Analysis of Dynamic Communicating Systems by Hierarchical Abstraction (Draft) *

Jörg Bauer and Reinhard Wilhelm

Informatik; Univ. des Saarlandes; Saarbrücken, Germany.
{*joba,wilhelm*}*@cs.uni-sb.de*

Abstract. We propose a new abstraction technique for verifying topology properties of dynamic communicating systems (DCS), a special class of infinite-state systems. DCS are characterized by unbounded creation and destruction of objects along with an evolving communication connectivity or topology. We employ a lightweight graph transformation system to specify DCS. Hierarchical Abstraction computes a bounded over-approximation of all topologies that can occur in a DCS directly from its transformation rules. Hierarchical Abstraction works in two steps. First, for each connected component, called cluster, of a topology, objects sharing a common property are summarized to one abstract object. Then isomorphic abstract connected components are summarized to one abstract component, called abstract cluster. This yields a conservative approximation of all graphs that may occur during any DCS run. The technique is implemented.

1 Introduction

Dynamic Communicating Systems (DCS) are widespread. Prominent examples include wireless ad-hoc networks and traffic control systems for cars, trains, and planes. Specifying and analyzing DCS is a very complex task due to their high dynamics. Both the number of objects (*e.g.* laptops, cars, trains, planes) and their communication connectivity vary over time. In general, DCS induce infinite-state transition systems that are notoriously hard to verify.

Graphs are a very natural choice to model the communication *topology* of a DCS, hence graph transformation systems [1] are a natural and intuitive choice to describe the evolution of a DCS. In this work, a notion of graph transformation systems tailored to model DCS is defined in Section 2.

A DCS graph transformation system induces an infinite-state transition system with graphs as states. The states of a transition system are not to be confused with object states to be defined later. The graphs themselves can be unbounded due to unlimited creation of objects. The challenge is to analyze an unbounded system of unbounded graphs.

One way to deal with this class of systems is abstraction in the sense of Abstract Interpretation [2]. In Section 3, Hierarchical Abstraction is introduced. By applying

* This work was 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 `www.avacs.org` for more information.

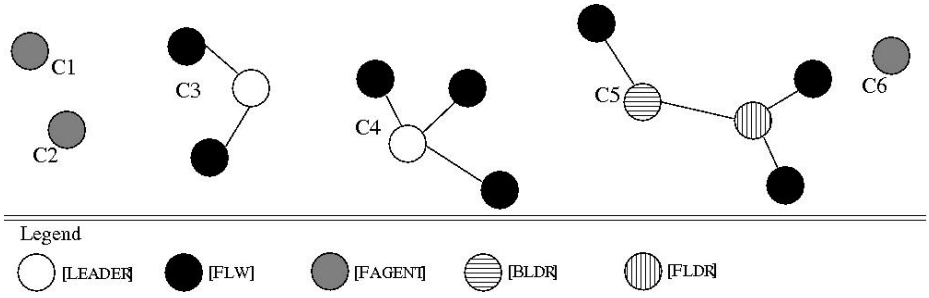


Fig. 1. A possible topology G_T for platoons, fill style of nodes indicating the state of objects represented by then nods.

Hierarchical Abstraction a *bounded* over-approximation of all possible graphs is computed directly from the graph transformation system specifying a DCS. It is shown that the abstraction is *sound* – the formal basis for proving safety properties of the concrete system.

Hierarchical Abstraction works in two steps. First, each connected subgraph of a graph is abstracted by quotient graph building *wrt.* an equivalence relation. Two objects are said to be equivalent, if they are in the same state, and if the set of labels of their adjacent objects are equal. Isomorphic abstract connected components, abstract clusters, are summarized in the second step resulting in an abstract topology, *i.e.* a set of abstract clusters. In Section 6.3, we show how to parameterize this notion of abstraction.

We demonstrate our ideas by an interesting, challenging, and practically relevant case study that is introduced in Section 2.1. Earlier approaches failed to verify crucial properties of this case study. Our technique is implemented and the tool is briefly introduced in Section 4.

Clearly, Hierarchical Abstraction was inspired by Canonical Abstraction as presented in the three-valued logic approach to Shape Analysis [11]. The relation between these to abstraction techniques are explained in Section 5.

2 Modeling Dynamic Communicating Systems

A *dynamic communicating system* (DCS) consists of a dynamically changing and unbounded number of objects with an evolving communication connectivity. Each object is always in a *state*, that can change over time. Two communicating objects connected via a *channel* are called *partners*. A maximally connected set of objects is called a *cluster*, and the connectivity at a certain point in time is called the current *communication topology* of the DCS. We assume *synchronous communication* over channels. A communication topology is modeled as a node-labeled, directed graph with objects as nodes, the current state of an object as node label and channels as edges. In figures objects are drawn as circles with different fill styles denoting their states. DCS are characterized by the following key properties:

1. Destruction and unbounded creation of objects.
2. Evolving communication topologies.
3. Objects changing their state depending on their own *and* their partners' state. This captures the intuition of communicating objects and is called the *Transition Principle*.
4. Clusters, set of objects able to communicate with each other, are the crucial entity of interest.

2.1 Case Study: Platoons

Our case study is taken from the California PATH project [3, 4], the relevant part of which is concerned with cars driving on a highway. In order to make better use of the given space, cars heading for the same direction are supposed to drive very near to each other building *platoons*. Platoons consist of a *leader*, the foremost car, along with a number of *followers*. A leader without any followers is called *free agent* and is considered a special platoon. Within a platoon there are communication channels between the leader and each of its followers. Inter-platoon communication is only between leaders. In this work, the platoon *merge maneuver* is studied. It allows two platoons heading in the same direction and driving close to each other to merge. Verifying the merge maneuver is a crucial step in the verification of the platoon case study. Platoons are prototypical for DCS, because there is no control of how many cars enter and leave the highway, and because maneuvers like merge imply non-trivial changes in the communication topology. It was tried to verify platoon protocols such as merge within the PATH project using the model checker COSPAN [5]. However, the methods were inappropriate, because they did not take into account any dynamics. Only a static scenario with a fixed number of cars and platoons was considered. The proposed method remedies this deficit.

Platoons are modeled using five different states that a car can assume. In figures such as Figure 1, we will use filling styles to distinguish the states of the cars.

