not all possible combinations of concretizations of abstract clusters actually occur in the concrete graph semantics. If they did, the concrete semantics would consist of graphs of n singleton nodes for any n. In fact, the concrete semantics consists of graphs of $2n$ singletons for any n.

The reason for that problem is, that two clusters are created simultaneously thus dependently As the abstract graph semantics is a set, it cannot record these dependencies. If all rule applications used up the graph they match and created exactly one connected graph, this source of dependencies could be avoided. A very simple, statically checkable, sufficient criterion avoiding this problem is the connectedness of right graphs of transformation rules.

If the partner graph grammar has the cluster multiplicity property and the right graphs of all rules are connected, then an arbitrary number of

concretizations of each abstract cluster in arbitrary combinations may be derived by the partner graph grammar. Because of cluster completeness, each concretization of a given abstract cluster is possible.

Theorem 4.3.6 (Decidability) *Let $\mathfrak{G} = (\mathcal{R}, \mathcal{I})$ be a friendly partner graph grammar and let*

$$k \geq \max\{|\,\ell_L(\nu)^{-1}\,| \mid (L, \_, \_, \_) \in \mathcal{R}, \nu \in \mathcal{N}\}$$

The abstract graph semantics $[\![\mathfrak{G}]\!]^k$ is word decidable, if it is cluster complete, and if R is connected for all $(L, h, p, R) \in \mathcal{R}$.

The formal proof of Theorem 4.3.6 is given in Appendix A. Notice that the abstract graph semantics of the simple platoon merge implementation given in Figure 4.7 is even word decidable, because all rules of Table 3.1 have connected right graphs. To be exact, word decidability can only be proven for an abstraction using $k = 2$ – in contrast to $k = 1$ that is used in Figure 4.7. In the course of the case study evaluation in Section 4.5 more details about the completeness status of the remaining platoon case studies of Section 3.1 are revealed.

4.4 Property Preservation

The aim of this section is to investigate which $\mathcal{GL}$ properties are preserved under abstraction. More precisely, a property is preserved, if the fact, that it holds of the concrete case implies that it also holds for the abstract case. As $\mathcal{GL}$ properties will be evaluated on abstract graph semantics', it may then be concluded that a property does not hold of a partner graph grammar, if it can be shown not to hold of the abstract graph semantics. The inverse direction is both more difficult to show and happens more seldom. However, for complete abstract semantics', some results can be achieved.

The preservation of first-order properties under partner abstraction is explored in the beginning of this section. This means given a *first-order logic formula* φ and a concrete graph G, what is the relation between $G, \rho \models \varphi$ and $\dot{\cup}\alpha_k(G), \hat{\rho} \models \varphi$?

The result depends both on the kind of formula and on the considered graphs. Only existential positive properties are preserved in the general case. In the case of abstract clusters with lone summary nodes, however, arbitrary first-order formulas are preserved under partner abstraction.

Note that no new definition for the evaluation of first-order formulas on abstract graphs is necessary, because they are ordinary graphs except for

their special node identities. The meaning of node identities (a node being summary node, information about its adjacent nodes), however, is not visible to a first-order formula.

As for the preservation of temporal properties, almost all structure of the concrete graph transition system induced by a partner graph grammar is lost by going to the abstract graph semantics. The notions of node and cluster evolution defined in Section 4.2 mainly served as a means of defining the completeness notions of Section 4.3. Therefore, only *invariants* and *possibilities* are investigated, whereas general temporal properties are left for further research. In the scope of this section are formulas of the form $\mathsf{AG}_R\ \phi$ and $\mathsf{EF}_R\ \phi$ for first-order formulas ϕ. Again, due to the simplicity of these cases, no new definition of a $\mathcal{GL}$ semantics for the abstract case becomes necessary, as will be seen in the theorems of this section.

4.4.1 Property Preservation and Partner Abstraction

Initially, some classes of first-order logic formulas are identified and named. All first order formulas will be used in prenex form, *i.e.*, in the form $Q_1 x_1 \ldots Q_n x_n.\ \phi$, where Q_i are existential or universal quantifiers and where ϕ consists of quantifier-free disjunctions and conjunctions of positive or negative literals. Here, literals are of either one of the forms $\nu(x)$, $\beta(x_1, x_2)$, or $x_1 = x_2$. A formula is called *existential positive*, if $Q_i = \exists$ for all i and all literals occur without negation. Dually, a formula is called *universal negative* if $Q_i = \forall$ for all i and all literals are negative. Finally, *existential formulas* are those, where $Q_i = \exists$ for all i without any restrictions on the literals. Analogously, one speaks of *universal* formulas, if $Q_i = \forall$ for all i.

As partner abstraction constitutes a morphism – *c.f.* Lemma 4.1.1 – existential positive properties are preserved under partner abstraction, which is a very general result (*c.f.* [Ros05] for current research on property preservation under morphisms).

Lemma 4.4.1 (Property Preservation) *Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ and let $k \geq 1$. For all existential positive formulas ϕ holds: If $G, \rho \models \phi$, then $\dot{\cup}\alpha_k(G), \hat{\rho} \models \phi$, where $\hat{\rho} = \lambda x.\xi(\rho(x))$.*

The almost trivial proof is given in Appendix A. A much stronger result can be established in the case of lone summary nodes and unique partners. Lemma 4.3.2 showed that all concretizations of ground canonical graphs with lone summary nodes are essentially S-materializations. The following theorem states that S-materializations preserve *precisely* the validity of first-order formulas without equality. In other words, the only loss of information is in

the number of of nodes summarized to ∞-summary nodes, whereas structure is completely preserved. The theorem is stated in terms of connected graphs to simplify the proof. Its proof is given in Appendix A.

Theorem 4.4.2 (Property Preservation) *Let $C \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a connected graph and let $k \geq 1$. Let ϕ be a first order formula without equality. If $\{\hat{C}\} = \alpha_k(C)$ with induced morphism ξ has lone summary nodes and unique partners, then*

- *$C, \rho \models \phi \;\Rightarrow\; \hat{C}, \hat{\rho} \models \phi$, for any assignment ρ and $\hat{\rho} = \lambda x.\xi(\rho(x))$.*
- *$\hat{C}, \hat{\rho} \models \phi \;\Rightarrow\; C, \rho \models \phi$ for all assignments $\hat{\rho}$ and for all assignments ρ, such that $\rho(x) \in \xi^{-1}(\hat{\rho}(x))$.*

The choice of partner abstraction was mainly motivated by the application domain of dynamic communication systems, where the similarity of objects not only depends on their states but also on the states of their communication partners. The previous theorem presents another strong justification. Although being a merely technical and logical reason, partner abstraction has exactly the right precision to precisely preserve first-order properties without equality in some well-defined cases. Such graphs that are indistinguishable by first-order formulas without equality are equal under partner abstraction.

4.4.2 Invariants

After the evaluation of first-order formulas on partner abstracted graphs has been clarified, the stage is set to investigate the preservation of temporal formulas in the abstract graph semantics. As mentioned in the introductory remarks to this section, only formulas of the form $\mathsf{EF}_R\ \phi$ and $\mathsf{AG}_R\ \phi$ are considered here. Also the treatment of restrictions R is postponed to a later paragraph. So $R = \mathcal{R}$ is assumed for now, and the R index to the temporal quantifiers is left out.

The first theorem deals with forbidden subgraphs and reads as follows. If there finally exists a graph in the concrete graph transition system, such that an existential positive formula ϕ holds true of it, then ϕ will hold true of the disjoint graph union of all abstract clusters in the abstract graph semantics. Note that this theorem holds for all partner graph grammars. It is called forbidden subgraph, because it is usually applied in its counterpositive form. If a formula does not hold in the disjoint graph union of all abstract clusters in the abstract graph semantics, there will never occur a graph in the concrete graph transition system of which the formula holds true. Moreover, existential positive formulas essentially describe subgraphs, hence the name of the theorem.

Theorem 4.4.3 (Forbidden Subgraphs) *Let $\mathfrak{G}$ be a partner graph grammar and let $k \geq 1$. Furthermore, let ϕ be a closed, existential positive formula. If $\mathfrak{G} \models \mathsf{EF}\ \phi$, then $\dot{\cup}[\![\mathfrak{G}]\!]^k, [] \models \phi$.*

The proof is trivial using soundness (Theorem 4.1.8) and the preservation of existential positive formulas as stated in Lemma 4.4.1. In fact, using a coding trick in the partner graph grammar, even soundness alone suffices to show the absence of a spurious subgraph: Consider graph F to be the forbidden subgraph. Add a special error label to the set of node labels and add an extra rule to the grammar, that has F as its left graph and a single node labeled by the error label as the right graph. This error label must not occur anywhere else in the grammar. If this error labeled node does not occur in the abstract graph semantics, then, by soundness, it does not occur anywhere in the concrete graph semantics proving the absence of F.

The next and much stronger theorem holds only for abstract graph semantics without spurious clusters and for word decidable abstract graph semantics, respectively. It makes use of the strong property preservation result holding for abstract clusters with lone summary nodes. The statements of the theorem read as follows. For a cluster complete abstract graph semantics without spurious clusters and with lone summary nodes: If there exists an abstract cluster in the abstract graph semantics, of which an existential formula holds true, then eventually, on some direct derivation in the concrete graph transition system, this formula will hold of a graph. Moreover, a universal formula that does not hold of some abstract cluster, implies, that the same formula cannot hold globally of all graphs on all paths in the concrete graph transition system.

The results are even much stronger in the case of word decidable abstract graph semantics. In this case, the abstract graph semantics captures precisely the concrete graph semantics (but not the concrete graph transition system). It comes as no surprise, that all invariant properties of the concrete graph transition system are thus checkable in the abstract graph semantics.

Note that Theorem 4.4.4 enables the user to infer almost arbitrary invariants of the concrete graph transition system by simply looking at the bounded size abstract graph semantics. In particular, the existential and universal formulas may contain both positive and negative literals at the same time. The proofs are an immediate consequence of the cluster completeness or the word decidability theorem together with the property preservation theorem. They are stated in Appendix A.

Theorem 4.4.4 (Preservation in Complete Cases) *Let $\mathfrak{G}$ be a partner graph grammar and let $k \geq 1$. Furthermore, let $\phi_\forall$ be a closed universal*

formula and let $\phi_\exists$ be a closed existential formula. If $[\![\mathfrak{G}]\!]^k$ is cluster complete and all $\hat{C} \in [\![\mathfrak{G}]\!]^k$ have lone summary nodes, then

1. *If there exists a $\hat{C} \in [\![\mathfrak{G}]\!]^k$, such that $\hat{C}, [] \models \phi_\exists$, then $\mathfrak{G} \models \mathsf{EF}\, \phi_\exists$.*
2. *If there exists a $\hat{C} \in [\![\mathfrak{G}]\!]^k$, such that $\hat{C}, [] \not\models \phi_\forall$, then $\mathfrak{G} \not\models \mathsf{AG}\, \phi_\forall$.*

If $[\![\mathfrak{G}]\!]^k$ is word decidable and all $\hat{C} \in [\![\mathfrak{G}]\!]^k$ have lone summary nodes, then for any closed first order formula ϕ that may contain equality

a. *$\mathfrak{G} \models \mathsf{EF}\, \phi$ if and only if there exists an S_1-materialization M of $\dot{\cup}[\![\mathfrak{G}]\!]^k$, such that $M, [] \models \phi$.*
b. *$\mathfrak{G} \models \mathsf{AG}\, \phi$ if and only if for all S_1-materializations M of $\dot{\cup}[\![\mathfrak{G}]\!]^k$ holds, that $M, [] \models \phi$.*

4.4.3 Extensions

Two theorems have stated sufficient conditions that can be evaluated on the bounded-size abstract graph semantics guaranteeing the absence or the presence of invariants in the concrete graph transition system. These theorems may be extended in several ways that are not fully formalized in this work. Rather they are left for further investigation.

Beyond existential positive As made clear in the proof of Lemma 4.4.1, the existential requirement is not necessary in the statement of the lemma. Because of the totality and surjectivity of the abstraction induced morphism ξ, universal positive formulas would work, too.

Rule Restrictions The results so far were only concerned with unrestricted temporal operators. This means that all rules of a partner graph grammar are allowed to be applied. As there is no nesting of temporal operators thus no nesting of restrictions, the same results can be obtained for a restriction R, if the abstract graph semantics of the restricted system $(R, \mathcal{I})$ is used instead of the abstract graph semantics of the original system $(\mathcal{R}, \mathcal{I})$.

Partial Cluster Completeness The requirements of Theorem 4.4.4 are rather strong. They may be relaxed for statements 1 and 2 of the theorem. If cluster completeness is shown by Theorem 4.3.4, a generating order on the abstract graph semantics is required. If there was a generating order on a *subset* of the abstract graph semantics including the abstraction of the right graphs of create rules, cluster completeness

could be guaranteed for the abstract clusters of this subset. This is intuitively clear and becomes formally clear by looking at the proof of the cluster completeness theorem. Statements 1 and 2 of Theorem 4.4.4 also hold, if $\hat{C}$ is part of this (partial) generating order.

General Temporal Properties In order to reason about general temporal properties like until, or nested temporal quantification, a better notion of cluster evolution is necessary. This should not be a binary relation between clusters but should take all the affected clusters into account, too. Also abstracting from the number of clusters makes life hard for temporal verification. A method how to count clusters is proposed in Section 4.6. Another direction of future research may hence be to incorporate model-checking techniques into the verification process as suggested in [BSTW06]. There is some hope, that they are able to better deal with the temporal aspects, whereas the technique of this work is ideally suited to analyze structural properties.

Node Evolution The same argument about refinement of cluster evolutions applies for node evolution, too. Properties of the form $\exists x.\mathsf{AG}\,\phi$ are out of scope of reasoning so far. Such a formula expresses, that there is some node u in the initial graph such that property ϕ holds for this fixed u during its lifetime. Logics and systems arguing about such properties include [DKR04] and [YRSW03]. These techniques, however, work only for settings less expressive than partner graph grammars.

Whereas node evolution as defined here is tailored to express relations between ∞-summary nodes, a new notion reasoning about non-summary nodes seems promising in order to tackle formulas of the above kind.

4.5 Evaluation of Case Studies

The purpose of this section is to evaluate the case studies of Chapter 3 experimentally. It is divided into two major parts. First, the implementation of partner abstraction based abstract interpretation of graph grammars, the `hiralysis` tool, is introduced. Second, the implementation of the car platooning case studies are given to the tool and the results are commented and explained. Also, the car platooning examples are examined *wrt.* completeness and property preservation.

The section concludes by giving some outlook on how to tackle further case studies in the future. Although being amenable to partner abstraction

based abstract interpretation of partner graph grammars in general, they are not amenable to the current implementation yet. In some cases, even the technique as is needs to be extended. Potentially worthy extensions will be described in a subsequent section on counting and general clusters.

4.5.1 Implementation: `hiralysis`

The `hiralysis` tool is written in 3000 lines of standard C code. Its functionality is rather simple. It gets as input a textual description of a partner graph grammar $\mathfrak{G}$ adhering to the abstract syntax given in Appendix B. It may then be called with an optional number of iterations to be executed. The tool produces a representation of the abstract graph semantics $[\![\mathfrak{G}]\!]^1$ of the specified partner graph grammar. This means, that the partner abstraction parameter of `hiralysis` is fixed to $k = 1$. Apart from that, the tool also visualizes the specified partner graph grammar in a separate output file. In case an iteration number n was specified, it produces as output a representation of $[\![\mathfrak{G}]\!]^1_n$. The graph is represented in the `.gdl` format, an XML-like description of graphs. This output can be visualized with the `aisee`® tool that builds on VCG graph visualization [San94]. A free version is downloadable for non-commercial usage from `www.aisee.com`. Sample input and output will be displayed and explained later in this section and in Appendix B.

After the functionality of `hiralysis` has been described, some implementation details follow. Most interesting among them are the internal representation of graphs, the match finding, the handling of isomorphism checks and the implementation of materialization. The overall algorithm is a fixpoint computation closely following the definition of the abstract graph semantics in Definition 4.1.10. Starting from the abstracted initial graph, all newly computed abstract clusters are added to the current abstract graph semantics in each iteration until a fixpoint is reached, which is guaranteed by Theorem 4.1.8.

Graphs are represented internally as *adjacency lists*, which is one of the two standard ways of representing graphs the other being adjacency matrices. A detailed comparison of the two data structures can be found in Section 23.1, pages 465ff of the standard data structure and algorithms book [CLR89]. There are several reasons why adjacency lists are to be preferred to adjacency matrices for this particular application. First of all, one of the most often applied operations on the data structure is finding the set of all adjacent nodes, partners, of a given node. This is important for computing partner abstractions and connected components. It is straightforward in adjacency

lists, whereas it requires some overhead in the matrix representation. Furthermore, the considered graphs seem to be reasonable sparse, where sparse graphs are more efficiently implemented using lists. Finally, space becomes an issue, if there are many graphs around, which is conjectured to happen in abstract graph semantics. In this case the more space-efficient adjacency lists win over the matrix representation.

Matching left graphs against abstract clusters is an exponential operation in the worst case, because it amounts to subgraph isomorphism. It is not that bad, however, because, usually, left graphs are reasonably small and because all graphs are labeled. This restricts the set of candidates to match a node to equally labeled nodes. In each abstract cluster, there are supposedly not too many equally labeled nodes (even though there may be an exponential number of them, if they all differ by their partners' labels). For each node in a transformation rule pointers to equally labeled nodes in all abstract clusters are stored with it. This helps to compute new matches. Analogously, pointers to the abstract clusters that a connected component of a left graph has matched, are stored with this connected component. In each iteration step, only the most recently added abstract clusters are searched for matches. The number of the iteration, in which it was computed is stored together with an abstract cluster. This guarantees that the same rule with the same match is not applied more than once.

Abstract clusters that have been found to be in the abstract graph semantics are stored in a binary search tree. The key of an abstract cluster in the search tree corresponds to a sorted list of all canonical names of all its nodes. Lemma 4.1.2 guarantees that isomorphic canonical graphs are in fact equal (if the multiplicities coincide). A necessary criterion is certainly the equality of the node sets. The latter corresponds to checking key equality, where the keys are also called footprints of abstract clusters. Footprint equality can be checked efficiently. In case footprints are equal an additional isomorphism check becomes necessary. However, the potential isomorphism is already determined by the equal node identities simplifying the search. It becomes effectively linear in the size of the graph. In any case, the management of the search tree can be handled efficiently. Note that the search tree only supports addition of clusters due to the monotone nature of the fixpoint computation.

The last implementation detail is about how to compute materializations. The definition of materialization from Definition 4.1.6 is not constructive mainly for mathematical reasons. Especially, the soundness proof is simplified a lot using the declarative notion of Definition 4.1.6. Note that also the notion of an abstract match is easily formalized using the declarative

notion of materializations. Lemma 4.1.4 about the existence of materialization gave an equivalent characterization of an abstract match using non-injective morphisms. In fact, this is the characterization that is implemented in `hiralysis`.

