CLUSTER ABSTRACTION OF GRAPH TRANSFORMATION SYSTEMS

Dissertation

zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften der
Naturwissenschaftlich-Technischen Fakultäten der
Universität des Saarlandes

von

Peter Backes

Saarbrücken
2015

Tag des Kolloquiums:	13. November 2015
Dekan:	Prof. Dr. Markus Bläser
Prüfungsausschuss:	
Vorsitzender:	Prof. Dr. Bernd Finkbeiner Universität des Saarlandes, Saarbrücken
Gutachter:	Prof. Dr. Dr. h.c. Reinhard Wilhelm Universität des Saarlandes, Saarbrücken
	Prof. Dr. Jan Reineke Universität des Saarlandes, Saarbrücken
	Prof. Dr. Barbara König Universität Duisburg-Essen
Akademischer Mitarbeiter:	Dr. Tobias Mömke

Impressum

Herstellung und Verlag: epubli GmbH, Berlin, www.epubli.de
Printed in Germany
ISBN: 978-3-7375-7665-9

Abstract

This dissertation explores the problem of analyzing the reachable graphs of graph transformation systems. Such systems rewrite graphs according to subgraph replacement rules; we allow negative application conditions to be specified in addition. This problem is hard because the number of reachable graphs is potentially unbounded.

We use abstract interpretation to compute a finite, overapproximated representation of the reachable graphs. The main idea is the notion of a cluster: We abstract the graph locally for each of its nodes such that we obtain a bounded cluster with the node and its immediate neighborhood. Then, we eliminate duplicate clusters such that we obtain a bounded abstraction for the entire graph. We lift concrete rule application to this abstraction, eventually obtaining an overapproximation of all reachable graphs.

We present ASTRA, an implementation of cluster abstraction, and the merge protocol from car platooning, our main test case. This protocol enables autonomous cars to form and merge platoons consisting of a leader car and several followers, such that the leader controls speed and lane. The abstraction does well with the merge protocol, and also manages to analyze several other standard case studies from the literature, as well as test cases automatically generated from a higher-level formalism.

Zusammenfassung

Diese Dissertation untersucht, wie sich die erreichbaren Graphen eines Graphtransformationssystems analysieren lassen. Solche Systeme verändern Graphen gemäß Teilgraphersetzungsregeln; wir lassen zusätzlich negative Anwendungsbedingungen zu. Dieses Problem ist schwierig, da die Anzahl der erreichbaren Graphen potentiell unbeschränkt ist.

Wir benutzen abstrakte Interpretation, um eine endliche überapproximierte Darstellung der erreichbaren Graphen zu berechnen. Die Hauptidee ist der Begriff des Clusters: Wir abstrahieren den Graphen lokal für jeden seiner Knoten und erhalten einen Cluster beschränkter Größe mit diesem Knoten und seiner direkten Umgebung. Dann eliminieren wir doppelte Cluster, so dass wir eine Abstraktion beschränkter Größe für den gesamten Graphen erhalten. Wir führen dann die Regelanwendung auf dieser Abstraktion durch, wodurch wir letzlich eine Überapproximation aller erreichbaren Graphen erhalten.

Wir betrachten ASTRA, eine Implementierung der Cluster-Abstraktion, und als Hauptbeispiel das Merge-Protokoll aus dem Bereich automatisierter Kolonnenfahrten. Bei diesem Protokoll werden Kolonnen durch autonom fahrende Autos gebildet und verschmolzen, so dass das Führungsfahrzeug Geschwindigkeit und Spur kontrolliert. Die Abstraktion analysiert das gesamte Merge-Protokoll, mehrere weitere Standardfallbeispiele aus der Literatur, und auch Fallbeispiele, die aus einem Formalismus höherer Ordnung automatisch generiert wurden.

We have met the enemy and he is us. —Pogo

Looking back on it, I'd lived in an academic dreamland. —*Clifford Stoll,* Cuckoo's egg

The Tao of Programming flows far away and returns on the wind of morning.
—*The Tao of Programming*

Acknowledgements

First and foremost, I thank my advisor, Reinhard Wilhelm. I deeply appreciate the freedom and support he gave me to pursue my research, but also his patience with me when I was occasionally roaming in distant academic disciplines, trying to catch extravagant philosophic ideas. It is a fair judgement to say that his chair truly is academic dreamland. I really wish I had decided earlier to join and could have stayed forever.

Our secretaries Ilina Bach, Rosy Fassbender, Stefanie Haupert-Betz, Sandra Neumann and Carmen Rösch have been a great help to me by taking care of travel arrangements and administrative bureaucracy. I am indebted to Conny Clausen from the Patent Marketing Agency for managing copyright clearance with Saarland University, allowing me to release my implementation under a Free Software license.

My closest collaborator and de facto co-advisor has been Jan Reineke. The many discussions with him were enormously fruitful, productive and enjoyable. My sincerest thanks, Jan! Dmytro Puzhay assisted with tool implementation work and Jörg Kreiker (Bauer) provided his test cases.

I very much enjoyed working and sometimes relaxing with my colleagues Mohamed Abdel Maksoud, Sebastian Altmeyer, Tomasz Dudziak, Nico Fritz, Daniel Grund, Jörg Herter, Philipp Lucas, Claire Maiza (Burguière), Fritz Müller, Oleg Parshin, Markus Pister, Alejandro Salinger, Marc Schlickling, Björn Wachter and everyone else I have forgotten.

I thank the members of my thesis committee not mentioned yet, namely Bernd Finkbeiner for serving as the chairman, Barbara König for acting as referee of my thesis and Tobias Mömke for being the scientific assistant.

I am deeply indebted to family and friends for their support. In particular, if it had not been for Daniela, this thesis would never have been finished.

This work was partially supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `http://www.avacs.org/` for more information.

Contents

1 Introduction

This thesis presents a method for the analysis of formal description mechanisms that describe the evolution of, in principle, unbounded, linked or networked structures. This includes structures inside a computer, like heaps, as well as structures connecting a set of computers, like communication topologies; all evolution mechanisms that can be modeled as graph transformation systems are potential targets of the method. We use abstract interpretation to analyze such graph transformation systems. The method has been implemented in the ASTRA tool. Special focus is given to the car platooning case study as a running example.

Graph grammars. Graph grammars are a powerful and general approach with a wide range of applications that includes compilers, distributed systems, databases, object oriented development and even computational biology. Graph grammars can be used for any domain where entities and relationships are created and destroyed according to rules.

Practical use of graph grammars works roughly as follows: The user specifies a start graph and a set of transformation rules. The start graph describes, using nodes for the entities under consideration and edges for the relationships among those entities, an initial state, the initial working graph. The rules then match a subgraph of the working graph and replace it by a different subgraph, as specified by the *left hand side* and the *right hand side*, respectively, of the rule. This yields a new working graph. If several matches are equally possible, one is picked by nondeterministic choice. This happens until no further matches are possible. Thus, graphs are used to model a state and transformation rules are used to model its evolution over time.

This thesis is concerned with the analysis of graph transformation systems, in particular with the analysis of the reachable graphs. This is of interest because a system might exhibit undesired properties in some of the reachable graphs. To find such errors in the system, it is not possible to explore all reachable graphs, since the system may have a very large or infinite number of them. To cope with this, we will employ abstract interpretation techniques; we will lift rule application to an abstraction that captures all possible concrete behaviors of the system under consideration. The specific abstraction that we employ is *cluster abstraction*, based on the idea of decomposing the concrete graph into overlapping clusters, one for each of its nodes.

Running example: Car platooning. Over the past couple of years, cars have been shipped with ever increasing capabilities to assist the driver on the highway, from lane departure warning systems to heading control. We are now on the verge of cars that drive completely autonomously and it is only a small step towards autonomous car platooning.

This means that autonomously driving cars search their vicinity for each other by wireless technology. The car in front becomes the leader, the others become its followers,

Car platooning increases fuel efficiency and reduces congestion on highways by maintaining short constant distances between consecutive cars. To this end, the leader car sends instructions on changing speed, distance and lane to its followers. Since cars have different destinations, they may enter and exit the highway at different points and thus, the set of cars belonging to a platoon is dynamically changing. Hence, protocols are needed to establish and transform logical communication topologies for the cars to maintain the physical relationships of a platoon.

One important protocol, which this thesis uses as a running example, is the merge protocol. This protocol is executed by all cars concurrently. Its purpose is to establish the leader and follower relationships of the platoon; the leader needs to know its followers and the followers need to know their leader for the coordination to work properly. Further, the protocol allows two such platoons to merge into a single one, with one single leader car that has the other cars as followers. The main result of this thesis is the first analysis of the merge protocol that does not require it to be fundamentally simplified. In particular, we want to ensure the absence of inconsistent states of the system topology, like two followers assuming each other to be the leader.

We model the merge protocol as an infinite-state graph transformation system which we then analyze using abstract interpretation. We will show that our abstraction method is not restricted to the merge protocol, but general enough to also cope with several other standard test cases, like AVL and red-black trees, dining philosophers, firewalls, resource sharing, singly-linked lists, circular buffers and Euler walks.

Outlook. Chapter 2, originally published as [BR10b], gives a detailed introduction into the merge protocol problem setting. Chapter 3, originally published as [BR10a], introduces a rather simple abstraction, *star abstraction*, and shows that it is strong enough to analyze a significant part of the protocol, but, as a negative result, not strong enough to analyze the protocol in its entirety. Chapter 4, originally published as [BR15a] describes the more precise abstraction method *cluster abstraction*, which is based on star abstraction by keeping the size of its abstract domain the same, but making the domain elements more precise by annotating each "star" with additional constraints, thus preserving additional information. We use the name *cluster* for such more precise stars. The chapter also provides a basic evaluation of the abstraction, done with the tool implementation ASTRA. Finally, Chapter 5, originally published as [BR15b], describes use of ASTRA more in detail, and demonstrates the robustness of the approach. The original specification of the merge protocol was done in the high-level formalism DCS; we show that not only a manual, but also an automatic translation of several variants of the merge protocol, from a DCS representation into a graph transformation system, can be analyzed just fine. Chapter 6 contains an extensive review of related work. Chapter 7 wraps up the thesis with a summary and discussion of future work.

2 A Graph Transformation Case Study for the Topology Analysis of Dynamic Communication Systems

This chapter was initially published as [BR10b].

Abstract: We propose a case study for the Transformation Tool Contest 2010 that concerns dynamic communication systems (DCS). DCS are systems of autonomous processes that interact to achieve their goals. For this purpose, the processes exchange messages with each other. In contrast to distributed algorithms, the number of processes of the system is unbounded. The specific dynamic communication systems we want to investigate are so-called platoons. Platoons are groups of cars that drive on a highway with constant speed and constant distance to conserve energy. To form such platoons, each car follows the so-called merge protocol, which guides its local behaviour. We are interested in properties of the communication topologies that may emerge in this platoon scenario. Hence, we ask you to analyze a graph transformation system that generates the possible topologies of the merge protocol. The goal is to do this with as many processes as possible. The case study aims to improve understanding of how useful existing tools are for state space exploration and topology analysis.

2.1 Introduction

2.1.1 Context of the case

Dynamic communication systems are systems that have an unbounded and dynamically changing number of processes. Those processes communicate with each other in order to establish and transform communication topologies (see [BW07] for a more detailed description). In this case study, we want you to compute the topologies that may occur for the merge protocol, a communication protocol which is used in car platooning. Car platooning [HESV91] concerns cars that drive on a highway with constant speed and constant distance from each other, to conserve energy.

The cars are equipped with wireless technology that allows them to communicate via messages. These messages are used to coordinate actions of the platoon, such as have new cars join the platoon or have the platoon change the lane. For this to work, one car per platoon acts as a leader of the platoon and the other cars—the followers—receive command messages from the leader so that the platoon as a whole acts in the desired manner. Accordingly, the platoon leader has to remember all its followers and

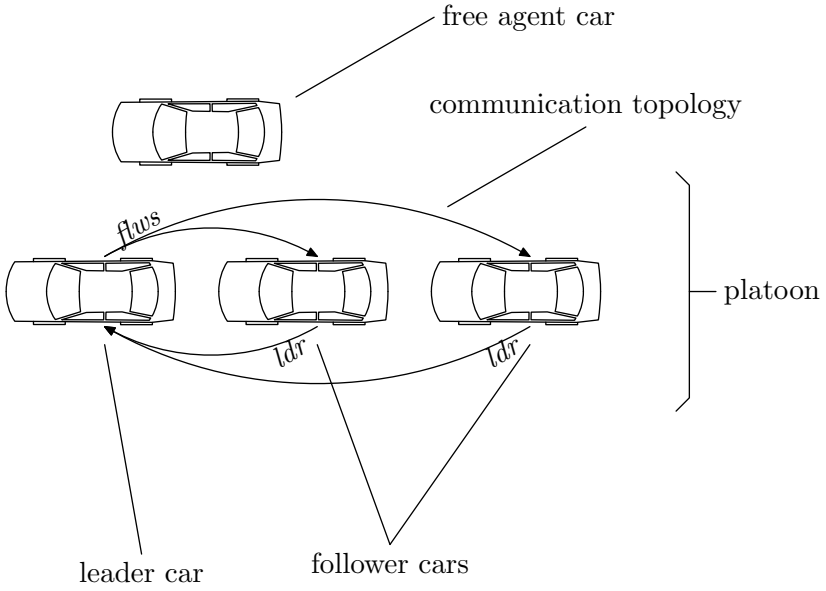


Figure 2.1: Car platooning

each follower has to remember the leader. We call these relations among the cars the communication topology of the platoon.

The merge protocol describes how the cars use communication to join existing platoons and how two platoons manage to merge, so that only one platoon with one platoon leader is left once the merge has finished.

2.1.2 Purpose from a larger perspective

Our case study is supposed to shed light on how useful existing graph transformation tools are for network protocol and other concurrent systems analysis. We think that it has two major benefits:

(a) It shows us how well existing graph transformation can do reachability analysis of network protocols, and to what number of processes they scale. Reachability analysis explores the state space of a network protocol and checks each state that is reachable for undesired properties; or, put differently, it searches for bugs in the protocol. While more sophisticated tools are available for reachability analysis, they often require specialist knowledge about the inner workings of the respective algorithm. Graph transformation, on the other hand, is an intuitive and general approach that can be understood and used easily.

(b) Reachability analysis of unbounded systems can only be partial, since it computes only a finite subset of a finite set of states. We still hope that results from such an analysis provide a good heuristic for the construction and evaluation of abstraction techniques for network protocol topology analysis like [BRKB07]. Such analyses

serve two purposes: To directly verify safety properties—for example that two followers never assume each other to be their leader—and to provide invariants of the protocol that improve precision and efficiency of related analyses like [Tob08].

2.1.3 Challenges that are involved

As the state space of the entire system grows rapidly with the number of processes, it is your goal to show that your analysis can scale well in that respect. This means that runtime and memory consumption should be kept as low as possible. How many processes can your tool handle? We suspect that it is possible only for a small single-digit number of processes.

The graph transformation system that implements the protocol to be analyzed uses rules with simple left hand sides (at most three nodes). During the state space exploration, many similar graphs arise that are matched by the same rules. Can your tool handle such cases efficiently?

2.2 The subject to be modeled

In this section, we describe the background of dynamic communication systems and the intuition behind the merge protocol. It is not necessary to understand all the details for the challenge—we will provide you with a set of graph transformation rules modeling the merge protocol.

2.2.1 Dynamic communication systems

Dynamic communication systems [BSTW06] consist of a finite but arbitrary number of processes that are in one of a finite set of states. Each process has a separate FIFO queue of unbounded length for messages from each of the other processes. Each message can optionally carry the identity of another process as a parameter, so processes can refer to other processes in their communication. Further, each process has a special queue for environment messages. Environment messages may be sent unconditionally by the environment, that is, they may be added to the queue at any time. They are necessary because they are the only way for disconnected parts of the system to get known to each other. In car platooning, for example, a sensor that is built in each of the cars might notice that another car is in its communication range. As we abstract from the physical locations of cars, we model such sensors by the nondeterministic environment. Finally, each process locally maintains a finite set of channels. Each such channel holds a subset of the process identities of the entire system. Channels are a logical construct, not a physical one; they are rather like local address tables (if you assume that the cars use IPv6 to communicate and their identities are IP addresses), not like global wave frequencies shared by all processes; i.e., Channels : $Id \to 2^{Id}$, not Channels $\subseteq 2^{Id}$. So two different processes may store different identities in the channels of the same name at the same time. A process communicates with other processes by sending a message

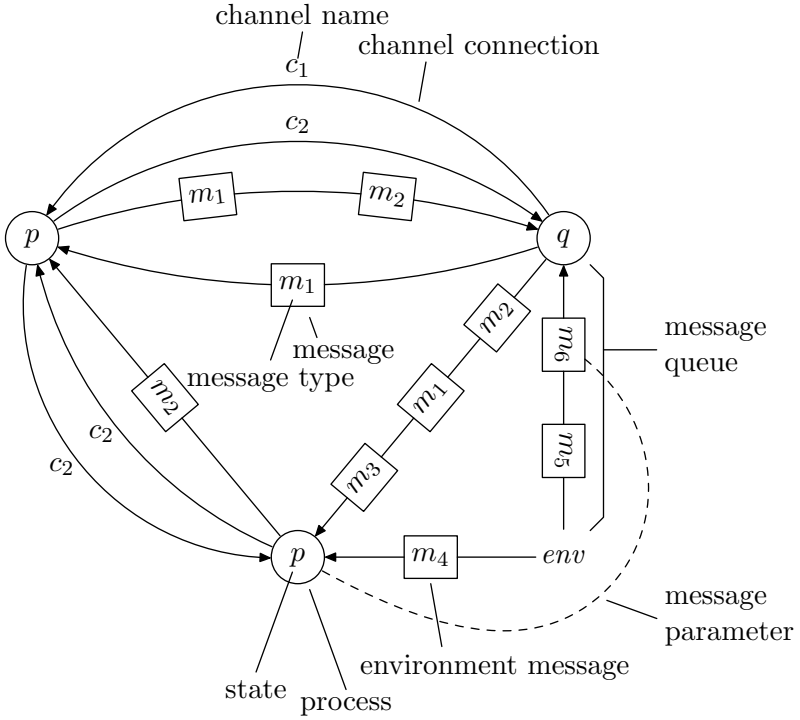


Figure 2.2: Concepts of dynamic communication systems

to all processes in one of its channels. From a global view, the channels make up the communication topology.

A protocol specifies the behavior of a process of a dynamic communication system. It consists of the states that the processes can be in and of the transitions among these states. Transitions are annotated with statements:

- $?(m, c, op)$ and $?(m)$ are guard statements and cause the respective transition to be executed only if a message of type m is at the front of one of the queues. For the first version, the parameter of the message is to be combined with channel c by the set operation op. In both cases, the message that has been received is consumed from the respective queue. For example, $?(ca, ldr, =)$ consumes a ca ("car ahead") message from the queue, clears all process identities from its channel ldr ("leader") and stores the identity that was attached to the consumed ca message into that channel.

- $c = \emptyset$ is a guard, too, and allows a transition to be taken only if the channel c is empty.

- $!(m, c_1, c_2)$ sends a message of type m to all processes on channel c_1 and attaches the identity of one randomly chosen process on c_2 as a parameter. In case of c_2 being the special channel id ("identity"), the process attaches its own identity. For example, $!(req, ldr, id)$ sends a req message to the process(es) in channel ldr (the

processes using the merge protocol happen to never store more than one identity in their *ldr* channel) and attaches the identity of the sending process itself attached as a parameter. $!(newf, ldr, flws)$ picks one of the identities from channel *flws* ("followers"), attaches it to a *newf* ("new follower") message and sends it to the process(es) from channel *ldr*.

- (c_1, op, c_2) combines c_1 and c_2 with the set operation op and saves the result in c_1 again. For example, $(bldr, \setminus, bldr)$ removes all identities from the channel *bldr* ("back leader").

A subset of the states, the initial states, specify in which states a new process may come into existence (initially with empty queues and channels).

2.2.2 The merge protocol

The merge protocol implements three tasks: Building a platoon out of two processes so that one of them becomes a leader and the other a follower; having a process join an existing platoon; and merging two platoons into one.