1. [LEADER]: The state of a leader car that is currently not involved in a merge maneuver. In figures, this state is depicted as an empty circle.
2. [FLW]: The state of a follower car that is currently not involved in a merge maneuver, depicted as a circle with black filling.
3. [FAGENT]: The state of a car that is moving around alone, depicted in gray filling.
4. [BLDR]: This state is needed to distinguish the two leaders of merging platoons. The back leader is the one that joins the platoon in front by handing over its followers. Depicted as nodes filled with horizontal lines.
5. [FLDR]: The front leader receiving followers from the back leader; nodes filled with vertical lines.

Figure 1 displays a typical topology that might occur during the run of the platoon system. We have six clusters, five of which are platoons, *i.e.* a cluster containing [BLDR] and [FLDR] is not a platoon. Note that undirected edges suffice for our running example. This is a special instance of the general technique presented in this work.

A merge action is initiated by opening a channel between two distinct platoon leaders, *i.e.* leaders or free agents. In reality such an event is triggered by sensors. On opening this channel the two leaders involved change their state according to their relative position. Their new states are [BLDR] and [FLDR], respectively. Then the back leader passes its followers one by one to the front leader. Finally, when there are no followers to the back leader left, the back leader and the front leader change their states to [FLW] and [LEADER], respectively.

The aim of this work is to prove topology properties of platoons. These are safety properties like some cars are never connected shown in Section 4.

2.2 System Evolution

The evolution of a DCS is specified by *graph transformation rules*. In order to fit the needs of the application domain, DCS, the single pushout (SPO) approach to graph transformation as formally introduced in Chapter 4 of [1] is used in a much simplified form here. Our notion of *DCS graph transformation (DCS-GTS)* is easy to use and analyze yet expressive enough to capture the essence of DCS.

In general, a *graph transformation system* $(\mathcal{R}, I)$ consists of a set $\mathcal{R}$ of *rules* and an *initial graph* I. Informally, a rule is a pair (L, R) of graphs, called *left* and *right* graph with a mapping from the nodes and edges of L to the nodes and edges of R. A rule (L, R) can be applied to a graph G, if L *matches* G. A match is a structure- and label-preserving mapping m from L to a subgraph of G. After that, the subgraph matched by L is replaced with R.

In the analysis of DCS, however, the full complexity of general graph transformation systems is not needed. It suffices to consider three types of rules.

1. *Create* rules have an empty left graph
2. *Destroy* rules have an empty right graph
3. *Edge* rules have a node-bijective mapping from left to right graph

Additionally, we deviate from the standard SPO approach by

- allowing *injective* matches only
- allowing easy node-label changes (as compared to [6])
- allowing a mild form of *negative application conditions* as introduced in [7]: *Partner Constraints* can constrain the set of neighbors of a node in order for a rule to match

Definition 1 (DCS-GTS). *Let $\mathfrak{L}$ be a finite set of* node labels*.*

1. *A* graph G *over $\mathfrak{L}$ is a triple $G = (V, E, \mathfrak{n})$, where V is the set of* nodes *and $E \subseteq V \times V$ is the set of* edges*. The mapping $\mathfrak{n} : V \to \mathfrak{L}$ assigns labels to nodes. The set of all graphs over $\mathfrak{L}$ is written $\mathcal{G}(\mathfrak{L})$. Given a graph G, V_G, E_G, and $\mathfrak{n}_G$ denote the nodes, edges, and the node-labeling of G, respectively.*
2. *A* DCS graph transformation rule r *is a 4-tuple (L, h, p, R), where $L, G \in \mathcal{G}(\mathfrak{L})$, $p : V_L \rightharpoonup 2^{\mathfrak{L}}$, $h : V_L \to V_R$ and either*
   *(a) $V_L = \emptyset$ (*create rule*)*

(b) $V_R = \emptyset$ *(*destroy rule*)*
(c) $V_L, V_R \neq \emptyset$ *and* h *is bijective (*edge rule*)*

3. *A* DCS graph transformation system (DCS-GTS) *is a pair* $\mathcal{GTS} = (\mathcal{R}, \mathcal{I})$*, where* $\mathcal{R}$ *is a finite set of DCS graph transformation rules and* $\mathcal{I} \in \mathcal{G}(\mathfrak{L})$ *is the* initial graph*.*
4. *A DCS-GTS rule* $r = (L, h, p, R)$ matches *graph* $G \in \mathcal{G}(\mathfrak{L})$*, if there is a pair* $m = (m_n, m_e)$ *of injective mappings, where* $m_n : V_L \to V_G$*,* $m_e : E_L \to E_G$*, such that*
   (a) $\mathfrak{n}_L(v) = \mathfrak{n}_G \circ m_n(v)$ *for all* $v \in V_L$
   (b) $m_e(v_1, v_2) = (m_n(v_1), m_n(v_2))$ *for all* $(v_1, v_2) \in E_L$
   (c) If $v \in dom(p)$*, then* $p(v) = \{\mathfrak{n}_G(v') | (v', m_n(v)) \in E_G \text{ or } (m_n(v), v') \in E_G\}$
   Conditions (a,b) define a graph homomorphism*. The pair* m *is called a* (concrete) match *from* r *to* G*. If* m *is a match, then* m_n *and* m_e *denote the node and edge mappings, respectively.*
5. *Let* $r = (L, h, p, R)$ *be a transformation rule matching* G *via match* m*. The result of the* application of r to G *wrt.* m *is the graph* H *with*
   (a) $H = G \dot\cup R$*, if* r *is a create rule, and* $\dot\cup$ *is the disjoint union of graphs.*
   (b) $V_H = V_G \setminus m_n(V_L)$*,* $E_H = E_G \cap (V_H \times V_H)$*,* $\mathfrak{n}_H = \mathfrak{n}_G \mid_{V_H}$*, if* r *is a destroy rule.*
   (c) If r *is an edge rule, then*

$$\begin{aligned} V_H &= V_G \\ E_H &= (E_G \setminus m_e(E_L)) \cup \{(m_n(v_1), m_n(v_2)) \mid (h(v_1), h(v_2)) \in E_R\} \\ \mathfrak{n}_H(v) &= \begin{cases} \mathfrak{n}_R(h(v')) \text{ if } v = m_n(v') \\ \mathfrak{n}_G(v) \qquad \text{otherwise} \end{cases} \end{aligned}$$