The constructive definition is tedious to write down, because a lot of cases need to be distinguished depending on the connectivity of left graphs in rules. For the purpose of this presentation, only two cases are explained in more detail: left graphs of transformation rules consisting of one or two connected components, respectively.

The declarative notion of materialization is also very liberal *wrt.* the multiplicities of nodes materialized from a summary node $\hat{u}$. It merely requires that the finite sum of the materialized nodes yields the multiplicity of $\hat{u}$. It is sufficient to materialize only as many non-summary nodes from a summary node $\hat{u}$ as there are nodes in the left graph matching $\hat{u} \in V_{\hat{C}}$. (Again, the non-injective match characterization of abstract matches is used here.) Assume nodes $u_1, \ldots, u_n$ of a left graph matched to the ∞-summary node $\hat{u}$ by a match morphism m. For $n > 1$, there are three node materializations that sufficiently describe all possible cases. The first one splits $\hat{u}$ in n non-summary nodes, the second one splits $\hat{u}$ in $n+1$ non-summary nodes, and the last one splits $\hat{u}$ in one ∞-summary node and n non-summary nodes. Notice that for $k = 1$ as in the implementation, all summary nodes are also ∞-summary nodes. For $n = 1$, only two cases are necessary: two non-summary nodes, and one non-summary, one ∞-summary node. The three cases above capture all possible behavior that may be triggered by the abstract cluster $\hat{C}$.

Apart from node materialization, also cluster materialization needs to be defined constructively for implementation purposes. If the left graph of a rule consists of a single connected component only, matching and thus cluster materialization is easy. A match can only occur within one abstract cluster. If an abstract cluster $\hat{C}$ is matched, an identical copy of it is made and the matching rule applied to that copy.

If the left graph of a rule consists of two connected components L_1 and L_2, there are three cases to be distinguished. First, $L_1 \dot{\cup} L_2$ may match a single abstract cluster $\hat{C}$. In this case, either one or two copies of $\hat{C}$ are made and updated according to the match. Thirdly, L_i may match $\hat{C}_i$ for distinct $\hat{C}_1$ and $\hat{C}_2$. In this case, one identical copy of each $\hat{C}_1$ and $\hat{C}_2$ has to be made, and their disjoint graph union needs to be updated as specified by the transformation rule. Without dwelling on details, it becomes clear that this leads to a combinatorial blowup if a left graph consists of more than two connected components, which is, on the other hand, not very likely to

happen.

If the user is certain as to when a rule with a left graph of several connected components is to be applied, she can aid the tool using the keywords `connected` and `disjoint` to simplify the materialization process. Each of the keywords can be added to a rule. If a rule is tagged to be `connected`, although the left graph has several connected components, the tool only seeks matches within one abstract cluster. On the other hand, a `disjoint` rule of n connected components in the left graph is only matched against n *distinct* abstract clusters. Examples of the usage of the connectivity keywords are numerous in Appendix B.

4.5.2 Car Platooning

Recall the car platooning case study described in Section 3.1. It consisted of three categories of examples:

1. A simplified version, where actual messages are abstracted away.
2. A version with explicit message queues.
3. General ideas how to implement and how to overcome faulty channels.

All of these instances of the platoon scenario were implemented as a partner graph grammar and fed into the `hiralysis` tool. The results and experiences are discussed below. Some of the material, especially graphical, was put into Appendix B for illustration.

Idealized Platoons – Section 3.1.1 The merge part of the idealized platoon scenario has occurred frequently throughout this work. In particular, its abstract graph semantics was presented in Figure 4.7. In Section 4.3, it was shown, that this particular abstract graph semantics is even word decidable. Hence the sample properties (3.1) and (3.2) are amenable to verification given the abstract graph semantics according to Theorem 4.4.4. In fact, they both turn out to hold for the idealized merge. The abstract graph semantics of this grammar consists of 12 abstract clusters and is computed in 5 iterations using 19 rule applications.

After considering the merge in isolation, the splitting maneuvers are investigated in isolation taking the representation of a platoon of three cars as initial graph. Remember that the split maneuver was modeled by two sets of rules in Table 3.2 and Table 3.3. The transformation rule [2Split], which is an element of the latter set, makes use of a non-simple partner constraint.

Hence it is not amenable to the technique of partner abstraction based abstract interpretation that is only sound for partner constraints without "at most" restrictions.

The first set of rules models the split in the middle of a platoon. It was implemented as given in Table 3.2. The first experiment yielded an abstract graph semantics of more than 500 abstract clusters revealing a flaw in the specification, that was due to the fact, that several cars wanted to initiate a split within the same platoon. After this possibility was excluded in the model, the abstract graph semantics as computed by `hiralysis` is made up of 13 abstract clusters that were computed in 4 iterations applying 15 rules. The abstract graph semantics cannot be proven to be cluster complete. In fact, it is not, because the initial graph represents only three cars and no other cars are created. But even under the assumption that there were platoons of all possible sizes to start with, completeness could not be proven using the completeness theorems. This gives rise to certain points, where the sufficient conditions for completeness could be relaxed. For instance, one could think of d-increment paths with $d \leq 0$ instead of 0-increment paths in the definition of generating orders. This would justify the cluster completeness of the split partner graph grammar. Property (3.1) holds for the split case, whereas (3.2) does not, because a single leader that is not a free agent is an element of the abstract graph semantics.

The validity of these properties was not inferred by automatic reasoning, but by human reasoning. Still, the occurrence of a single leader without followers in the abstract graph semantics may be suspicious to a protocol designer.

Eventually, merge and split are put to run in parallel, *i.e.*, a new partner graph grammar was built using the union of the rules for merging and splitting. The simple union resulted in an abstract graph semantics of 169 abstract clusters. This demonstrates impressively, that partner graph grammar specifications are not very modular. Putting two specifications together leads to a lot of undesired interferences as suggested by this example. Here, those interferences result from initiating merge operations for currently splitting platoons. If these interferences are manually excluded in the protocol, only 22 abstract clusters remain. This corresponds to the sum of the sizes of the abstract graph semantics' of merge and split alone except for 3 clusters that are in the intersection of the abstract semantics' – free agents, platoons of three or more cars, and platoons of two cars.

Merge with Queues – Section 3.1.2 While the idealized scenarios studied so far abstract from concrete messages, actual messages are around in the

asynchronous communication based platoon implementation. First attempts to implement the protocol as specified in Section 3.1.2 revealed numerous flaws, that are all easily detectable. If the abstract graph semantics of the erroneous partner graph grammar is computed, it generates thousands of abstract clusters or even makes `hiralysis` run out of memory. A quick look at the abstract clusters immediately reveals the origin of the explosion. The `hiralysis` tool proved very helpful in correcting the flaws and designing a more reliable protocol, since the reasons for flaws are easily identifiable given the graphical notion.

An example of such discovered errors: The concept of environment messages is too liberal as is. Such messages can easily swamp the message queues of arbitrary processes. (Nodes represent processes in terms of the specification given in Section 3.1.2.) It is easily possible to have several merges start simultaneously. This lack of mutual exclusion was already present in the idealized version, so it comes as no surprise. The key to restricting the system behavior – as already observed in [BSTW06] – is the introduction of acknowledgement messages into the protocol.

Apart from protocol-inherent problems, there were also some technical properties that need to be discussed. The repaired protocol yielded an abstract graph semantics of reasonable size (several 100s), but with a large amount of spurious abstract clusters. These were mainly due to summary messages, *i.e.* summary nodes representing messages. Once such a summary message occurs, it will, at least in some clusters, always prevail, because information about the concrete number of such messages is lost. This spoils the precision about the content of the queues. It may, however, still be enough to show properties of the mere communication topology regardless of message queues, if the queues are abstracted from the results. For the final implementation, whose results are reported here, the queue length was restricted to 1.

A minor technical problem is the control structure that needs to be included in the graph grammar to model the broadcast of messages. This issue was already discussed in detail in Section 3.1.2. Therefore, broadcast was replaced by picking a random identity among those present in a channel.

The so obtained partner graph grammar yields an abstract graph semantics of 159 abstract clusters that were computed using 163 rules in 40 iterations. Most notably, 34 partner constraints were used in only 30 rules to obtain this partner graph grammar. This demonstrated how many restrictions needed to be introduced to make the protocol work. As expected, the result cannot be shown to be complete regardless of the completeness notion. Among the $\mathcal{GL}$ properties stated in Section 3.1.2, only (3.4) is an invariant

of the implementation. It states that each car that is not a leader or a free agent has a unique leader. The abstract graph semantics does not contradict this, *i.e.*, it does not reveal a forbidden subgraph.

Faulty Channels – Section 3.1.3 In the section on how to model faulty channels with partner graph grammars, four options were named how a protocol can be analyzed *wrt.* its robustness concerning unreliable channels. These options contained the possibilities of arbitrary edges to disappear with/without notification of the incident nodes and with/without a persistent logical link. Among these options, those without notification of incident objects were chosen: [FAULT1], that does not introduce new logic links, and [FAULT3] that does.

The partner graph grammar augmenting the idealized merge with rules that delete arbitrary edges is given in Appendix B. The computation of the abstract graph semantics took 6 iterations and 53 rule applications to yield 20 abstract clusters. It presents valuable information as to which failure situations may occur when communication fails non-deterministically. If $\mathfrak{G}$ is the original partner graph grammar and $\mathfrak{G}'$ the augmented one, the difference $[\![\mathfrak{G}']\!]^k \setminus [\![\mathfrak{G}]\!]^k$ describes precisely the dangerous situations.

In the case of the idealized merge, this difference consists of 8 abstract clusters:

- 4 clusters of a single follower, leader, rear and front leader, respectively.
- Platoons of either one or more than one follower, where the leader is replaced with a front or rear leader, respectively.

If one is to *design an error recovery strategy*, the enumerated 8 situations are the ones to consider. It was also tested to add the inverse of the link deletion to the grammar as an default recovery strategy. This attempt failed, because an enormous amount of abstract clusters was created. Almost anything could be connected to anything else. This is obviously due to the fact, that the designated recovery rules also apply in situations, where no failure has occurred.

In the [FAULT3] scenario, failing physical communication links are replaced with *logical* links. This implies doubling the set of edge labels. The so extended idealized merge partner graph grammar leads to an abstract graph semantics of 450 abstract clusters in 20 iterations using 1206 rule applications. This rather large number results in iterated failures. Clusters that remain connected because of logical links, can have other physical links fail leading to a combinatorial blowup. Anything like 450 abstract clusters

	#rules	#node labels	#edge labels	#pconstraints
merge	8	5	1	1
split	4	6	1	4
combined	12	6	1	5
combined+	12	6	1	9
queues	30	18	4	34
faulty1	32	5	1	1
faulty3	32	5	2	1

(a) Partner Graph Grammars

	#Abstract Clusters	#Iterations	# Rules applied
merge	12	5	19
split	13	4	15
combined	169	10	302
combined+	22	7	34
queues	159	40	163
faulty1	20	6	53
faulty3	450	10	1206

(b) Computation of abstract graph semantics

Tab. 4.1: Experimental evaluation of the platoon case studies. The table on top shows the size of the specifications of the examples, whereas the bottom one shows the size and the complexity of the computation needed to obtain the abstract graph semantics. The **merge** program implements the idealized merge, and **split** implements the idealized split. The two versions of **combined** are obtained by simply taking the union of **merge** and **split** and by augmenting the union with more constraints. Finally, there are implementations of asynchronous communication and two selected scenarios using unreliable channels.

is hard to digest for a human reader. Therefore, the [Fault3] scenario lends itself to *testing a recovery strategy*, where additional rules take care of failed links, before even more links can fail. One may consider this scenario as orthogonal to the [Fault1] case that is better suited to design rather than to test recovery strategies by identifying potential failure situations.

To conclude this section, it must be said that even a protocol as simple as the idealized merge is far from robust *wrt.* faulty channels.

4.5.3 Experiences

Table 4.1 summarizes the numerical results obtained from running `hiralysis` on the test cases described in this work. This subsection on experiences is partitioned *wrt.* three major aspects:

- Performance of the tool and how it is influenced.
- Practicability of partner graph grammars as specification mechanism
- Usefulness of the results, that is the abstract graph semantics', that are computed by the `hiralysis` tool.

Performance The `hiralysis` tool achieved execution times in the order of fractions of second for each of the examples stated in Table 4.1. In the worst-case, partner abstraction based abstract interpretation suffers from a heavy complexity, because the number of possible abstract clusters explodes exponentially with the number of node and edge labels. In practice, however, the number of node and edge labels was not found to influence the running-time of the analysis tool to such an extent. The most complex example, **queues**, has clearly most labels but a comparably small number of abstract clusters in the abstract graph semantics, whereas an example like **faulty3** has much fewer labels, but a much bigger state space.

The number of applied rules is a much better indicator of the execution time. This number is, of course, related to the number of distinct matches of left graphs in rules. On the one hand, small left graphs in dynamic communication systems models reduce the matching burden. On the other hand, small left graphs potentially lead to many matches and may corrupt the execution time of the analysis. Therefore, grammars with more partner constraints provide more restrictive matching and less superfluous rule applications. The two extreme cases are **faulty3** and **queues**. The latter uses many partner constraints and computes a new abstract cluster with almost every rule application (159 vs. 163). In contrast, **faulty3** computes only one new abstract cluster per 3 rule applications.

In cases, where the tool ran out of memory, tests with gradually growing iteration bounds showed an explosion of the number of applied rules rather than an explosion of the number abstract clusters.

Partner Graph Grammars as a Specification Tool The `hiralysis` input language (consult Appendix B for its abstract syntax) does not provide many syntactical conveniences yet. This becomes clear by looking at the

implementation of **faulty1** (also available in Appendix B). The next step in the tool development must provide such conveniences, *e.g.* label constraints as defined in Definition 2.1.11 and frequently used in this work. In the long run, one may even think of a graphical input language. So far, `hiralysis` mainly provides the core analysis engine.

A more general problem of using graph grammars became visible at several points in this work. They are not very suited to model control in a programming language sense. There are extensions of graph grammars towards that direction, but they go beyond the pure formalism and may render themselves hard to verify.

Despite the disadvantages, `hiralysis` comes with a major strength: partner constraints to restrict the applicability of rules. In experiments, partner constraints came to help, whenever the state space exploded. Partner constraints, as a special instance of negative application conditions, proved to be absolutely essential for specification purposes. Without the ability to say, when a rule does not match, none of the examples could have been implemented. As with the disadvantages, most of them are inherent using any kind of graph grammar.

Usefulness of the Results Recall that Section 3.1 described in detail the approach to verifying car platooning in the PATH project. It was clarified that even the specification of the protocols was inadequate. This claim is strengthened by the experimental evaluation. None of the flaws discovered in even the simplest implementations is in scope of the COSPAN based, static verification methods used in the PATH project. Rather these mistakes were due to the inherent, dynamic behavior of the car platooning case study. Despite the complexity of the case study, many interesting properties could be verified or refuted by the results of the `hiralysis` implementation. In simple cases, even completeness (word decidability) results could be established.

Unfortunately, the sufficient conditions stated in the completeness theorems prove to be rather strong and only applicable in simple cases. There are two possible ways to overcome this weakness: try to relax the sufficient criteria and/or use partial completeness in a property driven fashion. Both options were already elaborated on in Section 4.4.

A general problem of using graphs in specification and verification of systems is the accessibility and readability of large graphs by humans. When a certain size is exceeded, no valuable information can be gathered from such a graph by just looking at it. There need to be ways to automatically access and query these graphs. The use of $\mathcal{GL}$ for property specification is a first step towards this goal, because such a specification may be automatically checked

against the resulting abstract graph grammar. This model-checking process is not yet integrated into the `hiralysis` tool, but offers great potential value of future extensions.

This paragraph concludes by identifying a surprising but very useful application domain of the `hiralysis` tool: protocol design. Given verbal descriptions of communication protocols, one may start with implementing an abstracted formal description as a partner graph grammar and run the `hiralysis` tool on it at this early design stage. The experience gained from the platoon case study showed that many subtle errors may be discovered already at this early stage.

Later on, one should gradually refine the protocol implementation. At all stages, a quick partner graph grammar implementation is possible. It can then be easily checked by using `hiralysis`.

This process was exemplified in the platoon case study by starting with understanding the idealized scenario, before this scenario was refined to contain explicit message queues or the possibility of communication failures.

Other Case Studies The remaining case studies presented in Chapter 3 were either not implemented in terms of partner graph grammars or are beyond the possible applications of the `hiralysis` tool. They mostly identify future work.

4.6 Extensions

This section introduces two extensions to the standard partner abstraction based abstract interpretation, which will be referred to as standard analysis from now on. The key idea of the first extension is to *count abstract clusters*. So far, only the number of nodes summarized to a summary node was tracked up to some k. The number of abstract clusters, however, was completely forgotten. This works fine for partner graph grammars that have the cluster multiplicity property. In general, the loss of information may be severe. The new abstraction will be called *counting abstraction*, the induced analysis will be called *counting analysis*.

The second extension – besides counting abstraction – is called *general cluster* analysis. The idea is to parameterize the analysis in terms of the definition of a cluster. So far, cluster was a synonym for a connected component. That is, a cluster of a graph was a connected component of the graph. Technically, a cluster used to be an equivalence class of a graph *wrt.* the "connected" equivalence. One may get beyond this notion by allowing clusters to

be equivalence classes *wrt.* other equivalences than just connectedness. On the abstraction level, abstract graphs will not be sets or multisets of abstract clusters, but graphs that have abstract clusters as subgraphs.

4.6.1 Counting Clusters

After a motivating example, some technical details of the counting abstraction are explained, starting from the actual abstraction mapping, proceeding with abstract matches, and concluding with abstract updates.

The theory is not developed from scratch as in Section 4.1. In particular, no formal theorems are stated or proven. However, the story-telling of this section is closely along the lines of Section 4.1. The presented material should be profound enough to define the technical basis and make clear how the counting analysis is set up.

Motivation The most obvious partner graph grammar, where counting analysis enhances precision is the following. Take the partner graph grammar modeling an idealized merge from Table 3.1 without the [CREATE] and [DESTROY] rules. Complement the rules with an initial graph consisting of a single free agent. The concrete graph semantics of this partner graph grammar consists of this single free agent only, because no rule matches in the concrete. The standard analysis, however, will assume an arbitrary amount of free agents and start exploring the search space, until it reports the famous twelve abstract clusters of Figure 4.7 as result. This could be prevented, if it was known, that there was only one free agent around. The counting analysis addresses this issue.