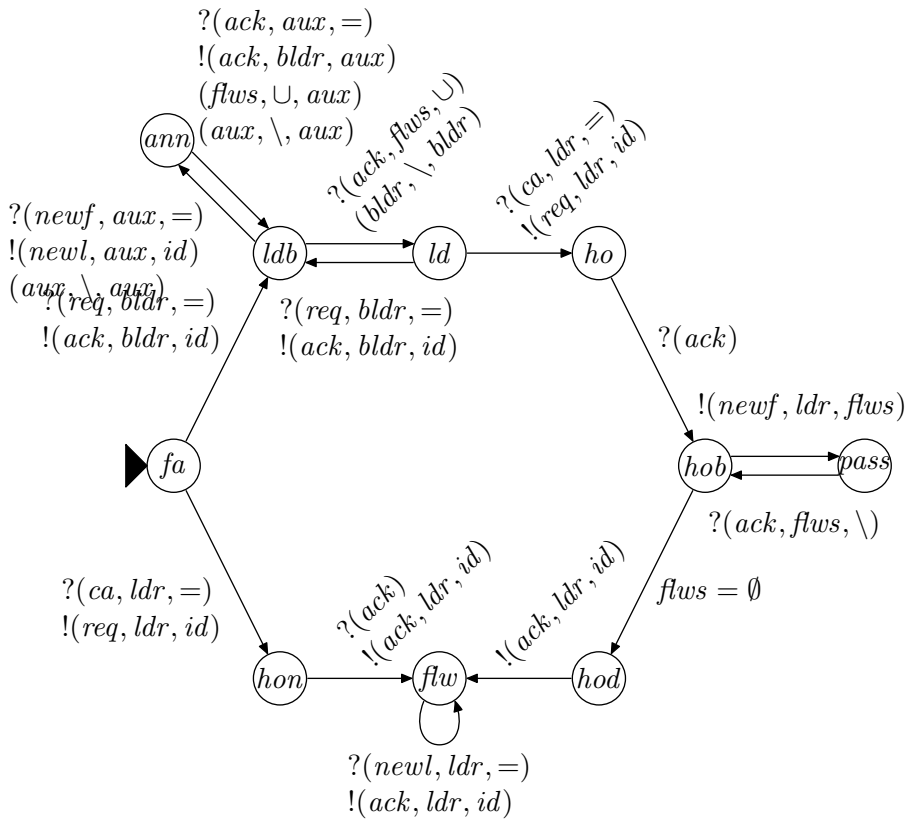
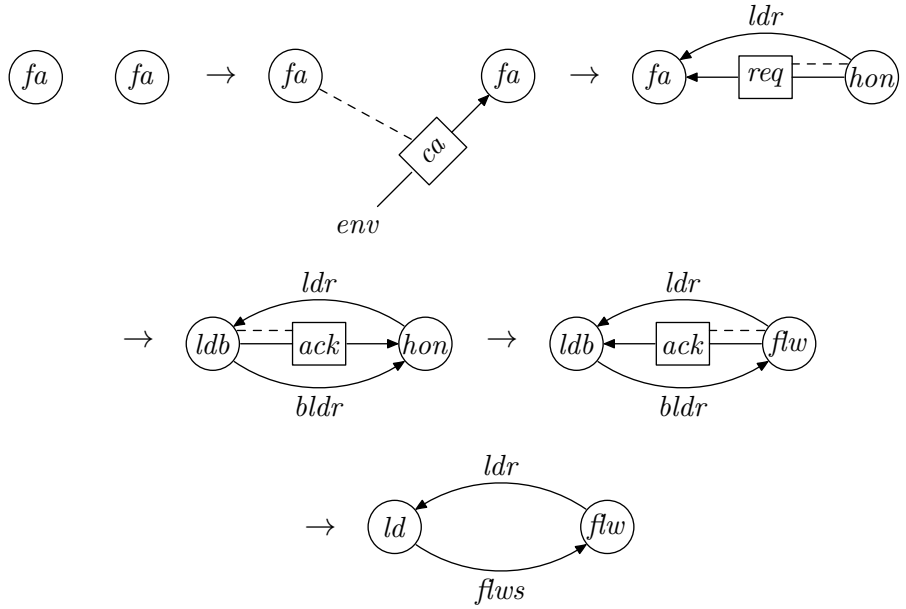


Figure 2.3: Transition system of the merge protocol

The merge protocol specifies that each process starts in the only initial state *fa* ("free agent") (denoted in 2.3 by the triangle). As can be seen in Figure 2.3, from this state, a

process may essentially take two different routes: The upper one, to become a leader, or the lower one, to become a follower. The decision between these two options is made based on whether the process receives the environment message *ca* ("car ahead") with another process as a parameter, or whether it is the other way around and it is attached as a parameter to a car ahead message received by a different process. Let us assume the former happens. That means that it takes the lower route: It stores the attached process identity in its *ldr* ("leader") channel, sends back a *req* ("request") message to that process with its own identity as parameter and changes its state to *hon* ("hand over nothing"). The process that receives the request message will then take the upper route: It will receive the message in state *fa* and so add the identity of the process that sent the message, which it knows from the parameter of the *req* message, to its *flws* ("followers") channel and send back an *ack* ("acknowledgment") message with its own process identity. It then changes to state *ldb* ("leader"). As soon as the other process, assuming it is still in state *hon*, receives the message, it changes to state *flw* ("follower") and returns another acknowledgment, which causes the other process to switch to state *ld* ("leader"). Once that has been done, the two cars have formed a platoon: The car that initially received the car ahead message has become a follower and the car that was attached as a parameter to that message has become the platoon leader. Here is a graphical representation of this evolution of the platoon:

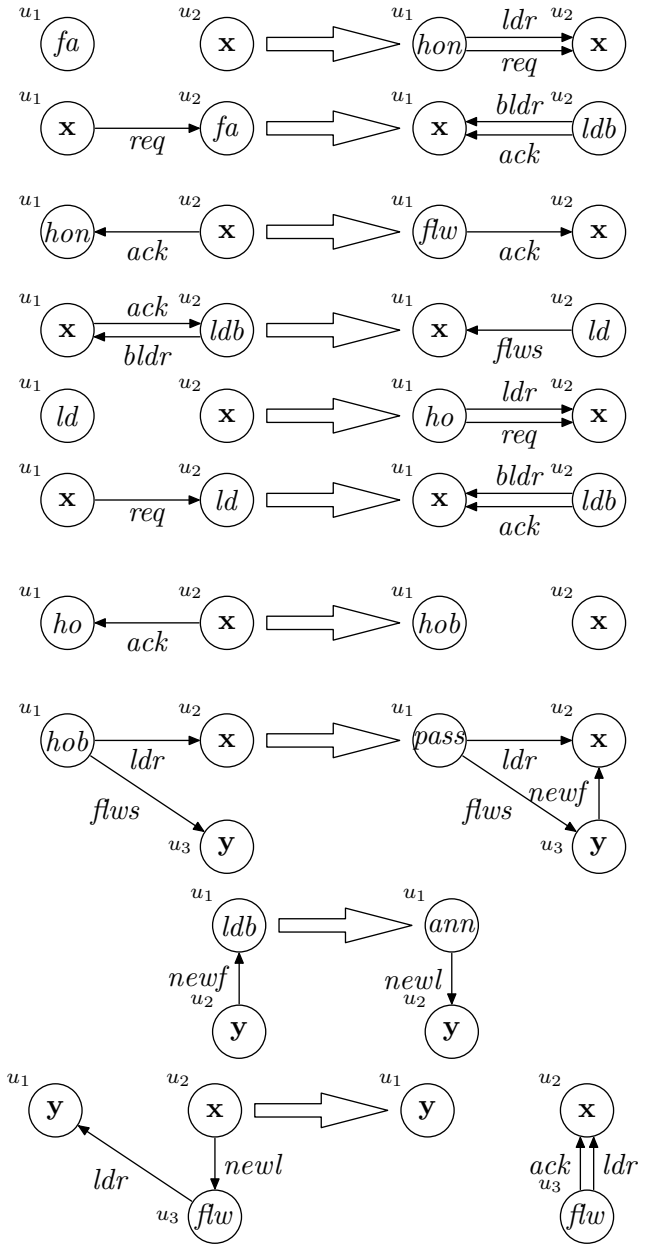


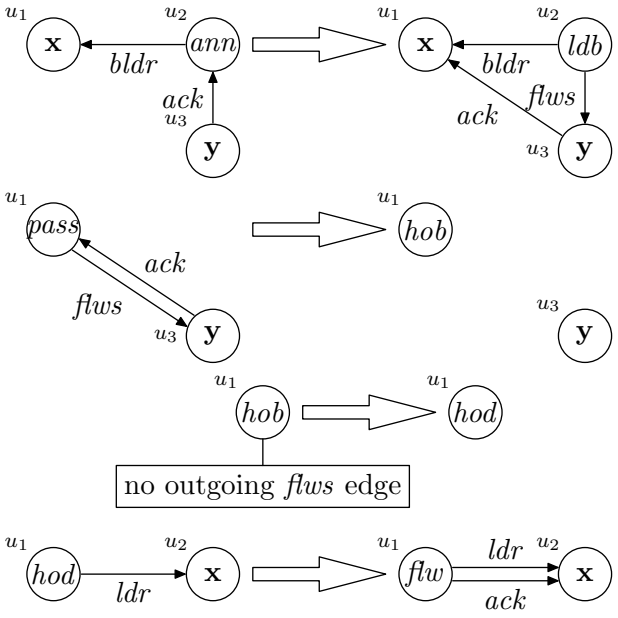
The second task, having a car join an already existing platoon, is mostly similar to the first one, except that the process attached to the car ahead message as a parameter is initially in state *ld* instead of *fa*. For merging (the third task), we have both the recipient of the car ahead message and the parameter process in state *ld*. Both enter a transmission loop to hand over the followers from the one leader to the other: The new platoon leader—the front leader—repeatedly switches between *ldb* (leader, expecting follower identities) and *ann* (waiting for acknowledgment after announcing itself as a new leader to a follower), the back leader repeatedly switches between *hob* (during handover,

transmitting a follower identity) and *pass* (waiting for acknowledgment after passing a follower to the new leader), after having been temporarily in state in state *ho* (start of handover). Finally, the back leader goes to *hod* ("handover done") before becoming a follower itself. The auxiliary channel *aux* allows processes to temporarily store identities during transitions.

2.3 Implementation remarks

We provide a set of graph transformation rules (single pushout):





These rules model the merge protocol in the following way:

- The configurations of the dynamic communication systems are modeled as graphs with node and edge labels
  - The node labels represent the state of the processes.
  - The edge labels represent the channels that make up the communication topology.
- The state transitions are modeled as graph transformation rules.
- Queues are represented as edges, and message types by edge labels (that are different from the labels used for the communication topology). Note that this is a simplification of the original system, as we disregard the order and exact number of messages in queues.
- **x** and **y** are variables. That mean that the corresponding nodes of the rule's left hand side should match nodes with arbitrary labels. All left hand side nodes with such variables in the rules above have corresponding right hand side nodes with exactly the same variable, which means that the rule application should not change the label of these nodes.
- The second to last rule, which implements the $\mathit{flws} = \emptyset$ transition, uses a negative application condition. The rule should only be applied to nodes with the label *hob* that do not have any outgoing edge with the label *flws*.

The rules use the following tricks:

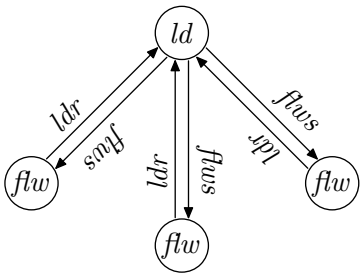
- We do not model environment messages. Because they can be sent at any time, they can also be received at any time.

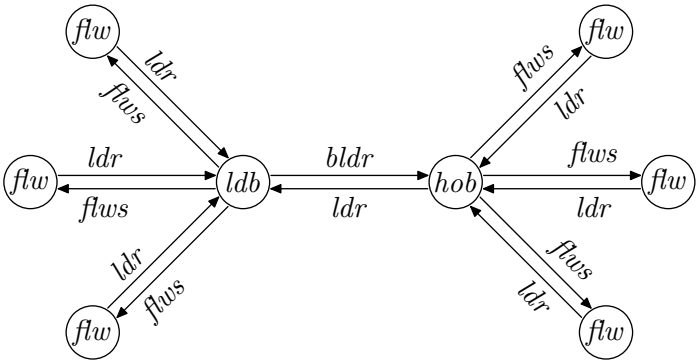
- We do as many things as possible in one graph transformation rule.

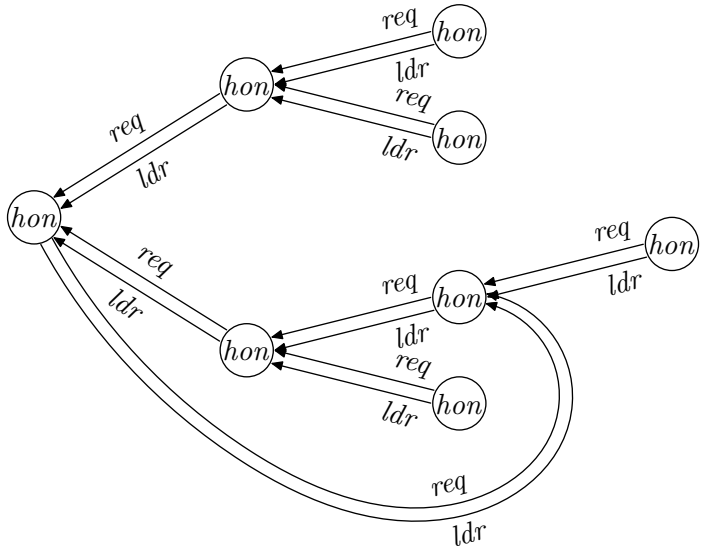
In short, your tool should first read the transformation rules and the start graph. The start graph should consist of nodes that represent the processes, all in their initial state, i.e.*fa*, and with no edges. Then, the tool should compute by fixpoint iteration all reachable graphs of the graph transformation system. In each iteration, it has to find all matches of rules to any of the new graphs, apply the corresponding rule and add the result as a new graph (but only if no isomorphic graph has been computed before).

2.4 Example topologies

Here are some topologies that will emerge from the graph transformation system rules described above that model the merge protocol (assuming sufficiently many processes):







2.5 Goals

2.5.1 Core characteristics

- Output the topologies your tool computed. You may choose the output format freely.

- Feel free to cut down the problem to a reasonable size and ignore everything that you consider as an obstacle, even if you only analyze a small part of the protocol. The description of the DCS protocol in Section 2.2 should help you to adjust the graph transformation rules to your needs.

2.5.2 How the model should be used

- The graphs from the result are used for evaluating structural predicates on them, like "is there a node with label a and a node with label b such that an edge with label c points from the one to the other." Here is a list of properties that should be satisfied by the merge protocol:
  - No two nodes labeled *flw* are connected to each other with an edge labeled *ldr*.
  - A node labeled *pass* or *ld* always has at least one node labeled *flw* connected via some edge labeled *flws*.
  - If a node labeled *flw* has outgoing edges labeled *ldr* and *newf*, respectively, to two different nodes, then those two nodes are connected via an edge labeled *bldr*.

  How easy is it to evaluate such properties in your framework?

- The result should also be used for graphically displaying and exploring the topology structure of the platoons admitted by the protocol. Is it easy for the user to filter the displayed topology to eg. not include edges corresponding to messages?

2.5.3 Extensions

- We are also interested in a transition metagraph that models the evolution of the graph transformation system. It should have the graphs resulting from the analysis as nodes. Edges should be labeled with the respective rules that caused the transformation. This allows the inspection of traces.
- The graph transformation system we provide does not accurately reflect the DCS protocol with respect to message queues. Are you able to perform a queue analysis, either as part of the system analysis, or in a separate step, using graph transformation?
- Can you analyze the protocol in a general way using abstraction techniques such that the number of processes is not limited?

2.5.4 Evaluation criteria

We propose the following evaluation criteria:

- Completeness of the used transformation system: Less, same, more precise than reference transformation system? (more is better)
- Completeness of analysis: Systems with how many processes ($2 \dots \infty$) were you able to analyze? (more is better)
- Performance: What is the memory consumption and runtime of the analysis for the largest analyzable system? (less is better)
- Flexibility of output: Do you merely allow output of topologies as they are, or do you allow filtering of edges/nodes according to labels, or even according to more complex filter specifications? (more is better)
- Flexibility of property evaluation: How powerful is your check for desired properties of topologies? Merely subgraph matching? More complex expressions over graphs with node and edge labels? (more is better)

3 Abstract topology analysis of the join phase of the merge protocol

This chapter was initially published as [BR10a].

Abstract: We present a partial solution to the TTC2010 topology analysis case study. We pick a small part of the merge protocol, namely the part where cars join a leader to form a platoon. Using abstract interpretation, we compute an approximation of the arising topologies, without limiting the number of cars.

3.1 Introduction

In our case study, we ask "Can you analyze the protocol in a general way using abstraction techniques such that the number of processes is not limited?" In this solution to the case study, we achieve this goal for a part of the protocol. Section 2.2 of the case study mentions that the protocol implements three tasks. The part that we analyze consists of the first and the second task: Building a platoon out of two processes so that one of them becomes a leader and the other a follower, and having a process join an existing platoon.

If we apply the merge protocol graph transformation rules from the case study to a start graph with finitely many nodes, then they produce a finite set of graphs. This is so because none of the rules adds new nodes. If, on the other hand, we use an empty start graph, and add a new rule that merely creates free agent nodes, the result will be an infinite set of graphs. Accordingly, such a system cannot be analyzed using classic graph transformation tools.

Abstract interpretation allows us to compute approximations of such systems. The state of the art technique using this paradigm is partner abstraction [Bau06, BW07], implemented in the tool `hiralysis`. However, partner abstraction was designed for simple topologies only and requires a human expert to supply additional invariants—partner constraints—to cut off parts of the merge protocol that involve more complicated topologies. Such more complicated topologies do occur in the merge protocol [Bac08]. Without cutting off these parts, partner abstraction will suffer from state space explosion, and `hiralysis` will not terminate.

The more complicated topologies cut off for partner abstraction analysis already arise for the two mentioned tasks. We aim at an abstraction that does not need manual intervention to cope with these topologies.

In Section 2, we introduce our abstraction. Section 3 presents which parts of the protocol we analyze and the abstract result that we get after running `astra`, our analysis

tool. We talk about the evaluation of properties on the abstract result in Section 4. Section 5 sums up our work and presents future research.

3.2 Star abstraction

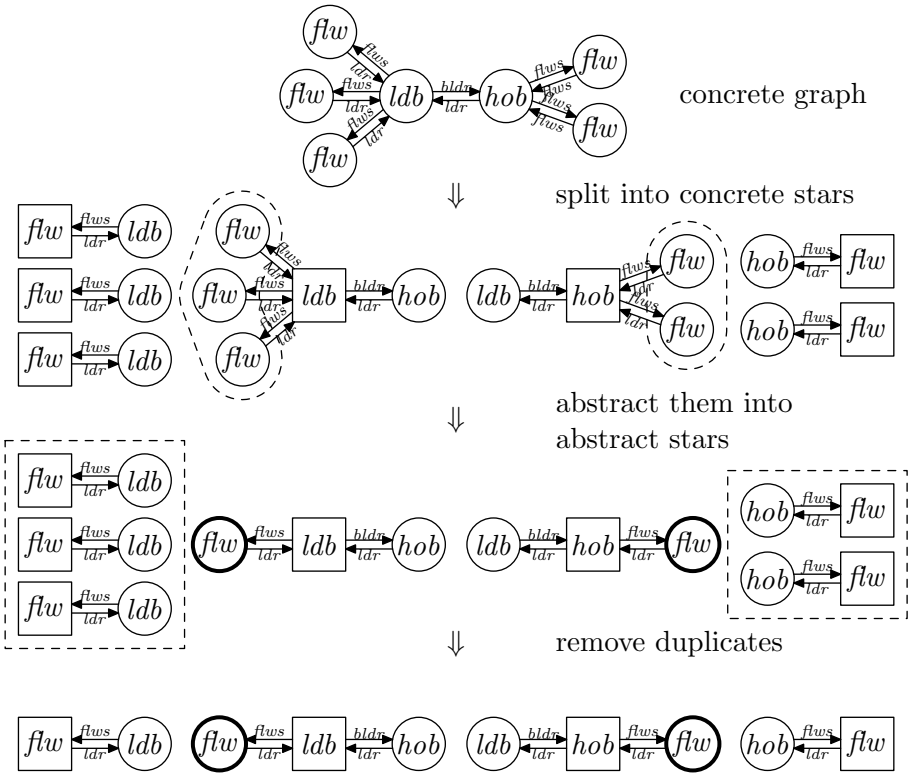


Figure 3.1: Our abstraction applied step by step to a simple example

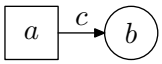
Our abstraction is sketched in Figure 3.1. We abstract graphs in three steps: First, we build the corresponding *star* for all nodes v of the graph. We obtain the star by removing all nodes from the graph except for v and its partner nodes, and by removing all the edges that are not incident to v. We call v the *core node* (displayed as a square in Figure 3.1) and the other nodes the *outer nodes*.

The next step is done for each star separately. We identify sets of outer nodes that cannot be distinguished from each other with respect to their label and the labels and directions of the edges incident to them. For each such set that contains two or more indistinguishable nodes, we merge them all into a summary node. We are then left with *abstract stars*. We represent the abstract stars as a tuple (l, E, A, S) with $l \in \mathcal{N}$ being the label of the core node, $E \subseteq \mathcal{E}$ being the self-loops of the core node, $A \subseteq \mathcal{N} \times 2^{\mathcal{E}} \times 2^{\mathcal{E}}$ being the *axes* and $S \subseteq A$ specifying which of those axes have summary nodes. Each axis (l, in, out) represents an outer node with label l and the edges incident to it. in contains those edges that point from the core node to the outer node and out the remaining ones. By construction, it follows that at least one connection must exist, that is, $in \cup out \neq \emptyset$.

In the final step, we ensure that each abstract star is unique. This is accomplished by keeping at most one copy from each class of isomorphic abstract stars.

There are $|\mathcal{N}| \cdot 2^{|\mathcal{E}|} \cdot 3^{|\mathcal{N}| \cdot (2^{2 \cdot |\mathcal{E}|} - 1)}$ different stars: Each star can have $|\mathcal{N}|$ different node labels for the core node and can have any subset of the $|\mathcal{E}|$ edge labels for self-loops. Between the core node and each outer node, there can be any edge with one of the $|\mathcal{E}|$ edge labels, either pointing from the core node to the outer node, or the other way around; with the exception that in total, at least one edge must be present. The outer node of an axis can have any of the $|\mathcal{N}|$ node labels, and each axis is either present, absent or present as a summary axis. We denote the set of all possible stars over a node label set $\mathcal{N}$ and an edge label set $\mathcal{E}$ as $\mathfrak{S}(\mathcal{N}, \mathcal{E})$.

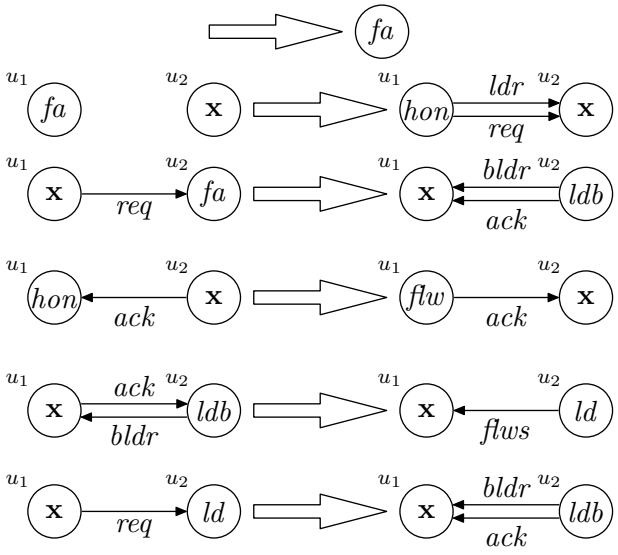
We do not yet have a result on the exact size of the star domain itself and merely know that it is bounded by $2^{|\mathfrak{S}(\mathcal{N},\mathcal{E})|}$. It is non-trivial, because not every subset of the stars is a valid abstraction of an actually existing concrete graph. For example, a set containing only the star



is not a abstraction of any graph, since there is no corresponding star with a core node that has label b.