6. *We write* $G \rightsquigarrow_r H$ *if* H *is the result of applying rule* r *to* G*. We write* $G \rightsquigarrow_{\mathcal{R}} H$*, if there exist a rule* $r \in \mathcal{R}$ *such that* $G \rightsquigarrow_r H$*.*
7. *Given a DCS-GTS* $\mathcal{GTS} = (\mathcal{R}, \mathcal{I})$*, we write* $\rightsquigarrow^*_{\mathcal{R}}$ *for the reflexive, transitive closure of the relation* $\rightsquigarrow_{\mathcal{R}}$*. The* semantics *of* $\mathcal{GTS}$ *is defined to be*

$$[\![\mathcal{GTS}]\!] =_{\text{def}} \{G \mid \mathcal{I} \rightsquigarrow^*_{\mathcal{R}} G\}$$

A sequence $G_1 \rightsquigarrow_{\mathcal{R}} G_2 \rightsquigarrow_{\ldots} \rightsquigarrow_{\mathcal{R}} G_n$ *is called a* run *of length* n*. The pair* $([\![\mathcal{GTS}]\!], \rightsquigarrow_{\mathcal{R}})$ *is called the* DCS transition system induced by $\mathcal{GTS}$*.*

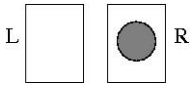
Later on, the notion of connected components is used frequently. Here is the formal introduction.

Definition 2 (Connected Components). *Let* $(G = (V, E, \mathfrak{n})$ *be a graph over an arbitrary, finite set of node labels. The set of connected components of* G *is defined* wrt. *the equivalence relation* $\sim_c$ *on* V*:*

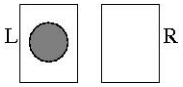
$$v_1 \sim_c v_2 \iff_{\text{def}} \exists u_1, \ldots, u_n \in V.\, u_1 = v_1 \wedge u_n = v_2 \wedge \bigwedge_{1 \leq i < n} (u_i, u_{i+1}) \in E \vee (u_{i+1}, u_i) \in E$$

A connected component *of* G *is a graph whose nodes are an element of* $V/\!\sim_c$*. The set of all connected components of* G *is written* $cc(G)$*. A graph is* connected*, if all its nodes are* $\sim_c$ *equivalent. The* set of all connected graphs *over node labels* $\mathfrak{L}$ *is written* $\mathcal{CC}(\mathfrak{L})$*.*

Specification of Platoons Now, the set $\mathcal{P}$ of transformation rules specifying platoons is introduced.

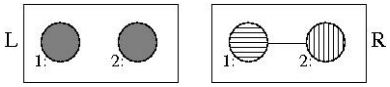


[CREATE]

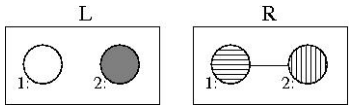


[DESTROY]

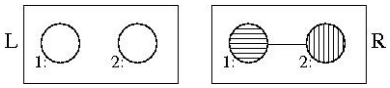
[CREATE] and [DESTROY] are the most simple rules of $\mathcal{P}$. At any time a new free agent can be created by [CREATE], an existing free agent can be destroyed by [DESTROY]. This captures the dynamics of creation and destruction that are typical to DCS. [CREATE] matches any graph and can always be applied, whereas [DESTROY]can be applied to a graph with at least one [FAGENT]. [CREATE] and [DESTROY] are obviously instances of create and destroy rule, whereas all the remaining rules are edge rules.



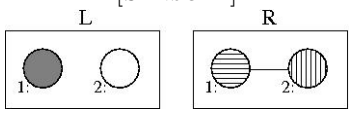
[SENSOR1]



[SENSOR2]

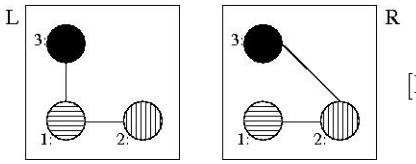


[SENSOR3]



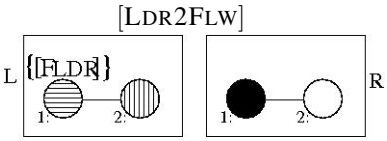
[SENSOR4]

These four rules model the sensor-triggered opening of a channel between platoon leaders. Recall that also free agents can be considered platoon leaders. In [SENSOR1] two distinct free agents get connected by a channel, while changing their states to [BLDR] and [FLDR], respectively. The same happens to two distinct [LEADER] cars, or to [LEADER] and [FAGENT] cars, respectively in rules [SENSOR2-4].



[PASS]

[LDR2FLW]



[PASS] formalizes the hand-over of a follower from a back to a front leader. Whenever we find a topology with connected back and front leaders and a follower connected to the back leader, the [BLDR]–[FLW] edge can be deleted and a new [FLDR]–[FLW] edge can be inserted. The set of nodes remains unchanged.

Note the $\{[\text{FLDR}]\}$ annotation of the [BLDR] car, v, in rule [LDR2FLW]. This is an instance of a *partner constraint*. It means that, in order for [LDR2FLW] to match a graph G via mapping m, the set of labels of $m(v)$'s partners in G must equal $\{[\text{FLDR}]\}$. In this case, it codes, that the back leader has only front leaders as partners and therefore no more followers left.

Figure 2 shows a sample run of this DCS-GTS.

The Cluster Multiplicity and the Transition Principle Before bounded abstraction of both graphs and a whole DCS transition system are introduced, two fundamental prop-

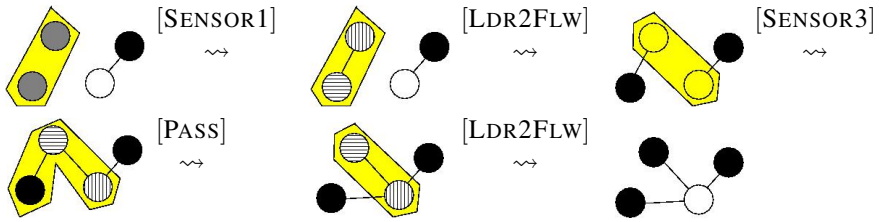


Fig. 2. A platoon sample run of length 6: The names of rules applied label the transition, the nodes that a rule matches are enclosed in gray polygons. The run merges two free agents to build a new platoon, which afterwards merges with another platoon.

erties of the whole class of DCS are presented. The first is concerned with cluster multiplicity and is called the *Cluster Multiplicity Principle*:

If some cluster can evolve from the empty graph, then arbitrarily many of these clusters can arise.