Abstraction While an abstract graph in the standard abstraction was a set of abstract clusters, the counting abstraction uses *multisets* of abstract clusters as abstract graph domain. Therefore, a counting analysis is parameterized by two integer bounds k and c, where k is used for counting nodes and c is used for counting clusters. The domain of abstract graphs in the counting abstraction, given a set $\mathcal{N}$ of node and a set $\mathcal{E}$ of edge labels, is thus

$$\wp_m(\mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k) \times \mathbb{N}_c)$$

with a typical element being $\hat{G} = \{\!\{(\hat{C}_1, n_1), \ldots, (\hat{C}_t, n_t)\}\!\}$. Note that the multiset notion together with explicit counters attached to an abstract cluster provide two notions of counting, whereas one notion seems sufficient. However, the given domain simplifies reasoning when it comes to cluster

materializations. This corresponds tightly to node multisets together with multiplicities being part of canonical names.

The abstraction mapping $\alpha_{c,k}$ is defined in terms of a *partner* and a *collapse* mapping just like in the standard case. A bit of care must be taken in order to get the two-layered counting right. Here are the definitions:

$$\begin{aligned} \mathit{partner}_{c,k}(C) &= (\mathit{partner}_k(C), 1) && (4.20)\\ \mathit{collapse}_{c,k}(C,t) &= (\mathit{collapse}_k(C), t_c) && (4.21)\\ \mathit{Collapse}_{c,k}(M) &= \bigoplus^{c} \{\!\!\{ \mathit{collapse}_{c,k}(C) \mid C \in_m M \}\!\!\} && (4.22)\\ \alpha_{c,k}(G) &= \mathit{Collapse}_{c,k} \{\!\!\{ \mathit{partner}_{c,k}(C) \mid C \in \mathit{cc}(G) \}\!\!\} && (4.23) \end{aligned}$$

The computation of partner information for nodes is exactly like in the standard case. This can be seen in (4.20), where the c parameter – the bound for cluster counting – is effectively ignored. This definition works for clusters C only and faithfully records 1 being the number of instances of the argument clusters C.

The $\mathit{collapse}_{c,k}$ mapping applied to a cluster C, whose nodes are already canonical names, ignores the c, too, except for taking the c-bound version of the number t of instances of C. This mapping summarizes nodes with equal partner information.

The $\mathit{Collapse}_{c,k}$ mapping takes care of both summarizing and counting partner equivalent nodes and isomorphic abstract clusters. It gets a multiset of pairs of abstract clusters and cluster multiplicities as argument and adds up equal abstract clusters. The bound c is used for the purpose of cluster counting, where the operator in (4.22) is defined by:

$$\bigoplus^{c}(M) = \{\!\!\{ (\hat{C}, t) \mid t \neq 0;\ \exists q \in \mathbb{N}_c.(\hat{C}, q) \in_m;\ t = \oplus^c \{\!\!\{ q, (\hat{C}, q) \in_m M \}\!\!\} \}\!\!\}$$

Equation (4.23) completes the definition of the abstraction mapping. First, canonical names are computed for each node in each connected component of a graph, then these nodes are summarized and counted up to bound k. Finally, equal abstract clusters are summarized and counted up to bound c.

Abstract Updates Like the definition of counting abstraction was split into the computation of the canonical names and the summarization of nodes and clusters – just like in the standard case – the definition of materialization becomes straightforward. It is almost literally the same as in Definition 4.1.6. The second requirement in this definition forced a materialization to contain at most one ∞-summary node of a given canonical name. The same requirement needs to be added on cluster level. Otherwise, the definitions stay the

same. This is another advantage of the declarative nature materializations. In order to define materializations constructively for the counting analysis, even more care would have to be taken than in the standard case.

Having defined materializations, the next step on the way to an abstract update is abstract partner constraint satisfaction. For the concrete case, it was defined in Definition 4.1.7. Again, this definition applies almost literally to the counting abstraction as well, because it is defined in terms of connected components. Counting abstraction on this level is just the standard abstraction. The only syntactical change involves replacing $\hat{C}$ with $(\hat{C}, 1)$ to cater for the cluster counting.

Abstract matching by materialization is equivalently characterized by non-injective matches in the abstract graph with additional multiplicity constraints. This is formalized in Lemma 4.1.4 for the standard analysis. This lemma almost holds for counting abstraction as well. Instead of taking the disjoint graph union $\dot{\cup}\hat{G}$ of the abstract graph to be matched, one needs to unfold the counting abstracted graph $\hat{G}$ according to the given cluster multiplicities in order to obtain the equivalence result. Formally, $\dot{\cup}\hat{G}$ is replaced by $\dot{\cup}\{\mathit{unfold}(\hat{C}, t) \mid (\hat{C}, t) \in_m \hat{G}\}$, where

$$\mathit{unfold}(\hat{C}, t) = \underbrace{\hat{C}\dot{\cup} \ldots \dot{\cup}\hat{C}}_{n \text{ times}}$$

where $n = t$ if $t \neq \infty$ and $n = c + 1$ otherwise for bound c.

The definition of an abstract update in the counting abstraction reads as follows. Abstract graph $\hat{G}$ evolves into $\hat{H}$ by application of rule r, if and only if there exists a materialization $\hat{M}$. The latter is partitioned into clusters of multiplicity 1 and into summary clusters: $\hat{M}_1 = \{\!\{(\hat{C}, 1) \mid (\hat{C}, 1) \in_m \hat{M}\}\!\}$ and $\hat{M}_{>1} = \hat{M} \setminus_m \hat{M}_1$. Moreover, there exists the concrete direct derivation $\dot{\cup}\hat{M}_1 \rightsquigarrow_r \hat{M}'$. The result of the application $\hat{H}$ is the defined to be:

$$\hat{H} = \mathit{Collapse}_{c,k}(\{\!\{\mathit{partner}_{c,k}(C) \mid C \in \mathit{cc}(\hat{M}')\}\!\} \dot{\cup}\hat{M}_{>1})$$

The update part is joined with the non-affected part of the materialization and abstracted again.

Defining the abstract graph semantics in the case of counting abstraction completes the definition of the counting analysis. Therefore, Definition 4.1.10 is adapted in the following way. The maximum union operator $\dot{\cup}_c$ replaces the $\bigcup$ in the second case of the definition. It is defined to be:

$$\hat{M}_1 \dot{\cup}_c \hat{M}_2 = \{\!\{(\hat{C}, t) \mid t = \max\{q \mid (\hat{C}, q) \in_m M_1 \vee (\hat{C}, q) \in_m M_2\}\}\!\}$$

This operator summarizes all abstract clusters of two multisets and assigns the maximal (*wrt.* bound c) multiplicity to each summarized cluster. All

remaining union operators in Definition 4.1.10 are replaced with the operator in (4.23).

Properties of Counting Abstraction The special maximum operator $\dot{\cup}_c$ takes care – to a certain amount – of clusters that disappear due to a rule application. This is yet another precision gain as compared to standard analysis. Also, the maximum operator $\dot{\cup}_c$ preserves monotonicity in the definition of the abstract graph semantics, which is the key to the termination result, which certainly holds for the counting analysis, too. However, the partial order necessary to prove termination is not just subset inclusion, but the combination of subset inclusion on clusters and smaller than or equal on the cluster multiplicities.

Soundness can be proven as easily as for the standard analysis. Finally, the following observation claims the coherence of the counting analysis and the standard analysis. If a partner graph grammar has the cluster multiplicity property, the abstract graph semantics as computed by the counting analysis coincides with the abstract graph semantics computed by the standard analysis.

4.6.2 Generalized Clusters

In the previous subsection, it was demonstrated, how the standard analysis can gain precision by counting clusters on top of counting partner equivalent nodes. In contrast, this subsection reveals a parameterization that not necessarily increases precision. Rather, it allows to tailor the standard analysis to applications, where connected components are not as crucial as they are in the setting of dynamic communication systems. The notion of a cluster will be parameterized in terms of a defining equivalence relation. In the standard case, two nodes are *cluster equivalent*, if and only if they are connected. Arbitrary equivalence relations are allowed to replace connectedness in the *generalized cluster* analysis. This subsection is more sketchy than the previous one and merely describes an idea that is currently being explored.

Motivation Recall the routing case study of Section 3.2. This case study does not lend itself to the standard analysis, because typically, one is interested in one, large connected component only. One may even be interested in proving, that all nodes are connected at all times. Although the standard analysis may be applied to that, the power of the two-layered abstraction is completely lost. Moreover, an ad hoc network does not evolve in a very

structured or controlled manner. Rather changes are arbitrary, before the network re-structures itself.

However, the Safari routing protocol does impose structure on a network. It is partitioned into cells, the set of cells is partitioned into super-cells, and so on. Each of the cells has distinguished drums announcing themselves to everyone in reach. In this case, it might be useful, if the cells were considered to be clusters instead of the connected components. As cells are arbitrarily nested, this raises the need for deeper hierarchies than available in the standard analysis. In the following, only the first issue is addressed: arbitrary cluster definitions.

Abstraction While the abstract graph domain consisted of sets and multisets of abstract clusters, respectively, in the cases of the standard and the counting abstraction, abstract graphs now will be graphs that have abstract clusters as subgraphs in the generalized cluster analysis. For simplicity assume unlabeled edges for this extension. In the remainder, $\sim_G$ denotes an equivalence relation on the nodes of a graph G. The reader may think of connectedness or "being in the same cell" as examples of this equivalence relation.

An abstract graph in the generalized cluster analysis is an element of

$$\wp(\mathcal{G}_{\text{can}}(\mathcal{N}, k)) \times (\wp(\mathcal{G}_{\text{can}}(\mathcal{N}, k)) \times \mathit{canonicalNames}(\mathcal{N}, k))^2$$

Therefore, an abstract graph is in fact a graph, whose nodes are canonical names. There are two kinds of edges.

- *intra-cluster* edges between canonical names in the same cluster and
- *inter-cluster* edges between canonical names belonging to different abstract clusters.

That said, the abstract graph domain of the standard abstraction becomes a special case of this more general domain, where the set of inter-cluster edges is empty.

The abstraction of a graph G is defined to be $\alpha_k^{\sim}(G) = (\alpha_k^V(G), \alpha_k^E(G))$, where

$$\alpha_k^V(G) = \{\mathit{collapse}_k \circ \mathit{partner}_k(C) \mid V_C \in G/\sim_G, C = G\mid_{V_C}\}$$

Let $\xi^{\sim}$ be the morphism induced by α_k^V. Then $((\hat{C}, \hat{u}), (\hat{C}', \hat{u}')) \in \alpha_k^E$, if and only if there exist $u, u' \in V_G$, such that $u \not\sim_G u'$, $(u, u') \in E_G$, $\xi^{\sim}(u) = \hat{u}$, and $\xi^{\sim}(u') = \hat{u}'$.

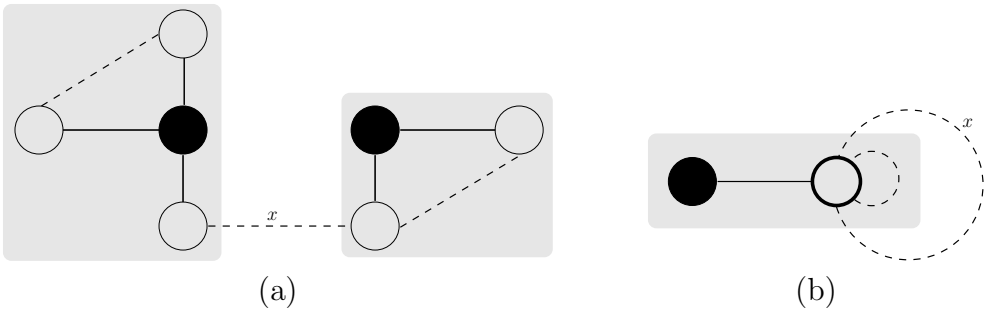


Fig. 4.10: Generalized cluster abstraction at work. Part (a) shows a sample graph G having two sorts of edge labels β_1 (solid) and β_2 (dashed). Node labels are represented by the filling style of nodes (black, white). Two nodes are cluster equivalent, written $u \sim v$, if and only if they are in the same connected connected of the graph $(V_G, E_G^{\beta_1}, \ell_G)$. Part (b) shows the graph $\alpha_1^{\sim}$. Clusters are shaded. Intra-cluster edges are within shaded areas; inter-cluster edges between shaded areas.

In order to define the notion of materialization, the concept of *flattening* is introduced. Flattening is applied to abstract graphs. It basically breaks up the difference between inter- and intra-cluster edges, combining them into an ordinary graph. Node materialization remains the same again, whereas cluster materialization becomes more involved. As no real working examples of the generalized cluster analysis exist so far, the major technical burden of introducing these more complicated notions is avoided and delegated to future research. Figure 4.10 shows a generalized cluster abstraction at work on a simple example. Note that there are two parallel edges, one of them is an intra- and the other an inter-cluster edge. This is due to the $u \not\sim_G u'$ clause in the definition of inter-cluster edges.

Discussion The power of generalized cluster abstraction remains unexplored until now. Clearly, it provides a strong, parameterized abstraction, which may be applicable to all the case studies that do not suite the standard analysis. As indicated in the beginning of this section, cluster equivalence may be instantiated to characterize cells in the Safari routing environment. Apart from that, routing protocols in ad-hoc networks offer the ideal application domain for extending the presented analyses to hierarchies of abstraction deeper than two.

Dynamically allocated heap data structures are another interesting domain for generalized cluster analysis. A good choice for cluster equivalence may be "belonging to the same data structure", *e.g.* the same, unshared list.

In this application domain, it may be a step towards a hierarchical shape analysis. All the applications of the technique surveyed in [SRW02] are in some sense flat. Even if there is a hierarchical structure conceptually – *e.g.*, in [YRSW03] – it is flattened to be fed into the TVLA [LAS00] engine. If one reformulated the generalized cluster analysis in terms of logic and predicate update formula, *c.f.* Chapter 5 – a way of bringing more structure to shape analysis may be near. This said, one needs to stress, that this is still speculative and requires further research.

Before this section concludes, it must be said, that there is no conceptual obstacle, that disallows the combination of counting and generalized cluster analysis, even though this will come at the price of an increased complexity.

A way to decrease complexity at the cost of efficiency not only applies to the generalized cluster analysis, but also to the standard case, because it is independent of the actual definition of a cluster. In all variants isomorphic clusters were summarized on the second layer of abstraction. One may also think of having weaker or incomparable ways of summarizing clusters, whose nodes are canonical names. Two examples are given below, more may make sense.

- Summarize clusters C and D, if their nodes have equal sets of labels, *i.e.*, if $\ell_C(V_C) = \ell_D(V_D)$. This is weaker and easier to check than isomorphism, but yields coarser abstractions.

- Summarize clusters C and D, if they have the same number of nodes, *i.e.*, if $|V_C| = |V_D|$.

Chapter 5

Related Work

While many related works have been mentioned throughout this thesis, the most relevant ones are summarized and looked at in more detail in this section. They include general graph grammars and their verification, the relation to the three-valued logic based shape analysis technique [SRW02], and other approaches to the analysis of dynamic communication systems. Section 2.3 provided a detailed overview of the relation between partner graph grammars and other notions of graph grammars. It constitutes a related work section of its own and is thus not repeated here. Nor is the underlying theory of abstract interpretation part of this section.

Graph Grammar Verification Verification techniques such as abstract interpretation or model-checking become increasingly important in the graph grammar community.

[Hec98] presents an approach to the formal verification of graph grammars based on model-checking. However, no abstraction is used there, making it inappropriate for the verification of unbounded systems.

Another orthogonal approach [BCK04] proved very successful, but is not based on abstract interpretation. Rather it is based on the unfolding semantics of the given graph grammar [BCM99] and approximates its behavior by means of Petri nets. Recently, this approach was equipped with counter-example guided abstraction refinement [KK06]. It is strong enough to show properties of programs manipulating red-black trees [BCE+05].

The only abstract interpretation based approach other than the one presented here was developed independently in [RD06]. The underlying abstraction [Ren04a] relies on counting incoming and outgoing labeled edges, *i.e.*, it is rather edge-centric compared to the node-centric approach used in this thesis. The idea of materialization-update-abstraction occurs in [RD06], too. The major drawbacks of [RD06] seem to be the requirement of deterministic

graphs, the lack of any form of negative application conditions, and, to some extent, missing tool support. These issues are considered in more detail in the following.

- Graphs are deterministic, if no node has two outgoing β-edges to two distinct nodes. This property is surprisingly close to the notion of unique partners that is essential for proving the matching theorem. In the work presented here, however, this determinism is not needed to make the analysis work at all.

- The lack of any form of negative application conditions may be the most severe drawback of the approach proposed in [RD06]. The importance of such conditions was stressed at many places throughout this thesis. In the partner abstraction based approach, such conditions come in the form of partner constraints.

- Although it is still at the prototype phase, the `hiralysis` tool provides specially tailored support for partner abstraction based analysis. It exploits all the properties of the analysis making it efficient. On the other hand, `hiralysis` lacks a GUI and syntactic conveniences. In contrast, the technique of [RD06] was implemented as an add-on to a general purpose tool, potentially leading to less efficiency.

The partner abstraction based approach of this work provides clear advantages over [RD06]. It is not restricted to deterministic graphs, has some form of negative application conditions, and provides a two-layered abstraction. The latter is more precise, when it comes to systems of many connected components as typical of dynamic communication systems.

Three-Valued Logic Based Shape Analysis Originally, partner abstraction was inspired by the work on three-valued logic based shape analysis [SRW02]. In that approach, concrete graphs are coded as logical structures. Abstraction, called *canonical abstraction*, works by summarizing objects indistinguishable under a set of abstraction predicates. Abstract states are three-valued logical structures, where the third logical value denotes uncertainty. The concrete semantics is described in terms of *predicate transformers* that are also used for updates on abstract states. This paragraph clarifies two issues.

- How can partner graph grammars be encoded into logical structures and predicate transformers?

- Given the first issue, how do partner abstraction and canonical abstraction relate?

There is an obvious correspondence between graphs of $\mathcal{G}(\mathcal{N}, \mathcal{E})$ and logical structures over the set $\mathcal{N}$ of unary and the set $\mathcal{E}$ of binary predicates. The application of a transformation rule or of a predicate transformer corresponds to direct derivations between graphs. As a first step to examine the relation between (the application of) graph transformation rules and predicate transformers, *matching* is encoded into logical terms. The work reported in [Ren04b] handles this issue in quite some detail. To be more precise, the referenced work investigates how the applicability of transformation rules, in particular, in the presence of negative application conditions, can be expressed in terms of first-order logical formulas. It is shown that general negative application conditions as described in the previous subsection correspond to the $\exists\neg\exists$ fragment of first-order logic. As a consequence, partner constraints being a restricted version of negative application conditions can be encoded in that fragment, too. It must be stated, that general first-order logic, in particular if augmented with transitive closure operators as in [SRW02], is more expressive, when it comes to specifying, where an update is to happen. Also, the constructive definition of direct derivations in partner graph grammars can be easily coded in logic. However, the experience gained using the TVLA tool [LAS00] shows that it can be extremely difficult and tedious to write down update formulas, whereas the use of partner graph grammars seems more elegant and less technical.