3.3 Results

Our tool, `astra`, expects a file in `hiralysis` format as input. The file contains a set of graph transformation rules and a start graph. We use the rules from the case study, except for the ones that deal with platoon merging and follower handover:



generate free agents

free agent requesting to join a platoon

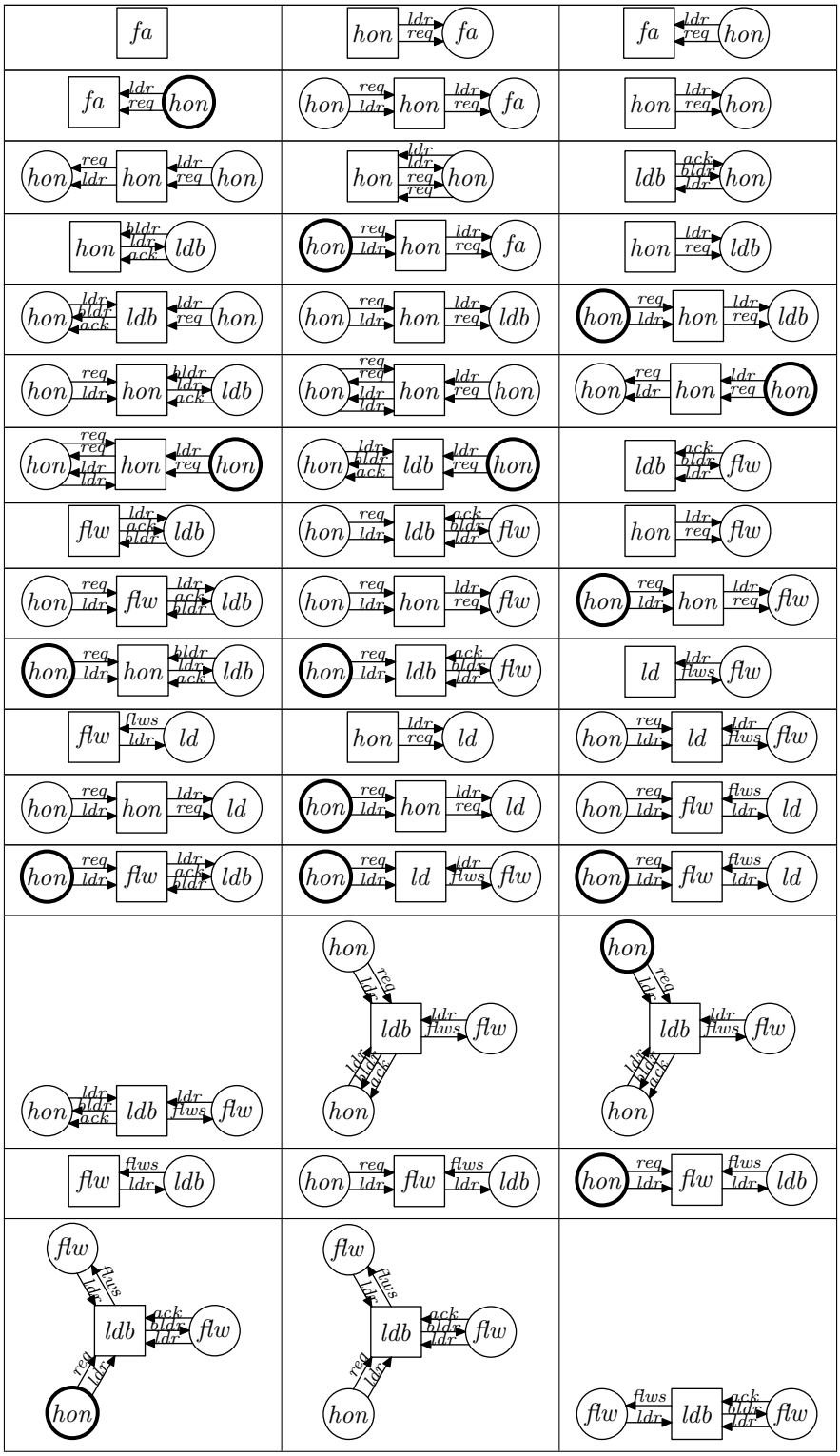
free agent acknowledging a join request

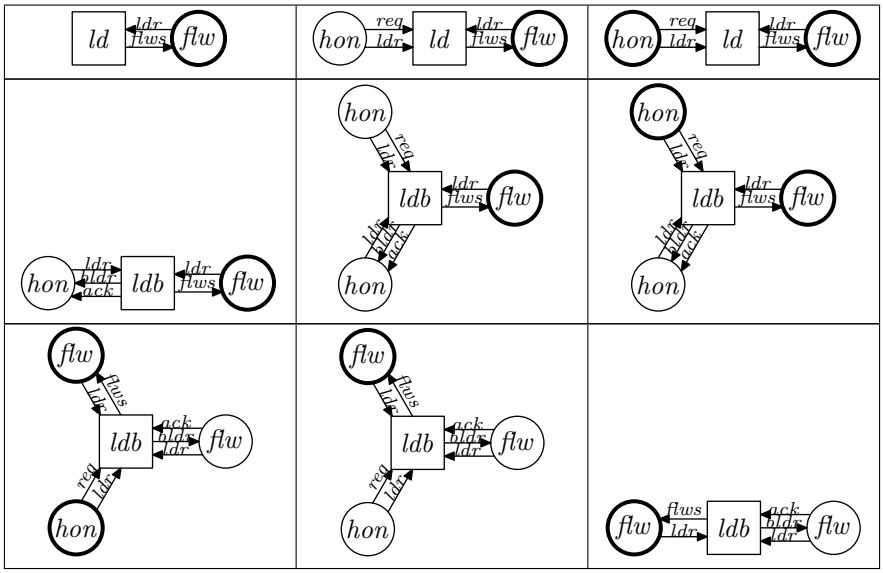
new follower receiving acknowledgment to join the platoon

leader adding new follower after its acknowledgment

leader receiving merge or join request

`astra` computes an abstraction of all graphs that can be generated by the system, resulting in the following set of stars:

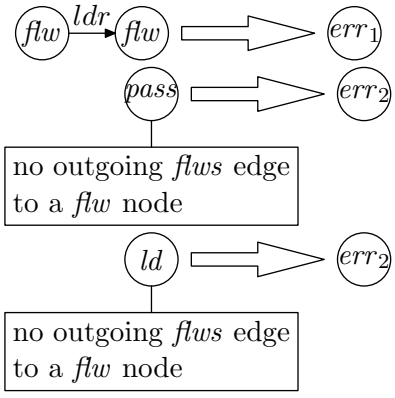




Our tool outputs the result in graphviz, GDL, XGDL, Tulip and METAPOST format and such that it can be rendered/displayed with any tool capable of processing these formats. The graphical representation above is the METAPOST output.

3.4 Property evaluation

Star abstraction easily allows for evaluation of properties involving two nodes and the connections among them, such as the first and second example given in Section 5.2 of the case study. This is because the stars contain each adjacent node of each node and the edges between them. That is the only thing that the first two example properties talk about. Using the additional rules



we mark parts of the graph with error labels where the properties are violated. It is then easy to scan the result for such labels. We can verify any property that can be expressed using such rules.

Our abstraction overapproximates the set of concrete topologies that might arise. Thus, if a property of the mentioned kind (one that can be expressed by a transformation rule adding an error node) is satisfied by all topologies represented by our abstract result, we know that it is satisfied by all reachable concrete topologies as well. However, if is not satisfied in the abstract result, it might still be satisfied for the concrete result.

The computing time and memory consumption of our tool is negligible (< 1 MB, $<$ 1 sec) on any reasonably modern machine.

3.5 Conclusion

Our analysis has proven powerful enough to analyze the join phase of the merge protocol in a general way, without limit to the number of processes. The resulting topologies are more complex than what can be analyzed with existing approaches, since they are not limited with respect to path length.

On the other hand, the abstraction we employed is too weak to deal with the characteristic topology structures occurring during handover. If we try to analyze the full merge protocol, the abstraction runs into state space explosion. This is caused by the inability of the abstraction to preserve the fact that the topology has a triangular shape during handover. Accordingly, the abstraction does not preserve enough information to verify the third example property from the case study.

Since the results are still promising, we are currently implementing a new tool with an extended abstraction that is able to cope with topologies involving triangular shapes.

4 Analysis of infinite-state graph transformation systems by cluster abstraction

This chapter was initially published as [BR15a].

Abstract: Analysis of distributed systems with message passing and dynamic process creation is challenging because of the unboundedness of the emerging communication topologies and hence the infinite state space. We model such systems as graph transformation systems and use abstract interpretation to compute a finite overapproximation of the set of reachable graphs. To this end, we propose cluster abstraction, which decomposes graphs into small overlapping clusters of nodes. Using `astra`, our implementation of cluster abstraction, we are for the first time able to prove several safety properties of the merge protocol. The merge protocol is a coordination mechanism for car platooning where the leader car of one platoon passes its followers to the leader car of another platoon, eventually forming one single merged platoon.

4.1 Introduction

Distributed message-passing systems such as car platoons and drone swarms consist of an unbounded and dynamically changing number of agents. These agents act in a coordinated fashion using wireless ad-hoc networks to achieve common goals. For this purpose, they assume different roles in a logical communication topology that is established on top of the physical communication medium. These communication topologies, which consist of unidirectional channels between pairs of agents, are formed by distributed protocols that all agents execute concurrently.

The purpose of our analysis is to determine the emerging topologies, which can then be used to evaluate safety properties, ensuring that the system will never reach a state with an undesired topology.

We model such systems by graph transformation systems, i.e., graphs modified by transformation rules. Graph transformation is a lingua franca with a broad range of applications in systems modeling, all of which become potential use cases for our method. Many domain-specific models can be translated automatically into graph transformation systems.

In the graph transformation framework, we represent agents as labeled nodes and communication channels and message queues as labeled, directed edges of a graph. We model the dynamics of the system, like agents sending and receiving messages,

detecting each other's presence and setting up and closing communication channels, as transformation rules that are applied to the graphs. Those rules match subgraph patterns in a graph, optionally restricted by application conditions, and replace them by modified subgraphs.

The main challenges with respect to the analysis of the systems under consideration are the unboundedness of the graphs, caused by the unboundedness of the number of agents, and the concurrency of the computations of the participating agents. In particular, the state space of such systems is infinite, and naive state-space exploration cannot be used for our purpose. Instead, we use abstract interpretation, overapproximating the graphs by abstract representations of bounded size.

To compute this overapproximation, we lift rule application to the abstract level, reducing the infinite concrete state space to a finite abstract one: We match the rules on the abstract representation, partly undo the abstraction, just enough to apply the rule, and restore abstraction on the result. By fixed-point iteration, we end up with one final abstract topology, an overapproximation of all graphs the system may produce.

The crucial idea of our abstraction is to decompose graphs into overlapping, simultaneously evolving *clusters*, one per node of the graph—*cluster abstraction*. Each cluster consists of a *core node*, corresponding to the specific node under consideration, and *peripheral nodes*, corresponding to the immediate neighborhood of the core node, i.e., its adjacent nodes. We keep the edges between peripheral nodes and the core node, as well as the core node itself, completely concrete. The neighborhood of a node may be unbounded, e.g., in some protocols a leader may have an unbounded number of followers. To arrive at a finite abstract domain, we use approximated counting: two or more neighborhood nodes that are similar become one summary node in the periphery. By a three-valued abstraction, we preserve information about the neighborhood edges where possible.

We have implemented cluster abstraction in a tool called `astra`. In addition to benchmarks from the literature, ranging from red-black trees to firewalls, we successfully apply `astra` to the merge protocol. The merge protocol is a coordination mechanism for car platooning that could not be fully analyzed with previous approaches.

Outline. In Section 2, we describe the graph transformation framework our work is based upon. Section 3 introduces cluster abstraction and the computation of the corresponding abstract transformer. In Section 4 we present our tool implementation `astra` and experimental results. After discussing related work in Section 5, we conclude the paper in Section 6.

4.2 Background

4.2.1 Graph Preliminaries

Our framework is based on directed graphs with edge and node labels. We allow several edges between the same pair of nodes, but only as long as their direction or edge label differ.

Definition 1 (Graph). *Let $\mathcal{V}$ be a set of node names, $\mathcal{N}$ a set of node labels and $\mathcal{E} = \{\beta_1, \ldots, \beta_{|\mathcal{E}|}\}$ a set of edge labels. A* graph *G is a tuple $(V_G, E_G^{\beta_1}, \ldots, E_G^{\beta_{|\mathcal{E}|}}, \ell_G)$ where $V_G \subseteq \mathcal{V}$ is the set of nodes, $\ell_G : V_G \to \mathcal{N}$ is the node label assignment and $E_G^\beta \subseteq V_G \times V_G$ is the set of edges with label $\beta \in \mathcal{E}$.*

For simplicity, we assume a globally unique set $\mathcal{V}$ of node names, plus a globally unique set of node labels $\mathcal{N}$ and edge labels $\mathcal{E}$. Note the difference between node names and node labels: Nodes may share the same node label and nodes from different graphs may share the same node name, but nodes from the same graph always have different node names. We use mappings over node names to relate nodes of different graphs. We denote the set of graphs as $\mathcal{G}$.

Graph morphisms map the nodes of one graph to the nodes of another graph such that the node labels agree and all edges are preserved. The existence of a graph morphism means that one graph is a subgraph of another.

Definition 2 (Partial and total graph morphism, subgraph relation). *Let G and H be graphs. An injective partial function $h : V_G \rightharpoonup V_H$ is a* partial graph morphism *iff $\ell_G \cap (\mathrm{def}(h) \times \mathcal{N}) = h \circ \ell_H$ and for all $\beta \in \mathcal{E}$, $h(E_G^\beta) \subseteq E_H^\beta$. We call h a* (total) graph morphism *iff it is a total function, i.e., $h : V_G \to V_H$. If an injective graph morphism exists, G is a* subgraph *of H, denoted by $G \lesssim_h H$.*

For the purpose of abstraction, we will later need to consider *spokes* between nodes, not merely individual edges. Spokes represent the configuration of edges, i.e., direction and edge label of edges between two given nodes.

Definition 3 (Spoke). *Let G be a graph and $v, v' \in V_G$. Then the* spoke *between v and v' in G is the pair $SP_G(v, v') := (\{\beta \in \mathcal{E} \mid (v, v') \in E_G^\beta\}, \{\beta \in \mathcal{E} \mid (v', v) \in E_G^\beta\})$ We denote the set of all spokes $2^\mathcal{E} \times 2^\mathcal{E}$ by $\mathcal{SP}$. An alternative notation for the empty spoke $(\emptyset, \emptyset)$ shall be $\emptyset$.*

4.2.2 Graph Transformation Systems

Graph transformation systems rewrite graphs according to transformation rules, starting with some initial graph. Rule application can be restricted via negative application conditions. In this paper, we consider negative application conditions specified by partner constraints. A partner constraint prohibits incident edges with a specific direction and label to an adjacent node with a specific label.

Definition 4 (Partner constraint). *A* partner constraint *is a tuple $(d, \beta, l) \in \mathcal{PC} = \{in, out\} \times \mathcal{E} \times \mathcal{N}$ where d is a direction, β an edge label and l a node label.*

Transformation rules consist of a left hand side graph matched against the host graph, a right hand side graph by which the left hand side graph is replaced, and a mapping that describes node correspondence between the left and the right hand side graph. Additionally, for each left hand side node, an optional set of partner constraints can be specified.

Definition 5 (Graph transformation rule)**.** *A* graph transformation rule *is a tuple* (L, h, p, R) *where* L *(the* left hand side*) and* R *(the* right hand side*) are graphs,* $h : V_L \rightharpoonup V_R$ *is an injective partial mapping from the left to the right hand side and* $p : V_L \rightharpoonup 2^{\mathcal{PC}}$ *specifies the partner constraints.*

For simplicity, in the following, we assume one globally unique set of graph transformation rules $\mathcal{R}$ and an initial graph I, which, together with node and edge labels, form the graph transformation system $\mathcal{S} := (\mathcal{N}, \mathcal{E}, I, \mathcal{R})$. We further assume for simplicity that in each rule, either all or none of its right hand side nodes are newly created.

For a rule to match, its left hand side must be a subgraph of the host graph and all negative application conditions need to be satisfied: We check each partner constraint against the matched node and its neighborhood.

Definition 6 (Match, partner constraint satisfaction)**.** *Let* $r = (L, h, p, R)$ *be a rule,* G *a graph and* $m : V_L \to V_G$*. Then* m *is a* match *from* r *to* G *iff* $L \lesssim_m G$ *such that the partner constraints* p *are satisfied: For each* $v \in \mathrm{def}(p)$ *and* $\beta \in \mathcal{E}$*, we have* $p(v) \cap E = \emptyset$*, where*

$$\begin{aligned} E = \{&(out, \beta, \ell_G(u')) \mid (m(v), u') \in E_G^\beta\} \\ \cup \{&(in, \beta, \ell_G(u')) \mid (u', m(v)) \in E_G^\beta\})\} \end{aligned}$$

Rule application requires that the left hand side matches the host graph. A result graph is the host graph with labels of matched nodes changed as specified by h, nodes and edges of the left hand side removed and nodes and edges of the right hand side added as specified by the rule. Added nodes may be assigned any unused node name, thus the result is not unique. We obtain a mapping from the unchanged nodes of the host graph to the result graph as a byproduct. A graph is directly derived from a host graph according to some rule iff there is any way to apply the rule and obtain this graph as the result.

Definition 7 (Rule application, direct derivation)**.** *Let* $r = (L, h, p, R)$ *be a rule,* G, H *graphs,* $m : V_L \to V_G$ *an injective graph morphism and* $D := m(V_L \setminus \mathrm{def}(h))$ *the set of deleted nodes. Then* H *is a result of the* application *of* r *to* G *with respect to* m*, written* $G \overset{r,m}{\rightsquigarrow} H$*, iff there is an injective mapping* $m' : V_R \setminus h(V_L \setminus \mathrm{def}(m)) \to V_H$ *such that* $m = h \circ m'$*,* $V_H \cap D = \emptyset$ *and*

$$\begin{aligned} \ell_H &= (\ell_G \setminus (D \times \mathcal{N}) \cup (m'^{-1} \circ \ell_R) \\ V_H &= (V_G \setminus D) \cup m'(V_R) \\ E_H^\beta &= ((E_G^\beta \setminus m(E_L^\beta)) \cap (V_H \times V_H)) \cup m'(E_R^\beta) \end{aligned}$$

The direct derivation relation $\overset{r}{\rightsquigarrow}$ *is a relation over* $\mathcal{G} \times \mathcal{G}$ *where* $G \overset{r}{\rightsquigarrow} H$ *iff there is a match* m *such that* $G \overset{r,m}{\rightsquigarrow} H$*.*

In this paper, we are interested in reachability properties, i.e., is a graph with a particular property reachable or not? Therefore, we define the semantics of the graph transformation system simply as the set of reachable graphs.

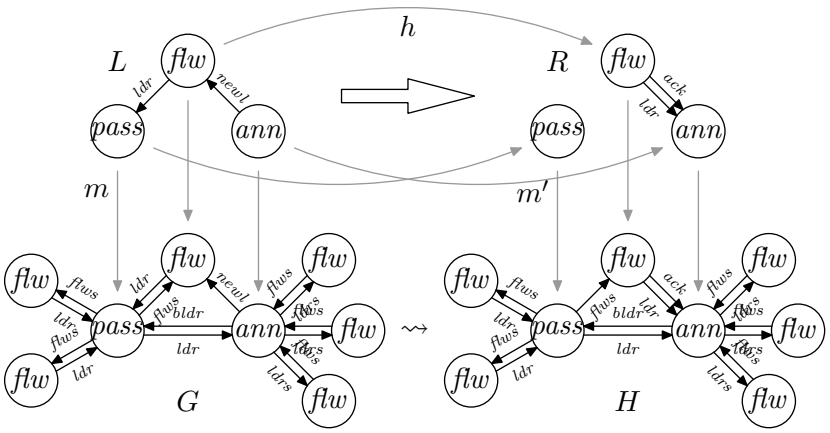


Figure 4.1: An example of rule application: A rule $(L, h, \emptyset, R)$ transforming graph G into graph H, as it occurs in the merge protocol [BR10b].

Definition 8 (Graph transformation system semantics)**.** *The* semantics of a graph transformation system $\mathcal{S}$ *is the smallest set such that* $I \in [\![\mathcal{S}]\!]$ *and, if there are graphs* $G \in [\![\mathcal{S}]\!]$ *and* H *and a rule* $r \in \mathcal{R}$ *such that* $G \overset{r}{\rightsquigarrow} H$*, then* $H \in [\![\mathcal{S}]\!]$*.*

4.2.3 The Merge Protocol

Our main benchmark is a graph transformation system modeling the *merge protocol* [HESV91, BR10b]. This protocol is used in car platooning, where autonomous cars on highways form platoons driving at constant speed and distance to save fuel. Its purpose is to allow (1) two cars to form a platoon with the car in front becoming the platoon leader and the other becoming its follower, (2) a car joining an existing platoon as a new follower and (3) merging of two platoons, with the leader on the back handing over all its followers to the leader in front, eventually itself becoming one of the followers.

What makes the merge protocol so difficult to analyze is the vast range of topological configurations all present and evolving at the same time, caused by the protocol's massively distributed nature. For example, a car may receive at any time a request to form a platoon, at the same time receive a request to merge with another platoon, all while being in the middle of any intermediate step of a merge operation, or sending such a request itself—and this happening with an arbitrary number of cars in different contexts at once. This is different from the typical setting of shape analysis, i.e., the static analysis of heap-manipulating programs, where data structures typically have a regular global structure modified only at some select points, those referenced by pointers from the stack. On the other hand, shape analyses are often employed to prove *global* invariants about the heap structure, such as the sortedness of a binary tree, whereas in the analysis of the merge protocol, our goal is to show that undesired *local* configurations never occur.

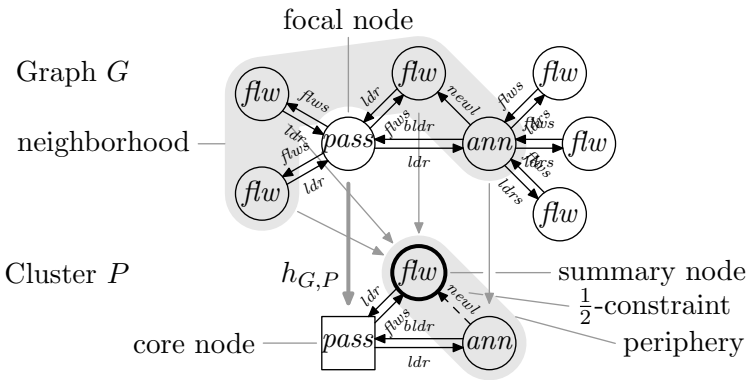


Figure 4.2: An example of local abstraction: Graph G with the *pass*-labeled node as focal node is abstracted into cluster P. The periphery of P is an abstraction of the neighborhood of the focal node in G.

4.3 Analysis

4.3.1 Cluster Abstraction

In the graph transformation systems we consider, unbounded numbers of nodes may be created dynamically. Thus, the state space of such systems is infinite in size, making exact analysis by concrete state-space exploration impossible. To overcome this challenge, we employ a bounded abstraction: each concrete graph of arbitrary size is represented by an abstract graph of bounded size, reducing the infinite state space to a finite one.

We apply *local abstraction* to each node of a given graph, obtaining a bounded set of *clusters*. Local abstraction focusses on one specific node in the graph, henceforth called the *focal node*. It abstracts from all nodes in the graph except for the focal node and its immediate neighborhood, referred to as the *periphery* in the abstraction. The neighborhood consists of the incident edges and the adjacent nodes of the focal node. In addition, neighborhood nodes are merged into summary nodes if they are connected to the focal node by the same spoke (see Definition 3). Further, edges among neighborhood nodes are abstracted into three-valued *constraints*. This yields a *cluster*, which consists of the core node and its periphery. The core node shall have the unique name $\mathit{core} \in \mathcal{V}$. Figure 4.2 illustrates local abstraction, which is formally defined later.

Local abstraction asymmetrically preserves information about one specific node and some information about its neighborhood only, none about the rest of the graph. To capture the structure of the entire graph, we apply local abstraction to all of its nodes. As the neighborhoods of nodes are overlapping, this preserves some information about the global graph structure.

While a concrete graph may contain an arbitrary number of nodes, the set of distinct clusters is bounded. Thus the abstraction is bounded.

The process described above yields a set of clusters that may contain clusters that differ only with respect to constraints between peripheral nodes. To reduce analysis complexity, such clusters are merged by loosening the constraints.