If some cluster C can be constructed starting with the empty graph, then all the nodes that need to be created to construct an isomorphic copy C' of it can be created arbitrarily often and independently of the rest of the topology due to unlimited creation in DCS. Also, the existence of C does not prevent C' from being created, because partner constraints only affect changes within one cluster and do not affect other clusters. The Cluster Multiplicity Principle resembles a Pumping Lemma for DCS.

The second principle reflects the way objects can change their state in DCS and is thus called the *Transition Principle*.

The change of an object's state depends on its own state and *the states of its partners.*

Intuitively, object states correspond to roles in protocols, and the role of an object and the set of roles of partners determine the next transition. Due to communication between them, partners mutually influence their states. The Transition Principle is reflected in the partner constraints and in cluster abstraction, which is introduced in Section 3.

3 Computing the Abstract Transition System

Verifying properties of a DCS induced by a DCS graph transformation system $\mathcal{GTS}$ means proving properties of unbounded sequences of graphs of unbounded size. The challenge is to find a bounded approximation of this behavior preserving relevant properties. First, *Hierarchical Abstraction* abstracts arbitrary graphs to *abstract topologies*. After that, the concrete graph transformation rules are directly applied to such abstract topologies.

3.1 Abstract Topologies

For the rest of this section, if not mentioned otherwise, let $\mathfrak{L}$ be a set of node labels and let $G \in \mathcal{G}(\mathfrak{L})$ be a graph over these labels.

Hierarchical Abstraction consists of two consecutive phases: *cluster abstraction* and *topology abstraction*. Cluster abstraction is based on the Transition Principle, whereas topology abstraction is based on the Cluster Multiplicity Principle.

Cluster abstraction is defined in terms of quotient graph building with respect to an equivalence relation, where, additionally, some multiplicity information is kept in the quotient graph. Equivalence of objects means agreement on the set of possible transitions, *i.e.* the choice of the relation is motivated by the Transition Principle. As a potential change of an object's state is mainly determined by its own and its partners' states, two objects can be considered equivalent (with respect to the next transition) if they are in the same state and if their partners are in the same states. In this case they are *summarized*, *i.e.* collected into one equivalence class represented by an abstract object.

The cluster abstraction of a given graph is a quotient graph *wrt.* node equivalence. It has node labels $\mathfrak{L} \times \{0, 1\}$. There is an edge between two equivalence classes, if there is an edge between two members of the two classes. The label of an equivalence class is the label of its representatives plus a multiplicity bit. If an equivalence class consists of more than one node, it is specially marked by a label $(\_, 1)$. Such a node is called a *summary node*. In figures, summary nodes are drawn with double lines. The building of two quotient graphs is shown exemplarily in Figure 3.

Definition 3 (Cluster Abstraction). *Let $G \in \mathcal{CC}(\mathfrak{L})$ be a connected graph. The nodes $v_1, v_2 \in V_G$ are equivalent, written $v_1 \sim_c v_2$, if and only if*

1. $\mathfrak{n}(v_1) = \mathfrak{n}(v_2)$
2. $\{\mathfrak{n}(v) \mid (v, v_1) \in E_G\} = \{\mathfrak{n}(v) \mid (v, v_2) \in E_G\}$
3. $\{\mathfrak{n}(v) \mid (v_1, v) \in E_G\} = \{\mathfrak{n}(v) \mid (v_2, v) \in E_G\}$

The quotient graph $G/\!\sim_c \in \mathcal{G}(\mathfrak{L} \times \{0, 1\})$ with

$$G/\!\sim_c =_{\text{def}} (V/\!\sim_c, \{([v_1], [v_2]) \mid (v_1, v_2) \in E\}, \lambda[v].(\mathfrak{n}(v), [v] > 1)$$

is called the cluster abstraction *of G and written $\alpha_c(G)$. A node v with label $(\_, 1)$ in a cluster abstraction is called a* summary node*.*

Note, that cluster abstraction is well-defined, because $\sim_c$ defines an equivalence relation, and because all nodes summarized to an abstract one have the same label – implying the well-definedness of the node-labeling. Two examples of quotient graph building *wrt.* $\sim_c$ are given in Figure 3.

For any connected graph G the abstraction $\alpha_c(G)$ has at most $|\mathfrak{L}| \cdot 2^{2 \cdot |\mathfrak{L}| + 1}$ nodes. Each node can then be identified with its unique pair of label and set of partner labels, its canonical name. This gives rise to the canonical representation of $\alpha_c(G)$. The canonical representation of isomorphic graphs $\alpha_c(G)$ and $\alpha_c(H)$ is identical simplifying the handling of isomorphic graphs.

Definition 4 (Canonical Names). *Let $G \in \mathcal{G}(\mathfrak{L})$ be a connected graph and let $H = \alpha_c(G)$ be its cluster abstraction. The* canonical name *of node $v \in H$ is the pair*

$$k_n(v) =_{\text{def}} (\mathfrak{n}_H(v), \{(\mathfrak{n}_H(v'), 0), (\mathfrak{n}_H(v''), 1) \mid (v, v') \in E_H, (v'', v) \in E_H\}, |v| > 1)$$

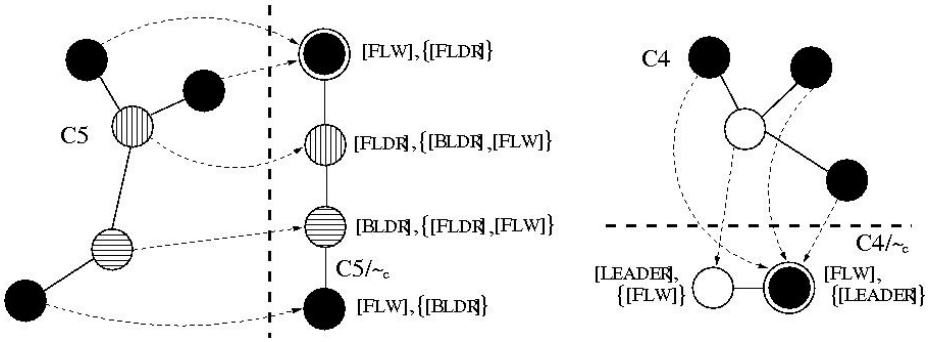


Fig. 3. Building a quotient graph *wrt.* $\sim_c$ by "crossing the dashed line": Summary nodes are drawn with double lines. Arrows indicate, to which equivalence class a node belongs. The additional information attached to the nodes in the quotient graph characterizes the equivalence class by stating label and set of partner labels.

