

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

Peter Backes¹ and Jan Reineke²

¹ Universität des Saarlandes, Saarbrücken, Germany
`rtc@cs.uni-sb.de`

² University of California, Berkeley
`reineke@eecs.berkeley.edu`

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.

1 Introduction

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 [2] 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 [4] concerns cars that drive on a highway with constant speed and constant distance from each other, to conserve energy.

* This work was partly 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.

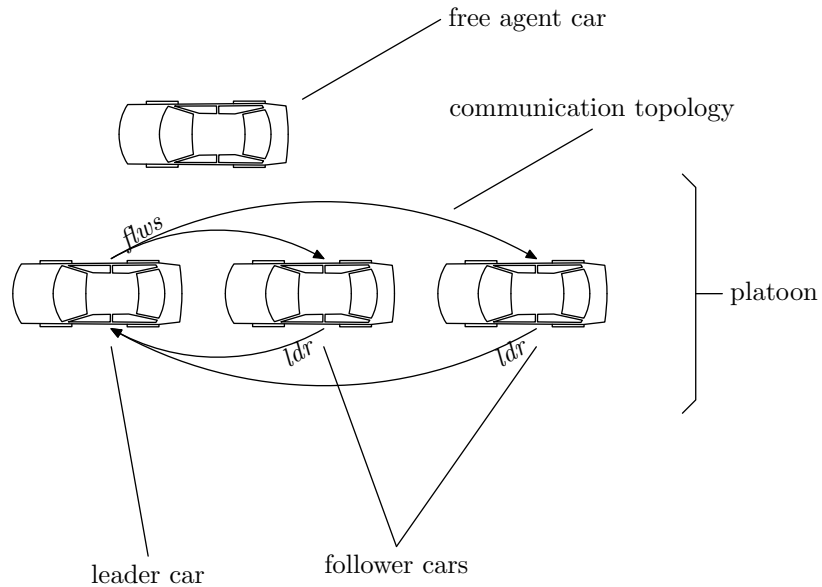


Fig. 1. Car platooning

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 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.

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:

1. 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.

2. 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 [3]. 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 [5].

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 The subject to be modelled

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 modelling the merge protocol.

2.1 Dynamic communication systems

Dynamic communication systems [1] 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; ie., $\text{Channels} : Id \rightarrow 2^{Id}$, not $\text{Channels} \subseteq 2^{Id}$. 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 to all processes in one of its channels. From a global view, the channels make up the communication topology.

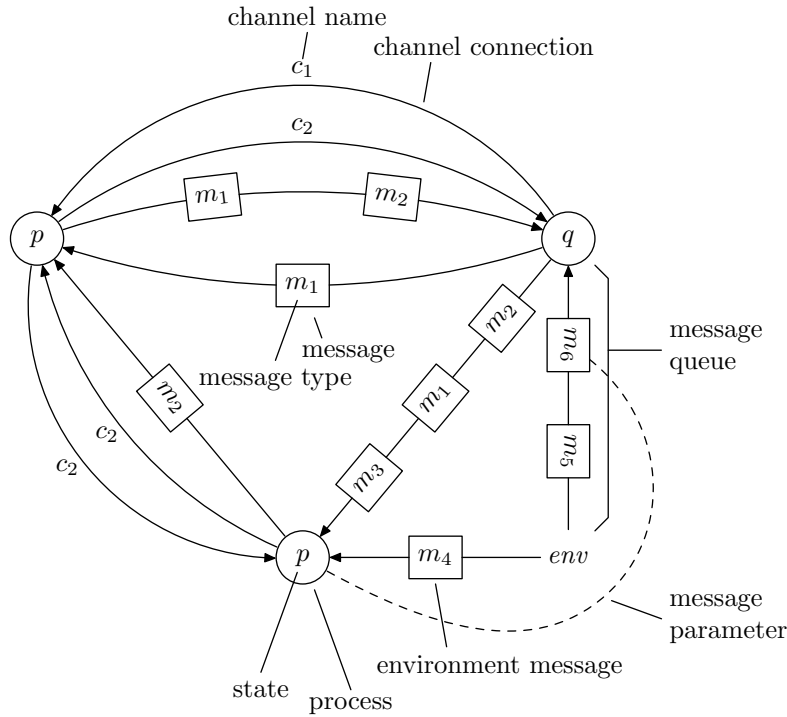


Fig. 2. Concepts of dynamic communication systems

A protocol specifies the behaviour 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 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.