Definition 9 (Cluster)**.** *A* cluster P *is a tuple* $(G_P, S_P, C_P^{\beta_1}, \ldots, C_P^{\beta_{|\mathcal{E}|}})$ *where* $G_P = (V_P, E_P^{\beta_1}, \ldots, E_P^{\beta_{|\mathcal{E}|}}, \ell_P)$ *is a graph,* $\{core\} \subseteq V_P \subseteq \{core\} \cup \mathcal{SP} \times \mathcal{N}$ *are the node names,* $S_P \subseteq D_P = V_P \setminus \{core\}$ *is a set of* summary nodes*, with* D_P *the set of* peripheral nodes*,* $C_P^\beta : ((D_P \times D_P) \setminus \{(v,v) \mid v \in D_P \setminus S_P\}) \to \{0, 1, \frac{1}{2}\}$ *are the* peripheral constraints*,* $E_P^\beta \subseteq (\{core\} \times V_P) \cup (V_P \times \{core\})$ *for any* $\beta \in \mathcal{E}$ *and* $SP_P(core, v) \neq \emptyset$ *for all* $v \in D_P$*. We denote the set of all clusters by* $\mathcal{P}$*.*

Given a graph G and one of its nodes v, local abstraction yields a cluster P with a core node that corresponds to the focal node v. P has one peripheral node per uniquely connected neighborhood node of v, that is, with a unique configuration of neighborhood node label plus non-empty spoke.

The edges connecting the neighborhood nodes are abstracted as follows: If, in G, there are β-labeled edges from all source nodes V_1 to all target nodes V_2, both sets each corresponding to a (possibly summary) node in P, then there is a peripheral 1-constraint in P that involves two nodes corresponding to V_1 and V_2. If there are some, but not all such β-labeled edges, we use a $\frac{1}{2}$-constraint instead. And if there are no such β-labeled edges at all, a 0-constraint. Note that peripheral constraints do not contain information about self-loops of the corresponding concrete nodes.

The byproduct of local abstraction is a mapping $h_{G,P}$. It maps nodes of G to corresponding nodes in P, if any. $h_{G,P}$ is not necessarily injective: If the abstraction contains a summary node, then all corresponding concrete nodes will be mapped to it.

Definition 10 (Local abstraction, induced mapping)**.** *The* local abstraction *of a graph* G *with respect to a* focal node $v \in V_G$*, denoted by* $\alpha(G, v)$*, is the cluster* P *that satisfies the following conditions:*

- $V_P = h_{G,P}(V_G)$
- $E_P^\beta = h_{G,P}(E_G^\beta \cap ((\{v\} \times V_G) \cup (V_G \times \{v\})))$
- $S_P = \{u \in D_P \mid |{h_{G,P}}^{-1}(\{u\})| \geq 2\}$
- $C_P^\beta(u_1, u_2) = \begin{cases} 0 & : \forall v_1 \neq v_2 : (h_{G,P}(v_1), h_{G,P}(v_2)) = (u_1, u_2) \Rightarrow (v_1, v_2) \notin E_G^\beta \\ 1 & : \forall v_1 \neq v_2 : (h_{G,P}(v_1), h_{G,P}(v_2)) = (u_1, u_2) \Rightarrow (v_1, v_2) \in E_G^\beta \\ \frac{1}{2} & : \textit{else} \end{cases}$
- $\ell_P = {h_{G,P}}^{-1} \circ \ell_G$

where $h_{G,P} : V_G \rightharpoonup V_P$ *is the* induced mapping *of concrete nodes from* G *to abstract nodes in* P*, defined as*

$$h_{G,P} = \{(v, core)\} \cup \{(u, u') \in (V_G \setminus \{v\}) \times (\mathcal{SP} \times \mathcal{N}) \mid SP_G(v, u) \neq \emptyset \text{ and } u' = (SP_G(v, u), \ell_G(u))\},$$

The information order compares the information content of two peripheral constraints. It expresses that a $\frac{1}{2}$-constraint is less precise than both a 0 and a 1 constraint.

Definition 11 (Information order)**.** *For $l_1, l_2 \in \{0, 1, \frac{1}{2}\}$, we write $l_1 \sqsubseteq l_2$ iff $l_1 = l_2$ or $l_2 = \frac{1}{2}$.*

Using information order, we define a partial order on clusters P and P' that considers P to be less than or equal to P' if P and P' are equal except for peripheral constraints, and each constraint of P is less than or equal (with respect to the information order) to the corresponding constraint of P'.

Definition 12 (Cluster order)**.** *Let P and P' be clusters. We write $P \sqsubseteq P'$ iff $G_P = G_{P'}$, $S_P = S_{P'}$ and $C^\beta_P(v, v') \sqsubseteq C^\beta_{P'}(v, v')$ for any $\beta \in \mathcal{E}$ and $v, v' \in V_P$. We say that P' is an upper bound of P.*

Note that both information order and cluster order are partial orders, so the notion of least upper bounds is applicable to them. A least upper bound exists for clusters as long as they differ in peripheral constraints only. It yields a cluster with peripheral constraints that are just weak enough to be consistent with both clusters. In effect, a constraint becomes $\frac{1}{2}$ whenever it differs in the two clusters (or is already $\frac{1}{2}$).

Our abstract domain consists of sets of clusters, such that no pair of clusters is comparable according to the cluster order:

Definition 13 (Abstract topology)**.** *An* abstract topology *is a set $S \subseteq \mathcal{P}$, where for no pair $P_1 \neq P_2 \in S$ there is a mutual upper bound $P' \in \mathcal{P}$.*

To obtain such an abstract topology from the clusters produced by local abstraction, we impose an order on sets of clusters, with an induced least upper bound. Cluster set S is less than or equal to cluster set S' according to this induced order iff for each cluster P in S, S' contains a cluster P', such that $P \sqsubseteq P'$ according to the cluster order.

Definition 14 (Cluster set order)**.** *Let S, S' be sets of clusters. We write $S \sqsubseteq S'$ iff for each $P \in S$, there is a $P' \in S'$ such that $P \sqsubseteq P'$.*

We split the set of clusters into singleton sets, each containing one of the clusters. Then we consider the least upper bound over all of those singleton sets. This means joining any clusters that can be joined and taking the union for those that cannot. At the end, this yields the abstract topology we were looking for. We call this abstract topology the topologization of the cluster set under consideration.

Definition 15 (Topologization)**.** *The* topologization *of a set of clusters $S \subseteq \mathcal{P}$ is the abstract topology $\bigsqcup S = \bigsqcup\{\{P\} \mid P \in S\})$.*

For each equivalence class of clusters from S identical except for peripheral constraints, topologization yields a single, joined, less precise cluster in the resulting topology. Note that we overload the $\bigsqcup$ operator, denoting topologization if applied to a set of clusters, and denoting the least upper bound on cluster sets if applied to sets of sets of clusters. Note further that, given $S \sqsubseteq S'$, we have $\bigsqcup S \sqsubseteq \bigsqcup S'$ and $S \sqsubseteq \bigsqcup S'$, but not necessarily $\bigsqcup S \sqsubseteq S'$.

The full abstraction of a graph is the topologization of the set of clusters obtained by local abstraction of each node of the graph. Each of these nodes corresponds to the core node of one of the clusters in the resulting abstract topology. Conversely, we define topology concretization.

Definition 16 (Cluster abstraction and concretization)**.** *Let* $\mathfrak{G} \subseteq \mathcal{G}$*. Then the* cluster abstraction *of* $\mathfrak{G}$ *is the abstract topology* $\alpha(\mathfrak{G}) = \bigsqcup\{\alpha(G, v) \mid v \in V_G \wedge G \in \mathfrak{G}\}$*. An abstract topology* S *represents the set of concrete graphs* $\gamma(S) = \{G \in \mathcal{G} \mid \alpha(\{G\}) \sqsubseteq S\}$*.*

4.3.2 Abstract Transformer

Thus far, we considered how to apply rules on concrete graphs and how to abstract a graph into an abstract topology. Now, we discuss the application of rules on abstract topologies instead of concrete graphs. We obtain an abstract topology capturing the graphs we would obtain in the concrete. We sacrifice some precision in the abstract transformation to allow for a tractable and efficient implementation.

Rule application to all graphs from the cluster concretization induces an abstract derivation relation between clusters for a given rule and abstract topology. The relation holds if the core nodes of source and target cluster relate to corresponding nodes in the respective host and result graph.

Definition 17 (Induced abstract derivation)**.** *The* induced abstract derivation *is a relation* $\overset{r,S}{\Rightarrow} \subseteq \mathcal{P} \times \mathcal{P}$ *where* $P' \overset{r,S}{\Rightarrow} Q$ *iff there are graphs* G, H*, a match* $m : V_L \to V_G$ *from* r *to* G *and a node* $v \in V_G$*, such that* G *is in the cluster concretization of* S*,* $P \sqsubseteq P'$*,* $G \overset{r,m}{\rightsquigarrow} H$ *and* $\alpha(H, v) = Q$*, where* $r = (L, h, p, R)$*,* $P = \alpha(G, v)$ *with induced mapping* $h_{G,P} : V_G \to V_P$ *and* $m \circ h_{G,P} \neq \emptyset$*.*

The induced abstract topology is the topology we obtain if we apply full abstraction to the initial graph and then iteratively compute abstract topologies until we reach a fixpoint: We apply any rule in any possible way to any graph from the cluster concretization of the abstract topology from the previous iteration, add the resulting clusters to those that already existed, and take the least upper bound on the cluster set thus obtained.

Definition 18 (Induced abstract topology)**.** *The* induced abstract topology *is the set* $[\![\mathcal{S}]\!]^\sharp = [\![\mathcal{S}]\!]^\sharp_n$ *where* $n = \min\{i \in \mathbb{N} \mid [\![\mathcal{S}]\!]^\sharp_i = [\![\mathcal{S}]\!]^\sharp_{i+1}\}$ *and* $[\![\mathcal{S}]\!]^\sharp_i$ *defined recursively as follows:*

- $[\![\mathcal{S}]\!]^\sharp_0 = \alpha(\{I\})$
- $[\![\mathcal{S}]\!]^\sharp_i = \bigsqcup([\![\mathcal{S}]\!]^\sharp_{i-1} \cup \{Q \in \mathcal{P} \mid \exists P \in \mathcal{P}, r \in \mathcal{R} : P \overset{r, [\![\mathcal{S}]\!]^\sharp_{i-1}}{\Rightarrow} Q\})$

Note that the existence of the induced abstract topology follows from the fact that $[\![\mathcal{S}]\!]^\sharp_i \sqsubseteq [\![\mathcal{S}]\!]^\sharp_{i+1}$ and the finiteness of the domain.

Proposition 1. *The induced abstract topology overapproximates the graph transformation system semantics, i.e.,* $[\![\mathcal{S}]\!] \subseteq \gamma([\![\mathcal{S}]\!]^\sharp)$*.*

Induced abstract derivation, and, consequently, the induced abstract topology involves rule application to an infinite number of graphs. For an implementation, we need to reduce this to a finite number. That this is possible follows from the fact that our domain is finite.

We capture the characteristics of a sufficient, yet finite subset using the notion of abstract matches. While a concrete match relates a left hand side L of a rule to the nodes of a host graph G, the abstract match relates it to a cluster P. The core node of P has a corresponding node in a host graph G. This node has a corresponding focal node in the result graph H. (Recall that we do not permit node deletion.) Local abstraction on the result graph will yield the relevant cluster Q. Q primarily depends on P and the node and edge modifications as stipulated by the rule. Thus, the main components of an abstract match are P and the relation $h_{L,P}$ between the left hand side and the matched nodes of P.

However, indirectly, and perhaps contrary to intuition, Q also depends on some nodes and edges of the host graph G that are *neither* matched *nor* determined by P:

- For each match to a summary node, only one concrete instance will be matched. Thus, Q may depend on the number of additional unmatched instances (captured by *mater* in the following definition). We need to distinguish only the cases of zero, one, and more than one instances, since the latter will always become a summary node after abstraction.

- A $\frac{1}{2}$-constraint in P may become a 0 and 1 constraint in Q, and sometimes remain as is: (a) If two matched peripheral nodes have an unmatched $\frac{1}{2}$-constraint in between, the corresponding concrete edge will be either present or absent in G, captured by *cc*. (b) The concrete edge corresponding to a $\frac{1}{2}$-constraint between a pair of unmatched peripheral nodes will be either present or absent in G. Concrete edges incident to residual materializations of a summary node v with $mater(v) \geq 1$ may be present for all, none or some of the corresponding concrete node pairs. Both cases are captured by *dd*. (c) The same possibilities exist for edges between an unmatched peripheral node and a matched node. The mapping *dc* specifies these edges. In this case, the matched node does not even have to be in P, since it might just be about to become connected to the focal node through application of the rule.

In addition, the match requires that a closure exists, that is, we have a graph G from the cluster concretization for which the match holds.

Definition 19 (Abstract match)**.** *Let $r = (L, h, p, R)$ be a rule and S be an abstract topology. An abstract match from r to S is a tuple $(P, h_{L,P}, mater, dc, cd, dd)$ where*

- *$P \in \mathcal{P}$ is the matched cluster,*
- *$h_{L,P} : V_L \rightharpoonup V_P$ maps the left hand side to the nodes of P,*
- *$mater : D_P \to \{0, 1, 2\}$ specifies the residual materialization count of summary nodes in P,*

- $dc : (V_L \times D_P \times \{-1, 1\} \times \mathcal{E}) \to \{0, 1, \frac{1}{2}\}$ *specifies the materialization of edges from peripheral to matched nodes and vice versa,*
- $dd : (D_P \times D_P \times \mathcal{E}) \to \{0, 1, \frac{1}{2}\}$ *specifies the peripheral edge materialization*
- $cc : V_L \times V_L \to 2^{\mathcal{E}}$ *specifies the materialization of edges among matched nodes*

and the following conditions are satisfied:

- $P \sqsubseteq P'$ *for some* $P' \in S$
- $h_{L,P}(V_L) \neq \emptyset$
- $|{h_{L,P}}^{-1}(core)| < 2$
- *the following conditions hold for* $matched : D_P \to \mathbb{N}$*, the induced number of matches, defined as* $matched(u) := |{h_{L,P}}^{-1}(\{u\})|$*:*

$$matched(u) = 0 \Rightarrow mater(u) = \begin{cases} 2 & \text{if } u \in S_P \\ 1 & \text{otherwise} \end{cases}$$

$$matched(u) = 1 \Rightarrow mater(u) \in \begin{cases} \{1, 2\} & \text{if } u \in S_P \\ \{0\} & \text{otherwise} \end{cases}$$

$$matched(u) > 1 \Rightarrow u \in S_{P_v}$$

- *there is a graph* G*, a match* $m : V_L \to V_G$ *from* r *to* G *and a node* $v \in V_G$ *such that* $\alpha(G, v) = P$ *with induced mapping* $h_{G,P} : V_G \rightharpoonup V_P$ *and*
  - $m \circ h_{G,P} = h_{L,P}$,
  - $mater(u) = \min\{|{h_{G,P}}^{-1}(\{u\}) \setminus m(V_L)|, 2\}$,
  - *for all* $u \in V_L \setminus m^{-1}(\{v\})$, $u' \in D_P$, $\beta \in \mathcal{E}$, *and* $d \in \{-1, 1\}$,

$$dc(u, u', d, \beta) = \begin{cases} 0 & : \forall v' \notin m(V_L) : h_{G,P}(v') = u' \Rightarrow (m(u), v') \in (E_G^\beta)^d \\ 1 & : \forall v' \notin m(V_L) : h_{G,P}(v') = u' \Rightarrow (m(u), v') \notin (E_G^\beta)^d \\ \frac{1}{2} & \text{otherwise,} \end{cases}$$

  - *for all* $u, u' \in D_P$*, for all* $\beta \in \mathcal{E}$,

$$dd(u, u', \beta) = \begin{cases} 0 & : \forall v_1 \neq v_2 \notin m(V_L) : (h_{G,P}(v_1), h_{G,P}(v_2)) = (u, u') \\ & \qquad \Rightarrow (v_1, v_2) \in E_G^\beta \\ 1 & : \forall v_1 \neq v_2 \notin m(V_L) : (h_{G,P}(v_1), h_{G,P}(v_2)) = (u, u') \\ & \qquad \Rightarrow (v_1, v_2) \notin E_G^\beta \\ \frac{1}{2} & \text{otherwise,} \end{cases}$$

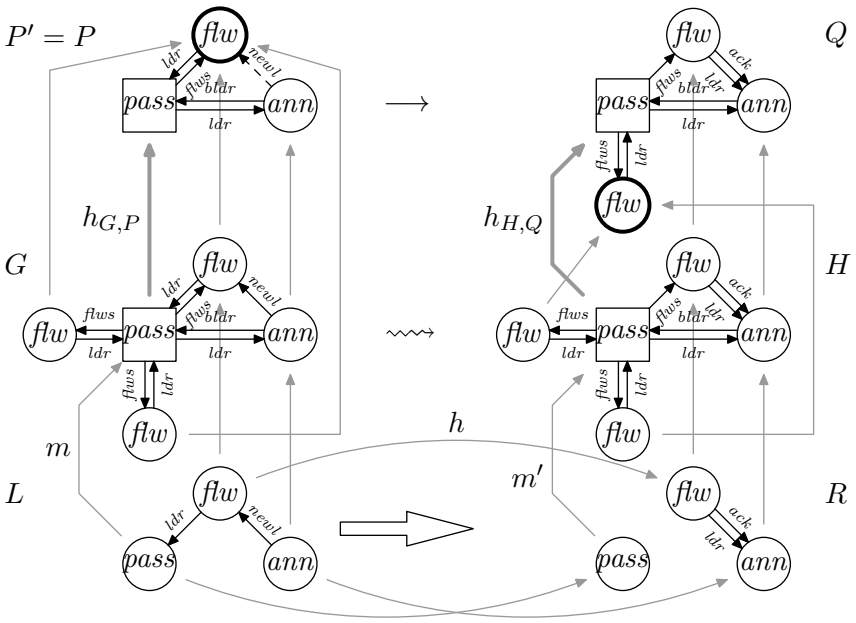


Figure 4.3: An example of the abstract transformer.

– *for all* $u, u' \in V_L$,

$$(cc(u,u'), cc(u',u)) = SP_G(m(u), m(u')),$$

– $G \in \gamma(S)$, *and*

– *the partner constraints* p *are satisfied.*

Since the number of abstract matches is finite, the definition is constructive and a computation method directly follows from it, except for the non-trivial closure check. However, the fact that we are looking for an overapproximation allows us to weaken this check, including the option to ignore it completely. This includes the check that partner constraints are satisfied. Note that at least the partner constraints for ${h_{G,P}}^{-1}(core)$ can be checked without knowledge of the entire graph G.

From the abstract matches, we generate partial concretizations. These are graphs with focal node and neighborhood, just sufficient to capture all potential changes to the cluster caused by rule application and local abstraction of the result. We do not need to consider the full graph, since this is taken care of by symmetry: The additional nodes it contains will be covered by other abstract matches with those nodes as the core node of a cluster. Those, in turn, have their own partial concretizations to account for the impact of the rule application.

Note that edges specified by *dd* and *dc* will never be modified by a rule, for that would require its adjacent nodes both to be matched, which is, by definition, not the case. The set A, in the following definition, splits the unmatched peripheral nodes into two subsets such that those in the set will have respective edges for the $\frac{1}{2}$ case and the complementary nodes will not.

Definition 20 (Partial concretization)**.** *The* partial concretization *function γ maps abstract matches $(P, h_{L,P}, mater, dc, dd, cc)$ to tuples $(G, m, h_{G,P})$ where G is a graph, $m : V_L \rightharpoonup V_G$ is an injective partial graph morphism from the left hand side to this graph and $h_{G,P} : V_G \rightharpoonup V_P$ is a mapping to the abstraction P, all defined as follows:*

- $V_G = \{core\} \cup (V_L \setminus {h_{L,P}}^{-1}(\{core\})) \cup \{(u, n) \in D_P \times \mathbb{N} \mid 1 \leq n \leq mater(u)\}$
- $h_{G,P}(u) = \begin{cases} core & \text{if } u = core \\ v & \text{if } u = (v, n) \\ h_{L,P}(u) & \text{if } u \in V_L \setminus {h_{L,P}}^{-1}(\{core\}) \end{cases}$
- $m = \{({h_{L,P}}^{-1}(core), core)\} \cup \{(u, u) \mid u \in (V_L \setminus {h_{L,P}}^{-1}(\{core\}))\}$
- $E_G^\beta = \{(u, u') \in A \times A \mid dd(h_{G,P}(u), h_{G,P}(u'), \beta) \geq \frac{1}{2}\}$
  $\cup \{(u, u') \in V_G \times V_G \mid dd(h_{G,P}(u), h_{G,P}(u'), \beta) = 1\}$
  $\cup \{(u, u') \in m(V_L) \times m(V_L) \mid \beta \in cc(m^{-1}(u), m^{-1}(u'))\}$
  $\cup \{(u, u') \in m(V_L) \times A \mid dc(m^{-1}(u), h_{G,P}(u'), 1, \beta) = \frac{1}{2}\}$
  $\cup \{(u, u') \in m(V_L) \times V_G \mid dc(m^{-1}(u), h_{G,P}(u'), 1, \beta) = 1\}$
  $\cup \{(u, u') \in A \times m(V_L) \mid dc(m^{-1}(u'), h_{G,P}(u), -1, \beta) = \frac{1}{2}\}$
  $\cup \{(u, u') \in V_G \times m(V_L) \mid dc(m^{-1}(u'), h_{G,P}(u), -1, \beta) = 1\}$
  $\cup {h_{G,P}}^{-1}(E_P^\beta)$
  where $A = (D_P \times \{1\}) \cap V_G$
- $\ell_G = (m^{-1} \circ \ell_L) \cup (h_{G,P} \circ \ell_P)$

The abstract transformer describes how clusters are affected by rule application. It presupposes the existence of an abstract match, constructs the corresponding partial concretization, applies the rule, and constructs the modified cluster by local abstraction of the focal node. See Figure 4.3.2 for an example.

Definition 21 (Abstract transformer)**.** *Let $r = (L, h, p, R)$ be a rule and S be an abstract topology. The* abstract transformer *(or* direct derivation*) is a relation $\overset{r,S}{\to} \subseteq \mathcal{P} \times \mathcal{P}$ where $P' \overset{r,S}{\to} Q$ iff there is a graph H and an abstract match $\hat{m} = (P, h_{L,P}, mater, dc, dd, cc)$ from r to S such that $P \sqsubseteq P'$, $\gamma(\hat{m}) = (G, m, h_{G,P})$, $G \overset{r,m}{\rightsquigarrow} H$ and $Q = \alpha(H, core)$*

The graph morphism m may be partial, i.e., some nodes of the left hand side may map to none of the nodes in G. Not even the focal node needs to be covered. In those cases, we waive the totality requirement that rule application puts on m, thereby modifying only those parts of the partial concretization that are matched. We obtain an abstract topology that overapproximates the system by abstracting the start graph and applying the abstract transformer in a fixpoint iteration.