What about the relation between canonical abstraction and partner abstraction? Canonical abstraction handles edges differently from partner abstraction. Edges in abstract graphs in this thesis correspond to 1/2-edges in canonical abstraction. Such edges denote that there *may* be such an edge in the concretization. Additionally, canonical abstraction offers 1- or *must*-edges that are not considered here. Put differently, canonical abstraction supports both $\forall\forall$ and $\exists\exists$ abstraction, whereas partner abstraction only supports the latter. (Even though it may be extended to have both.) Summary nodes in [SRW02] differ from summary nodes here, where the precision k comes as a parameter to partner abstraction. Canonical abstraction supports a subtle variation of the case $k = 1$.

The real strength of canonical abstraction comes through parameterization. The set of predicates used to encode the concrete semantics of a system (core predicates) can be augmented by arbitrarily defined instrumentation predicates to keep track of more information. This makes that technique very powerful and general. The more predicates used, the higher the worst-case complexity and the higher the specification burden for the designer of

an analysis. Each predicate has to be defined very carefully and, potentially, in a tedious manner.

Seemingly more expressive than partner abstraction, indeed, canonical abstraction is able to encode the former. Due to the power of the two layers of partner abstraction, this comes at a price of increased complexity, *i.e.*, of the need for more instrumentation predicates. For the encoding, specify two formulas: ψ_c and ψ_t, The first formula defines partner equivalence of nodes, which is easy enough in first-order logic. The second formula describes that two nodes are in the same connected component. As [SRW02] comes with transitive closure, this poses no problem, either. However, nodes v_1 and v_2 cannot simply be summarized, if both $\psi_t(v_1, v_2)$ and $\psi_c(v_1, v_2)$ hold in a concrete structure, because this would not result in a bounded abstraction. There can be arbitrarily many connected components. Rather v_1 and v_2 are summarized, if $\psi_c(v_1, v_2)$ holds *and* if the structures induced by $V_i = \{v \mid \psi_t(v, v_i)\}$ are isomorphic after quotient building under the equivalence relation specified by ψ_c. This is not straightforward to formalize using canonical abstraction.

The only possible way of encoding seems to use an exponential number of additional predicates, where there is a predicate for each possible abstract cluster (exponentially many) being true for all nodes in such a cluster. There must be update formulas for all of these many additional predicates, which may make the analysis very hard. It should be noted that this reasoning does not constitute a formal proof, that partner abstraction cannot be encoded efficiently in canonical abstraction.

Despite all arguments in favor of canonical abstraction, there are clear advantages of the proposed technique. It comes as a stand-alone technique that does not impose the instrumentation burden on the user. It is especially tuned to work for the application domain of dynamic communication systems, whereas TVLA is a general purpose tool. Being able to encode partner graph grammars and partner abstraction comes at a high price both on the specification and on the analysis side. Eventually, most advantages of three-valued logic based shape analysis boil down to more generality.

Other Analyses of Communication Systems The verification methods proposed in the PATH project defining the original platoon protocols were inadequate to handle the dynamics of dynamic communication systems. Only static instances were considered. Other works, where formal verification is applied to dynamic communication systems, typically use very different specification or verification techniques and are hard to compare with on a formal level. Therefore, only two more works are mentioned for

further reading. In [BDNN98], a control-flow analysis for the π-calculus was proposed. However, no explicit communication topologies are prevalent in this setting. UML implementations are checked against LSC specifications in [DW02]. Even though UML-style object creation and destruction has the dynamics of dynamic communication systems, and even though message queues can be explicitly encoded in UML, this setting reminds only vaguely of the setting in this thesis.

Chapter 6

Conclusion

This chapter concludes the thesis. It starts with a detailed summary of the major achievements of this work. After that, some areas of future research are identified and discussed.

6.1 Contribution

This thesis addresses the analysis of communication topologies as they typically occur in dynamic communication systems. *Partner graph grammars* are developed and identified to be an adequate formalism for specifying such systems. They are an expressive variant of the single-pushout approach of algebraic graph transformations. Partner graph grammars are characterized by injective matches and *partner constraints*, a restricted form of negative application conditions. Partner constraints prove essential for modeling systems. None of the case studies could be formulated without this feature. A constructive, rather than a declarative, algebraic definition of rule applications is given for partner graph grammars.

A significant case study that is prototypical of dynamic communication systems is presented in form of platoon maneuver protocols. Originally developed in the California PATH project, it is shown that protocol verification within PATH suffers from major drawbacks and is, as a whole, inadequate to deal with the complexity of the application. Neither the dynamic creation of objects nor the dynamically changing communication topology is considered in PATH. Also an unrealistic (for this scenario) shared memory model is chosen as means of communication, there.

In contrast, in the work presented here, a realistic way of communication via message queues is considered and the possibility of faulty channels is taken into account. Section 3.1 demonstrates that *all of these features* can

be conveniently expressed using partner graph grammars. In the evaluation to be discussed below, significant properties of the platoon case study are proven. Additionally, a number of further sophisticated case studies are presented: heap-manipulating programs, process calculi, and ad hoc network routing protocols. It is discussed how they could be implemented in partner graph grammars. The discussion of these further case studies leads to a set of features that may be added to partner graph grammars, such as probabilism or attributed graphs, and opened up a promising line of future research.

While Chapter 2 and Chapter 3 prove the expressiveness, usability, and adequateness of partner graph grammars in terms of *specifying* dynamic communication systems, Chapter 4 presents the key results of the analysis of partner graph grammars. First, an abstract interpretation of partner graph grammars based on *partner equivalence* is defined. Two nodes u_1 and u_2 in a graph are considered partner equivalent, if they have the same label *and* if the set of labels of u_1's adjacent nodes equals the set of labels of u_2's adjacent nodes (respecting the directions and labels of edges). In the point of view of dynamic communication systems, this means, that two objects are considered equal, if they are in the same state *and* if the set of states of their communication partners are equal. It is conjectured that this is precisely the information that determines the successor state of an object within a communication topology making partner equivalence the *ideal abstraction* for the analysis of such systems. Information may be lost, if the application of a transformation rule depends on the number of adjacent nodes with a given label, because this is only captured by partner equivalence up to some constant k. Also, if more information than just pure communication connectivity is encoded in graphs – as in the example of queue-based communication – partner equivalence may lose its precision while still yielding reasonably precise overall results.

Partner equivalence based abstraction is only the first layer of the proposed *two-layered* partner abstraction. Again, the second layer is driven by the communication systems domain. Two subsets of a communication topology that are not connected, *i.e.*, that cannot influence each other by communication, are independently abstracted using partner equivalence. If the results are isomorphic, the two abstracted subgraphs will not be distinguished any more in the final partner abstraction. This two-layered approach is novel.

Partner equivalence is the basic notion in the definition of an abstract graph. Direct derivation steps defined to work on non-abstract graphs are lifted to work directly on the abstract level. A fixpoint computation finally yields a conservative over-approximation of the graph semantics of the con-

crete partner graph grammar. Remember that the graph semantics denotes the set of all graphs derivable from an initial graph using the rules of a partner graph grammar. The fixpoint computation is proven to terminate and the soundness result is established formally.

A crucial notion for defining direct derivation steps on abstract graphs is the notion of an abstract match. An abstract match is just a non-injective match morphism, whereas concrete match morphisms must be injective. A strong theoretical result clearly identifies a class of abstract graphs, such that a rule matches the abstract graphs *if and only if* it matches all the concretizations of the abstract graph. This result is a first step towards a completeness results, because it states, that, for a certain class, the abstraction preserves precisely the applicability of rules.

Moreover, three notions of completeness are defined and statically checkable criteria are shown to be sufficient criteria for completeness. Partner graph grammars that are shown to be complete exhibit strong *property preservation* features. For a given class of graphs, partner abstraction preserves *exactly* the validity of first-order formulas without equality. The equality restriction is due to the fact that information about the number of nodes cannot be tracked precisely, if the abstraction is to be of bounded size.

Partner abstraction based abstract interpretation has been implemented in the `hiralysis` tool. It is experimentally evaluated on a significant set of examples taken from the platoon case study. All advanced features of the case study are analyzed quickly and interesting properties are shown. The advanced features include asynchronous, message queue-based communication and faulty channels. Based on experiences gained with `hiralysis`, the tool has been found to be extremely suitable not only for protocol verification, but in particular for *protocol design*.

The only verification problem is met in the presence of message queues. Clearly, more work needs to be invested there, in order to separate queue verification from pure topology verification. See below for some ideas. If one was to summarize the highlights of this thesis as briefly as possible, the following list would be a good candidate.

- Partner graph grammars are introduced as an adequate, powerful, yet analyzable specification formalism not only for dynamic communication systems. They gain their particular power from partner constraints, a special instance of negative application conditions.

- Partner abstraction seems like the ideal abstraction method for dynamic communication systems. It is also applicable in other scenarios.

- Partner abstraction based abstract interpretation has been implemented in the `hiralysis` tool. The tool has been experimentally evaluated on a significant set of examples taken from the platoon case study.
- The sophisticated platoon case study is successfully implemented in three realistic variants using partner graph grammars. Many interesting properties can be automatically verified using the `hiralysis` tool. For the idealized implementation, even completeness results can be proven. The original approaches to platoon verification are shown to be inadequate both for specification and verification.
- Some technical results beneficial on their own:
  - A class of graphs, for which abstract matches are *equivalent* to concrete matches.
  - Termination and soundness of the abstract interpretation of partner graph grammars.
  - Three notions of completeness for partner graph grammars; sufficient, statically checkable criteria guaranteeing completeness.
  - A class of graphs, for which partner abstraction preserves *exactly* the validity of first-order logical formulas (without equality).
  - Two possible extensions of partner abstraction that allow for counting of clusters and cluster definitions other than connected components.

6.2 Outlook

The problem of computing a conservative approximation of all possible communication topologies of a dynamic communication system has been solved – at least for pure communication topologies as in the idealized platoon case study. Further developments of the proposed technique should be application-driven, *i.e.*, they should contribute to making the technique applicable to richer application domains. Candidates were identified in Section 3.2. They raised the need for extensions like the following:

- Add stochastic features to partner graph grammars to model faulty channels more realistically than by non-determinism.
- Add more hierarchy and generalized clusters to partner abstraction in order to cope with hierarchies as they occur in ad hoc network routing.

- Define partner graph grammars and their abstract interpretation for *attributed graphs*. Queues, that proved to be an obstacle, could then be delegated to the attributes rather than being encoded in the graph. Also, sophisticated routing information in the Safari routing environment may be encoded using attributes.

Apart from these rather mid- to long-term goals, the immediate steps involve adding more convenience to the `hiralysis` tool, such as incorporating label constraints and augmented partner constraints in the grammar specifications or automatically checking for the occurrence of a given subgraph. Also, formalizing the extensions suggested in Section 4.6 is a challenge to be tackled soon.

The existence of the similar abstract interpretation based approach of [RD06] indicates that there may be a unified framework of abstract interpretation of graph grammars. This framework could be instantiated with partner abstraction, the abstraction proposed in [Ren04a], or others. If the layered abstraction shows to be a useful concept also for deeper hierarchies, the depth of such a hierarchy could serve as another parameter instantiating the framework. This parameterized framework may be much better comparable to the parameterized shape analysis framework proposed in [SRW02].

Appendix A

Proofs

Lemma 4.1.1 Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a non-canonical graph and let $k \geq 1$ be a natural number. There exists a surjective graph morphism ξ from G to the disjoint graph union of $\alpha_k(G)$ defined to be $\xi(u) := (\hat{C}, (\nu, P, n))$, where

1. C is the connected component of G containing u and $\alpha_k(C) = \{\hat{C}\}$
2. $partner_{C,k}(u) = (\nu, P, 1)$
3. $n = (|\{v \in V_C \mid u \bowtie_c v\}|)_k$

Proof

The proof is constructive. Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a non-canonical graph and let $\hat{G} = \dot{\cup}\alpha_k(G)$ for any $k \geq 1$. The morphism $\xi : V_G \to V_{\hat{G}}$ is defined as follows.

$$\xi(u) := (\hat{C}, (\nu, P, n))$$

where

1. C is the connected component of G containing u and $\{\hat{C}\} = \alpha_k(C)$
2. $partner_{C,k}(u) = (\nu, P, 1)$
3. $n = |\{v \in V_C \mid u \bowtie_c v\}|_k$

It is trivial to conclude the surjectivity of ξ.

Two requirements remain to be shown proving that ξ is indeed a morphism (*c.f.* Definition 2.1.4). First, it must be node label preserving, *i.e.* $\ell_G(u) = \ell_{\hat{G}}(\xi(u))$ for all $u \in V_G$. Let $u \in V_G$ be arbitrary and let $\ell_G(u) = \nu$. Then $partner_{G,k}(u) = (\nu, \_, \_)$ due to the definition of *partner*. Hence $\xi(u) = (\_, (\nu, \_, \_))$ due to the definition of ξ. Definition 4.1.4 concludes the

argument, since it defines the node labeling function of the resulting graph to be $\lambda(\nu, \_, \_).\nu$.

It remains to be shown that $\{(\xi(u), \xi(v)) \mid (u, v) \in E_G^\beta\} \subseteq E_{\hat{G}}^\beta$ for all $\beta \in \mathcal{E}$. Let $(u, v) \in E_G^\beta$ be arbitrary.
This implies that $(\mathit{partner}_{G,k}(u), \mathit{partner}_{G,k}(v)) \in E^\beta_{\mathit{partner}_{k(C)}}$ due to Definition 4.1.3, where C is the connected component of G containing both u and v. From the definition of node and edge sets in Definition 4.1.4, it becomes immediate that indeed $(\xi(u), \xi(v)) \in E_{\hat{G}}^\beta$.

Lemma 4.1.2 Let $k \geq 1$ be a natural number and let $G, H \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be connected, non-canonical graphs. Then, $\alpha_k(G) = \alpha_k(H)$, if and only if

- $G/\bowtie \cong H/\bowtie$ due to isomorphism ψ and
- $(|\, u \,|)_k = |\, \psi(u) \,|_k$ for all $u \in V_{G/\bowtie}$,

then $\alpha_k(G) = \alpha_k(H)$.

Proof

As a first step in the proof of Lemma 4.1.2, another lemma is proven: isomorphism preserves partners. It needs to be shown, that for isomorphic graphs A and B with isomorphism φ holds, that, for all $u \in V_A$ and any $k \geq 1$: $\mathit{partner}_{A,k}(u) = \mathit{partner}_{B,k}(\varphi(u))$. Let $\mathit{partner}_{A,k}(u) = (\nu, P, 1)$. This implies that $\ell_A(u) = \nu$ and – because of isomorphism – $\ell_B(\varphi(u)) = \nu$. As *partner* maps to multiplicity 1 for non-canonical graphs, it is known that $\mathit{partner}_B(\varphi(u)) = (\nu, P', 1)$. Because of the morphism property of φ, it holds that $P \subseteq P'$. The inverse follows, because φ is a morphism in the other direction, too.

Let G, H, k be like in the lemma, and let $\hat{G} = \dot{\cup}\alpha_k(G)$, $\hat{H} = \dot{\cup}\alpha_k(H)$, $G' = G/\bowtie$, and $H' = H/\bowtie$. To prove equality, it must be shown that $V_{\hat{G}} = V_{\hat{H}}$, $E^\beta_{\hat{G}} = E^\beta_{\hat{H}}$ for all $\beta \in \mathcal{E}$, and $\ell_{\hat{G}} = \ell_{\hat{H}}$ point-wise.
Show $V_{\hat{G}} \subseteq V_{\hat{H}}$ Let $(\nu, P, n) \in V_{\hat{G}}$, where the name of the abstract cluster is dropped for this node in an abstract graph, because there is only one abstract cluster due to the required connectedness of G. As G and H are non-canonical, the application of *partner* in the definition of α_k always results in multiplicity 1. The $\oplus^k$ sum in the definition of $V_{\hat{G}}$ thus requires q nodes $u_1, \ldots, u_q \in V_G$ such that $(q)_k = n$ and $\mathit{partner}_{G,k}(u_i) = (\nu, P, 1)$ for all $1 \leq i \leq q$.

This implies that there are q partner equivalent nodes in G all of which being mapped to $(\nu, P, 1)$ by $\mathit{partner}_{G,k}$ where the equivalence class is called $x \in V_{G'}$. Because of the conditions of the lemma, there exists an equivalence

class $\psi(x) \in V_{H'}$ with q' elements such that $(q)_k = (q')_k$. Since partners are preserved by isomorphism, for all these q' elements u' holds $partner_{H,k}(u') = (\nu, P, 1)$. This argument together with $(q')_k = (q)_k = n$ and the definition of *collapse* yields $(q, \nu, n) \in V_{\hat{H}}$.

The proof of $V_{\hat{G}} \supseteq V_{\hat{H}}$ follows symmetrically. The proof of equal node labelings in $\hat{G}$ and $\hat{H}$ follows immediately from the node sets equality and the definition of node labels in abstract graphs. The proof of edge equality follows the same line as the proof of node equality.

Lemma 4.1.3 Let $k \geq 1$ and $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be arbitrary. There is a materialization $\hat{H}$ of $\alpha_k(G)$ such that $\dot{\cup}\hat{H} \cong G$ and $\hat{H}$ has no summary nodes.

Proof It is known from the definition of abstraction (Definition 4.1.5), that

$$\alpha_k(G) := \{collapse_k \circ partner_k(C) \mid C \in cc(G)\}$$

According to the definition of materialization (Definition 4.1.6),
$\hat{H} := \{partner_k(C) \mid C \in cc(G)\}$ is a materialization of $\hat{G}$. A close look at the definition of *partner* (Definition 4.1.3) eventually reveals, that $\dot{\cup}\hat{H}$ contains no summary nodes and is clearly isomorphic to G.

Lemma 4.1.4 Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph and $r = (L, h, p, R)$ a transformation rule over the same label sets. Let k be the maximal degree among the partner constraints in the image of p and $\hat{G} = \alpha_k(G)$ the abstraction of G. The following statements are equivalent.

1. Rule r matches $\hat{G}$.
2. There exists a morphism m from L to $\dot{\cup}\hat{G}$, such that $m(u), \dot{\cup}\hat{G} \models p(u)$ for all $u \in dom(p)$, and for each $C \in cc(L)$ and each $u = (\nu, P, n) \in m(V_C)$ holds $\mid m^{-1}(u) \mid_k \sqsubseteq_k \ n$.