The canonical representation *of H is the graph $k(H)$ with $V_{k(H)} = \{k_n(v) \mid v \in V_H\}$ and $E_{k(H)} = \{(k_n(v), k_n(v') \mid (v, v') \in E_H\}$ as well as $\mathfrak{n}_{k(H)} = \lambda v.\mathfrak{n}_H(v')$ if $v = k_n(v')$.*

For a given set $\mathfrak{L}$ of node labels, we write $\mathcal{CG}(\mathfrak{L})$ for the set of all graphs $G \in \mathcal{CC}(\mathfrak{L} \times \{0,1\})$ with $V_G \subseteq \mathfrak{L} \times \mathfrak{P}(\mathfrak{L} \times \{0,1\}) \times \{0,1\}$. An element of $\mathcal{CG}(\mathfrak{L})$ is called an abstract cluster. *A set of abstract clusters is called an* abstract topology.

If cluster abstraction was applied to a graph G with several connected components in the same way as defined above, it would yield a bounded abstraction of G due to the bounded number of nodes in cluster abstracted graphs. However, in order not to mix up nodes from different clusters – the main entity of interest in DCS – cluster abstraction will only be applied to single connected graphs. The following lemma is obvious. It states that cluster abstraction preserves connectivity of graphs.

Lemma 1. *If $G \in \mathcal{CC}\mathfrak{L}$, then $\alpha_c(G) \in \mathcal{CC}\mathfrak{L} \times \{0,1\}$.*

While cluster abstraction reflects the Transition Principle, the next abstraction step, called topology abstraction, reflects the Cluster Multiplicity Principle. topology abstraction is an isomorphic reduction reducing a set M of graphs to its maximal subset without distinct isomorphic elements. The isomorphic reduction is applied to the set of graphs resulting from cluster abstraction. It makes use of the newly defined notion of canonical representations.

The following definition introduces Hierarchical Abstraction and demonstrates it to be the result of two successive abstraction steps: cluster abstraction and topology abstraction. First, cluster abstraction is lifted to work element-wise on *sets* of connected graphs. This models – and is, in fact, isomorphic to – graphs with an arbitrary number of connected components. topology abstraction is the isomorphic reduction of this result computed using canonical representations.

Definition 5 (Hierarchical Abstraction). *Let $M \in \mathfrak{P}(\mathcal{CC}(\mathfrak{L}))$ be a set of connected graphs. The* cluster abstraction *of M is the set*

$$\alpha_c(M) =_{\text{def}} \{\alpha_c(H) \mid H \in M\}$$

The topology abstraction *of a set $N \in \mathfrak{P}(\mathcal{CC}(\mathfrak{L} \times \{0,1\}))$ is the set*

$$\alpha_t(N) =_{\text{def}} \{k(H') \mid H' \in N\}.$$

The Hierarchical Abstraction of M is the defined to be

$$\alpha(M) =_{\text{def}} \alpha_t \circ \alpha_c(M)$$

The bottom line of Figure 4 shows the abstract topology $\alpha(G_T)$, where G_T was given in Figure 1. Note the summary nodes representing more than one [FLW] node. The canonical names of the seven nodes in the bottom line are (from left to right):

1. $([\text{FAGENT}], \emptyset, 0)$
2. $([\text{LEADER}], \{([\text{FLW}], 0), ([\text{FLW}], 1)\}, 0)$
3. $([\text{FLW}], \{([\text{LEADER}], 0), ([\text{LEADER}], 1)\}, 1)$
4. $([\text{FLW}], \{([\text{BLDR}], 0), ([\text{BLDR}], 1)\}, 1)$
5. $([\text{BLDR}], \{([\text{FLW}], 0), ([\text{FLW}], 1), ([\text{FLDR}], 0), ([\text{FLDR}], 1)\}, 0)$
6. $([\text{FLDR}], \{([\text{FLW}], 0), ([\text{FLW}], 1), ([\text{BLDR}], 0), ([\text{BLDR}], 1)\}, 0)$
7. $([\text{FLW}], \{([\text{FLDR}], 0), ([\text{FLDR}], 1)\}, 0)$

Note that in the second component of the canonical names all neighbor labels occur with both 0 and 1, denoting outgoing and incoming edges. This is because our running example is an instance of an undirected graph.

Properties We will now state two algebraically interesting properties concerning our abstractions. First, α_c, α_t, and α are shown to form Galois connections given appropriate domains. So far, we have reasoned about connected graphs only. This seems reasonable, because it is much simpler to impose a partial order on sets of connected graphs by just taking subset inclusion. In the case of general graphs, a partial order would have to reason about subgraphs, too.

Lemma 2. *Let $\mathfrak{L}$ be a finite set of node labels. Let L_1, L_2, and L_3 be the complete lattices $\mathfrak{P}(\mathcal{CC}(\mathfrak{L}))$, $\mathfrak{P}(\mathcal{CC}(\mathfrak{L} \times \{0,1\}))$, and $\mathfrak{P}(\mathcal{CG}(\mathfrak{L}))$, respectively; all ordered by subset inclusion. The following four-tuples are Galois connections.*

1. $(L_1, \alpha_c, \gamma_c, L_2)$
2. $(L_2, \alpha_t, \gamma_t, L_3)$
3. $(L_1, \alpha, \gamma, L_3)$

where for for each $f \in \{\alpha_c, \alpha_t, \alpha\}$ the concretization is defined to be $\gamma_f(M) = \bigcup\{N \mid f(N) \subseteq M\}$.

The second algebraic observation concerns homomorphisms. Informally speaking, the abstractions are homomorphisms between graphs. The corresponding lemma is stated in terms of connected graphs and can, of course, be easily lifted to disjoint unions of connected graphs, because disjoint graphs cannot interfere with he homomorphism requirements.