Definition 22 (Derived abstract topology)**.** *The* derived abstract topology *is the set $[\mathcal{S}]^\sharp = [\mathcal{S}]^\sharp_n$, where $n = \min\{i \in \mathbb{N} \mid [\mathcal{S}]^\sharp_i = [\mathcal{S}]^\sharp_{i+1}\}$ and $[\mathcal{S}]^\sharp_i$ is defined recursively as follows:*

- $[\mathcal{S}]_0^\sharp = \alpha(\{I\})$
- $[\mathcal{S}]_i^\sharp = \bigsqcup([\mathcal{S}]_{i-1}^\sharp \cup \{Q \in \mathcal{P} \mid \exists P \in [\mathcal{S}]_{i-1}^\sharp, r \in \mathcal{R} : P \overset{r,[\mathcal{S}]_{i-1}^\sharp}{\rightarrow} Q\}$
  $\cup \{\alpha(\{R\}) \mid (\emptyset, \emptyset, \emptyset, R) \in \mathcal{R}\})$

Note that we assumed the absence of rules with non-empty left hand side that create new nodes. Because of this, we do not need to take care of new clusters that occur as a byproduct of the modification of an existing cluster. Instead, for each rule with empty left hand side, we add the clusters obtained by local abstraction for each right hand side node. This takes place unconditionally, pointing towards the equivalence of node creation and initial graphs in our domain.

Theorem 1. *The derived abstract topology overapproximates the induced abstract topology, i.e.,* $[\![\mathcal{S}]\!]^\sharp \sqsubseteq [\mathcal{S}]^\sharp$.

Corollary 1 (Soundness)**.** *The derived abstract topology overapproximates the graph transformation system semantics, i.e.,* $[\![\mathcal{S}]\!] \subseteq \gamma([\mathcal{S}]^\sharp)$.

Proof. This follows immediately from Proposition 1, Theorem 1, and the monotonicity of cluster concretization. □

4.4 Experimental Evaluation

4.4.1 Implementation

We implemented cluster abstraction in our tool `astra` 2.0. The implementation differs from theory in minor respects: (a) Partial concretization materializes clusters over the entire left hand side of a rule at once, exploiting symmetry and allowing us to properly check all partner constraints. (b) We do a rudimentary check for the existence of a closure, by checking whether peripheral constraints of unmatched nodes are satisfiable. (c) To cover cases with unmatched core nodes, for each match, we iterate over all possibilities in which one additional cluster can be attached in the periphery. (d) After each iteration, we apply a reduction step, eliminating any cluster whose existence can be ruled out easily, and concretizing $\frac{1}{2}$-constraints if more precise information is available. (e) In various places, we use overapproximation ad hoc in order to improve analysis time.

4.4.2 Selection of Benchmarks

With `astra` 1.0, we already succeeded to analyze a part of the *merge protocol* [BR10b] with star abstraction [BR10a], a precursor to the method described in this paper. (In a nutshell, cluster abstraction with all peripheral constraints being $\frac{1}{2}$.) It was sufficient to analyze platoon formation and car joining, but not platoon merging, for which state space explosion occurred: Follower handover requires ternary predicates, while star abstraction only preserves binary predicates. This causes a cascade of spurious abstract states, with the analysis eventually spending its time enumerating an intractable

Table 4.1: Benchmark analysis statistics. cl. = clusters, a.r. = active rules, i.e., applied at least once, m. = abstract matches, rule app. = rule applications, it. = iterations, vfy. = safety property verified. *safety property not expressible as forbidden subgraphs

Benchmark	# cl.	# a.r.	# m.	# rule app.	# it.	time	vfy.
Sync. merge	873	34	9674	349774	17	0m 14.057s	**yes**
Async. merge	3069	36	44553	36114603	21	14m 27.977s	**yes**
AVL trees	1876	302	114284	2221151967	38	757m 9.273s	**yes**
Firewall	31	4	139	1371	5	0m 0.012s	**yes**
Firewall 2	96	9	786	45525	7	0m 0.330s	no
Pub/priv s. 2	239	26	1633	102250	10	0m 1.030s	**yes**
Dining phil.	41	8	40	179	7	0m 0.006s	no*
Resources	32	7	100	207	4	0m 0.007s	**yes**
Mutual ex.	308	9	2419	1237361	17	0m 56.060s	**yes**
Red-black tr.	263	38	8769	24855500	11	10m 3.145s	**yes**
Singly-linked l.	7	2	15	13	3	0m 0.000s	**yes**
Circ. buffers	152	2	798	241234	17	2m 43.441s	no*
Euler walks	18	6	47	134	3	0m 0.008s	no*

number of combinatorial possibilities. The main goal of `astra` 2.0 was to analyze the full protocol. We did this for two versions. In addition, we analyzed the *AVL tree* benchmark from [Bac08] and various other benchmarks from the related work: *Firewall, public/private servers, dining philosophers, resources, mutual exclusion*, and *red-black trees* are benchmarks from the AUGUR package [KK08a]; *singly-linked lists, circular buffers* and *Euler walks* for GROOVE are from [Zam13].

The AUGUR package comes with additional benchmarks that we did not analyze: *connections, leader election protocol* and the *Needham–Schroeder protocol* all make use of numerical attributes, which are not yet supported by our tool. *External-internal processes* is merely a stripped-down version of *public and private server 2. Public and private server* contains a subset of the rules from *public and private server 2*. The same holds for the finite-state version of *dining philosophers* versus the infinite-state version, which we analyze. *Red-black trees converted* is a tweaked version of *red-black trees* to ease analysis with AUGUR.

We could analyze the GROOVE benchmarks without modifications. The AUGUR benchmarks, on the other hand, had to be translated from the tool's hyper-edge-based approach to one based on nodes and edges. In addition, we had to make a structure-preserving change to the *public/private server* grammar (replacing a specific edge with two edges connected by a node) in order to prevent combinatorial explosion that would otherwise have defied analysis. For *red-black* and *AVL trees*, we manually added invariants about the uniqueness of some labels over the entire graph. These invariants trivially follow from the respective graph transformation systems and it would in principle be easy

to find them automatically. However, uniqueness is not expressible in our abstraction, because clusters always represent an arbitrary number of concrete instances.

We checked the safety properties by adding rules specifying respective forbidden subgraphs, producing a node with an error label if found. This approach could not be taken for *dining philosophers*, *circular buffers* and *Euler walks*, since the respective safety properties quantify over an unbounded number of nodes and hence cannot be formulated as forbidden subgraphs.

4.4.3 Analysis Results

`astra` was able to analyze all benchmarks. See Table 4.1 for the number of iterations required for reaching the fixed point, the number of clusters in the final result and the processor time taken. We ran all analyses on an Intel Core 2 Quad CPU Q9550 (2.83GHz) with 4 GB of memory under Linux 3.15, though only 9 MB were used at the peak for the largest benchmark, *asynchronous merge*. Execution time given is the time in user mode as reported by `time`(1).

In all but one of the cases with safety properties expressible as forbidden subgraphs, verification succeeded. Verification failed for *firewall 2* because the abstraction was unable to distinguish locations in front of and behind the firewall.

4.5 Related Work

Petri graphs are unfoldings of graph transformation systems, abstracted by a cutoff after a defined depth [BK02]. Reachability can be checked with existing techniques for Petri nets. As we have seen, we were able to analyze a subset of their benchmarks. Once they support negative application conditions (which they currently list as future work), it will be interesting to investigate whether their tool AUGUR [KK08b] can analyze our main target, the merge protocol.

Bauer et al.'s partner abstraction [BW07] considers connected components instead of overlapping clusters and folds nodes according to neighborhood node and edge labels. In practice, it requires the system to obey friendliness properties that hold only for a simplified merge protocol where processes know each other's state [Bac08]. Rensink and Distefano [RD06] consider an abstraction similar in design and limitations. Ideas from both approaches were combined and extended in neighborhood abstraction [BKK+12]. No friendliness restriction applies, but lacking Bauer's decomposition into components, the GROOVE implementation runs out of memory even on Bauer's simplified merge protocol [Zam13].

Environment abstraction [CTV06] abstracts a system into one process and its environment, i.e., the set of states of all the other processes plus relations to them. Cherem and Rugina [CR07] propose a local abstraction for shape analysis that tracks individual heap cells and their immediate neighborhood. Bauer et al.'s daisy patterns [BBR09] and our star abstraction [BR10a] are graph abstractions based on the same idea, the former abstracting the transformation rules in addition to the graph. All these abstractions are less precise than cluster abstraction, since none of them tracks peripheral node relationships.

Saksena et al. [SWJ08] verify graph transformation systems by symbolic backward reachability analysis. Starting with the undesirable configurations, they compute, by backward rule application in a fixed point iteration, an overapproximation of the set of reachable predecessor configurations, checking whether an initial configuration is among them. While not guaranteed to terminate, their method succeeds in proving loop freedom of an ad hoc routing protocol.

Berdine et al. [BLAM+08] show that shape analysis of concurrent programs via canonical abstraction [SRW02] leads to state-space explosion even for a toy example. The complexity of expressing cluster abstraction via canonical abstraction confirms this: at least, one abstraction predicate would be needed for each spoke, which is exponential in the number of edge labels. Berdine et al.'s own solution allows efficient analysis of an unbounded number of threads manipulating an unbounded shared heap. However, their abstraction is unable to express direct relations between the state of the threads. Manevich et al. [MLAS+08] decompose the heap into a bounded number of overlapping components as specified by user-defined location selection predicates. In contrast, our method decomposes the graph by local abstraction of each of the unbounded number of nodes.

Zufferey et al. [ZWH12] provide an abstraction for depth-bounded systems (systems with a bound on the longest acyclic path), an expressive class of well-structured transition systems. Unfortunately, the merge protocol does not belong to this class unless one uses a simplified version similar to Bauer's.

4.6 Conclusions and Future Work

We have seen an abstraction for the analysis of the set of reachable graphs generated by infinite-state graph transformation systems. Using `astra`, our implementation of *cluster abstraction*, we were for the first time able to analyze the full merge protocol. In addition, our method has proven robust and precise enough to allow for the analysis of various benchmarks from the literature.

Future work: (1) We are going to check safety properties that cannot be expressed as forbidden subgraphs, such as quantification over an unbounded number of nodes. (2) We shall explore suitable approximations for the closure check, to preserve more of the global graph structure during rule application. (3) We are going to investigate opportunities to adjust the precision of our analysis. Especially, structure-preserving changes to the graph transformation system before the analysis seem to be an interesting way to give direction to the abstraction. For example, adding edges to the right hand side of rules with a new label that never occurs on a left hand side can keep nodes in the periphery of some clusters, thereby increasing precision. (4) If some cluster may occur at most once, we would like to retain this information. (5) We would like to allow integer values as node and edge attributes, in addition to regular labels. Lifted to the abstraction, it extends clusters by overapproximated values for those attributes, based on abstract domains on integers. (6) Based on a suitable fragment of μ-calculus, we plan to support abstract

model checking on an abstract labeled transition systems of clusters, preserving some non-trivial relationships for the transitions, such as size invariants on summary nodes. We plan to extend this to model checking over an abstract labeled transition system, based on a suitable fragment of μ-calculus.

4.7 Appendix 1: Notes About Evaluation

Various simplified versions of the merge protocol are part of the `hiralysis` implementation of partner abstraction. We were able to analyze all of them, but did not list them because they are essentially subsets of the full merge protocols.

All input files necessary to reproduce the analyses, as well as the ASTRA tool itself, are available at `http://rw4.cs.uni-sb.de/~rtc/astra/`. See the script `vmcai15.sh` provided with the tool for exactly how we analyze the benchmarks discussed in our paper.

ASTRA already has support for abstract transitions, and it outputs an abstract labeled transition system (LTS). Where feasible (i.e., the number of clusters is reasonable), we show this abstract LTS below, layouted with yEd.

Description of benchmarks and the corresponding safety properties:

Synchronous merge. Free agents form platoons of leaders and followers. Two leaders can merge by first determining a front and a back leader and then having the back leader hand over its followers to the front leader. The followers are transferred one by one, with the back leader waiting for acknowledgment after each step. Safety property: No two followers assume each other to be their leader, and leaders always have at least one follower. When about to hand over a follower, at least one follower is actually there.

Asynchronous merge. Like the synchronous version, except that the back leader hands over all followers in parallel.

AVL trees. We model AVL tree changes as a graph transformation system. Safety property: Nodes with balance level 0 must have zero or two children. Nodes with balance level -1 and 1 must have a left, respectively, a right child. If they have only one child, that child itself must not have any children. Note that these properties do not fully guarantee balancing.

Firewall. A network of secure and insecure locations, separated by a firewall and each connected to each other. Secure and insecure packets are exchanged over directed links and firewalls. Links, locations and packets can be added dynamically. Packets may cross firewalls only in one direction. Safety property: No insecure packet ever finds its way behind the firewall.

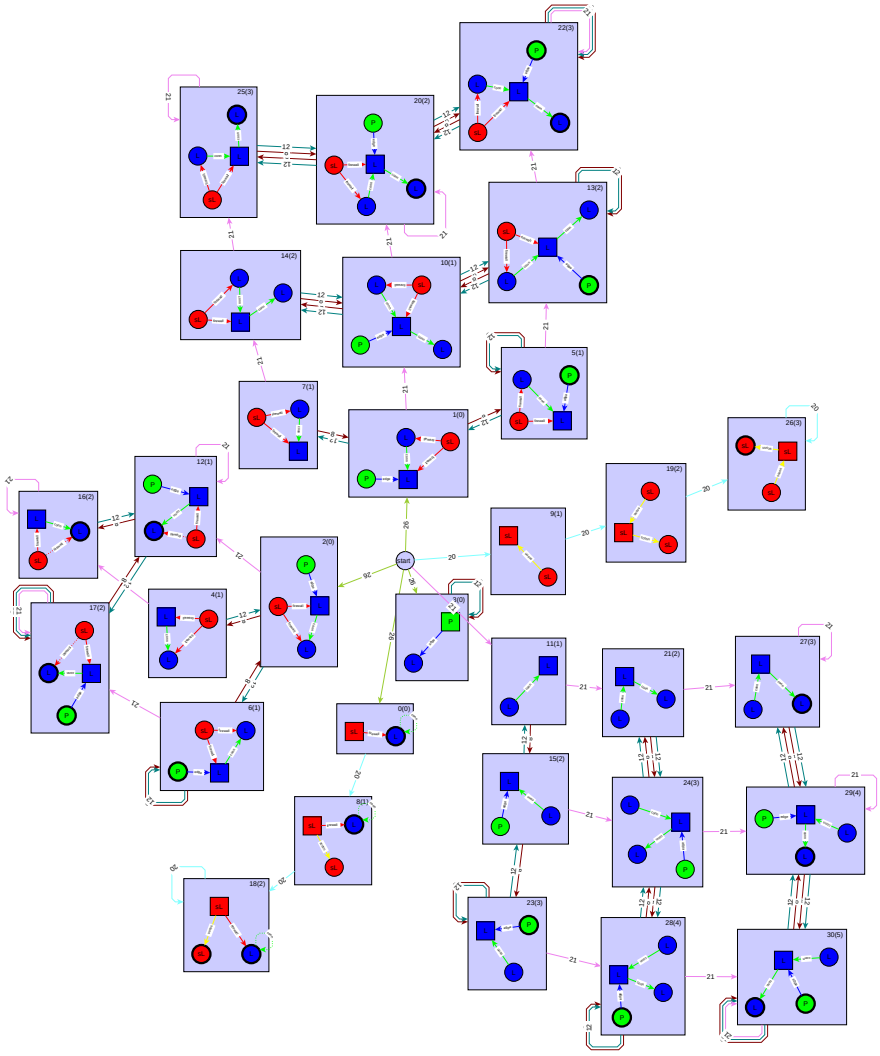


Figure 4.4: The abstract LTS for Firewall

Firewall 2. The same as firewall, but secure and insecure locations have the same label.

Public/private servers 2. A dynamically changing network of private and public servers is connected by directed network links, but never from a public to a private server. Private servers can create internal processes, external servers can create external processes. Both can switch to a different server over a network link. Safety property: A private server never runs an external process.

Dining philosophers. A dynamic infinite-state version of dining philosophers. In addition to taking forks, philosophers can replicate. Safety property: Absence of a deadlock state.

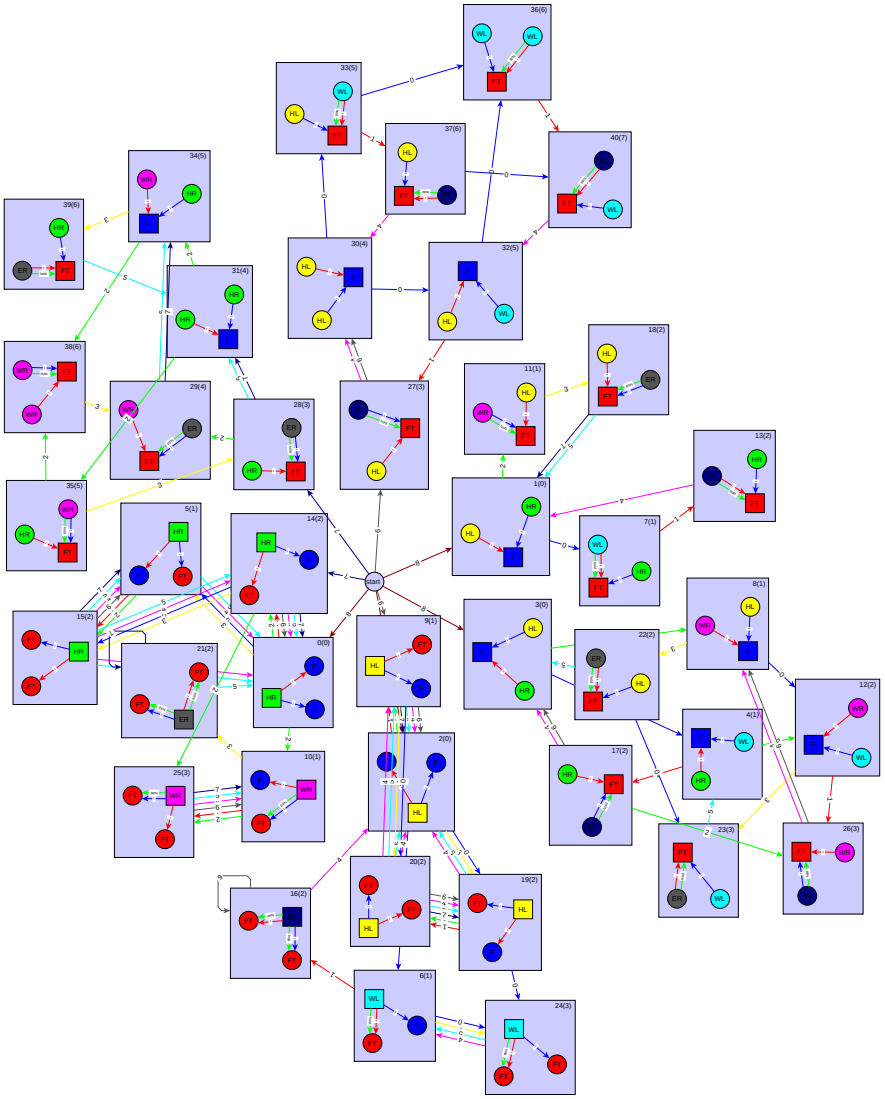


Figure 4.5: The abstract LTS for Dining philosophers

Resources. A system with a dynamically changing number of processes ensuring exclusive access to two resources by acquiring them in a fixed order. Safety property: The two resources are never claimed by two different processes.

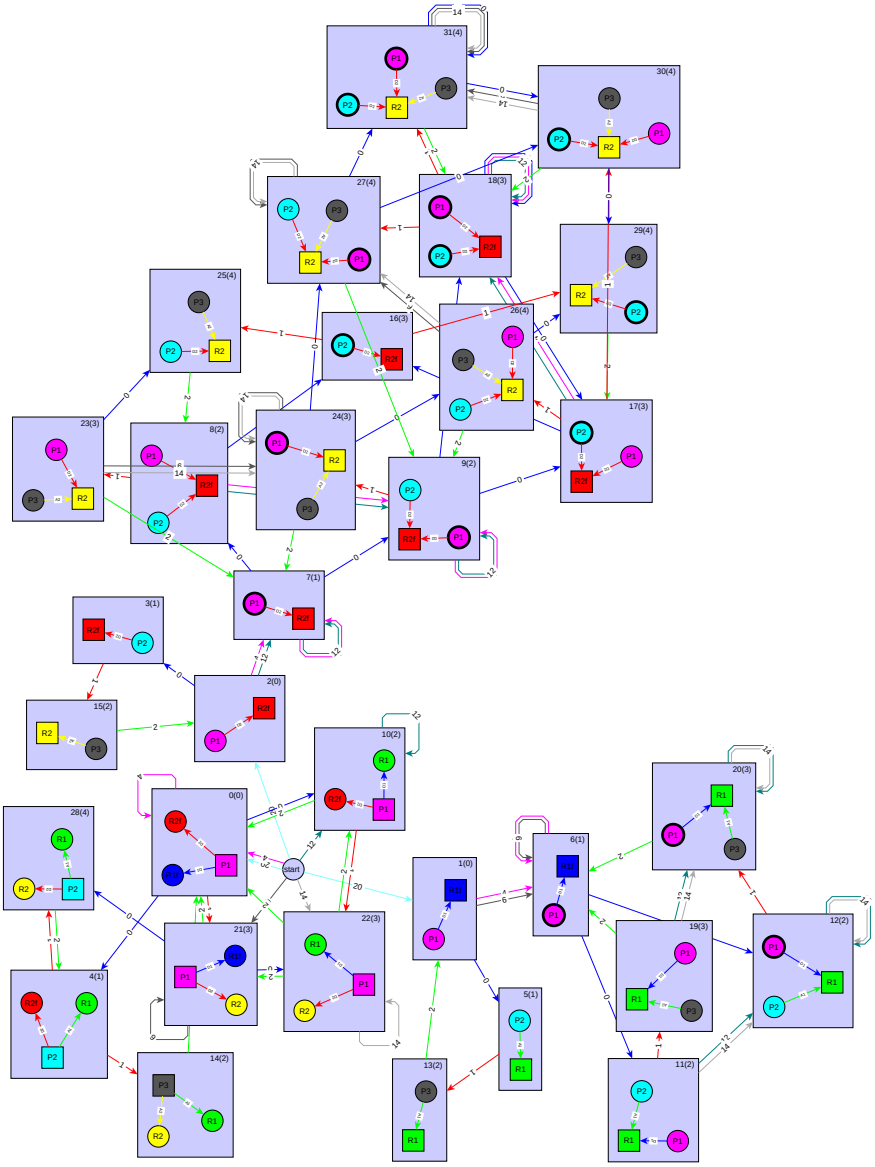


Figure 4.6: The abstract LTS for Resources

Mutual exclusion. Exclusive access to a resource in a system with dynamically added processes is ensured by token passing. Safety property: The resource is never claimed by two processes at once.

Red-black trees. Any possible red-black-tree is generated and an element is inserted, then balance is restored. Safety property: We never obtain a tree with two consecutive red nodes.