The merge protocol specifies that each process starts in the only initial state fa (“free agent”) (denoted in 3 by the triangle). As can be seen in Figure 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 (“acknowledgement”) message with its own process identity. It then changes to state ldb (“leader”). As soon

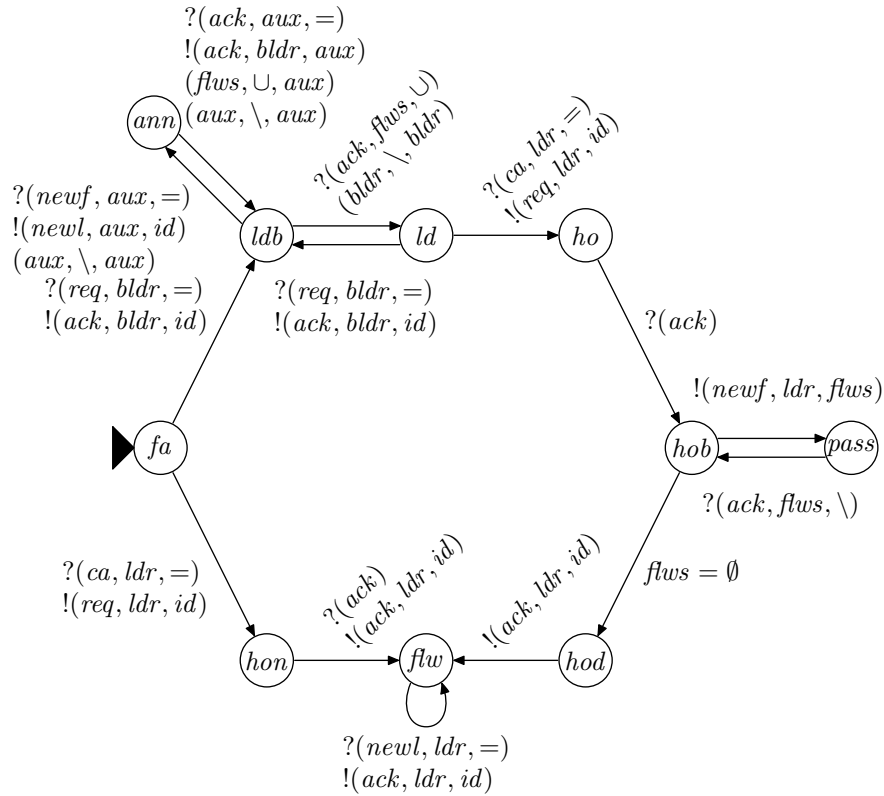
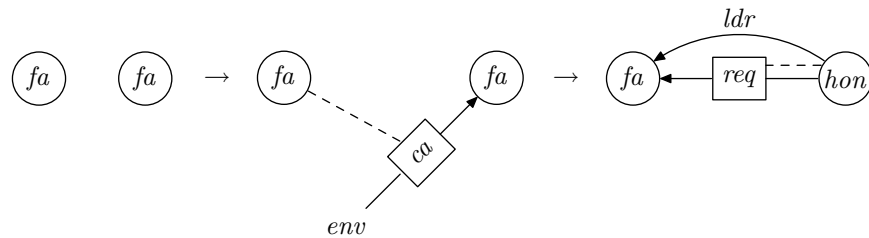
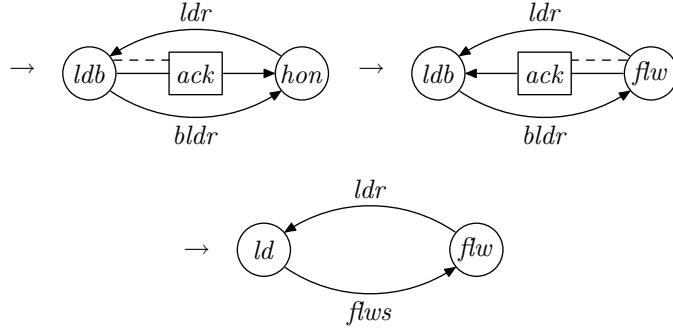


Fig. 3. Transition system of the merge protocol

as the other process, assuming it is still in state *hon*, receives the message, it changes to state *flw* (“follower”) and returns another acknowledgement, 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:

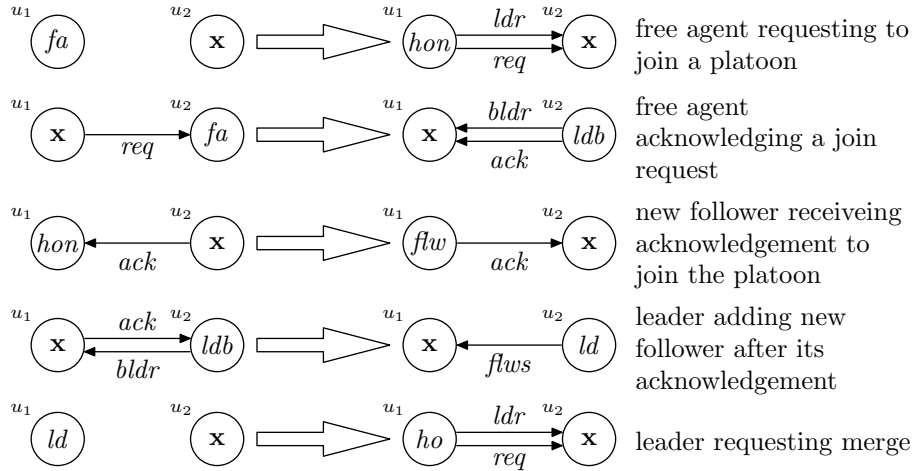


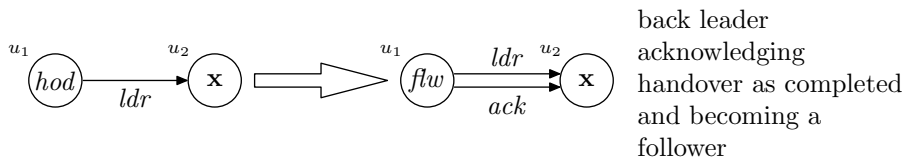
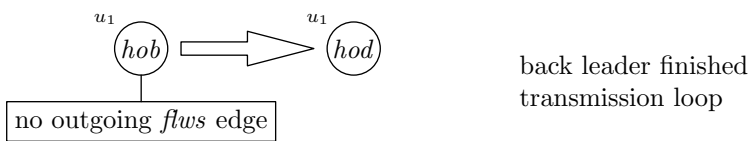
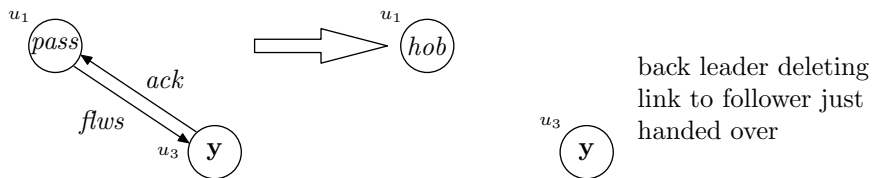
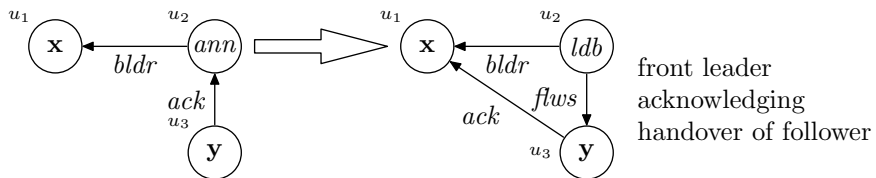
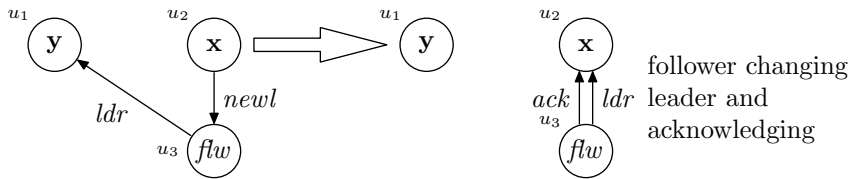
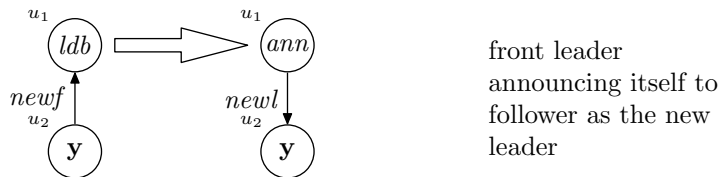
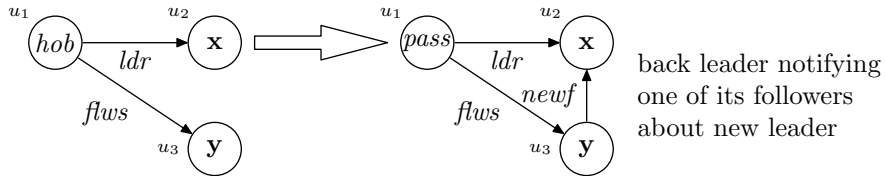
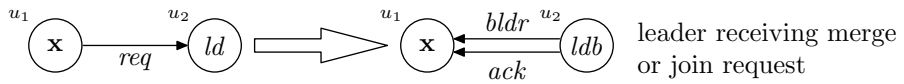


The second task, having a car join an already existing platoon, is mostly similar to the first one, except that the process attached to the car ahead message as a parameter is initially in state *ld* instead of *fa*. For merging (the third task), we have both the recipient of the car ahead message and 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 acknowledgement 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 acknowledgement 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.