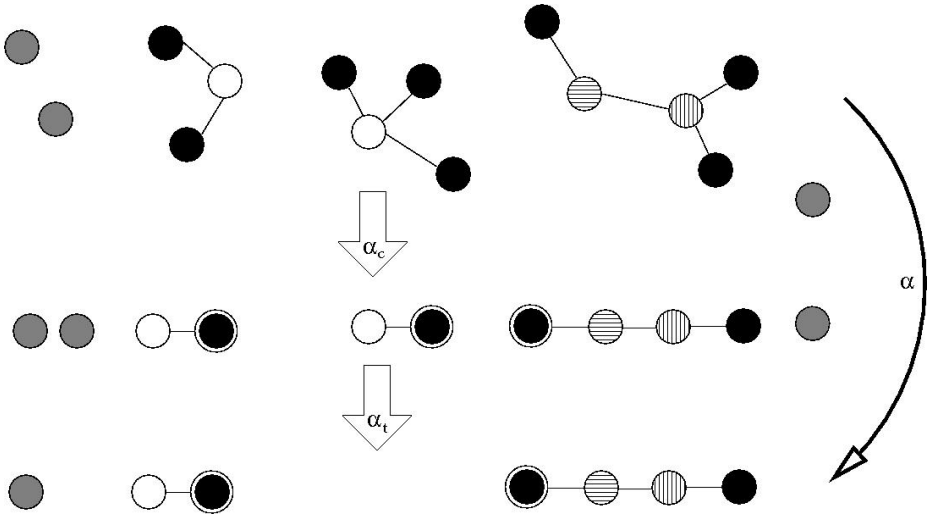


Fig. 4. Cluster abstraction, topology abstraction, and Hierarchical Abstraction applied to the graph G_T from Figure 1 resulting in abstract topology T.

Lemma 3. *For each connected component $C \in \mathcal{CC}(\mathfrak{L})$ and its abstraction $\alpha(\{C\}) = \{\hat{C}\}$, there is a homomorphism uniquely determined by $h : V_C \to V_{\hat{C}}$ by*

$$h(v) = k_n([v])$$

where $[\cdot]$ denotes the equivalence class wrt. *α_c.*

Statements about the precision of the abstraction will be made in Section 3.3.

3.2 Abstract Rule Applications

In this section, it will be explained, how the rules of a graph transformation system $\mathcal{GTS}$ are applied to abstract topologies. Informally, the major difference is that abstract matches are not required to be injective. This is of course due to the possibility of summary nodes in an abstract topology. Several nodes in the left side of a rule may thus match the same summary node. In order to apply the rule as defined for the injective match case, identical copies of the matching summary node are to be made. This process is called *node materialization*.

Due to cluster abstraction, an element of an abstract topology, *i.e.* an abstract cluster, may stand for an arbitrary number of concrete clusters. The consequence for abstract matches is the possibility to allow an arbitrary number of copies of abstract clusters. This concept is called *cluster materialization*. Hence, a rule is not matched against a set of abstract clusters, but a *multiset* of abstract clusters. The set of multisets of a set M is written $\mathfrak{P}_m(M)$.

Definition 6. *A rule* (L, h, p, R) matches *a set* $M \in \mathfrak{P}(\mathcal{CG}(\mathfrak{L}))$*, if there exist a multiset* $M' \in \mathfrak{P}_m(M)$ *and a pair* $m = (m_n, m_e)$ *of mappings*

$$m_n : V_L \to \dot{\bigcup}_{G \in M'} V_G$$
$$m_e : E_L \to \dot{\bigcup}_{G \in M'} E_G$$

such that

1. *For all* $v \in V_L$*, if* $m_n(v) \in V_G$ *for some* $G \in M'$*, then* $\mathfrak{n}_L(v) = (\mathfrak{n}_G \circ m_n(v), \_)$.
2. $m_e(v_1, v_2) = (m_n(v_1), m_n(v_2))$ *for all* $(v_1, v_2) \in E_L$.
3. *If* $v \in dom(p)$ *and* $m_n(v) \in G$ *for some* $G \in M'$*, then* $p(v) = \{(\mathfrak{n}_G(v'), \_) | (v', m_n(v)) \in E_G$ *or* $(m_n(v), v') \in E_G\}$
4. *If there are* $v_1 \neq v_2 \in V_L$ *and* $G \in M'$ *such that* $m_n(v_1), m_n(v_2) \in G$ *and* $m_n(v_1) = m_n(v_2)$*, then* $\mathfrak{n}_G(m_n(v_1)) = (\_, 1)$*,* i.e. *a summary node.*

The pair m *is called an* abstract match*.*

The first three conditions of the above definition closely resemble the conditions in the definition of concrete matches. There are two exceptions. Node labels in an abstract topology carry a summary bit, which is irrelevant for the first three conditions. Secondly, there are additional clauses picking one among a multiset of abstract clusters for formulating the requirements in the definition. In fact, given an abstract topology M, there can be many multisets M' meeting the requirements in Definition 6. The multiset formalization allows to specify cluster materialization concisely. New to the match definition is requirement 4. It states, that non-injective places in a match have to occur at summary nodes.

The issue of cluster materialization was taken care of by the multisets in Definition 6; node materialization needs to be considered now. Node materialization is done, when a summary node v is matched. As many identical non-summary copies of v as there are nodes in the left graph of the rule matching v are then made. After that, the update is computed as specified for the concrete case in Definition 1. The updated graph is abstracted again in order to guarantee boundedness. Each summary node that is materialized from induces a case distinction: Either the summary node stood for exactly as many nodes as it was matched by and it disappears after materialization; or it stood for one more node than it was matched by and becomes a non-summary node after materialization; or it remains unchanged. This yields a blow-up exponential in the number of matched summary nodes. Technically, the blow-up is hidden in R_0 and R_1 being arbitrary subsets of the set of matched summary nodes.

Definition 7. *Let* $r \in \mathcal{R}$ *be a graph transformation rule and let* $M \in \mathfrak{P}(\mathcal{CG}(\mathfrak{L}))$ *be an abstract topology, such that* r *matches* M*. Let* M' *be the multiset meeting the requirements from Definition 6 and let* m *be the corresponding abstract match. Finally, let* G *be the disjoint union of the abstract clusters in* M'*. The graph* G' *is a* node materialization *of* M wrt. *rule* r *and abstract match* m*, if there are disjoint sets*

$$R_0, R_1 \subseteq \{v \in V_G \mid \mathfrak{n}_G(v) = (\_, 1) \ \wedge \ \exists v' \in V_L . m_n(v') = v\}$$

such that

1. $V_{G'} = (G \dot\cup \{v \in V_L \mid \mathfrak{n}_G(m_n(v)) = (\_, 1)\}) \setminus R_0$
2. $E_{G'} = (E_G \cup \{(v, v') \mid v \in L \wedge (m_n(v), v') \in E_G\} \cup \{(v', v) \mid v \in L \wedge (v', m_n(v)) \in E_G\}) \cap (V_{G'} \times V_{G'})$
3. $\mathfrak{n}_{G'}(v) = \begin{cases} \mathfrak{n}_G(v) & \text{if } v \in V_G \setminus R_1 \\ (l, 0) & \text{if } v \in R_1 \wedge \mathfrak{n}_G(v) = (l, 1) \\ (\mathfrak{n}_L(v), 0) & \text{otherwise} \end{cases}$