Singly-linked lists. Elements are dynamically added to and deleted from a linked list. Safety property: The head cell has no predecessor, and the cells are not shared.

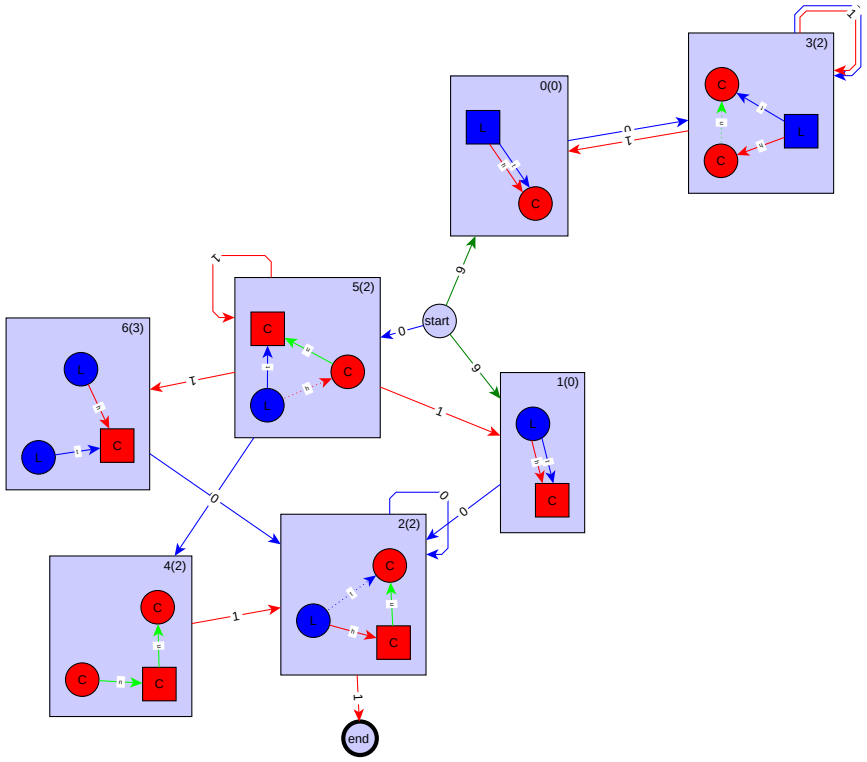


Figure 4.7: The abstract LTS for Singly-linked lists

Circular buffers. Elements are dynamically added to and deleted from a circular buffer. Safety property: Either all cells reachable from the first are unused and the last cell is the predecessor of the first, or the first cell is in use; if a cell is in use, either so is the next, or it is the last.

Euler walks. Euler walks are generated over a dynamically discovered map of areas connected via bridges. Safety property: All walks have either zero or two areas with an odd number of connecting bridges.

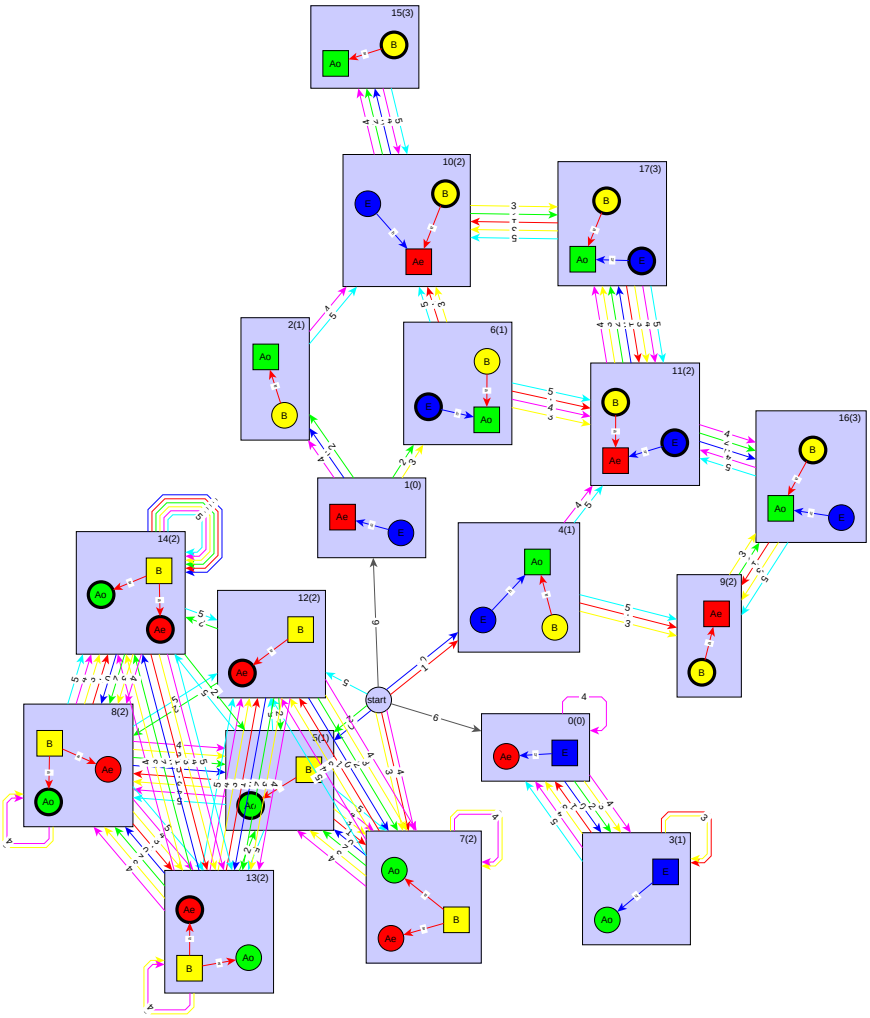


Figure 4.8: The abstract LTS for Euler walks

Figure 4.9 shows the analysis result for a trimmed merge protocol. The functions of the protocol can be nicely seen: Each process starts as a free agent, and depending on whether it sends or receives a join request, it either becomes a follower or a leader of a platoon with one follower. In the latter case, another free agent may request to join the platoon, so that, eventually, a platoon with more than one follower is built. At this point, the abstraction kicks in and further join request lead to the same abstract state. A free agent may also receive a merge request of an existing platoon, in which case handover of the back leader's followers takes place, eventually forming a platoon with one or more than one follower. Conversely, a platoon leader may send a merge request and hand over its followers, eventually becoming a follower itself.

Figure 4.9: ASTRA analysis result for the synchronous merge protocol (see Section 2.3). This is a trimmed-down version showing only the most important behavior (requests are sent only when they are expected). The full version would look very chaotic. As can be seen, ASTRA already builds a labeled transition system of clusters, the theoretical foundation for which is future work. Automatic organic layout by yEd. An aesthetic appeal cannot be denied ;-)

4.8 Appendix 2: Proof of Main Theorem

Recall our main theorem implying soundness of the analysis:

Theorem 1. *The derived abstract topology overapproximates the induced abstract topology, i.e.,* $[\![\mathcal{S}]\!]^\sharp \sqsubseteq [\mathcal{S}]^\sharp$.

sketch. It is sufficient to show that $\overset{r,S}{\Rightarrow}\subseteq\overset{r,S}{\rightarrow}$. Let $P' \overset{r,S}{\Rightarrow} Q$. Then, by Definition 17, there are

- graphs G, H

- a match $m : V_L \to V_G$ from r to G
- and a node $v \in V_G$

such that

- G is in the cluster concretization of S,
- $P \sqsubseteq P'$,
- $G \overset{r,m}{\rightsquigarrow} H$ and
- $\alpha(H, v) = Q$,

where

- $r = (L, h, p, R)$,
- $P = \alpha(G, v)$ with induced mapping $h_{G,P} : V_G \to V_P$ and
- $m \circ h_{G,P} \neq \emptyset$.

We need to show $P' \overset{r,S}{\to} Q$, which, by Definition 21, means: There is

- a graph H' and
- an abstract match $\hat{m} = (P^*, h_{L,P^*}, \mathit{mater}, \mathit{dc}, \mathit{dd}, \mathit{cc})$ from r to S

such that

- $P^* \sqsubseteq P'$,
- $\gamma(\hat{m}) = (G', m^*, h_{G',P^*})$,
- $G' \overset{r,m^*}{\rightsquigarrow} H'$ and
- $Q = \alpha(H', \mathit{core})$.

Let $h_{L,P}$, *mater*, *dc*, *dd* and *cc* be defined as required in Definition 19; let $P^* = P$, $h_{L,P^*} = h_{L,P}$ and $\hat{m} = (P^*, h_{L,P^*}, \mathit{mater}, \mathit{dc}, \mathit{dd}, \mathit{cc})$. It is now evident that $\hat{m}$ is an abstract match and that $P^* \sqsubseteq P'$.
Let $\gamma(\hat{m}) = (G', m^*, h_{G',P^*})$. To complete our proof, we need to show that $G' \overset{r,m^*}{\rightsquigarrow} H'$ such that $Q = \alpha(H', \mathit{core})$, i.e., $\alpha(H, v) = \alpha(H', \mathit{core})$. Let $Q' = \alpha(H', \mathit{core})$ and let $h_{H,Q}$ and $h_{H',Q'}$ be the induced mappings according to Definition 10. By Definition 9, we need to show:

- $G_Q = G_{Q'}$, that is, by Definition 1, $V_Q = V_{Q'}$, $E_Q^\beta = E_{Q'}^\beta$ for all $\beta \in \mathcal{E}$ and $\ell_Q = \ell_{Q'}$.

- $V_Q \subseteq V_{Q'}$: Case *core* is trivial. Let $u' \in D_Q$. By Definition 10, there is a $u \in V_H$ such that $h_{H,Q}(u) = u'$. By definition of $h_{H,Q}$, $u' = (SP_H(v,u), \ell_H(u))$, where $SP_H(v,u) \neq \emptyset$. That may be the case only because of two reasons, if not (1) $SP_G(v,u) \neq \emptyset$, then (2) the rule must have added an edge between v and u.
  - Case (1): Then, by Definition 10, V_P must contain the node $h_{G,P}(u) = (SP_G(v,u), \ell_G(u))$. Let $u^* = h_{G,P}(u)$.
    - Case (1a): There is some $u_L \in V_L$ with $u = m(u_L)$, i.e., u is matched by the left hand side. Then, by Definition 20, $u_L \in V_{G'}$, $SP_{G'}(core, u_L) = SP_G(v,u)$, $\ell_{G'}(u_L) = \ell_G(u)$ and $m^*(u_L) = u_L$. Thus, by Definition 7, $SP_{H'}(core, u_L) = SP_H(v,u)$ and $\ell_{H'}(u_L) = \ell_H(u)$. By Definition 10, hence $h_{H',Q'}(u_L) = (SP_H(v,u), \ell_H(u)) = u'$ and thus, $u' \in V_{Q'}$.
    - Case (1b): Otherwise, $u \notin m(V_L)$, i.e., u was unmatched. Then, by Definition 19, $mater(u^*) \geq 1$. Hence, by Definition 20, $(u^*, 1) \in V_{G'}$ with $SP_{G'}(core, (u^*,1)) = SP_G(v,u)$, $\ell_{G'}((u^*,1)) = \ell_G(u)$ and $(u^*,1) \notin m^*(V_L)$. Hence, by Definition 7, $SP_{H'}(core, (u^*,1)) = SP_{G'}(core, (u^*,1)) = SP_G(v,u) = SP_H(v,u)$ and $\ell_{H'}((u^*,1)) = \ell_{G'}((u^*,1)) = \ell_G(u) = \ell_H(u)$. Hence, by Definition 10, $h_{H',Q'}((u^*,1)) = (SP_H(v,u), \ell_H(u)) = u'$ and thus, $u' \in V_{Q'}$.
  - Case (2): Then the rule must add such an edge; $SP_G(v,u) = \emptyset$. By Definition 10, $u \notin \mathrm{def}(h_{G,P})$. Hence, by Definition 19, $u \notin \mathrm{def}(h_{L,P})$. Let $u_L = m^{-1}(u)$. By Definition 20, $SP_{G'}(core, u_L) = \emptyset$ and $m^*(u_L) = u_L$. Thus, by Definition 7, and analogous case (1a), $u' \in V_{Q'}$.
- $V_{Q'} \subseteq V_Q$: By Definition 10, Definition 20 and Definition 19, it holds that $\alpha(G', core) = P$. Thus, the proof for $V_Q \subseteq V_{Q'}$ can be applied backwards in an analogous fashion.
- $E^\beta_Q = E^\beta_{Q'}$ and $\ell_Q = \ell_{Q'}$ follow from $V_{Q'} = V_Q$ by Definition 10, since $u = (SP_{Q'}(core, u), \ell_{Q'}(u))$ for any $u \in V_{Q'} \setminus V_L$.

- $S_Q = S_{Q'}$:
  - $S_Q \subseteq S_{Q'}$: Let $u' \in S_Q$. By Definition 10, there are $u \neq w \in V_H$ such that $h_{H,Q}(\{u,w\}) = \{u'\}$. We can then argue analogous to $V_Q \subseteq V_{Q'}$.
  - $S_{Q'} \subseteq S'_Q$: Analogous to $V_{Q'} \subseteq V_Q$.

- $C^\beta_Q = C^\beta_{Q'}$ for all $\beta \in \mathcal{E}$: Let $\beta \in \mathcal{E}$, $u', w' \in D_Q$. Let $U = {h_{H,Q}}^{-1}(\{u'\})$ and $W = {h_{H,Q}}^{-1}(\{w'\})$. We decompose U and W into disjoint sets $U_{1a} \cup U_{1b} \cup U_2 = U$ and $W_{1a} \cup W_{1b} \cup W_2 = W$, each corresponding to the cases in the proof of $V_Q \subseteq V_{Q'}$. By analogy, we obtain disjoint sets $U'_{1a} \cup U'_{1b} \cup U'_2 = U'$ and $W'_{1a} \cup W'_{1b} \cup W'_2 = W'$ where $U' = {h_{H',Q'}}^{-1}(\{u'\})$ and $W' = {h_{H',Q'}}^{-1}(\{w'\})$. Let

$Abs(A, B) = 0$ if $A = \emptyset$, $Abs(A, B) = 1$ if $A \supseteq B$ and $Abs(A, B) = \frac{1}{2}$ otherwise and let $(A, B) \approx (A', B')$ iff $Abs(A, B) = Abs(A', B')$.

- $(E_H^\beta, U_{1a} \times W_{1a}) \approx (E_{H'}^\beta, U'_{1a} \times W'_{1a})$: Because the left hand side is completely material (cc)
- $(E_H^\beta, U_2 \times W_{1a}) \approx (E_{H'}^\beta, U'_2 \times W'_{1a})$, $(E_H^\beta, U_{1a} \times W_2) \approx (E_{H'}^\beta, U'_{1a} \times W'_2)$, $(E_H^\beta, U_2 \times W_2) \approx (E_{H'}^\beta, U'_2 \times W'_2)$: Analogous
- $(E_H^\beta, U_{1b} \times W_{1a}) \approx (E_{H'}^\beta, U'_{1b} \times W'_{1a})$: Because of dc
- $(E_H^\beta, U_{1a} \times W_{1b}) \approx (E_{H'}^\beta, U'_{1a} \times W'_{1b})$, $(E_H^\beta, U_2 \times W_{1b}) \approx (E_{H'}^\beta, U'_2 \times W'_{1b})$, $(E_H^\beta, U_{1b} \times W_2) \approx (E_{H'}^\beta, U'_{1b} \times W'_2)$: Analogous
- $(E_H^\beta, U_{1b} \times W_{1b}) \approx (E_{H'}^\beta, U'_{1b} \times W'_{1b})$: Because of dd

Therefore, $(E_H^\beta, U \times W) \approx (E_{H'}^\beta, U' \times W')$. Hence, by Definition 10, $C_Q^\beta = C_{Q'}^\beta$.

Therefore, the theorem holds. □

4.9 Appendix 3: Relations

We frequently used relations between nodes in our formalization. The most obvious application were the edges of a graph, but we also used them to relate nodes of different graphs to each other, such as the left and the right hand side of graph transformation rules, or the host graph before applying a transformation rule, and the result. Relations consist of pairs, which are a special case of ordered tuples.

Definition 23 (Tuple)**.** *Let $a_1, a'_1 \in A_1, \ldots, a_n, a'_n \in A_n$. Then a* tuple *over $A_1, \ldots, A_n$ is an object $(a_1, \ldots, a_n)$ such that $(a_1, \ldots, a_n) = (a'_1, \ldots, a'_n)$ if and only if $a_1 = a'_1, \ldots, a_n = a'_n$. We write $A_1 \times \cdots \times A_n$ for the set of tuples over $A_1, \ldots, A_n$.*

A relation consists of three sets, the domain, the codomain and a set of pairs. Each pair relates an element of the domain to an element of the codomain. It is admissible that elements of the domain and the codomain are not related in any way. Those elements from the domain that are related to some element of the codomain—not all need to be—form the domain of definition. One of the main uses of relations is to apply them to some subset of the domain, to get a subset of the codomain that contains any element that is related in any way to one of the elements of the domain subset.

We generalize this to sets of tuples whose components are elements of the domain, such that we can apply the relation to such a set, and obtain a set of tuples of the codomain elements, each tuple component of the result tuples being related to the respective tuple component of one of the input tuples. The main purpose of this generalization is to map edges from one graph to another, based on a relation of their nodes.

Definition 24 (Relation, Domain, Codomain, Domain of definition, application, image, inverse, equivalence class)**.** *A relation R over sets A and B is a subset $R \subseteq A \times B$. We call A the* domain *of R, denoted by* $\mathrm{dom}(R)$*, B the* codomain *of R, denoted by*

$\mathrm{codom}(R)$ *and* $\{a \in A \mid (a,b) \in R \text{ for some } b \in B\}$ *the* domain of definition *of* R*, denoted by* $\mathrm{def}(R)$*. The* application *of* R *to* $M \subseteq A^n$*, written as* $R(M)$*, is the set* $\{(b_1,\ldots,b_n) \in B^n \mid \exists (a_1,\ldots,a_n) \in M : \forall 1 \leq i \leq n : (a_i,b_i) \in R\}$*. We call* $R(A)$ *the* image *of* R*. We write* $[a]^R$ *for* $R(\{a\})$*. The* inverse R^{-1} *of a relation is the set* $\{(b,a) \in B \times A \mid (a,b) \in R\}$

A special case of a relation are functions. Functional relations relate ("map") each of its domain elements to at most one of its codomain elements. A total function, in addition, maps each of its domain elements to some codomain element.

Definition 25 (Function, Total function, Partial function)**.** *A* (partial) function $f : A \rightharpoonup B$ *is a relation over* A *and* B *such that there are no* $(a,b),(a,b') \in f$ *with* $b \neq b'$*. We write* $f(a) = b$ *for* $f(\{a\}) = \{b\}$ *and* $f(a) = \bot$ *for* $f(\{a\}) = \emptyset$*. We call* f *a* total function*, written* $f : A \to B$ *iff* $\mathrm{dom} f = \mathrm{def} f$*.*

Two relations can be composed into one. This yields a new relation that contains all pairs of elements that can be related to each other using the two relations via any third element. We will use composition to express the steps involved in application of graph transformation rules.

Definition 26 (Composition)**.** *Let* $R_1 \subseteq A \times B_1$ *and* $R_2 \subseteq B_2 \times C$ *be relations. Then the* composition *of* R_1 *and* R_2*, written* $R_1 \circ R_2$*, is the relation* $\{(a,c) \in A \times C \mid \exists b \in B_1 \cap B_2 : (a,b) \in R_1 \text{ and } (b,c) \in R_2\}$*.*

Partial orders are antisymmetric, transitive, reflexive binary relations and induce upper bounds. We will use the apparatus of least upper bounds on partially ordered sets for abstract interpretation.

Definition 27 (Partial order, upper bound, least upper bound)**.** *We call a relation* $\sqsubseteq$ *over* $A \times A$ *a* partial order *iff* $\{a \in A \mid a \sqsubseteq a\} \in \{\emptyset, A\}$*,* $(\sqsubseteq) \cap (\sqsubseteq)^{-1} = \{(a,a) \in A \times A\}$ *and* $(\sqsubseteq) \circ (\sqsubseteq) \subseteq (\sqsubseteq)$*. Let* $\sqsubseteq$ *be a partial order over* $A \times A$ *and* $a_1, a_2, b \in A$*. Then* b *is an upper bound of* a_1 *and* a_2 *iff* $a_1 \sqsubseteq b \sqsupseteq a_2$*.* b *is the* least upper bound *or* join *of* a_1 *and* a_2*, written* $b = a_1 \sqcup a_2$*, iff* b *is an upper bound and for every upper bound* $a \in A$*,* $b \sqsubseteq a$*.*

5 ASTRA: A tool for abstract interpretation of graph transformation systems

This chapter was initially published as [BR15b].

Abstract: We describe ASTRA (see `http://rw4.cs.uni-saarland.de/~rtc/astra/`), a tool for the static analysis of infinite-state graph transformation systems. It is based on abstract interpretation and implements cluster abstraction, i.e., it computes a finite overapproximation of the set of reachable graphs by decomposing them into small, overlapping clusters of nodes. While related tools lack support for negative application conditions, accept only a limited class of graph transformation systems, or suffer from state-space explosion on models with (even moderate) concurrency, ASTRA can cope with scenarios that combine these three challenges. Applications include parameterized verification and shape analysis of heap structures.

5.1 Introduction

Graph transformation is an intuitive formalism: One begins with a start graph and, by nondeterministic choice, matches and applies transformation rules to it, based on subgraph replacement. We are mainly interested in analysis of the graphs reachable by successive application of rules, to verify safety properties, for example.

One of the applications of graph transformation is modeling parameterized concurrent systems. Reasoning about such systems is hard because the state space is infinite. Hence, abstraction methods are required. In this paper, we present ASTRA, our tool for abstraction of graph transformation systems.

A number of tools are available that use abstract interpretation (each based on a different abstraction) to compute a finite over-approximation of the reachable graphs: AUGUR [KK08b] uses a Petri net based abstraction and had success with interesting examples of concurrent systems; it does not, however, support negative application conditions. `hiralysis` [Bau06] is based on partner abstraction. It does offer negative application conditions and can analyze some concurrent systems, but requires input grammars to satisfy some rather restrictive "friendliness" properties. GROOVE [Zam13] has an implementation of neighborhood abstraction, which has no such restriction, supports negative application conditions, but analysis of systems with concurrency leads to state space explosion.

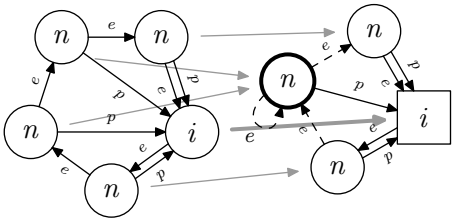


Figure 5.1: An example of how a cluster is obtained by abstracting the concrete graph with respect to one specific node (here, the i-labeled one). The tool lifts the application of graph transformation rules to this abstraction.