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 modelled 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 modelled 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.
- \mathbf{x} and \mathbf{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 $flws = \emptyset$ 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 $flws$.

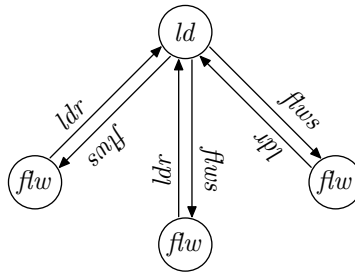
The rules use the following tricks:

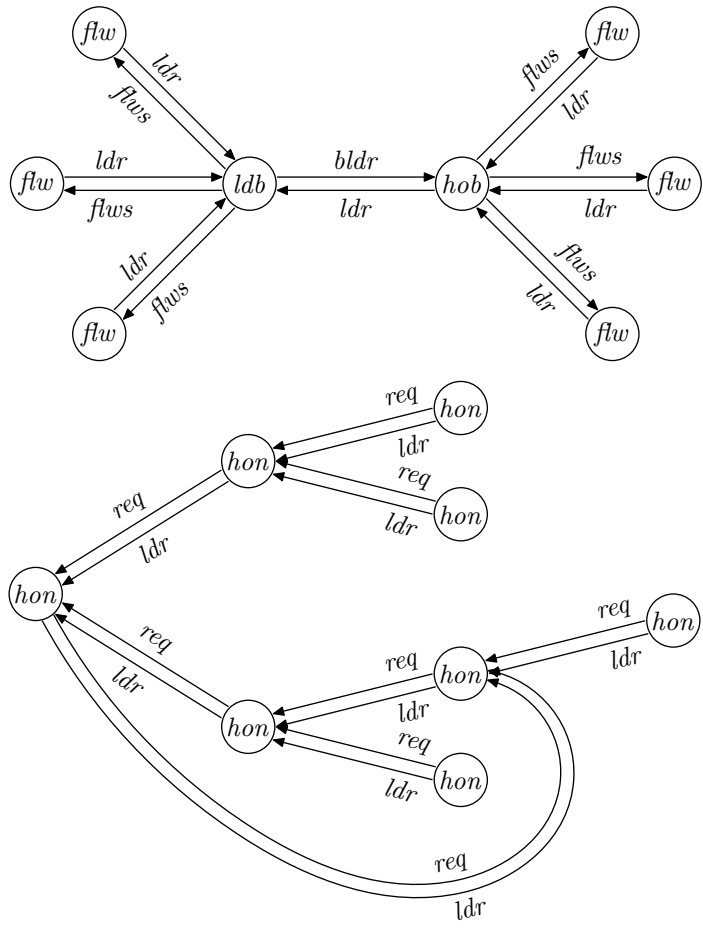
- We do not model environment messages. Because they can be sent at any time, they can also be received at any time.
- We do as many things as possible in one graph transformation rule.

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).

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):





5 Goals

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 should help you to adjust the graph transformation rules to your needs.

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 labelled *flw* are connected to each other with an edge labelled *ldr*.
- A node labelled *pass* or *ld* always has at least one node labelled *flw* connected via some edge labelled *flws*.
- If a node labelled *flw* has outgoing edges labelled *ldr* and *newf*, respectively, to two different nodes, then those two nodes are connected via an edge labelled *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?

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 labelled 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?

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)

References

1. Jörg Bauer, Ina Schaefer, Tobe Toben, and Bernd Westphal. Specification and verification of dynamic communication systems. In *Sixth International Conference on Application of Concurrency to System Design*, 2006.
2. Jörg Bauer and Reinhard Wilhelm. Static analysis of dynamic communication systems. In *14th International Static Analysis Symposium*, 2007.
3. Iovka Boneva, Arend Rensink, Marcos E. Kurban, and Jörg Bauer. Graph abstraction and abstract graph transformation. Technical Report TR-CTIT-07-50, Enschede, July 2007.
4. PATH Project. Vehicle platooning and automated highways. <http://www.path.berkeley.edu/PATH/Publications/Media/FactSheet/VPlatooning.pdf>, 1998.
5. Tobe Toben. Counterexample guided spotlight abstraction refinement. In K. Suzuki, T. Higashino, K. Yasumoto, and K. El-Fakih, editors, *Proceedings of the 28th IFIP WG6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2008)*, volume 5048 of *LNCS*, pages 21–36, Tokyo, Japan, June 2008. Springer-Verlag.