The (concrete) match $m' = (m'_n, m'_e)$, defined as follows, is called the materialization induced match*.*

$$m'_n(v) = \begin{cases} m_n(v) & \text{if } \mathfrak{n}_G(m_n(v)) = (\_, 0) \\ v & \text{otherwise} \end{cases} \qquad m'_e(v_1, v_2) = (m'_n(v_1), m'_n(v_2))$$

Some remarks about Definition 7:

- G in the definition is not an element of $\mathfrak{P}(\mathcal{CG}(\mathfrak{L}))$ any more. The disjoint graph union denies the possibility of canonical node naming.
- A materialized node carries the same "identity" as the original node in the left graph of the rule matching a summary node. These materialized nodes cannot be in the original abstract cluster, because it only contains canonical node names.
- The exponential blow-up mentioned recently is hidden in the R_i sets of Definition 7. The definition requires the R_i to be *some* subsets of the set of all summary nodes that are in the image of the match m.
- The pair m' defines in fact a (concrete) match. The crucial point is that m'_e is well-defined. This can be easily seen by plugging in the definition of m'_n and by then examining the definition of $E_{G'}$.

Definition 8. *Let $\mathcal{GTS} = (\mathcal{R}, \mathcal{I})$ be a graph transformation system. Let $r \in \mathcal{R}$ be a rule and let $M, M' \in \mathfrak{P}(\mathcal{CG}(\mathfrak{L}))$ be abstract topologies. M' is* a possible update of M *wrt. r, written $M \rightsquigarrow^{\alpha}_{r} M'$, if*

1. *r matches M by means of an abstract match m.*
2. *There is some node materialization G of M* wrt. *r and m, where the induced match is m'.*
3. *$G \rightsquigarrow_r G'$* wrt. *m'.*
4. *$M' = \alpha(cc(G'))$*

If M' is the union of all possible updates of M wrt. *r is written $M \rightarrow_r M'$. If M' is the union of all possible updates* wrt. *all rules in a set $\mathcal{R}$ of rules, we write $M \rightarrow_{\mathcal{R}} M'$. The abstract semantics of $\mathcal{GTS}$ is defined as follows:*

$$\begin{aligned} [\![\mathcal{GTS}]\!]^0_\alpha &=_{\text{def}} \alpha(cc(\mathcal{I})) \\ [\![\mathcal{GTS}]\!]^{i+1}_\alpha &=_{\text{def}} [\![\mathcal{GTS}]\!]^i_\alpha \cup M, \text{ where } [\![\mathcal{GTS}]\!]^i_\alpha \rightarrow^{\alpha}_{\mathcal{R}} M \\ [\![\mathcal{GTS}]\!]_\alpha &=_{\text{def}} \textstyle\bigcup_{i \geq 0} [\![\mathcal{GTS}]\!]^i_\alpha \end{aligned}$$

Definition 6, Definition 7, and Definition 8 are shown at work in Figure 5 and Figure 6. In the first of these figures, the rule [SENSOR1] is applied to the abstract topology T given in the bottom of Figure 4. In the second figure, [PASS] is applied to the same abstract topology. The set of all abstract clusters that can evolve for our case study is computed and presented using the implementation of our analysis in Section 4.

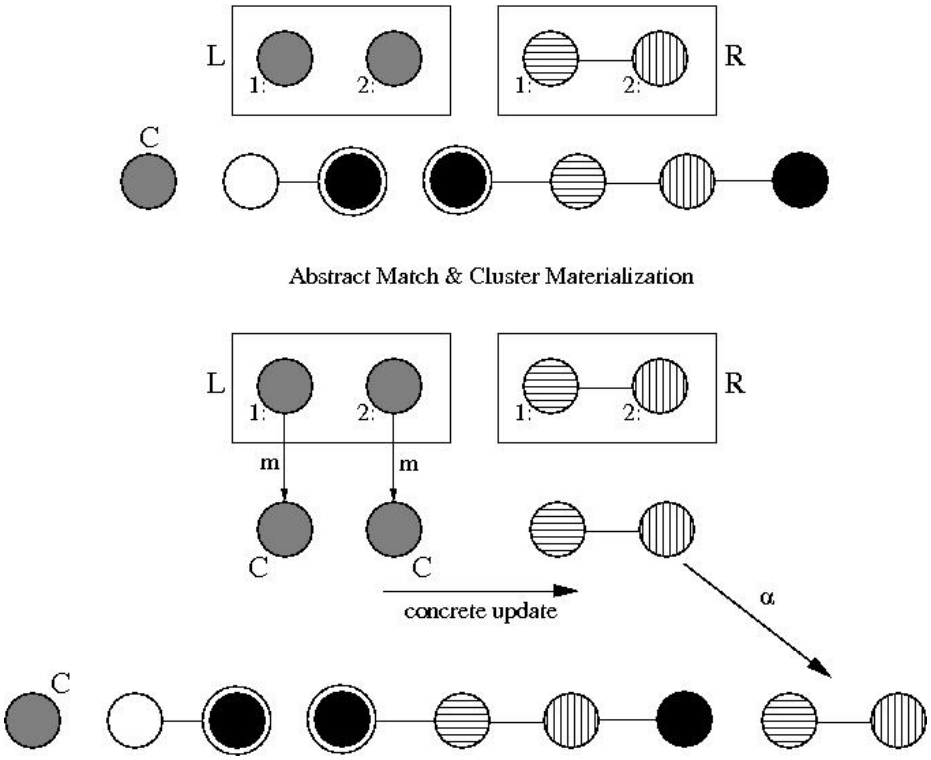


Fig. 5. Abstract Application of [SENSOR1] to abstract topology T: The left graph of the rule matches T by instantiating M' in Definition 6 to be the multiset $\{\!\{C, C\}\!\}$. This is called cluster materialization. As no summary nodes are involved [SENSOR1] can be applied to the disjoint graph union of $\{\!\{C, C\}\!\}$ like in the concrete case – the concrete update. The resulting cluster is then abstracted again and added to T. The final outcome of the application is shown in the bottom line. This is in fact the only possible application of r to T.

3.3 Properties

Lemma 4. *The abstract semantics $[\![\mathcal{GTS}]\!]_\alpha$ is a finite set for any graph transformation system $\mathcal{GTS}$*

This lemma is an immediate consequence of the bounded number of canonical names. Together with the monotonicity in the definition of $[\![\cdot]\!]^i_\alpha$ it guarantees the termination of an algorithm correctly implementing the technique.