Proof

First, (1) $\rightarrow$ (2) is proven. Assume there exists a materialization $\hat{H}$ of $\hat{G}$ such that $L \leq \dot{\cup}G$ by morphism m'. Define $m : V_L \rightarrow V_{\dot{\cup}\hat{G}}$ as $m = \zeta \circ m'$, where ζ is the morphism induced by *Collapse*. Mapping m is a morphism, because it is a concatenation of two morphisms.

Let $u = (C, (\nu, P, n)) \in m(V_C)$ be arbitrary for any $C \in cc(L)$, where the C will be dropped from u for brevity. Then

$$m^{-1}(u) \ = \ \{v \in V_L \mid \zeta \circ m'(u) = u\}$$

$$
\begin{aligned}
&= \{v \in V_L \mid m'(v) = (\nu, P, q_v)\} && \text{(A.1)}\\
&= \{v \in V_L \mid m'(v) = (\nu, P, 1)\} && \text{(A.2)}\\
&\subseteq \{(\nu, P, q) \mid \exists q.(\nu, P, q) \in \hat{H}\} && \text{(A.3)}
\end{aligned}
$$

Step A.1 follows from the definition of ζ induced by *Collapse*, whereas A.2 is an immediate consequence of the definition of an abstract match. Moreover, A.3 holds, because the number of $(\nu, P, 1)$ elements in a connected component is smaller than the number of (ν, P, q) elements in a connected component, if q is at least 1. As the $\oplus^k$ sum of the elements of the set in $A.3$ is just n (following the definition of $collapse_k$, $\mid m^{-1}(u) \mid_k \sqsubseteq_k n$ may be concluded.

In order to prove $m(u), \dot\cup\hat{G} \models p(u)$ for all $u \in dom(p)$, it needs to be shown that abstract partner constraint satisfaction is preserved by ζ, which is immediately obvious from the definitions of *collapse* and abstract partner constraint satisfaction.

To conclude the proof, (2) $\rightarrow$ (1) is shown by constructing a materialization given a morphism $m : V_L \rightarrow V_{\dot\cup\hat{G}}$. The multiset $\hat{H}$ will be defined and shown to be a valid materialization. This multiset is only one possible materialization, more cases may be obtained and are in fact implemented. As they are tedious to write down, only one exemplary materialization is given here.

For each connected component C of L, let $\hat{C} \in \hat{G}$ be the matched abstract cluster, hence let $\hat{C}_1, \ldots, \hat{C}_q$ be the matched abstract clusters for the q connected components of L. The multiset $\hat{H}' = \hat{G}\dot\cup\{\!\{\hat{C}_1, \ldots, \hat{C}_q\}\!\}$ is obviously a materialization of $\hat{G}$. It is called the cluster materialization of $\hat{G}$ as identical copies of each matched cluster are made.

For each connected component C of L, let $\hat{C}$ be the copy of the matched component, let $V_S = \{(\nu, P, n) \in V_{\hat{C}} \mid n \neq 1\} \cap m(V_L)$ be the set of matched summary nodes. For each $(\nu, P, n) = u \in V_S$ let $s(u) = (\mid m^{-1}(u) \mid)_k$, where $s(u) \sqsubseteq_k n$ by assumption of the lemma.

The node materialization of a summary node is defined by the mapping $mater : V_C \times \mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k) \rightarrow \mathcal{G}_{\text{can}}(\mathcal{N}, \mathcal{E}, k)$, where $mater(u, C) = D$ such that $u = (\nu, P, n)$ and

$$
\begin{aligned}
V_D &= V_C \setminus \{u\}\dot\cup\{(\nu, P, 1)\}\dot\cup \left\{ \begin{array}{ll} (\nu, P, \infty) & \text{if } n = \infty \\ (\nu, P, n-1) & \text{if } n \neq \infty \end{array} \right\}\\
E_D &= E_C \,\dot\cup\, \{\!\{(v, w) \mid (u, w) \in E_C\}\!\} \,\dot\cup\, \{\!\{(w, v) \mid (w, u) \in E_C\}\!\}\\
\ell_D &= \lambda(\nu', P', n').\nu'
\end{aligned}
$$

For each $u \in V_S$ there are $s(u)$ node materializations made. Node materialization after cluster materialization clearly defines a materialization $\hat{H}$

of $\hat{G}$ that is matched by L. The match morphism from L in the materialization is defined by mapping nodes to the materializations of their matches in $\hat{G}$. Partner constraint satisfaction is trivially preserved by this sort of materialization.

Lemma 4.1.5 Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph and $\hat{G} = \alpha_k(G)$ its abstraction for some $k \geq 1$. Let ξ be the induced morphism of the abstraction. Finally, let *pc* be a partner constraint. The following statements hold.

1. $\mathit{partner}_{\dot{\cup}\hat{G},k}(\hat{C}, (\nu, P, n)) = (\nu, P, n)$ for all $(\nu, P, n) \in V_{\hat{C}}$ and all $\hat{C} \in \hat{G}$.
2. $(\mathit{partner}_{G,k}(u)) \downarrow 2 = (\mathit{partner}_{\dot{\cup}\hat{G},k}(\xi(u))) \downarrow 2$ for all $u \in V_G$.
3. If *pc* is a simple partner constraint and $G, u \models \mathit{pc}$, then $\dot{\cup}\hat{G}, \xi(u) \models \mathit{pc}$.
4. If k is the degree of *pc* and $\dot{\cup}\hat{G}, \hat{u} \models \mathit{pc}$ for any $\hat{u} \in V_{\hat{G}}$, then $G, u \models p$ for all $u \in \xi^{-1}(\hat{u})$.

Proof Reasoning about partners in a graph certainly amounts to reasoning about partners in one connected component. Therefore, only connected components are considered below. Let C be a connected component of G such that $\alpha_k(C) = \{\hat{C}\}$ and let $(\nu, P, n) \in V_{\hat{C}}$ be arbitrary. The definition of $\mathit{partner}_{\hat{C},k}$ applied to a canonical name immediately yields $\mathit{partner}_{\hat{C},k}(\nu, P, n) = (\_, \_, n)$. Since $(\nu, P, n) \in V_{\hat{C}}$ and ξ is surjective, there exists a $u \in V_C$ such that $\xi(u) = (\nu, P, n)$. According to the definition of ξ this yields $\mathit{partner}_{C,k}(u) = (\nu, P, 1)$, hence $\ell_C(u) = \nu$. Since ξ is a morphism, $\ell_{\hat{C}}(\xi(u)) = \nu$ may be derived.

It has been shown that $\mathit{partner}_{\dot{\cup}\hat{G},k}(\hat{C}, (\nu, P, n)) = (\nu, \_, n)$. The following observation is helpful in the subsequent argument.

$$v \in \xi^{-1}(\nu, P, n) \quad \Leftrightarrow \quad (\mathit{partner}_{C,k}(v)) \downarrow 2 = P \tag{A.4}$$

The $\Rightarrow$ direction of (A.4) is obvious by definition of ξ. The $\Leftarrow$ direction holds, because C is connected. It is easily computed using the definition of *collapse*.

The following computation proves $\ell_{\hat{C}}((\nu, P, n) \triangleright^{\beta}_{\hat{C}}) = \ell_C(u \triangleright^{\beta}_{C})$ for any edge label β. The equation for incoming edges is derived symmetrically. Together with $\xi(u) = (\nu, P, n)$ this implies the claim of part (1). Let $C' = \mathit{partner}_k(C)$.

$$
\begin{aligned}
& \ell_{\hat{C}}((\nu, P, n) \triangleright^{\beta}_{\hat{C}}) \\
= \; & \{\nu' \mid ((\nu, P, n), (\nu', P', n')) \in E^{\beta}_{\hat{C}}\} \\
= \; & \{\nu' \mid \exists q, q', P'.((\nu, P, q), (\nu,', P', q')) \in E^{\beta}_{C'}\} && \text{Definition 4.1.4} \\
= \; & \{\nu' \mid \exists P'.((\nu, P, 1), (\nu,', P', 1)) \in E^{\beta}_{C'}\} && \text{multiplicities 1 in } C' \\
= \; & \{\nu' \mid \exists v, v' \in V_C.\mathit{partner}_{C,k}(v) = (\nu, P, 1), \\
& \qquad \mathit{partner}_{C,k}(v') = (\nu', P', 1), (v, v') \in E^{\beta}_{C}\} && \text{Definition 4.1.3} \\
= \; & \{\nu' \mid \exists v, v' \in V_C.u \bowtie_C v, (v, v') \in E^{\beta}_{C}, \ell_C(v') = \nu'\} \\
= \; & \{\nu' \mid \exists v' \in V_C.(u, v') \in E^{\beta}_{C}, \ell_C(v') = \nu'\} && \text{Definition 4.1.1} \\
= \; & \ell_C(u \triangleright^{\beta}_{C}) && \text{(A.4)}
\end{aligned}
$$

For the proof of part (2), assume $\hat{C} = \alpha_k(C)$, again, and use C instead of G. Let $\mathit{partner}_{C,k}(u) = (\nu, P, 1)$. By the definition of ξ this yields $\xi(u) = (\hat{C}, (\nu, P, n))$. Finally, applying part (1), yields $\mathit{partner}_{\hat{C},k}(\xi(u)) = (\nu, P, n)$ concluding the argument.

Part (3) is an immediate consequence of part (2) by looking at the definitions of (abstract) partner constraint satisfaction, because simple partner constraints only look at the neighborhood in a qualitative fashion.

Let $C, \hat{C}$ be as aforementioned, and let $\hat{C}, \hat{u} \models pc$ for a partner constraint of degree k. This implies (by Definition 4.1.7, part (2)) for all $(in, \beta, \mu, q) \in pc$, where $q \leq k$.

$$\oplus^k\{q' \mid ((\mu, P, q'), \hat{u}) \in E^{\beta}_{\hat{C}}\} \quad \leq q \tag{A.5}$$

Note that $\sqsubseteq_k$ may be substituted by $\leq$, because q is at most k. Let $u \in \xi^{-1}(\hat{u})$ be such that $C, u \not\models p$. Without loss of generality, assume that

$$\mid \{(v, u) \in E^{\beta}_{C} \mid \ell_C(v) = \mu\} \mid = q' > q \tag{A.6}$$

Requirement (1) of Definition 4.1.7 cannot be violated because of part (2) of this lemma. (A.6) implies by the definition of *partner*:

$$\mid \{\mathit{partner}_{C,k}(v) \mid (\mathit{partner}_{C,k}(v), \mathit{partner}_{C,k}(u)) \in E^{\beta}_{C'}\} \mid = q'$$

where $C' = \mathit{partner}_k(C)$. Plugging in the definition of *partner* yields

$$\mid \{(\mu, P', 1) \mid ((\mu, P', 1), \mathit{partner}_{C,k}(u)) \in E^{\beta}_{C'}\} \mid = q'$$

Finally, applying the definition of *collapse* and the induced morphism ζ yields

$$\oplus^k\{n' \mid ((\mu, P', n), \hat{u}) \in E^{\beta}_{\hat{C}}\} \sqsupseteq_k q'$$

contradicting (A.5).

Lemma 4.1.6 Let r be a transformation rule over $\mathcal{N}$ and $\mathcal{E}$ that features simple partner constraints only. Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a graph. For any $k \geq 1$ holds that, if r matches G, then r matches $\alpha_k(G)$.

Proof Let m' be the injective morphism from L to G that exists according to Definition 2.1.6, and let ξ be the morphism induced by the abstraction $\hat{G} = \alpha_k(G)$. Then $m = \xi \circ m'$ is a morphism from L in $\hat{G}$ as a concatenation of morphisms. By Lemma 4.1.4, it suffices to show, that m preserves partner constraint satisfaction and that

$$\forall C \in cc(L).\forall(\nu, P, n) \in m(V_C).(|\, m^{-1}(\nu, P, n)\,|)_k \;\sqsubseteq_k n \tag{A.7}$$

Partner constraint satisfaction for simple partner constraints is guaranteed by part (3) of Lemma 4.1.5. It remains to prove (A.7). Let $C \in cc(L)$ and $(\nu, P, n) \in m(V_C)$ be arbitrary. Let D be the connected component matched by elements in C.

$$\begin{array}{lll} & |\, m^{-1}(\nu, P, n)\,| & \\ = & |\{v \in V_C \mid \xi \circ m'(v) = (\nu, P, n)\}| & \\ = & |\{v \in V_C \mid m'(v) = (\nu, P, 1)\}| & \text{Lemma 4.1.1, part (2)} \\ \leq & |\{u \in V_D \mid u \bowtie_D m'(v)\}| & \\ = & n & \text{Lemma 4.1.1, part (3)} \end{array}$$

Certainly, the result for natural numbers lifts to finite counting concluding the proof.

Theorem 4.1.7 Let $C \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a connected graph and let $r = (L, h, p, R)$ be a transformation rule with simple partner constraints over the same set of labels such, where L is connected. Let $k \geq 1$ be arbitrary. If

1. r matches $\alpha_k(C)$, and the match morphism m according to *Lemma* 4.1.4 is injective.
2. $\alpha_k(C)$ has unique partners.
3. $\alpha_k(C)$ has no summary cycles.

then r matches C.

Proof As a first step in proving this theorem, an auxiliary statement is proven. It is a consequence of the unique partner requirement for the abstract graph. Let ξ be the morphism induced by the abstraction. Let $\{\hat{C}\} = \alpha_k(C)$.
Claim Let $(\hat{u}, \hat{u}') \in E^{\beta}_{\hat{G}}$. Then

a. For all $v \in \xi^{-1}(\hat{u})$, there exists a $v' \in \xi^{-1}(\hat{u}')$ such that $(v, v') \in E_C^\beta$.

b. For all $v' \in \xi^{-1}(\hat{u}')$, there exists a $v \in \xi^{-1}(\hat{u})$ such that $(v, v') \in E_C^\beta$.

In other words, the concrete relation between nodes that is abstracted to a single edge in the abstract graph is total (a) and surjective (b). The proof of part (b) of the claim is completely symmetric to the proof of part (a) and omitted. Assume $(\hat{u}, \hat{u}') \in E_{\hat{C}}^\beta$, where $\hat{u} = (\nu, P, n)$ and $\hat{u}' = (\nu', P', n')$. According to the definition of *collapse* this implies that there exists an edge $((\nu, P, 1), (\nu', P', 1)) \in E_{C'}^\beta$, where $C' = \mathit{partner}_k(C)$. According to the definition of *partner*, there must hence exist $(v, v') \in E_C^\beta$ such that $\mathit{partner}_{C,k}(v) = (\nu, P, 1)$ and $\mathit{partner}_{C,k}(v') = (\nu', P', 1)$. By the definition of ξ, $\xi(u) = \hat{u}$ and $\xi(v') = \hat{u}'$. So there exists $v \in \xi^{-1}(\hat{u})$ and $v' \in \xi^{-1}(\hat{u}')$ such that $(v, v') \in E_C^\beta$.

Now, let $w \in \xi^{-1}(\hat{u})$ be arbitrary. Obviously, $w \bowtie_C v$. As w is partner equivalent to v, there must be some $w' \in V_C$ such that $(w, w') \in E_C^\beta$ and $\ell_C(w') = \nu'$. It remains to be shown that $w' \bowtie_C v'$, because this implies that $w' \in \xi^{-1}(\hat{u}')$ using the equivalence (A.4) from a previous proof. Assume $w' \not\bowtie v'$. Then $\xi(w') \neq \xi(v')$, but $(\hat{u}, \xi(w')) \in E_{\hat{C}}^\beta$ and $\ell_{\hat{C}}(\xi(w')) = \nu'$ contradicting assumption (b) of unique partners. This concludes the proof of the claim.

In the second part of the proof, a concrete match m' of L to C is successively constructed from the abstract match m. Let $V_S \subseteq V_L$ be the set of nodes of the left graph matched to a summary node, *i.e.*

$$V_S = \{u \in V_L \mid m(u) = (\nu, P, n), n \neq 1\}$$

For each $u \notin V_S$, set $m'(u) = \xi^{-1}(m(u))$. This is well-defined, because for one connected component, ξ maps injectively to non-summary nodes.

Let F be the subgraph of $\hat{C}$ spanned by $m(V_S)$, *i.e.*, $F = \hat{C}\ |_{m(V_S)}$. If edges are considered undirected, then F is a *forest* of summary nodes, because, by assumption (3), there are no undirected summary node cycles in $\hat{C}$. Let $\{\hat{u}_1, \ldots, \hat{u}_n\}$ be a depth-first traversal of F traversing all nodes and edges, such that $\hat{u}_i = m(u_i)$ for $1 \leq i \leq n$. The bijective mapping between V_S and F is possible, because m is assumed to be injective (assumption (1)). Construct m' for elements of V_S inductively as follows.

1. Pick an arbitrary element $v_1 \in \xi^{-1}(\hat{u}_1)$. This is possible because ξ is surjective.

2. For $i > 1$, $\hat{u}_i$ has at most one adjacent node in F that is an element of $M = \{\hat{u}_1, \ldots, \hat{u}_{i-1}\}$ (If there were two, a cycle would be created contradicting assumption (3).) By the induction hypothesis, all elements

of M are already in the domain of m'. If $\hat{u}_i$ has no adjacent node in M, choose $v_i \in \xi^{-1}(\hat{u}_i)$ arbitrarily and set $m'(u_i) = v_i$. Without loss of generality assume $(\hat{u}_j, \hat{u}_i) \in E_F^\beta$ for some $j < i$ and some $\beta \in \mathcal{E}$. Let $m(u_j) = v_j$, hence $v_j \in \xi^{-1}(\hat{u}_j)$. According to the **Claim** above, there exists a $v_i \in \xi^{-1}(\hat{u}_i) \subseteq V_G$, such that $(v_j, v_i) \in E_G^\beta$. Set $m'(u_i) = v_i$.

The mapping m' is an injective morphism by construction, because m is injective and because C is connected. Partner constraint satisfaction is guaranteed by applying Lemma 4.1.5.

Theorem 4.1.8 Let $\mathfrak{G}$ be a partner graph grammar and let $k \geq 1$ be arbitrary. There exists an $n \geq 0$ such that $[\![\mathfrak{G}]\!]_n^k = [\![\mathfrak{G}]\!]_i^k$ for all $i \geq n$. Furthermore,

$$\bigcup_{G \in [\![\mathfrak{G}]\!]} \alpha_k(G) \subseteq [\![\mathfrak{G}]\!]^k$$

Proof Let $\mathfrak{G} = (\mathcal{R}, \mathcal{I})$ be a partner graph grammar with simple partner constraints. In order to prove termination of the analysis, two obvious observations suffice. First, the set of ground canonical graphs is of finite size, because the number of canonical names is bounded and statically determined by the number of node labels, edge labels, and by the abstraction parameter k. The mapping $\lambda i.[\![\mathfrak{G}]\!]_i^k$ is clearly monotone, because of the set union used in the construction. The set of ground canonical graphs is trivially transferred into the complete powerset lattice. By the theory of [CC77], such a computation is guaranteed to terminate.