5.2 Cluster abstraction

Our tool, ASTRA, implements *cluster abstraction* [BR15a]: We consider each node in the graph (to become the *core node* of a cluster) plus its respective adjacent nodes (to become the *periphery*). We merge two or more adjacent nodes into summary nodes if both their labels and configuration (*spoke*) of edges to the core node are equal. If, by this summarization, two merged nodes disagree on to existence of some edge to a third node, we replace it by a $\frac{1}{2}$ edge. After summarization, we are left with clusters of bounded size, and we eliminate any duplicate cluster by assuming (as a further overapproximation) that there can be any number of concrete instances. An example is shown in Figure 5.1. The initial graph is abstracted in this way, and then rule application is lifted to the abstraction.

In this paper we describe ASTRA 2.0. An earlier version, ASTRA 1.0 [BR10a], implemented a less precise precursor to cluster abstraction that assumed all edges in the periphery to be $\frac{1}{2}$.

5.3 Architecture and Usage

ASTRA is a command-line program that expects a start graph and graph transformation rules as input and outputs the clusters from the analysis. When running the analysis, it abstracts the start graph, then enters its main loop. The main loop searches for abstract matches; each left hand side node of each rule is matched against the core node of any cluster from the current working set, and the remaining nodes are matched to a subset of the respective peripheral nodes. In addition, one further cluster with unmatched core node, but matched peripheral nodes is materialized. Those matches are then combined into a partial concretization, with several checks done to rule out cases where no full concretization exists. Not all such cases are detected by the tool; but the result is still a valid over-approximation.

All clusters produced by rule application are added to a temporary set. After each iteration, the tool then, optionally, applies a post-pass reduction step to the temporary set, inspecting it for clusters that can be eliminated or refined. To do this, the tool

searches for all partial materializations bounded to three material nodes: If a cluster cannot be used in any of them, it is eliminated, and if an edge is always present or always absent, peripheral $\frac{1}{2}$ constraints are refined. Finally, the temporary set is joined with the working set.

The tool indicates progress as it goes from rule to rule, and from iteration to iteration. After each iteration, the current working set is dumped to disk, which is useful for inspecting the current state of the analysis when running the tool on complex cases that take some time.

The main loop is executed iteratively until the working set remains unchanged, ie., a fixpoint has been reached. The tool then dumps the output to disk, prints statistics and exits.

5.3.1 Input file format

ASTRA uses the same ASCII-based input file format as `hiralysis` (see [Bau06] Fig. B.1, p. 160), extended by additional application conditions. For example, the constraint `partner(x1)=neg{(out,p)}` restricts rules to apply only if the node matched by `x1` has no outgoing edge with label `p`.

Consider the following toy case as a running example. The input:

```
nodelabels n,Error,i; edgelabels e,p;
empty; // start graph
create [{x1:n,x2:n,x3:i},
   {(x1,x2):e,(x2,x3):e,(x3,x1):e,(x1,x3):p,(x2,x3):p}];// init
rule [{x1:i,x2:n},{(x1,x2):e}],                      // insert
     [{x1:i,x2:n,x3:n},{(x1,x3):e,(x3,x2):e,(x3,x1):p}];
rule [{x1:n},{},partner(x1)=neg{(out,p)}], [{x1:n,x2:Error},{}];
```

This example models singly-linked ring buffers into which an unbounded number of nodes are inserted dynamically. One special node is indicated with the label `i`. New nodes are inserted next to it with a back pointer. Here, we want to use astra to verify the safety property that each node has such a back pointer. We achieve this with the second rule. It uses a negative application condition to generate an error label if a node lacks the back pointer.

As can be seen, the input file format is mainly based on graphs, which are sets of node names, each with a label, and sets of edges (the name being a pair of node names), each with an edge label. The rules specify the subgraph to be replaced and the subgraph by which it is replaced. The node names imply a mapping from the left hand side to the right hand side.

5.3.2 Command-line interface

For our case study, consider the following tool run:

```
$ ./astra -Os -Op test023.gts
 0 [  2/  2] = 100% [+2, +-2]
```

```
 1 [  2/  2] = 100% [+1, +-1]
 2 [  2/  2] = 100% [+0, +-0]
done.
6 clusters, 5 matches, 1 active rules,
6 rule applications, 2 iterations
```

The `$` indicates the shell prompt; the remaining line is entered by the tool user. In this case, ASTRA is run on the input file of our running example (`test023.gts`).

In the example, we specify analysis options `-Os` and `-Op`, instructing ASTRA to apply a simple peripheral constraint satisfiability check and post-pass reduction, respectively. For our experiments, this proved to be the most practical option set, providing the best speed/precision trade-off. Removing one of the two options lead to drastic decrease in precision, while adding any other lead to merely minuscule gains. Only in specific cases where the analysis would otherwise run into state-space explosion, further analysis options were useful.

Option `-n` can be used to specify a cutoff iteration after which to prematurely terminate the analysis. This is useful to inspect the intermediate result. Run ASTRA without arguments for further details about the available options.

5.3.3 Status report

For each iteration, while running, the current iteration number, current rule, total number of rules and progress (current rule divided by total number of rules) is printed. After finishing the iteration, the number of clusters added and modified (i.e., with peripheral constraints weakened) is printed. Note that clusters added by the initial graph and by rules with empty left hand side are only accounted for in the final statistics printed after the fixpoint has been reached.

5.3.4 Output file formats

ASTRA supports DOT (as used by the graph layout tool Graphviz), GML (as used by OGDF and the GoVisual Diagram Editor, respectively), GDL (as used by VCG and its successor aiSee) and GraphML (as used by yEd and yComp, respectively). In addition, the tool supports its own native output format that is similar to the input format.

The output can be loaded or processed with any tool supporting any of those formats. The most common use will be a graph layout tool to inspect the output, but it can as well provide invariants for other analyses, like `hiralysis` [BSTW06].

For our running example, the tool outputs six clusters, visualized in Figure Figure 5.2. In addition to the full analysis, we show the intermediate results obtained by using option `-n`.

These drawings were done by METAPOST, based on an experimental output module built into ASTRA that does primitive circular graph drawing. For common use, aiSee and yEd have proven most useful, especially the organic and hierarchical layout engines.

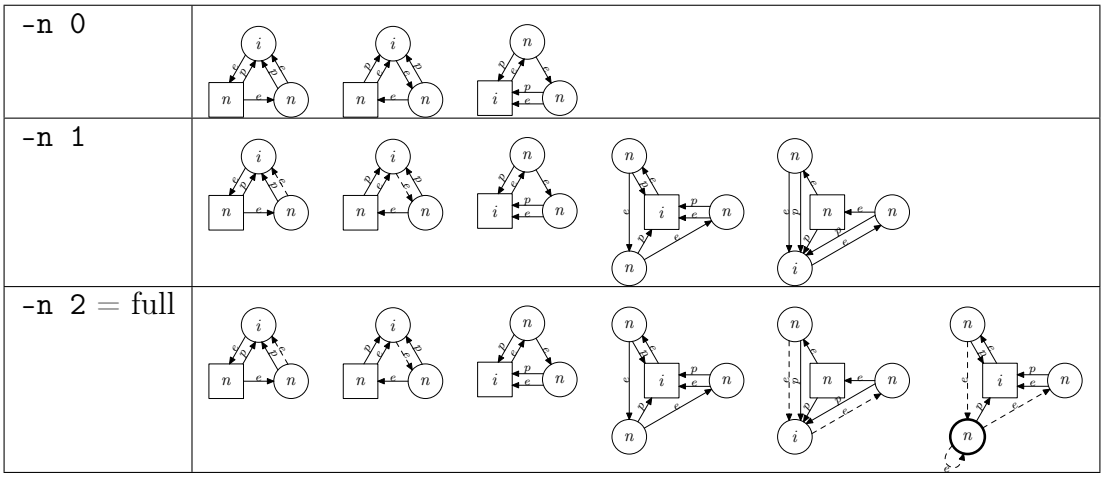


Figure 5.2: Analysis results on running example.

5.4 Experimental Evaluation

We already ran the tool on various test cases from the literature in [BR15a], including AVL trees, red-black trees, firewalls, public/private servers, dining philosophers, resources, mutual exclusion, singly-linked lists, circular buffers, Euler walks, and the merge protocol. The merge protocol, our main example, is a distributed car platooning coordination protocol that establishes a logical communication hierarchy on top of the physical communication medium. Analysis of the protocol is hard because of its massively distributed nature, caused by the vast range of topological configurations that may evolve concurrently.

However, all inputs from that case study were written by hand. To demonstrate the robustness of our tool, we apply it to graph transformation systems generated automatically from higher level models of the merge protocol, specified in the DCS formalism [Rak06, BTW07]. We used the tool `dcs2gts` [Bac07] to translate the DCS models into graph transformation systems suitable for analysis with ASTRA. We include two new variants, follower-controlled merge.

Synchronous (leader-controlled) merge in our former case study consisted of 402 rules (plus 3 for checking safety properties), the asynchronous version 313 (plus 2). The large number is caused by the fact that many rules are generated from templates that iterate over all node labels. The automatically generated versions use 788 and 835 rules, respectively. In contrast, the number of clusters in the analysis result increased from 873 to 22509 (factor 26) and from 3069 to 142326 (factor 46). This is because the automatically generated version uses intermediate steps to model topology changes. While those steps are serialized by special labels, and thus pose no combinatorial challenge, our analysis shows that the tool does well with all those intermediate configurations absent in the manually created inputs. See Table 5.1 for the full results.

Table 5.1: Benchmark analysis statistics. cl. = clusters, m. = abstract matches, rule app. = rule applications, it. = iterations.

Benchmark	# cl.	# m.	# rule app.	# it.	time	
Sync., leader-controlled	22509	75359	36685213	135	9m	34.200s
Sync., follower-controlled	24957	82569	43679468	144	22m	30.200s
Async., leader-controlled	142326	850889	1006759383	202	13136m	1.260s
Async., follower-controlled	58023	296310	83499253	157	3972m	37.560s

5.5 Conclusions and Future Work

We have seen how ASTRA can be used to analyze a simple graph transformation system, modeling insertion of elements into ring buffers. In contrast to related tools, it is not restricted to graph transformation systems of a special form, it supports negative application conditions and it does well when facing models involving concurrency. Our experimental evaluation showed that it is capable of handling very complex inputs generated automatically from higher-level specifications.

Future work: Our tool already has experimental support for generating an abstract labeled transition system of clusters, but the theory for actually using those with a model checker has still to be worked out. We would also like to provide more powerful application conditions, in particular non-existence of edges between two specific nodes and restrictions on the periphery of a node.

A promising way to considerably speed up analysis is parallelization. The structure of the analysis is very well suited for this and we expect a parallelized version to scale almost linearly.

6 Further notes on related work

6.1 Graph transformation framework

6.1.1 Relation to the algebraic approach

The algebraic approach aims at a graph transformation framework where transformation rules and application conditions are reducible to constructions based in category theory. The goal is to obtain simple proofs of rewriting properties, at the cost of weak operational features.

The two major algebraic frameworks are the single push-out (SPO) and the double push-out (DPO) approach. In essence, the basic SPO approach models rules without any application conditions, while DPO uses a second push-out to model a mandatory, built-in application condition in purely categorial terms. (The dangling condition and the identification condition: nodes may only be deleted if no incident edges are left; and nodes and edges may be matched twice only if they are preserved by the rule.) Restricting matches to be injective can be understood as a further application condition; the combination with non-injective morphisms connecting the left and the right hand side to the interface graph in DPO has been extensively studied in [HMP01] (where the authors make an interesting historical note to the effect that the original DPO paper required matches to be injective, while most more recent papers considered arbitrary matches only). For SPO, injectivity restrictions for matches are one of the application conditions discussed in [HHT96].

[HP02] extends DPO with relabeling. No SPO variant with relabeling exists. Edge replacement is used instead: If an edge label is to be changed, the edge will be deleted and a new one with the desired label will be created. Nodes are left unlabeled, and self-loops carry the desired label, with label changes done as for any other edge (cf. [EHK$^+$97, Section 5.1fn]).

The theorems that can then be proven using category theory state under which conditions certain properties hold: *Local Church-Rosser* and the *parallelism theorem* are relevant for the reinterpretation of transition systems that are based on an interleaved model of parallelism where rules are always applied one after another. Assuming instead a truly parallel computation semantics, where rules can also be applied in parallel (see [CMR$^+$96, Section 5.1]), derivations that are shift-equivalent can be reinterpreted as one single parallel derivation.

Embedding of derivations and *derived productions* are concerned with a subgraph that evolves independently from the rest over a sequence of derivation steps. In this case, assuming a suitable model of computation, the sequence can be be understood as a single

computation step that produces the effect of the sequence atomically.

Amalgamation and *distribution* apply to overlapping graphs that evolve separately except for synchronization via a common subgraph. Then the transition system that describes the global behavior can be split into separate parts that each cover local behavior only.

Like its ancestor in [Bau06], our graph transformation framework is, "in disguise," a subset of the SPO approach with application conditions from [HHT96]. While [Bau06, Section 2.3] claims a list of differences, all of them turn out to merely curtail that approach (cf. loc. cit., last two items not discussed here since they do not suggest the opposite):

(a) In unrestricted SPO, graphs may have multiple edges with the same label between them, which is not the case in our framework. This restriction can be formulated in terms of SPO: (1) Disallow such graphs as the start graph as well as as the left and right hand side of transformation rules. (2) Duplicate each rule that adds an edge between two existing nodes, one for the case that such an edge (with matching endpoints and label) does not already exist in the host graph (this can be expressed with a negative application condition such that the edge is added if it does not already exists), and one for the case that it already exists (this can be expressed by adding the edge between those two nodes to the left hand side such that no further edge is added if one already exists).

(b) While SPO uses a partial morphism for h, our approach merely uses a partial mapping, suggesting that it extends SPO with relabeling capabilities. However, this is equivalent to SPO having unlabeled nodes and self-loops carrying the node label instead, with the start graph and the transformation rules changed accordingly.

(c) While SPO uses categorial constructions, our approach is based on constructive definitions. However, this is merely for the sake of clarity; it is still equivalent to the categorial construction, and the respective theorems do apply.

(d) Our framework avoids the identification issue by injective matches. However, this is merely making an SPO application condition mandatory. Plus, it is not a real restriction, since non-injective matches can be taken care of by adding rules, one per possibility to identify sets of nodes on the left hand side. The identification condition can then be handled directly on the rules, and the rule be changed to delete nodes that were deleted by at least one of the identified nodes by the original rule.

In principle, it is possible to handle DPO with our abstraction: Since the identification condition can be checked directly on the rules, those rules for which it does not hold can be deleted. Further, checking the dangling condition can be lifted to the abstract, since each matched core node of a cluster also contains each incident edge.

Further, it seems quite possible to lift the algebraic considerations to the abstract (the following assumes DPO, but can be adopted for SPO):

(a) Parallelism: Check whether all possible matches of the two rules to a cluster, with at least one node of the cluster matched by both, overlap for a node deleted by at least one of those rules. If not so, parallelization is permitted in the abstract, since it is permitted for each possible concretization.

(b) Embedding: In a straight-forward sense, each cluster is already embedded into the concrete graph. Thus, a sequence of clusters evolving into each other linearly, with exactly one transition in between and no additional incoming or outgoing transitions, can be identified as one single, atomic rule application if no other transition can be taken in other parts of the graph. This requires checking whether for each cluster in the sequence, only the transition's rule matches, in only one way, each admissible global graph (induced by the set of clusters, and the cluster under consideration).

(c) Distribution: The three set of rules (two for subsystems and one for the interface) induce three separate sets of clusters and transitions, and respective morphisms among them. Using this information, check the distributed gluing condition and the connection condition in the abstract. If it holds, find one single unified set of clusters and transitions for the amalgamated rules without analyzing the whole system. This is done by merging the clusters of the two subsystems with respect to their spokes, as implied by the morphism from the interface graph. It is practical to restrict the interface clusters to have cluster count of at most one, for otherwise merging becomes too complex.

These algebraic considerations are orthogonal to cluster abstraction itself. The most obvious real value of parallelism and embedding is a more concise representation of the system evolution. Distribution, in contrast, also provides a basis for the separate design and analysis of the subsystems of a distributed system. It may, as a side-effect, make the analysis more precise, since clusters on opposite sides of the interface are kept separate that would be identified in an analysis of the whole system (cf. the firewall example). On the other hand, with cluster abstraction, distribution loses a lot of its original motivation, since a cluster abstraction analysis of the whole system is already a very fine-grained decomposition of the original system.

6.1.2 Relation to the algorithmic approach

The algorithmic approach tries to make the framework *operationally* powerful, well-suited for application, and allowing the specification of embedding relationships, control flow and priorities for rule application (see [Sch97]).

Rule priorities tell the graph transformation engine which rule to apply if several of them match the same graph. Cluster abstraction does not preserve the information necessary for rule priorities. To ensure that a rule with lower priority cannot be applied, we would have to know that a rule with higher priority can be applied; however, an abstract match does not guarantee a concrete one. Thus, we would have to consider

all possibilities anyway, except in the most trivial cases (where all matched clusters are known to materialize in the concrete graph).

Control flow can be done by annotating each cluster with a set of possible control flow locations, such that the cluster can only be matched if the respective rule is active at that location. To do control flow with already the existing approach, the start graph can be amended by a node that is connected to every other node. Then rules can also have such a node, connected to every other node on the left and right hand side, that is matched and changed according to the desired control flow. As a beneficial side effect, it keeps clusters at different control flow locations separate from each other.

The embedding relationship specifies how the unmatched edges between the matched nodes and their unmatched adjacent nodes (the context) are connected to the nodes of the right hand side. This can be done easily with cluster abstraction, since the clusters just happen to contain the information about this context. With respect to the comparison criteria from [Sch97, table 1], it should be possible to have all crucial features: changing of label and orientation, context label and edges test, and different treatment of identically labeled nodes.

6.2 Related abstractions

6.2.1 Partner abstraction

Partner abstraction [Bau06, BW07] abstracts a graph by splitting it into its connected components, merging—separately for each component—nodes with the same canonical name into summary nodes, and deleting any isomorphic duplicates among the resulting abstract components. As the canonical names of a node, it considers its label plus, for each incident edge, its edge label, direction and label of the corresponding adjacent neighborhood node. (NB: This is per edge, not per spoke.) For a class of "friendly" systems, partner abstraction is sound and complete. Hence, there is a guarantee for each abstract state and abstract transition to exist in the concrete system as well, and thus, abstract model checking, including liveness, is not an issue. Friendly systems are those where none of the abstract components contains an edge with a summary node on both ends. Unfortunately, this class is weak and most systems do not belong to it. In practice, partner abstraction requires systems to be friendly, since it runs into state-space explosion otherwise [Bac08]. As the merge protocol is not a friendly system, it cannot be analyzed with partner abstraction, only a simplified version where processes know each other's state.

Bauer presents counting of abstract components as an unimplemented extension and sketches generalized abstract components where nodes from different components can be connected by edges [Bau06, Section 4.6]. The counting yields a finite upper bound, but not information on which components may not occur together with others. However, the construction could be extended along the same lines as cluster counts for cluster abstraction (see Section 7.2).

To improve upon partner abstraction and to analyze the uncrippled merge protocol

was the original motivation for cluster abstraction. This is the reason for why cluster abstraction uses the same graph model as partner abstraction and the ASTRA implementation accepts the same input file format as hiralysis, which is the implementation of partner abstraction. ASTRA does well with all examples shipped with hiralysis, so it can be seen as the legitimate successor. Note that the term *cluster* is also used in partner abstraction, as the name for what we called abstract components above (which would have been a better fitting term from the outset). Cluster abstraction can be almost fully understood as a modification of the definition of the cluster notion, bringing it closer to the intuitive idea that it suggests.

6.2.2 Neighborhood abstraction and pattern abstraction

Distefano and Rensink [RD06] consider an abstraction based on an idea similar to partner abstraction. Nodes are summarized if the set of labels of the outgoing edges and the multiplicities (abstracted counts) of the incoming edges agree. Graphs are required to be deterministic (a node must not have two outgoing edges with the same label), nodes are unlabeled, and rule application mandates the dangling edge condition. The latter ensures that only matched nodes can change their canonical name, and no friendliness restriction (other than those mentioned) is necessary.

Neighborhood abstraction [BBKR08, BKK+12] combines ideas from partner abstraction and Distefano and Rensink's abstraction. Nodes are grouped into hierarchical equivalence classes: The equivalence class at the lowest level contains nodes with the same label, on higher levels, additionally, the multiplicities of edge connections to nodes from the predecessor level equivalence classes are taken into account (this causes node labels of the neighborhood to be taken into account implicitly). Hence, the equivalence classes at level i depend on each node's radius i neighborhood, that is, nodes and edges reachable via a path of at most i edges. The authors claim to be able to preserve and reflect properties for a modal logic (op. cit., Section 3.2), but they define reflection such that it relates one graph and its abstraction instead of the all reachable graphs and the abstract result of their method. Thus, their reflection properties do not quite come as much as a surprise as they claim.

The abstraction has been implemented [RZ10, RZ11] as an extension named ShapeGenerator for the tool GROOVE. It comes with a claim to the effect that liveness properties can be proven ("we can check, for example, that the following properties hold: ... (iii) rule get is applied infinitely often. ... (iii) is a liveness property. ... Since these properties hold in the abstract LTS, we can then conclude that they also hold in the infinite concrete state space." [RZ10, Section 4] The authors provide no explanation for how this conclusion could be admissible, given that their abstract labeled transition system is an over-approximation, which means that the existence of transitions in the abstract does not guarantee the existence in the concrete. In fact, their liveness property does not hold in the concrete to begin with, and thus of course not in the abstract: State s_3 of Figure 7 contains a self-loop for the transition put, which can be executed infinitely often once the state has been reached. This sequence even exists in the concrete; the left hand side of rule put matches its own right hand side as well as the start graph, thus, the rule

can be executed infinitely often.) Experimental evaluation [Zam13, Chapter 7] analyzes linked lists, a circular buffer, Euler walks and the firewall example and fails at the simplified merge protocol, the reason for which seems to be the lack of the component-wise abstraction that partner abstraction has. The author concludes that "the experiments ... can all be still considered toy models; ... performance was unfortunately sub-par [compared to] AUGUR and `hiralysis`. ... we believe that some of this inefficiency lies in the abstraction method itself, in particular ... the ... abstraction relation .. defined over a radius i ... is too coarse ... Usually, ... we want to look at different radii for different types of edges. This insight was the motivation behind ... *pattern abstraction*" (op. cit., p. 106). This conclusion is perhaps a tad too negative, given that none of those toy models was a "friendly" one doable with hiralysis; and in supporting edge multiplicities and being able to prove the Euler walk property, they are more advanced than ASTRA.

Pattern shape abstraction [RZ12] is based on the concept of pattern graphs, which expand the concrete graphs into layered, pattern-labeled graphs. Each such pattern is a subgraph of the concrete graph and in each layer, the number of subgraph edges increases: On the first layer, each pattern is a single node from the host graph; on the second layer, a single edge plus its end nodes, and from the third layer, the patterns are proper supergraphs of the patterns from the lower layer. The patterns of each layer connect to patterns of the next layer via morphisms. The abstraction works in two steps: (1) It discards any pattern not admitted by a pattern type graph (which contains a finite set of patterns that are allowed). This allows fine-tuning the abstraction, which is its main selling point. (2) It builds equivalence classes of nodes of the pattern graph and summarizes them using counter abstraction. This yields a bounded abstraction, pattern shapes. Rule application is then lifted to this abstraction. Since the abstraction has not been implemented yet, it is unclear how it performs in practice. Pattern shape abstraction seems to be a generalization of the earlier daisy pattern abstraction [BBR09], where the core idea happens to be similar to cluster abstraction: Consider each node in the concrete graph, its neighborhood nodes, and the edges in between, and then further approximate so as to obtain a finite abstraction.