As for soundness, assume some $H \in [\![\mathfrak{G}]\!]$. According to Definition 2.1.10, there exists a sequence of derivations

$$\mathcal{I} \rightsquigarrow_{r_1} G_1 \rightsquigarrow_{r_2} \ldots G_{n-1} \rightsquigarrow_{r_n} H$$

The proof that $\alpha_k(G) \subseteq [\![\mathfrak{G}]\!]^k$ is conducted using induction on the length of this sequence. The induction base $n = 0$ is easy to show, since $G = \mathcal{I}$ and by Definition 4.1.10 $\alpha_k(\mathcal{I}) = [\![\mathfrak{G}]\!]_0^k \subseteq [\![\mathfrak{G}]\!]^k$. The induction step amounts to proving the general result:

$$G \rightsquigarrow_r G' \wedge \alpha_k(G) \subseteq \hat{G} \quad \text{implies} \quad \hat{G} \rightsquigarrow_r \hat{G}' \wedge \alpha_k(G') \subseteq \hat{G}' \qquad \text{(A.8)}$$

The derivation step $G \rightsquigarrow_r G'$ implies that r matches G. By Lemma 4.1.6 and the fact that only simple partner constraints occur, one may conclude, that r matches $\hat{G}$. By Lemma 4.1.3, there exists a materialization $\hat{M}$ of $\alpha_k(G)$ such that $\dot{\cup}\hat{M} \cong G$. This implies the existence of $\hat{M}'$ such that $\dot{\cup}\hat{M} \rightsquigarrow_r \hat{M}'$

and $\hat{M}' \cong G'$. This yields, that $\hat{G} \rightsquigarrow_r^k \alpha_k(\hat{M}')$, where $\alpha_k(\hat{M}') = \hat{G}'$ and $\hat{G}' \supseteq \alpha_k(G')$, because $G' \cong \hat{M}'$.

Lemma 4.3.2 Let $\hat{C} \in \mathcal{C}(\mathcal{N}, \mathcal{E}, k)$ be a connected, ground canonical graph with lone summary nodes. $\hat{M}$ is a materialization of $\hat{C}$, if and only if it is an S-materialization of $\hat{C}$.

Proof Let $\hat{C}$ be an abstract cluster as in the statement of the lemma, and let $\hat{M}'$ be a materialization of it. According to Definition 4.1.6:

$$\mathit{collapse}_k(\hat{M}') \;=\; \hat{C} \tag{A.9}$$

Let $\zeta : V_{\hat{M}'} \to V_{\hat{C}}$ be the morphism induced by $\mathit{collapse}_k$, and set

$$S(\hat{u}) \;=\; \{\!\{ s \mid (\nu, P, s) \in \zeta^{-1}(\hat{u}) \}\!\} \tag{A.10}$$

for any $\hat{u} \in V_{\hat{C}}$. Mapping S is well-defined, because ζ is total and surjective. Following the definition of $\mathit{collapse}_k$ (Definition 4.1.4), S satisfies the arithmetic requirement in the definition of S-materializations (Definition 4.3.4). Hence $\hat{M}'$ is an S-materialization of $\hat{C}$. It needs to be shown that $\hat{M}' = \hat{M}$. The equalities $V_{\hat{M}} = V_{\hat{M}'}$ and $\ell_{\hat{M}} = \ell_{\hat{M}'}$ are straightforward. It remains to be shown, that, for all $\beta \in \mathcal{E}$, $E^{\beta}_{\hat{M}} = E^{\beta}_{\hat{M}'}$, *i.e.*

$$\begin{aligned} E^{\beta}_{\hat{M}'} &= \{\!\{ (v_1, v_2) \mid (\hat{u}_1, \hat{u}_2) \in E^{\beta}_{\hat{C}}, v_i \in \zeta^{-1}(\hat{u}_i), i = 1, 2 \}\!\} \\ &= E^{\beta}_{\hat{M}} \end{aligned} \tag{A.11}$$

First, show that $E^{\beta}_{\hat{M}'} \subseteq E^{\beta}_{\hat{M}}$. Let $((\nu_1, P_1, s_1), (\nu_2, P_2, s_2)) \in E^{\beta}_{\hat{M}'}$. Due to the definition of *collapse* and (A.9), this implies, that there exists n_1 and n_2, such that $((\nu_1, P_1, n_1), (\nu_2, P_2, n_2)) \in E^{\beta}_{\hat{C}}$ and hence $((\nu_1, P_1, s_1), (\nu_2, P_2, s_2)) \in E^{\beta}_{\hat{M}}$.

For the inverse direction, let

$$((\nu_1, P_1, n_1), (\nu_2, P_2, n_2)) \;\in\; E^{\beta}_{\hat{C}} \tag{A.12}$$

and let $\hat{u}_i = (\nu_i, P_i, n_i)$ for $i = 1, 2$. Furthermore, let $v_i \in \zeta^{-1}(\nu_i, P_i, n_1)$ be arbitrary. Due to the definition of *collapse* and (A.9), it holds that $v_i = (\nu_i, P_i, s_i)$ for some s_i and $i = 1, 2$. As $\hat{C}$ has lone summary nodes only, $n_1 \neq 1$ and $n_2 \neq 1$ is not possible at the same time. Without loss of generality, let $n_1 = 1$ hence $s_1 = 1$.

Moreover, according to Definition 4.1.6, (A.9) implies for $i = 1, 2$

$$(\mathit{partner}_{\hat{M}', k}(v_i)) \downarrow 2 \;=\; (\mathit{partner}_{\hat{C}, k}(\hat{u}_i)) \downarrow 2 \tag{A.13}$$

(A.13) and (A.12) imply that $(in, \beta, \nu_1) \in \mathit{partner}(v_2)$. There must thus exist a $v' = (\nu_2, P', s') \in V_{\hat{M}'}$ such that $(v', v_2) \in E^{\beta}_{\hat{M}'}$. Assume $v' \neq v_1$. As $s_1 = n_1 = 1$, v' cannot be partner equivalent to v_1 meaning that $P' \neq P_1$. Therefore $\zeta(v') \neq \zeta(v_1)$ and because of $(v', v_2) \in E^{\beta}_{\hat{M}'}$ and the definition of ζ, also $(\zeta(v'), \hat{u}_2) \in E^{\beta}_{\hat{C}}$ contradicting the unique partner requirement for $\hat{u}_2$ Hence $v' = v_1$ implies $(v_1, v_2) \in E^{\beta}_{\hat{M}'}$ and concludes the proof.

Lemma 4.3.3 Let $\hat{C} \in \mathcal{C}(\mathcal{N}, \mathcal{E}, k)$ be a connected, ground canonical graph with lone summary nodes. All graphs C such that $\alpha_k(C) = \hat{C}$ are isomorphic, if $\hat{C}$ does not contain an ∞-summary node.

Proof Consider the mapping S_1 from nodes of $\hat{C}$ to multisets of $\mathbb{N}_k$, where $S_1(\nu, P, t) = \{\!\{\underbrace{1, \ldots, 1}_{t \text{ times}}\}\!\}$ and $t \neq \infty$. Let C be a graph that is isomorphic to the S_1 materialization of $\hat{C}$. It is then obvious that $\alpha_k(C) = \{\hat{C}\}$. Assume D such that $\alpha_k(D) = \{\hat{C}\}$, then $\mathit{partner}_k(D)$ is a materialization of $\hat{C}$ and an S-materialization of $\hat{C}$ because of Lemma 4.3.2. As there are no ∞-summary nodes in $\hat{C}$, it is also the S_1-materialization of $\hat{C}$. A connected graph F is certainly isomorphic to $\mathit{partner}_k(F)$, hence D is isomorphic to C concluding the proof.

Theorem 4.3.4 Let $\mathfrak{G} = (\mathcal{R}, \mathcal{I})$ be a friendly partner graph grammar and let

$$k \geq \max\{|\, \ell_L(\nu)^{-1} \,|\; |\; (L, \_, \_, \_) \in \mathcal{R}, \nu \in \mathcal{N}\}$$

The abstract graph semantics $[\![\mathfrak{G}]\!]^k$ is cluster complete, if it has a generating order and all $\hat{C} \in [\![\mathfrak{G}]\!]^k$ have lone summary nodes and unique partners.

Proof As a first step in this proof, some notations and notions shall be fixed. A multiset consisting of n elements x will be denoted $\{\!\{x\}\!\}_n$. An S_1 materialization of $\hat{C}$ is an S-materialization, where $S(\hat{u}) \subseteq_m \wp_m(\{1\})$ for all $\hat{u} \in V_{\hat{C}}$.

Note that for each concretization of an abstract cluster there exists an isomorphic S_1-materialization and vice versa. This is a consequence of the lone summary assumption and of Lemma 4.3.2. Therefore, an abstract cluster $\hat{C}$ is said to *have all its concretizations*, if and only if for all S_1 materializations C of $\hat{C}$, there exists a $G \in [\![\mathfrak{G}]\!]$ and a $C' \in \mathit{cc}(G)$ such that $C' \cong C$. It suffices to show that each abstract cluster $\hat{C} \in [\![\mathfrak{G}]\!]^k$ has all its concretizations.

The notion of S-materializations may be trivially lifted to sets of abstract clusters (in contrast to Definition 4.3.4). A materialization of an abstract

graph $\hat{G}$ is an S-materialization, if and only if all clusters in the materialization of $\hat{G}$ are an S-materialization of an abstract cluster $\hat{C} \in \hat{G}$.

A simple observation makes life easier. Partner equivalent nodes in an S materialization C of $\hat{C}$ have *equal neighbors*, *i.e.*

$$u \bowtie_C v \Leftrightarrow \forall \beta \in \mathcal{E}.u{\lhd}_C^\beta = v \lhd_C^\beta \ \wedge \ u{\rhd}_C^\beta = v{\rhd}_C^\beta$$

Assume $u \bowtie_C v$, $u \neq v$, and $(u, w) \in E_C^\beta$ for an S materialization C. If $\ell_C(w) = \mu$, then $\mu \in \ell_C(u{\rhd}_C^\beta)$. As u and v are partner equivalent, $\mu \in \ell_C(v{\rhd}_C^\beta)$, too. If there was a $w' \neq w \in V_C$ such that $\ell_C(w') = \mu$ and $(v, w') \in E_C^\beta$, this would either contradict the lone summary or the unique partner requirement for $\hat{C}$ and is thus not possible. Hence $(v, w) \in E_C^\beta$. All other steps in the proof are analogous.

Obviously, nodes with equal partners are affected in the same manner by an update *wrt.* to their label and their partners' labels, *i.e.* they remain partner equivalent after an update.

It will now be proven, that node evolution between abstract clusters can be faithfully mimicked in the concrete graph semantics *wrt.* to the given Δ. More formally:

Claim Let $(\hat{C}, \hat{u})$ $\mathfrak{E}(r, n)$ $(\hat{C}', \hat{u}')$. If $\hat{u} = (\nu, P, \infty)$ is the only ∞-summary node in $\hat{C}$ and $\hat{C}$ has a concretization isomorphic to an S_1 materialization of $\hat{C}$, where $S_1(\hat{u}) = \{\!\{1\}\!\}_s$, then $\hat{C}'$ has a concretization whose corresponding S_1'-materialization satisfies $S_1'(\hat{u}') = \{\!\{1\}\!\}_{s+n}$. Let

$$S(\hat{u}) = \{\!\{1\}\!\}_q \dot{\cup} \{\!\{t_0\}\!\}_1 \dot{\cup} \{\!\{t_1, \ldots, t_n\}\!\}$$

be the materialization triggering the node evolution, and let q be the number of nodes matched by rule r in $\hat{C}$, such that where t_0 corresponds to the t whose existence is required by point 3 of the definition of node evolution.

For all $1 \leq i \leq n$ holds $t_i \neq \infty$, because otherwise $s^- = \infty$ in contrast to the requirement of a 1-increment path. Hence the following computations may be conducted using general integers instead of counting to k. Let $d = \Sigma_{i=1}^n t_i$. Then $s^- = d + q$. Let

$$q' = \mid \{\hat{u} = (\nu, P, 1) \in m(V_L) \cap V_C \mid \hat{u} \notin V_{C'} \text{ or } \mathit{partner}_{C',k}(\hat{u}) \neq (\nu, P, 1)\} \mid$$

Hence q' is the number of matched nodes in C that are affected by the update (disappear or change their canonical name). Additionally, let

$$d' = \Sigma\{a \mid \hat{u} = (\mu, Q, a) \notin V_C, \mathit{partner}_{C',k}(\hat{u}) = (\nu, P', a)\}$$

be the number of nodes (rather the sum of their multiplicities) not in C that migrate to the same connected component as (ν, P, t_0) after the update. Also

$d' \neq \infty$, because otherwise $s^+ = \infty$. This yields $s^+ = d + (q - q') + d'$ and thus

$$\begin{aligned} n &= s^+ - s^- \\ &= (d + (q - q') + d') - (d + q) \\ &= d' - q' \end{aligned}$$

This shows that n is independent of the particular choice of $t_0, t_1, \ldots t_n$. As partner equivalent nodes are affected in the same way by an update and because all other clusters involved in the materialization have all their concretizations by assumption, the same update can be performed, if C is an S_1 materialization of $\hat{C}$ with $S_1(\hat{u}) = \{\!\{1\}\!\}_s$ for $s > q$. The latter is implied by $s > k \geq q$. (This is where the requirement on k comes onto the stage.) So assume C is really an S_1-materialization. The number s' of nodes with canonical name $(\nu, P', 1)$ in C' is then computed as follows, where $t = 1$ because C is an S_1-materialization.

$$\begin{aligned} s &= d + q + t \\ s' &= d + t + (q - q') + d' \\ &= (d + q + t) + (d' - q') \\ &= s + n \end{aligned}$$

This concludes the proof of the claim, because of the correspondence between S_1 materializations and concretizations for the case of lone summary nodes.

Now, the core of the proof – a well-founded induction on the generating order – may be conducted as follows. It must be shown, that each $\hat{C} \in [\![\mathfrak{G}]\!]^k$ has all its concretizations. This is trivial for the minimal clusters, because they have no ∞-summary nodes, thus only one unique S_1-materialization (due to lone summary nodes) corresponding to the unique concretization that is isomorphic to the right graph of some create rule. Non-minimal abstract clusters without ∞-summaries have all their concretizations, because they result from an abstract update from clusters with all their materializations (by induction hypothesis). In this abstract update, an S_1-materialization can be used instead of the actually triggering materialization by the introductory reasoning of this proof.

Now, consider an abstract cluster $\hat{C}$ minimal among the ∞-clusters. Let $\hat{u}$ be the ∞-summary node of $\hat{C}$. By definition of a generating order there must be a 1-increment path from a cluster $\hat{C}'$ equal to $\hat{C}$ up to $\hat{u}$. An inductive argument using the claim about node evolution shows that all S_1-materializations of $\hat{C}$ exist. The same claim proves the existence of all materializations of non-maximal clusters, because they have at most one ∞ summary node.

The only part left to show is the case of a maximal cluster $\hat{C}$, that may have several ∞ summary nodes named $\hat{u}_1$ through $\hat{u}_t$. Due to point 6 in the definition of generating order, there exists an abstract update

$$\{\hat{C}_1, \ldots, \hat{C}_n\} \rightsquigarrow_r^k \{\hat{C}, \ldots\}$$

and $i_1, \ldots, i_t$ such that $(\hat{C}_{i_j}, \hat{v}_j)$ $\mathfrak{E}(r, 0)$ $(\hat{C}, \hat{u}_j)$ for j from 1 to t. Due to the cluster multiplicity property and the induction hypothesis the proof can be concluded, if all C_{i_j} are distinct.

Assume now, there exist $(\hat{D}, \hat{v})$ evolving to distinct $\hat{u}_1$ and $\hat{u}_2$ in $\hat{C}$. If this is to occur due to the same rule application, it must be due to two clusters materialized from $\hat{D}$. Otherwise point 4 of the node evolution definition cannot hold, because the same node would have different partners in the updated cluster C'. Due to the cluster multiplicity property these two materialized cluster can have arbitrary concretizations in any possible combination independently. Hence $\hat{u}_1$ and $\hat{u}_2$ can have all possible concretizations independently concluding the well-founded induction and the whole proof.

Theorem 4.3.5 Let $\mathfrak{G}$ be a friendly partner graph grammar and let

$$k \geq \max\{|\,\ell_L(\nu)^{-1}\,| \mid (L, \_, \_, \_) \in \mathcal{R}, \nu \in \mathcal{N}\}$$

The abstract graph semantics $[\![\mathfrak{G}]\!]^k$ is free of spurious clusters, if it has an almost generating order, and lone summary nodes and unique partners for all clusters except the maximal ones.

Proof The proof is an immediate consequence of the proof of Theorem 4.3.4. The difference between generating and almost generating orders is only in the maximal clusters *wrt.* the strict order. In both cases, in particular in the almost generating one, all but the maximal clusters have all concretizations in the concrete graph semantics. Hence also the derivation leading to a maximal abstract clusters may be mimicked in the concrete, because all necessary smaller clusters have all their concretizations. The only thing that is not guaranteed, is that the maximal clusters have all their concretizations. They must have at least one, however, proving the absence of spurious clusters.

Theorem 4.3.6 Let $\mathfrak{G} = (\mathcal{R}, \mathcal{I})$ be a friendly partner graph grammar and let

$$k \geq \max\{|\,\ell_L(\nu)^{-1}\,| \mid (L, \_, \_, \_) \in \mathcal{R}, \nu \in \mathcal{N}\}$$

The abstract graph semantics $[\![\mathfrak{G}]\!]^k$ is word decidable, if it is cluster complete, and if R is connected for all $(L, h, p, R) \in \mathcal{R}$.

Proof Consider the cluster multiplicity Lemma 4.3.1. It states, for friendly graph grammars $\mathfrak{G}$, that $G_1, G_2 \in [\![\mathfrak{G}]\!]$ implies $G_1 \dot\cup G_2 \in [\![\mathfrak{G}]\!]$. If all right graphs of all transformation rules in $\mathfrak{G}$ are known to be connected, also the inverse direction of this lemma holds. Whenever $G_1 \dot\cup G_2 \in [\![\mathfrak{G}]\!]$ then also $G_1, G_2 \in [\![\mathfrak{G}]\!]$. Together with cluster completeness implied by the assumption of the theorem the inverse cluster multiplicity yields the desired decidability result.