6.2.3 Petri graph abstraction

The Petri graph [BCK01, BK02, BCK08] method is based on approximations of the unfolding semantics of hypergraph transformation systems. The unfolding semantics is a Petri graph, a hybrid between a hypergraph and a Petri net, with nodes and edges of the former being the places of the latter. It is constructed as follows: Take the start hypergraph and mark all nodes, yielding a Petri graph. Then, perform the following step infinitely often: Find all matches into the current Petri graph and add the nodes and edges to be added by the rule, but keep the nodes and edges to be deleted. Finally, add a transition to the net that consumes the markings from the deleted nodes and edges, consumes and produces markings for the kept nodes and edges and produces markings for the added nodes and edges. The basic idea for obtaining a finite overapproximation, called k-covering of the unfolding semantics, is to merge occurrences of the left hand side in the Petri graph that are reachable by a depth of more than k rule applications.

Conversely, an underapproximation, the k-truncation, can be obtained by truncating the unfolding process after a depth of k rule applications. The reachable graphs of the hypergraph transformation system are the reachable markings of the unfolding semantics. Thus, the k-covering corresponds to a superset and the k-truncation to a subset of the reachable graphs. There is also support for model checking [BCKL07]: Using the k-covering, safety properties can be checked, and the k-truncation may be able to check liveness properties, though no proof of concept seems to exist.

The method has been implemented in the tool AUGUR [KK05, KK08b], with which the authors were able to analyze a number of examples (see [KK08a, Section 7] for a list). Where it performs better than ASTRA is the firewall example with locations on both sides having the same label. A weakness is that AUGUR does not support negative application conditions, as it would require inhibitor arcs in the Petri graph [KK05, Section 2], which would make reachability undecidable in general. An interesting extension is their support for attributed hypergraphs [KK08c]. It should be possible to add this feature to ASTRA in a similar way.

It is an interesting question whether ideas from the Petri graph approach and cluster abstraction could be combined. In particular, it may be interesting to investigate whether cluster Petri graphs could work, where cluster are places.

6.2.4 Canonical abstraction

Canonical abstraction [SRW99, SRW02] was originally proposed as a shape analysis method for heaps. It is based on logical predicates that express unary properties of and binary relations among the heap objects, in particular stack references, points-to and reachability. The properties can be interpreted as node and edge attributes, allowing the heap to be represented as a graph. The abstraction summarizes objects if they have the same canonical name (which consists of the values of the finite number of properties). Binary predicates are then abstracted using three-valued logic (which avoids the problems that partner abstraction has in non-friendly cases), quite similar to how cluster abstraction handles the peripheral edges. The abstraction can be parameterized with user-specified predicates; the analysis then lifts them to the abstract.

Canonical abstraction has been implemented in the tool TVLA [LARSW00]. Extensions of the tool allow decomposition of the heap into (weakly-)connected components [MBC+07], the same idea as used in partner abstraction, or into a finite number of overlapping components according to location selection predicates [MLAS+08]. TVLA works well with single-threaded programs, but shape analysis of programs with an unbounded number of threads manipulating an unbounded shared heap leads to state-space explosion even for trivial cases unless TVLA is extended to thread-quantified invariants [BLAM+08]. However, this still requires modeling each direct relationship between threads (which is our setting) over the shared heap. Implementing cluster abstraction with basic TVLA seems not to be feasible either, as the number of abstraction predicates needed would be exponential in the number of edge labels.

With an extension, TVLA outputs a transition system that can be used for analysis of safety [Yah01, Yah13] and liveness [YRSW03] properties, although the latter has met

some skepticism [Wac05, Section 5.5].

As the shapes of canonical abstraction are graphs, it seems natural to use the tool for analysis of graph transformation systems. A front-end [SWW10] allows the specification of graph transformation rules in TVLA instead of actions.

6.2.5 Counter and environment abstraction

Counter abstraction [PXZ02] and environment abstraction [CTV06] are directly motivated by model checking and capture only a minimum of necessary information about the system structure. Counter abstraction abstracts the state of the system as a list of multiplicities that count the number of processes in each of the possible process states. Environment abstraction supports relations among processes and takes a "Ptolemaic" view of the system, abstracting it as the state of one specific process and how it relates to its "environment", that is, for each of the relations in the system, a list of states of the other processes to which it holds. The idea is very close to cluster abstraction. One of its benefits is the ability to treat different relations independently from each other, thus reducing the state space, while cluster abstraction preserves the exact set of edges among the core and the periphery (environment abstraction can still do that, if one uses one relation per set of edges, so it is more powerful). On the other hand, in contrast to cluster abstraction, environment abstraction preserves no information about the relations among processes in the environment. The approaches can prove safety and liveness properties of the Szymanski and Lamport's Bakery algorithms.

6.2.6 Other related abstractions

Cherem and Rugina [CR07] propose a local abstraction of heaps that tracks for each heap cell reference counts of that cell, reference counts for the neighbor cells, and the points-to relations between the cell and its neighbors. They prove invariants of doubly-linked lists. Again, no relations among neighbors are captured by the abstraction.

Ideal abstraction [WZH10, ZWH12] overapproximates the reachable states of depth-bounded systems, the class of well-structured transition systems (WSTS) with a bound on the length of the longest acyclic path. In fact, this class of systems seems to be a superset of the friendly systems of Bauer, where not only nodes, but entire structures can be summarized. Based on the same idea, fair termination can be proven [BKWZ13]. Cluster abstraction, in contrast, can analyze systems that do are not depth-bounded, and the merge protocol is one of them, unless one uses a simplification like the one that partner abstraction can do. Further, cluster abstraction is well-suited for depth-bounded systems, since the overlap can preserve the exact paths if they are bounded by a length. On the other hand, cluster abstraction cannot preserve cycles of length bounded by more than 3.

Saksena et al. use symbolic backward reachability analysis to check for undesirable global configurations of the topology. Their method is not guaranteed to terminate, but capable of proving loop freedom of an ad hoc routing protocol.

König and Stückrath [KS12] study a subclass of graph transformation systems with negative application conditions that are well-structured with respect to the minor ordering. Starting with a bad graph, they check by backwards search whether the start graph is reachable. The method does not terminate for all types of negative application conditions.

Rieger and Noll [RN08] analyze heaps by backward hyperedge replacement. They introduce an upper bound on the size of the heap, beyond which a sink hole represents an arbitrary subgraph.

Meyer [Mey08] analyzes a class of pi calculus systems by translating them to Petri nets. The class is characterized by exhibiting only finitely many patterns of connections at runtime.

Steenken et al. [SWW11] annotate abstract shape graphs with additional shape constraints to obtain a sound and complete "abstraction". Their analysis does not necessarily terminate. It is unclear whether infinite-state graph transformation systems can be evaluated.

Venet [Ven98] analyzes, by abstract interpretation, unbounded topologies for a class of friendly pi calculus systems.

Feret [Fer00, Fer05] analyzes, by abstract interpretation, whether private information on mobile systems in hostile contexts, as specified in pi calculus, can be communicated only to authorized agents. It is unclear whether the method has been tested in practice.

7 Conclusions

7.1 Summary of contributions

This thesis presented the theory of *cluster abstraction*, as well as ASTRA, its implementation. Practical evaluation showed that cluster abstraction is precise enough to analyze several versions of the merge protocol, a protocol with typical features of distributed message-passing systems that consist of an unbounded number of processes. Various benchmarks from the literature could be analyzed as well with the method. A less precise variant of cluster abstraction, *star abstraction*, could already analyze a significant part of the merge protocol, though not the entire protocol. The ASTRA implementation is the first tool for abstract graph transformation that is not subject to friendliness criteria, that supports negative application conditions and that can cope with unbounded concurrent processes at the same time.

Cluster abstraction abstracts a graph by considering, for each node, its immediate neighborhood and relationships among the neighbors. The success of this abstraction is to be explained by the fact that human designers follow a "Ptolemaic" approach, by taking the view of the objects under consideration and how they relate to their immediate neighbors, but not further.

7.2 Future work

7.2.1 Cluster count

It should often be possible to preserve, for each cluster, whether each concrete graph has at most one node with local abstraction being (or subsumed by) that cluster; i.e., whether the cluster is unique. Initially, that property can be obtained for nodes in the start graph. (Note that, with the naive abstraction, the start graph was treated the same as the right hand side of a rule with empty left hand side.) Each abstract rule application has to check for preservation of the property; i.e., if a unique cluster is modified by rule application, the modified cluster can, under some circumstances, also be assumed unique. In particular, the abstract transition system must not have incoming edges which are endpoints of paths from two different source clusters. The hardest part is to figure out whether a cluster modified in the periphery is unique, but it is possible to take care about that if the rules ensure that the core label is unique.

7.2.2 Model checking

The non-existence of a subgraph in all reachable graphs can be checked by adding rules with the left hand side being the subgraph and the right hand side adding a node with an error label. Temporal safety can be checked by keeping track of the cluster evolution, yielding an abstract labeled transition system with clusters as states and rule applications as transitions. The basic idea for safety is: If a certain state cannot be reached in the overapproximation, then it cannot be reached in the concrete. Interestingly, a similar idea allows liveness to be checked: If a certain state is reached in all paths even in the overapproximation, then it must also be reached in the concrete. For this, we need to assume compassion and justice properties (see [CTV06] for more details), and ensure that abstract loops are executed only a bounded number of times. In particular, the properties ensure that a rule that is always enabled (can always be applied) *must* be applied after a finite number of steps. Further, if a rule changes exactly one label in a loop, then it can be executed only a bounded number of times, since the number of processes is bounded. Finally, we need to ensure local progress, which can be done by adding complementary rules doing busy wait. Rules with empty left-hand side guarantee existence.

7.2.3 Closure

It is an interesting question and important to compute the best abstract transformer whether, given a set of clusters, a concrete graph exists where local abstraction for some node yields a certain cluster, that is, whether a "closure" exists for that cluster. Reineke and Puzhay have found out that this problem is decidable for cluster sets where all periphery is $\frac{1}{2}$, since it can be reduced to a set of linear Diophantine equations. If the condition does not hold, the problem seems to be undecidable, since it can be reduced to Post's correspondence problem. However, decidability is still given if we take into account cluster multiplicity, since the number of cases is bounded where clusters have bounded materialization. Clusters with constraints other than $\frac{1}{2}$ can be taken partly into account, by annotating the spokes with the set of possible connecting clusters, according to a bounded closure check.

Bibliography

[Bac07] Peter Backes. *dcs2gts – An interface between XML-coded DCS protocols and the hiralysis representation of graph transformation grammars*. Fopra report, Saarland University, January 2007. 53

[Bac08] Peter Backes. Topology analysis of dynamic communication systems. Diploma thesis, Saarland University, March 2008. 15, 35, 36, 58

[Bau06] Jörg Bauer. *Analysis of Communication Topologies by Partner Abstraction*. PhD thesis, Saarland University, 2006. 15, 49, 51, 56, 58

[BBKR08] Jörg Bauer, Iovka Boneva, Marcos Kurbán, and Arend Rensink. A Modal-Logic Based Graph Abstraction. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT 2008*, volume 5214 of *LNCS*, pages 321–335. Springer Berlin Heidelberg, 2008. 59

[BBR09] Jörg Bauer, Iovka Boneva, and Arend Rensink. Graph abstraction by daisy patterns. Privately circulated, May 2009. 36, 60

[BCK01] Paolo Baldan, Andrea Corradini, and Barbara König. A Static Analysis Technique for Graph Transformation Systems. In Kim G. Larsen and Mogens Nielsen, editors, *CONCUR 2001*, volume 2154 of *LNCS*, pages 381–395. Springer Berlin Heidelberg, January 2001. 60

[BCK08] Paolo Baldan, Andrea Corradini, and Barbara König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, 206(7):869–907, July 2008. 60

[BCKL07] Paolo Baldan, Andrea Corradini, Barbara König, and Alberto Lluch Lafuente. A Temporal Graph Logic for Verification of Graph Transformation Systems. In José Luiz Fiadeiro and Pierre-Yves Schobbens, editors, *WADT 2006*, volume 4409 of *LNCS*, pages 1–20. Springer Berlin Heidelberg, January 2007. 61

[BK02] Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *ICGT 2002*, volume 2505 of *LNCS*, pages 14–29, January 2002. 36, 60

[BKK+12] Iovka Boneva, Jörg Kreiker, Marcos Kurbán, Arend Rensink, and Eduardo Zambon. Graph abstraction and abstract graph transformations (amended version). Technical Report TR-CTIT-12-26, University of Twente, Enschede, the Netherlands, October 2012. 36, 59

[BKWZ13] Kshitij Bansal, Eric Koskinen, Thomas Wies, and Damien Zufferey. Structural counter abstraction. In Nir Piterman and Scott A. Smolka, editors, *TACAS 2013*, volume 7795 of *LNCS*, pages 62–77, 2013. 62

[BLAM+08] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Thread quantification for concurrent shape analysis. In Aarti Gupta and Sharad Malik, editors, *CAV 2008*, volume 5123 of *LNCS*, pages 399–413, 2008. 37, 61

[BR10a] Peter Backes and Jan Reineke. Abstract topology analysis of the join phase of the merge protocol [using astra]. In *TTC 2010*, volume WP10-03 of *CTIT Workshop Proceedings*, pages 127–133, Enschede, 2010. University of Twente. 2, 15, 34, 36, 50

[BR10b] Peter Backes and Jan Reineke. A graph transformation case study for the topology analysis of dynamic communication systems. In *TTC 2010*, volume WP10-03 of *CTIT Workshop Proceedings*, pages 107–118, Enschede, 2010. University of Twente. 2, 3, 25, 34

[BR15a] Peter Backes and Jan Reineke. Analysis of Infinite-State Graph Transformation Systems by Cluster Abstraction. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *VMCAI 2015*, volume 8931 of *LNCS*, pages 135–152. Springer Berlin Heidelberg, January 2015. 2, 21, 50, 53

[BR15b] Peter Backes and Jan Reineke. ASTRA : A tool for abstract interpretation of graph transformation systems. In *SPIN 2015*, volume 9232 of *LNCS*, 2015. 2, 49

[BRKB07] I. B Boneva, A. Rensink, M. E Kurban, and J. Bauer. Graph Abstraction and Abstract Graph Transformation. Internal report, University of Twente, 2007. 4

[BSTW06] Jörg Bauer, Ina Schaefer, Tobe Toben, and Bernd Westphal. Specification and Verification of Dynamic Communication Systems. In *ACSD 2006*, pages 189–200, Los Alamitos, CA, USA, 2006. IEEE Computer Society. 5, 52

[BTW07] Jörg Bauer, Tobe Toben, and Bernd Westphal. Mind the Shapes: Abstraction Refinement via Topology Invariants. Reports of SFB/TR 14 AVACS 22, SFB/TR 14 AVACS, June 2007. 53

[BW07] Jörg Bauer and Reinhard Wilhelm. Static analysis of dynamic communication systems by partner abstraction. In Hanne Riis Nielson and Gilberto

Filé, editors, *SAS 2007*, volume 4634 of *LNCS*, pages 249–264, 2007. 3, 15, 36, 58

[CMR+96] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach. In Grzegorz Rozenberg, editor, *Handbook of graph grammars and computing by graph transformation*, volume 1, pages 163–246. World Scientific, 1996. 55

[CR07] Sigmund Cherem and Radu Rugina. Maintaining doubly-linked list invariants in shape analysis with local reasoning. In Byron Cook and Andreas Podelski, editors, *VMCAI 2007*, volume 4349 of *LNCS*, pages 234–250, 2007. 36, 62

[CTV06] Edmund Clarke, Muralidhar Talupur, and Helmut Veith. Environment abstraction for parameterized verification. In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI 2006*, volume 3855 of *LNCS*, pages 126–141, 2006. 36, 62, 66

[EHK+97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In Grzegorz Rozenberg, editor, *Handbook of graph grammars and computing by graph transformation*, volume 1, pages 247–312. World Scientific, 1997. 55

[Fer00] Jérôme Feret. Confidentiality Analysis of Mobile Systems. In Jens Palsberg, editor, *SAS 2000*, volume 1824, pages 135–154, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. 63

[Fer05] Jérôme Feret. *Analysis of mobile systems by abstract interpretation*. PhD, École Polytechnique, Paris, February 2005. 63

[HESV91] Ann Hsu, Farokh Eskafi, Sonia Sachs, and Pravin Varaiya. Design of platoon maneuver protocols for IVHS. Technical report, Institute of Transportation Studies, UC Berkeley, 1991. 3, 25

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3):287–313, January 1996. 55, 56

[HMP01] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(05):637–688, October 2001. 55

[HP02] Annegret Habel and Detlef Plump. Relabelling in Graph Transformation. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz

Rozenberg, editors, *Graph Transformation*, volume 2505 of *LNCS*, pages 135–147. Springer Berlin Heidelberg, 2002. 55

[KK05] Barbara König and Vitali Kozioura. Augur—a tool for the analysis of graph transformation systems. *EATCS Bulletin*, 87:125–137, 2005. 61

[KK08a] Vitaly Kozyura and Barbara König. *Augur 2—A tool for the analysis of (attributed) graph transformation systems using approximative unfolding techniques*, April 2008. 35, 61

[KK08b] Barbara König and Vitali Kozioura. Augur 2—a new version of a tool for the analysis of graph transformation systems. In Roberto Bruni and Dániel Varró, editors, *GT-VMT 2006*, volume 2011 of *ENTCS*, pages 201–210, 2008. 36, 49, 61

[KK08c] Barbara König and Vitali Kozioura. Towards the Verification of Attributed Graph Transformation Systems. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT 2008*, volume 5214 of *LNCS*, pages 305–320. Springer Berlin Heidelberg, January 2008. 61

[KS12] Barbara König and Jan Stückrath. Well-Structured Graph Transformation Systems with Negative Application Conditions. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *ICGT 2012*, volume 7562 of *LNCS*, pages 81–95. Springer Berlin Heidelberg, January 2012. 63

[LARSW00] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 26–38, Portland, Oregon, United States, 2000. ACM. 61

[MBC+07] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In Michael Huth Orna Grumberg, editor, *TACAS 2007*, volume 4424 of *LNCS*, pages 3–18, 2007. 61

[Mey08] Roland Meyer. *Structural Stationarity in the π-Calculus*. PhD thesis, University of Oldenburg, 2008. 63

[MLAS+08] Roman Manevich, Tal Lev-Ami, Mooly Sagiv, Ganesan Ramalingam, and Josh Berdine. Heap decomposition for concurrent shape analysis. In María Alpuente and Germán Vidal, editors, *SAS 2008*, volume 5079 of *LNCS*, pages 363–377, 2008. 37, 61

[PXZ02] Amir Pnueli, Jessie Xu, and Lenore Zuck. Liveness with (0,1, ∞)-Counter Abstraction. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV 2002*, volume 2404 of *LNCS*, pages 107–122. Springer Berlin Heidelberg, January 2002. 62

[Rak06] Jan Rakow. *Verification of Dynamic Communication Systems*. Master, Carl-von-Ossietzky Universität Oldenburg, April 2006. 53

[RD06] Arend Rensink and Dino Distefano. Abstract graph transformation. In *SVV 2005*, ENTCS, pages 39–59, May 2006. 36, 59

[RN08] Stefan Rieger and Thomas Noll. Abstracting Complex Data Structures by Hyperedge Replacement. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT 2008*, volume 5214 of *LNCS*, pages 69–83, 2008. 63

[RZ10] Arend Rensink and Eduardo Zambon. Neighbourhood Abstraction in GROOVE - Tool Paper, 2010. 59

[RZ11] A. Rensink and Eduardo Zambon. Neighbourhood Abstraction in GROOVE, April 2011. 59

[RZ12] Arend Rensink and Eduardo Zambon. Pattern-Based Graph Abstraction. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *ICGT 2012*, volume 7562 of *LNCS*, pages 66–80, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 60

[Sch97] Andy Schürr. Programmed Graph Replacement Systems. In Grzegorz Rozenberg, editor, *Handbook on Graph Grammars: Foundations*, pages 479–546. World Scientific, 1997. 57, 58

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL 1999, pages 105–118, New York, NY, USA, 1999. ACM. 61

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, May 2002. 37, 61

[SWJ08] Mayank Saksena, Oskar Wibling, and Bengt Jonsson. Graph grammar modeling and verification of ad hoc routing protocols. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS 2008*, volume 4963 of *LNCS*, pages 18–32, 2008. 37

[SWW10] Dominik Steenken, Heike Wehrheim, and Daniel Wonisch. Towards a shape analysis for graph transformation systems. CoRR abs/1010.4423, arXiv, 2010. 62

[SWW11] Dominik Steenken, Heike Wehrheim, and Daniel Wonisch. Sound and Complete Abstract Graph Transformation. In Adenilso Simao and Carroll Morgan, editors, *SBMF 2011*, volume 7021 of *LNCS*, pages 92–107, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. 63

[Tob08] Tobe Toben. Counterexample Guided Spotlight Abstraction Refinement. In Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, editors, *FORTE 2008*, volume 5048 of *LNCS*, pages 21–36. Springer Berlin Heidelberg, 2008. 5

[Ven98] Arnaud Venet. Automatic Determination of Communication Topologies in Mobile Systems. In Giorgio Levi, editor, *SAS 1998*, volume 1503 of *LNCS*, pages 152–167. pringer Berlin Heidelberg, 1998. 63

[Wac05] Björn Wachter. *Checking universally quantified temporal properties with three-valued analysis*. Diploma, Universität des Saarlandes, 2005. 62

[WZH10] Thomas Wies, Damien Zufferey, and Thomas A. Henzinger. Forward Analysis of Depth-Bounded Processes. In Luke Ong, editor, *FOSSACS 2010*, volume 6014, pages 94–108, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 62

[Yah01] Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36:27–40, March 2001. 61

[Yah13] Eran Yahav. Solving The Apprentice Challenge with 3vmc, January 2013. 61

[YRSW03] Eran Yahav, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying Temporal Heap Properties Specified via Evolution Logic. In Pierpaolo Degano, editor, *ESOP 2003*, volume 2618 of *LNCS*, pages 204–222. Springer Berlin Heidelberg, January 2003. 61

[Zam13] Eduardo Zambon. *Abstract graph transformation : theory and practice*. PhD thesis, University of Twente, 2013. 35, 36, 49, 60

[ZWH12] Damien Zufferey, Thomas Wies, and Thomas A. Henzinger. Ideal abstractions for well-structured transition systems. In Viktor Kuncak and Andrey Rybalchenko, editors, *VMCAI 2012*, volume 7148 of *LNCS*, pages 445–460, January 2012. 37, 62