The inverse cluster multiplicity lemma remains to be proven, so assume $G_1 \dot\cup G_2 \in [\![\mathfrak{G}]\!]$. Assume further a direct derivation of $G_1 \dot\cup G_2$ from the empty initial graph. Without loss of generality assume, that G_1 and G_2 are connected. Since create rules are always applicable, the rule applications in the derivation of $G_1 \dot\cup G_2$ may be re-ordered, such that all create rules come first. The proof that there are direct derivations of G_1 and G_2 is by induction on the length of the remaining derivation (without create rules). If this length is 1, then only one rule application after all create rules leads to $G_1 \dot\cup G_2$. Since this rule has a connected right graph, its application *either* yields G_1 or G_2, where the respective other graph was created by a create rule. Obviously both G_1 and G_2 may be created alone and independently of each other in this case. The induction step is analogous.

Lemma 4.4.1 Let $G \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ and let $k \geq 1$. For all existential positive formulas ϕ holds: If $G, \rho \models \phi$, then $\dot\cup\alpha_k(G), \hat\rho \models \phi$, where $\hat\rho = \lambda x.\xi(\rho(x))$.

Proof Clearly, for a positive formula it suffices to show that the $G, \rho \models \nu(x)$ implies $\dot\cup\alpha_k(G), \hat\rho \models \nu(x)$, and $G, \rho \models \beta(x_1, x_2)$ implies $\dot\cup\alpha_k(G), \hat\rho \models \beta(x_1, x_2)$, and $G, \rho \models x_1 = x_2$ implies $\dot\cup\alpha_k(G), \hat\rho \models x_1 = x_2$. Remember that $\hat\rho(x) = \xi(\rho(x))$ for the morphism ξ induced by the abstraction.

$$\begin{aligned} G, \rho \models \nu(x) &\Rightarrow \ell_G(\rho(x)) = \nu \\ &\Rightarrow \ell_{\dot\cup\alpha_k(G)}(\xi(x)) = \nu \\ &\Rightarrow \ell_{\dot\cup\alpha_k(G)}(\hat\rho(x)) = \nu \end{aligned}$$

$$\begin{aligned} G, \rho \models \beta(x_1, x_2) &\Rightarrow (\rho(x_1), \rho(x_2)) \in E^\beta_G \\ &\Rightarrow (\xi(\rho(x_1)), \xi(\rho(x_2))) \in E^\beta_{\dot\cup\alpha_k(G)} \\ &\Rightarrow (\hat\rho(x_1), \hat\rho(x_2)) \in E^\beta_{\dot\cup\alpha_k(G)} \end{aligned}$$

$$\begin{aligned} G, \rho \models x_1 = x_2 &\Rightarrow \rho(x_1) = \rho(x_2) \\ &\Rightarrow \xi(\rho(x_1) = \xi(\rho(x_2)) \\ &\Rightarrow \hat\rho(x_1) = \hat\rho(x_2) \end{aligned}$$

All derivations merely require ξ to be a morphism, in fact, they hold for any homomorphic abstraction. They hold also for universal formulas, if ξ is surjective, which is the case here.

Theorem 4.4.2 Let $C \in \mathcal{G}(\mathcal{N}, \mathcal{E})$ be a connected graph and let $k \geq 1$. Let ϕ be a first order formula without equality. If $\{\hat{C}\} = \alpha_k(C)$ with induced morphism ξ has lone summary nodes and unique partners, then

- $C, \rho \models \phi \;\Rightarrow\; \hat{C}, \hat{\rho} \models \phi$, for any assignment ρ and $\hat{\rho} = \lambda x.\xi(\rho(x))$.
- $\hat{C}, \hat{\rho} \models \phi \;\Rightarrow\; C, \rho \models \phi$ for all assignments $\hat{\rho}$ and for all assignments ρ, such that $\rho(x) \in \xi^{-1}(\hat{\rho}(x))$.

Proof Let ξ be the morphism induced by $\alpha_k(C) = \{\hat{C}\}$. As $\hat{C}$ has lone summary nodes, it is safe to assume, that C is isomorphic to an S-materialization of $\hat{C}$ (by Lemma 4.3.2). The proof is by structural induction on the shape of the formula ϕ. Note that ϕ is assumed to be in prenex form with negations propagated down to literals.

- $\phi \equiv \nu(x)$: Part 1 of the theorem is proven analogous to the proof of Lemma 4.4.1. Part 2 is proven by the following reasoning.

$$\begin{aligned} \hat{C}, \hat{\rho} \models \nu(x) &\Rightarrow \ell_{\hat{C}}(\hat{\rho}(x)) = \nu \\ &\Rightarrow \forall v \in \xi^{-1}(\hat{\rho}(x)).\ell_C(v) = \nu \qquad \text{(A.14)} \\ &\Rightarrow C, \rho \models \nu(x) \end{aligned}$$

  Concluding (A.14) is possible, because ξ is a surjective morphism. The final conclusion of this computation holds of course for any assignment ρ as stated in part 2 of the theorem.

- $\phi \equiv \beta(x_1, x_2)$: Again, part 1 follows from Lemma 4.4.1. Part 2 is derived as follows.

$$\begin{aligned} \hat{C}, \hat{\rho} \models \beta(x_1, x_2) &\Rightarrow (\hat{\rho}(x_1), \hat{\rho}(x_2)) \in E^{\beta}_{\hat{C}} \\ &\Rightarrow \forall v_i \in \xi^{-1}(\hat{\rho}(x_i)).(v_1, v_2) \in E^{\beta}_{C} \qquad \text{(A.15)} \\ &\Rightarrow C, \rho \models \beta(x_1, x_2) \end{aligned}$$

  It is crucial to stress that the correctness of (A.15) follows from the definition of S-materializations, Definition 4.3.4. As C is isomorphic to an S_1-materialization, the clause for edges in an S-materialization shows, that there are edges among *all* pairs of nodes belonging to connected equivalence classes in $\hat{C}$.

- $\phi \equiv \neg\nu(x)$: Analogous to the non-negated case.

- $\phi \equiv \neg\beta(x_1, x_2)$: As for part 1, assume that $(\rho(x_1), \rho(x_2)) \notin E^{\beta}_{C}$. The fact that this implies $(\xi(\rho(x_1)), \xi(\rho(x_2))) \notin E^{\beta}_{\hat{C}}$ is again derived by the definition of the edge set in an S-materialization: If $(\hat{\rho}(x_1), \hat{\rho}(x_2)) \in E^{\beta}_{\hat{C}}$, this definition implies $(v_1, v_2) \in E^{\beta}_{C}$ for all $v_i \in \xi^{-1}(\hat{\rho}(x_i))$.

  As for part 2, assume $(\hat{\rho}(x_1), \hat{\rho}(x_2)) \notin E^{\beta}_{\hat{C}}$ and there exists $v_i \in \xi^{-1}(\hat{\rho}(x_i))$ such that $(v_1, v_2) \in E^{\beta}_{C}$. This contradicts Lemma 4.4.1 and is thus excluded. This case is only possible because of the lone summary node property of $\hat{C}$. It is not typical of arbitrary homomorphic abstractions at all.

- The cases for boolean combinations are trivial.

- $\phi \equiv \forall x.\varphi$: Universal quantification amounts to a finite (because all universes under consideration are of finite size) conjunction, where the bound variable ranges over all elements of the universe. Again, part 1 is obvious by applying the induction hypothesis finitely many times knowing that ξ is total. In order to show part 2, the surjectivity of ξ in conjunction with the induction hypothesis suffices to prove the statement.

Theorem 4.4.4 Let $\mathfrak{G}$ be a partner graph grammar and let $k \geq 1$. Furthermore, let $\phi_\forall$ be a closed universal formula and let $\phi_\exists$ be a closed existential formula. If $[\![\mathfrak{G}]\!]^k$ is cluster complete and all $\hat{C} \in [\![\mathfrak{G}]\!]^k$ have lone summary nodes, then

1. If there exists a $\hat{C} \in [\![\mathfrak{G}]\!]^k$, such that $\hat{C}, [] \models \phi_\exists$, then $\mathfrak{G} \models \mathsf{EF}\,\phi_\exists$.

2. If there exists a $\hat{C} \in [\![\mathfrak{G}]\!]^k$, such that $\hat{C}, [] \not\models \phi_\forall$, then $\mathfrak{G} \not\models \mathsf{AG}\,\phi_\forall$.

If $[\![\mathfrak{G}]\!]^k$ is word decidable and all $\hat{C} \in [\![\mathfrak{G}]\!]^k$ have lone summary nodes, then for any closed first order formula ϕ that may contain equality

a. $\mathfrak{G} \models \mathsf{EF}\,\phi$ if and only if there exists an S_1-materialization M of $\dot{\cup}[\![\mathfrak{G}]\!]^k$, such that $M, [] \models \phi$.

b. $\mathfrak{G} \models \mathsf{AG}\,\phi$ if and only if for all S_1-materializations M of $\dot{\cup}[\![\mathfrak{G}]\!]^k$ holds, that $M, [] \models \phi$.

Proof Only one statement will be proven for the cluster completeness and word decidable case each. The remaining statements have very similar proofs.

As for part 1, assume there exists a $\hat{C} \in [\![\mathfrak{G}]\!]^k$ such that ϕ holds of $\hat{C}$. As $\hat{C}$ is required to have lone summary nodes and as ϕ is closed and assignments can be neglected, ϕ holds of every concretization C of $\hat{C}$. As $[\![\mathfrak{G}]\!]^k \subseteq [\![\mathfrak{G}]\!]$, there exists an $G \in [\![\mathfrak{G}]\!]$ and a $C \in cc(G)$, such that $\alpha_k(C) = \hat{C}$. As the formula is existential without equalities and holds of C, it also holds of G proving the statement.

Statement (a) is obvious due to the definition of word decidable abstract graph semantics. A graph is in $[\![\mathfrak{G}]\!]$, if and only if it is isomorphic to a concretization (in this case an S_1 materialization) of $\dot{\cup}[\![\mathfrak{G}]\!]^k$.

Appendix B

Tool Samples

Appendix B illustrates the usage of the `hiralysis` implementation of partner abstraction based abstract interpretation of partner graph grammars. It is partitioned into a number of figures as described below.

1. The abstract syntax of the partner graph grammar input to `hiralysis`.
2. An implementation of the $(\mathcal{R}_{\text{merge}}, E)$ partner graph grammar for the specification of an idealized platoon merge maneuver.
3. The abstract graph semantics visualization of the above partner graph grammar together with the visualization of the rules as they are output by the tool.
4. A stepwise computation of the same graph grammar illustrating the iteration bound feature of the tool.
5. A partner graph grammar specifying faulty channels for the idealized platoon merge.
6. A rather involved excerpt from the queue implementation of the platoon merge maneuver.
7. A code snippet from the `hiralysis` implementation showing the main fixpoint iteration.

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{nodeDecl}\ \underline{;}\ \mathit{edgeDecl}\ \underline{;}\ \mathit{graph}\ \underline{;}\ \mathit{rules} \\
\mathit{nodeDecl} & ::= & \underline{\texttt{nodelabels}}\ \mathit{nlist} \\
\mathit{edgeDecl} & ::= & \underline{\texttt{edgelabels}}\ \mathit{elist} \\
\mathit{nlist} & ::= & \nu \mid \nu\ \underline{,}\ \mathit{nlist} \\
\mathit{elist} & ::= & \beta \mid \beta\ \underline{,}\ \mathit{elist} \\
\mathit{graph} & ::= & \underline{\texttt{empty}} \\
 & \mid & \underline{[}\ \mathit{nodeset}\ \underline{,}\ \mathit{edgeset}\ \underline{]} \\
 & \mid & \underline{[}\ \mathit{nodeset}\ \underline{,}\ \mathit{edgeset}\ \underline{,}\ \mathit{pconstraints}\ \underline{]} \\
\mathit{nodeset} & ::= & \underline{\{\ \}} \mid \underline{\{}\ \mathit{nodes}\ \underline{\}} \\
\mathit{edgeset} & ::= & \underline{\{\ \}} \mid \underline{\{}\ \mathit{edges}\ \underline{\}} \\
\mathit{nodes} & ::= & x\underline{:}\nu \mid x\underline{:}\nu\ \underline{,}\ \mathit{nodes} \\
\mathit{edges} & ::= & \underline{(}x\underline{,}x\underline{):}\beta \mid \underline{(}x\underline{,}x\underline{):}\beta\ \underline{,}\ \mathit{edges} \\
\mathit{pconstraints} & ::= & \varepsilon \mid \underline{\texttt{partner}(}x\underline{)} \equiv \underline{\{\ \}} \\
 & \mid & \underline{\texttt{partner}(}x\underline{)} \equiv \underline{\{}\ \mathit{nlist}\ \underline{\}} \\
\mathit{rules} & ::= & \underline{\texttt{create}}\ \mathit{graph}\underline{;} \\
 & \mid & \underline{\texttt{destroy}}\ \mathit{graph}\underline{;} \\
 & \mid & \underline{\texttt{rule}}\ \mathit{graph}\ \underline{,}\ \mathit{graph}\underline{;} \\
 & \mid & \underline{\texttt{rule}}\ \mathit{graph}\ \underline{,}\ \underline{\texttt{connected}}\ \underline{,}\ \mathit{graph}\underline{;} \\
 & \mid & \underline{\texttt{rule}}\ \mathit{graph}\ \underline{,}\ \underline{\texttt{disjoint}}\ \underline{,}\ \mathit{graph}\underline{;} \\
 & \mid & \mathit{rules}\ \mathit{rules}
\end{array}
$$

Fig. B.1: Abstract syntax of the `hiralysis` input language specifying partner graph grammars. Terminals are underlined. In addition to the syntactical categories above, the categories $\nu \in \mathcal{N}$ of node labels, $\beta \in \mathcal{E}$ of edge labels, and $x \in \mathit{Var}$ of node variables are used. Create and destroy rules are just special notations for rules with empty left or right graph, respectively. The first *graph* in a program is the initial graph. Note that simple partner constraints are simplified even further to only restrict the set of labels of adjacent nodes regardless of their connection to the constrained node.

```
nodelabels ldr, fa, flw, rl, fl;

edgelabels l;

empty;

create [{x:fa},{}];

destroy [{x:fa},{}];

rule [{x1:fa,x2:fa}, {}], disjoint,
     [{x1:rl,x2:fl}, {(x1,x2):l} ];
// merge between two free agents

rule [{x1:ldr,x2:ldr}, {}], disjoint,
     [{x1:rl,x2:fl}, {(x1,x2):l} ];
// merge between two leaders

rule [{x1:fa,x2:ldr}, {}], disjoint,
     [{x1:rl,x2:fl}, {(x1,x2):l} ];
// merge between free agent and leader

rule [{x1:ldr,x2:fa}, {}], disjoint,
     [{x1:rl,x2:fl}, {(x1,x2):l} ];
// merge between leader and free agent

rule [{x1:rl, x2:fl, x3: flw}, {(x1,x2):l, (x1,x3):l}],
     [{x1:rl, x2:fl, x3: flw}, {(x1,x2):l, (x2,x3):l}];
  // passing a follower from back to front leader

rule [{x1:rl,x2:fl}, {(x1,x2):l}, partner(x1)={fl} ],
     [{x1:flw, x2:ldr},{(x2,x1):l}];
// re-establish platoon after all followers are handed over;
// an example of a "partner constraint"
```

Fig. B.2: The `hiralysis` implementation of the idealized merge protocol graph grammar $(\mathcal{R}_{\text{merge}}, E)$. Note that there is an instance of a partner constraint. The mapping – usually called h – from left to right graph is given implicitly by naming.

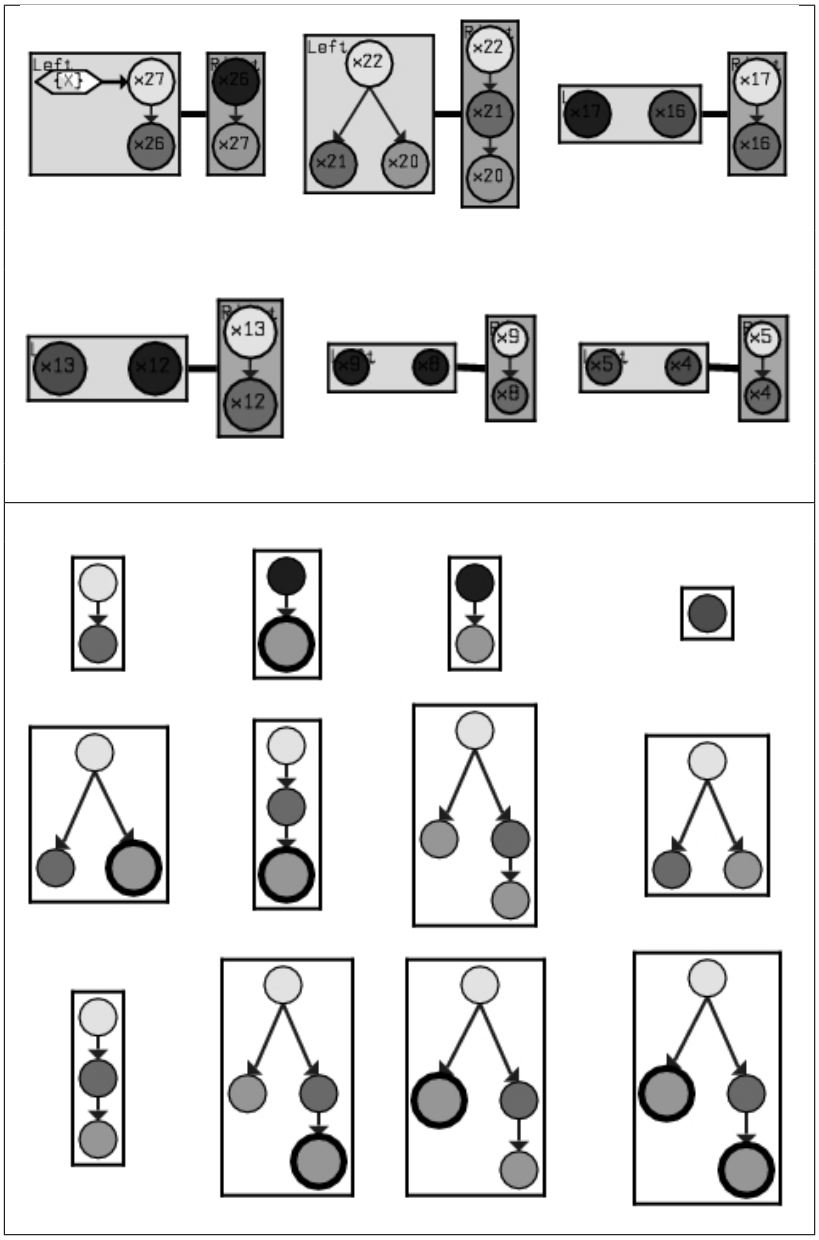


Fig. B.3: On top, there are six of the eight rules constituting the partner graph grammar modeling the idealized platoon merge. The correspondences between left and right graph is given by naming. The top, left rule demonstrates how partner constraints are drawn. On the bottom, the abstract graph semantics ($k = 1$) of the idealized platoon merge partner graph grammar is given. It is generated by `hiralysis` and visualized by `aisee`®. Note the similarities to the hand-drawn version given in Figure 4.7.

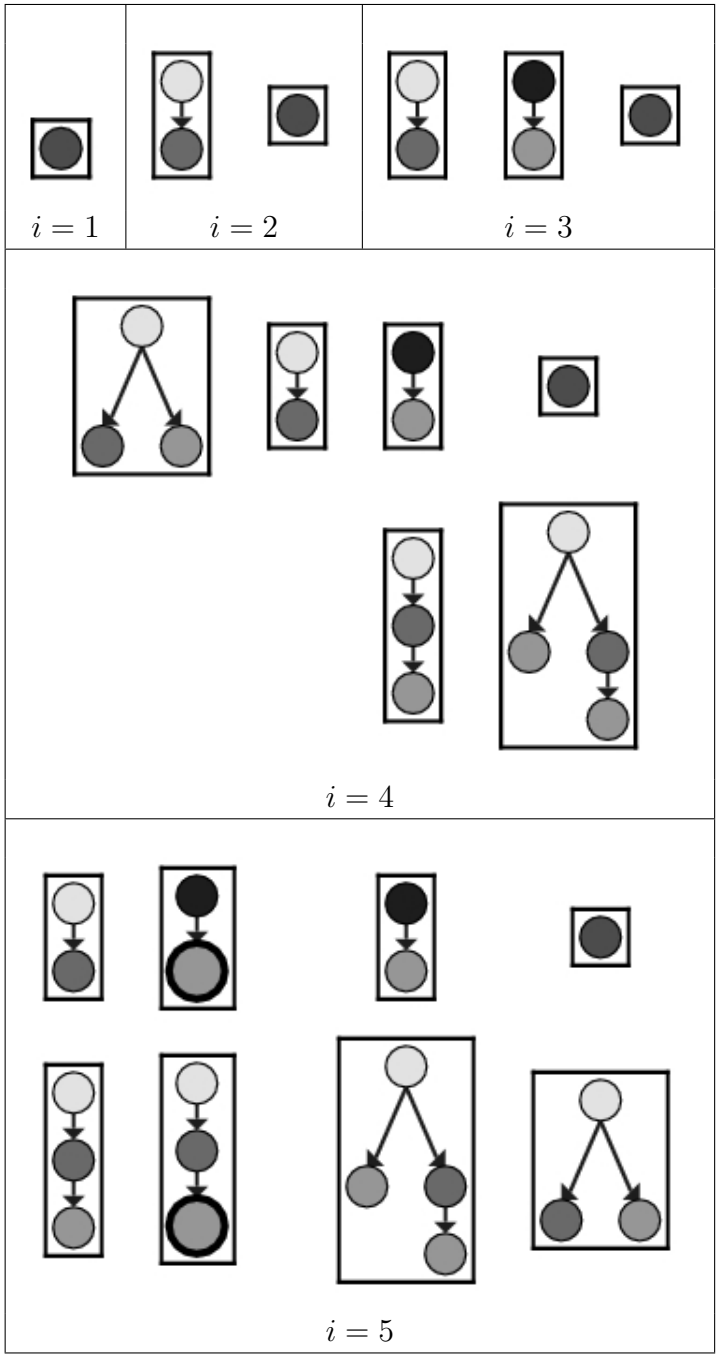


Fig. B.4: The single iteration steps in the computation of the abstract graph semantics of the idealized platoon merge. The graphs were generated using the iteration command line option of the tool. The case of $i = 0$ is empty, because the initial graph is empty, whereas the case of $i = 6$ is already the whole abstract graph semantics displayed in Figure B.3.

```
...
// destroy any possible link

rule [{x1:fa,x2:fa},{(x1,x2):l}], [{x1:fa,x2:fa},{}];
rule [{x1:fa,x2:ldr},{(x1,x2):l}], [{x1:fa,x2:ldr},{}];
rule [{x1:fa,x2:flw},{(x1,x2):l}], [{x1:fa,x2:flw},{}];
rule [{x1:fa,x2:rl},{(x1,x2):l}], [{x1:fa,x2:rl},{}];
rule [{x1:fa,x2:fl},{(x1,x2):l}], [{x1:fa,x2:fl},{}];
rule [{x1:ldr,x2:fa},{(x1,x2):l}], [{x1:ldr,x2:fa},{}];
rule [{x1:ldr,x2:ldr},{(x1,x2):l}], [{x1:ldr,x2:ldr},{}];
rule [{x1:ldr,x2:flw},{(x1,x2):l}], [{x1:ldr,x2:flw},{}];
rule [{x1:ldr,x2:rl},{(x1,x2):l}], [{x1:ldr,x2:rl},{}];
rule [{x1:ldr,x2:fl},{(x1,x2):l}], [{x1:ldr,x2:fl},{}];
rule [{x1:flw,x2:fa},{(x1,x2):l}], [{x1:flw,x2:fa},{}];
rule [{x1:flw,x2:ldr},{(x1,x2):l}], [{x1:flw,x2:ldr},{}];
rule [{x1:flw,x2:flw},{(x1,x2):l}], [{x1:flw,x2:flw},{}];
rule [{x1:flw,x2:rl},{(x1,x2):l}], [{x1:flw,x2:rl},{}];
rule [{x1:flw,x2:fl},{(x1,x2):l}], [{x1:flw,x2:fl},{}];
rule [{x1:fl,x2:fa},{(x1,x2):l}], [{x1:fl,x2:fa},{}];
rule [{x1:fl,x2:ldr},{(x1,x2):l}], [{x1:fl,x2:ldr},{}];
rule [{x1:fl,x2:flw},{(x1,x2):l}], [{x1:fl,x2:flw},{}];
rule [{x1:fl,x2:rl},{(x1,x2):l}], [{x1:fl,x2:rl},{}];
rule [{x1:fl,x2:fl},{(x1,x2):l}], [{x1:fl,x2:fl},{}];
rule [{x1:rl,x2:fa},{(x1,x2):l}], [{x1:rl,x2:fa},{}];
rule [{x1:rl,x2:ldr},{(x1,x2):l}], [{x1:rl,x2:ldr},{}];
rule [{x1:rl,x2:flw},{(x1,x2):l}], [{x1:rl,x2:flw},{}];
rule [{x1:rl,x2:fl},{(x1,x2):l}], [{x1:rl,x2:fl},{}];
rule [{x1:rl,x2:rl},{(x1,x2):l}], [{x1:rl,x2:rl},{}];

// standard merge rules
...
```

Fig. B.5: The `hiralysis` implementation of the [FAULT1] scenario of Section 3.1.3. The given rules allow any possible edge between two nodes to disappear. These rules are combined with the rules of Figure B.2 to make up the scenario described in Section 4.5.2. As this is a tedious combinatorial exercise, one of the next development steps is to equip the `hiralysis` tool with a command line option, that automatically augments a partner graph grammar with the ability to delete any edge at arbitrary times. Also, the other faulty scenarios of Section 3.1.3 will be implemented in that way.

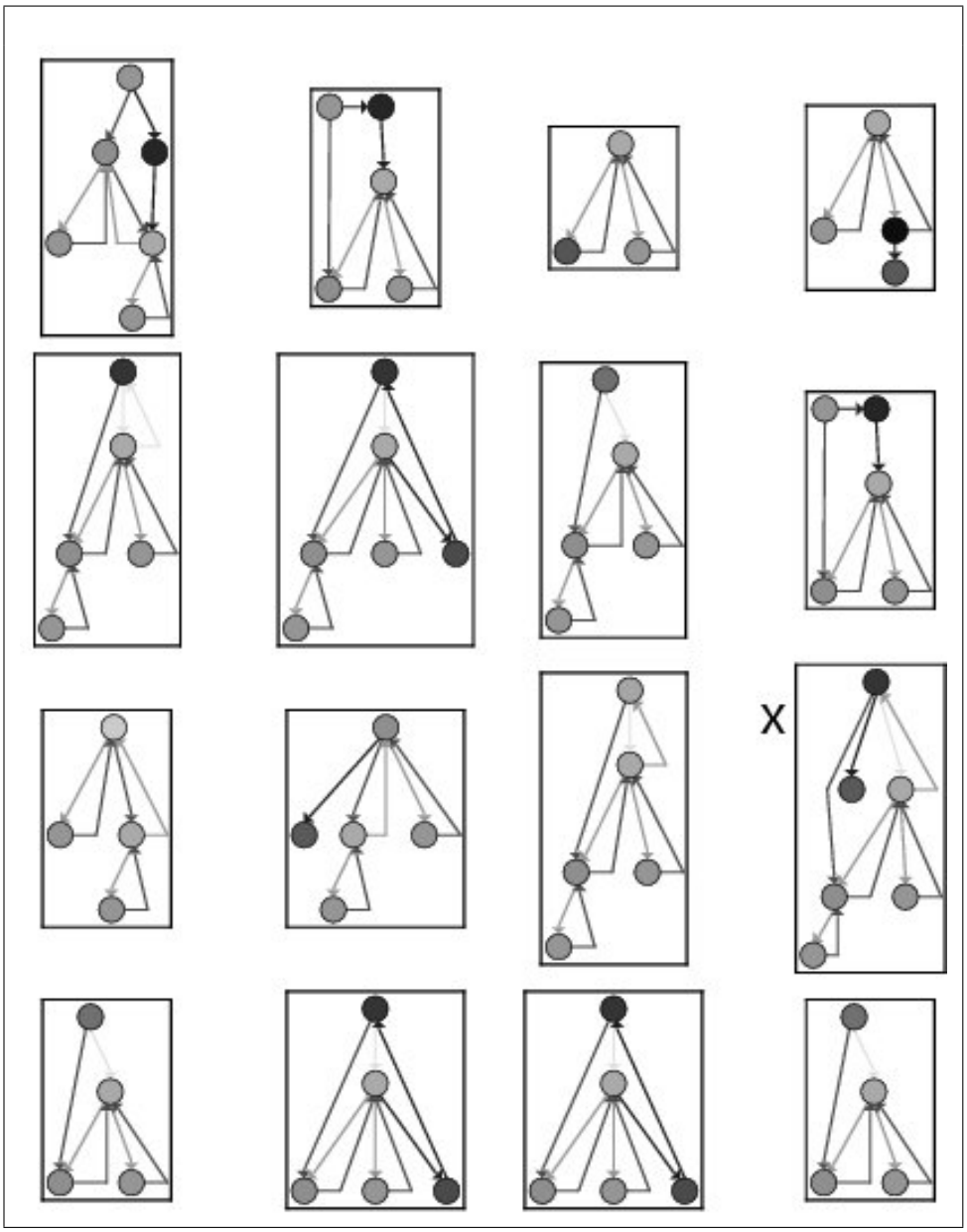


Fig. B.6: An excerpt of the 159 abstract clusters constituting the abstract graph semantics of the queue based platoon implementation. This is an example featuring much more complicated abstract clusters compared to the idealized version. Consider the X-marked abstract cluster. The four lighter nodes represent two merging platoons. The top dark node is a follower being handed over. Attached to it is a dark acknowledgement message.

```
int newMatch;
int applications;

CLUSTER *analyze(GTS *gts, int loop)
{
  CLUSTER *result;
  ANARULE *rules;
  int iter;

  iter = 0;
  newMatch = 1;
  applications = 0;

  result = NULL;
  result = initResult(gts);

  rules = NULL;
  rules = initRules(gts->rules);
  while (newMatch && iter < loop+1) {
    newMatch = 0;
    findNewMatches(result, rules, iter);
    result = apply(result, rules, iter);
    iter++;
  }

  return result;
}
```

Fig. B.7: The implementation of the main fixpoint iteration in `hiralysis`. The function gets the partner graph grammar and a loop bound and returns the abstract graph semantics. First the result is initialized to contain the partner abstraction of the initial graph. The parsed rules need some massaging before being applied. The `newMatch` flag is set to 1, whenever a new abstract cluster is added to the search tree of abstract clusters representing the currently computed abstract graph semantics.

Bibliography

[BCE+05] Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel, Barbara König, and Vitali Kozioura. Verifying red-black trees. In *Proc. of COSMICAH '05*, 2005. Proceedings available as report RR-05-04 (Queen Mary, University of London).

[BCK04] Paolo Baldan, Andrea Corradini, and Barbara König. Verifying finite-state graph grammars: An unfolding-based approach. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2004.

[BCM99] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Unfolding and event structure semantics for graph grammars. In *FoSSaCS*, pages 73–89, 1999.

[BDNN98] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1998.

[BKR04] Paolo Baldan, Barbara König, and Arend Rensink. Graph grammar verification through abstraction. Dagstuhl Seminar Proceedings 04241, 2004.

[BSTW06] Jörg Bauer, Ina Schaefer, Tobe Toben, and Bernd Westphal. Specification and verification of dynamic communication systems. In *Proc. of the 6th Conference on Application of Concurrency to System Design (ACSD 2006)*. IEEE Computer Society, 2006. to appear.

[BW95] Michael Barr and Charles Wells. *Category theory for computing science*, volume - of *Prentice Hall International Series in Computer Science*. Prentice-Hall, 2nd ed. edition, 1995.

[BW06] Jörg Bauer and Reinhard Wilhelm. Analysis of dynamic communicating systems by hierarchical abstraction. Dagstuhl Seminar Proceedings 06081, 2006.

[Car46] Rudolf Carnap. Modalities and quantification. *Journal of Symbolic Logic*, pages 33–64, 1946.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.

[DH01] Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, July 2001.

[Dis03] Dino Distefano. *On Model Checking the Dynamics of Object-based Software*. PhD thesis, University of Twente, 2003.

[DKC+04] S. Du, A. Khan, S. Chaudhuri, A. Post, A.K. Saha, P. Druschel, D.B. Johnson, and R. Riedi. Self-organizing hierarchical routing for scalable ad hoc networking. Technical Report TR04-433, Rice University, Houston, Texas, 2004.

[DKR04] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. Who is pointing when to whom? In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 250–262. Springer, 2004.

[DP05] Brian A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge University Press, CAmbridge, 2. ed. edition, 2005.

[DW02] Werner Damm and Bernd Westphal. Live and let die: Lsc-based verification of uml-models. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors,

FMCO, volume 2852 of *Lecture Notes in Computer Science*, pages 99–135. Springer, 2002.

[EEPPR04] Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors. *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*, volume 3256 of *Lecture Notes in Computer Science*. Springer, 2004.

[FH05] Christian Ferdinand and Reinhold Heckmann. Verifying timing behavior by abstract interpretation of executable code. In Dominique Borrione and Wolfgang J. Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 336–339. Springer, 2005.

[GRS05] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In Palsberg and Abadi [PA05], pages 338–350.

[Hec98] Reiko Heckel. Compositional verification of reactive systems specified by graph transformation. In *FASE*, pages 138–153, 1998.

[HESV91] A. Hsu, F. Eskafi, S. Sachs, and P. Varaiya. The design of platoon maneuver protocols for IVHS. Technical Report UCB-ITS-PRR-91-6, University of California, Berkley, 1991.

[HK87] Z. Har'El and P.R. Kurshan. COSPAN user's guide. AT&T Bell Laboratories, Murray Hill, NJ, 1987.

[HLM04] Reiko Heckel, Georgios Lajios, and Sebastian Menge. Stochastic graph transformation systems. In Ehrig et al. [EEPPR04], pages 210–225.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[HP02] Annegret Habel and Detlef Plump. Relabelling in graph transformation. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *ICGT*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2002.

[KK06] Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *Proc. of TACAS '06*. Springer, 2006. LNCS. to appear.

[KL06] Piotr Kosiuczenko and Georgios Lajios. Simulation of generalised semi-markov processes based on graph transformation systems. In *Proc. of Workshop on Graph Transformation for Verification and Concurrency*, 2006.

[Kön00] Barbara König. A graph rewriting semantics for the polyadic pi-calculus. In *Proc. of GT-VMT '00 (Workshop on Graph Transformation and Visual Modeling Techniques)*, pages 451–458. Carleton Scientific, 2000.

[Kri63] Saul Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[LARSW00] Tal Lev-Ami, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA*, pages 26–38, 2000.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *Static Analysis Symposium*. Springer, 2000. http://www.math.tau.ac.il/∼ rumster.

[Lew68] David Lewis. Counterpart theory and quantified modal logic. *Journal of Philosophy*, LXV(5):113–126, 1968.

[LKW93] Michael Löwe, Martin Korff, and Annika Wagner. An algebraic framework for the transformation of attributed graphs. *Term Graph Rewriting: Theory and Practice*, pages 185–199, 1993.

[LL98] John Lygeros and Nancy A. Lynch. Strings of vehicles: Modeling and safety conditions. In Thomas A. Henzinger and Shankar Sastry, editors, *HSCC*, volume 1386 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 1998.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[ML71] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, New York - Heidelberg - Berlin, 1971.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[NNP04] Hanne Riis Nielson, Flemming Nielson, and Henrik Pilegaard. Spatial analysis of bioambients. In Roberto Giacobazzi, editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2004.

[PA05] Jens Palsberg and Martín Abadi, editors. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. ACM, 2005.

[PAT03] PATH. California partners for advanced transport and highway, 1986-2003. http://www.path.berkeley.edu/.

[Pie91] Benjamin C. Pierce. *Basic category theory for computer scientists*, volume - of *Foundations of computing series, research reports and notes*. MIT Press, Cambridge - London, 1991.

[RBR+05] Noam Rinetzky, Jörg Bauer, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In Palsberg and Abadi [PA05], pages 296–309.

[RD06] Arend Rensink and Dino Distefano. Abstract graph transformation. *Electr. Notes Theor. Comput. Sci.*, 157(1):39–59, 2006.

[Ren04a] Arend Rensink. Canonical graph shapes. In David A. Schmidt, editor, *ESOP*, volume 2986 of *Lecture Notes in Computer Science*, pages 401–415. Springer, 2004.

[Ren04b] Arend Rensink. Representing first-order logic using graphs. In Ehrig et al. [EEPPR04], pages 319–335.

[Ros05] Benjamin Rossman. Existential positive types and preservation under homomorphisms. In *LICS*, pages 467–476. IEEE Computer Society, 2005.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[San94] Georg Sander. Graph layout through the vcg tool. In Roberto Tamassia and Ioannis G. Tollis, editors, *Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pages 194–205. Springer, 1994.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[SW01] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[Yah01] Eran Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, pages 27–40, 2001.

[YRSW03] E. Yahav, T. Reps, S. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In P. Degano, editor, *Proc. ESOP'03*, number 2618 in LNCS, pages 204–222. Springer-Verlag, 2003.