The first and most important theorem is a soundness theorem. Each connected graph that can evolve from the initial graph is in the concretization of one abstract cluster from the abstract semantics, *i.e.* the abstract semantics is an *over-approximation* of the concrete semantics.

Theorem 1 (Soundness). *Let $\mathcal{GTS}$ be a graph transition system. For every $G \in [\![GTS]\!]$ and every connected component C of G, there is an abstract cluster $\hat{C} \in [\![\mathcal{GTS}]\!]_\alpha$ such that $\alpha(\{C\}) = \{\hat{C}\}$.*

Proof Let $\mathcal{GTS} = (\mathcal{R}, \mathcal{I})$. The proof is by induction on the length n of the sequence deriving G.

$$\mathcal{I} = G_0 \rightsquigarrow_{\mathcal{R}} G_1 \rightsquigarrow_{\mathcal{R}} \ldots \rightsquigarrow_{\mathcal{R}} G_n = G$$

Induction Base $n = 0$. In this case $G = \mathcal{I}$ and the correctness follows from the definition of $[\![\mathcal{GTS}]\!]^0_\alpha$ and the properties of α.
Induction Step: Assume we have proven the property up to some $n-1$, *i.e.* for the above derivation the induction hypothesis reads as follows. For each $C \in cc(G_{n-1})$ there is some abstract cluster $\hat{C} \in [\![\mathcal{GTS}]\!]_\alpha$ such that $\alpha(\{C\}) = \{\hat{C}\}$. Let $r \in \mathcal{R}$ be the rule leading to $G_{n-1} \rightsquigarrow_r G_n$.

If r is a create rule, the result follows immediately, because create rules can already be applied to $[\![\mathcal{GTS}]\!]^0_\alpha$, *i.e.* because of (4) in Definition 8, $[\![\mathcal{GTS}]\!]^1_\alpha \supseteq \alpha(cc(R))$ for each right graph R of each rule.

If $r = (L, h, p, R)$ has a non-empty left graph, the proof comprises two major subproofs: (i) A concrete match implies an abstract match; (ii) Node materialization works fine. The rest is obvious from clauses (3) and (4) of Definition 8.
Concrete match $\Rightarrow$ abstract match. Because of $G_{n-1} \rightsquigarrow_r G_n$, there exist a concrete match $m = (m_n, m_e)$ matching r against G. Let M denote the set of connected components of G_{n-1} matched by m, *i.e.*

$$M = \{C \in cc(G_{n-1}) \mid V_C \cap m_n(V_L) \neq \emptyset\}$$

Because of the induction hypothesis, we know that the multiset

$$M' = \dot{\bigcup}^m_{C \in M} \alpha(\{C\})$$

exists in $[\![\mathcal{GTS}]\!]_\alpha$. This is the choice for the multiset required in Definition 6. This point clarifies the concept of cluster materialization. We need a multiset here, because several concrete connected components may be abstracted to the same abstract cluster. From now on, we deliberately confuse M' and the disjoint union of its elements.

Lemma 3 showed that there is a graph homomorphism from M to M', the induction hypothesis implies that there is a homomorphism from L to M. We define m' to be the functional composition of these two, which is also a homomorphism. Homomorphism m' is our choice for the abstract match. As it is a homomorphism, we have proved requirements (1) and (2) of Definition 6. Requirement (3) states the fulfillment of partner constraints, which again follows immediately from the homomorphism property of m'. Partner constraints are obviously invariant under homomorphisms.

Consider requirement (4) and assume $m'_n(v_1) = m'_n(v_2)$ for $v_1 \neq v_2 \in V_L$. This means $k_n([m_n(v_1)]) = k_n([m_n(v_2)])$. As m_n is injective and k_n maps only α_c equivalent nodes to the same canonical name, the size of $[m_n(v_1)]$ must be at least 2 – meaning that $\mathfrak{n}_{M'}(m'_n(v_1)) = (\_, 1)$.
Node materialization. The crucial part in Definition 7 is how to pick the sets R_0 and R_1. Once we have shown that one choice coincides with the application of r, the rest of the

proof follows immediately, because all possible choices of R_0 and R_1 are considered in Definition 8. The set R_0 denotes those matched summary nodes that will disappear after materialization; R_1 stands for those becoming non-summary nodes. Hence the sets are determined by the difference of α_c equivalent nodes in G and how many of those nodes are matched. The set S of matched summary nodes is given by

$$S = \{v \in V_{M'} \mid \mathfrak{n}_{M'}(v) = (\_1) \wedge \exists v' \in V_L . m'_n(v') = v\}$$

The difference between matched and all nodes in an equivalence class is defined for any $v \in S$ and $v' \in V_L$ such that $m'_n(v') = v$:

$$d(v) = |[m_n(v')]| - |\{u \in V_L \mid m_n(u) \in [m_n(v')]|$$

yielding the following choice for R_i ($i = 1, 2$).

$$R_i = \{v \in S \mid d(v) = i\}$$

□

Corollary 1. *No connected graph $C \in \mathcal{CC}(\mathfrak{L})$ with $\alpha(\{C\}) \not\subseteq [\![\mathcal{GTS}]\!]_\alpha$ is a subgraph of a graph in $[\![\mathcal{GTS}]\!]$.*

This result can be lifted to edges between nodes with distinct labels rather straightforwardly.

Corollary 2. *If in any connected graph $C \in \mathcal{CC}\mathfrak{L}$ there is an edge between a node labeled l_1 and a node labeled l_2, then there is an abstract cluster in $[\![\mathcal{GTS}]\!]_\alpha$ containing two connected nodes labeled l_1 and l_2, respectively.*

Considering the abstract semantics for our platoon case study as given in Figure 8, we can prove that no each follower has exactly one leader at a time and that there are no such links as between a [LEADER] and a [BLDR] or a [LEADER] and a [FLDR].

More such results can be obtained by considering the precision of the abstraction. Here is one result that applies to all abstract clusters, in which summary nodes are only connected to one non-summary node. Such abstract clusters are precise up to the number $n > 2$ of nodes that a summary node represents. This is in particular the case for all abstract clusters in the abstract semantics of our platoon case study.

Lemma 5. *Let $\hat{C} \in \mathcal{CG}(\mathfrak{L})$ be an abstract cluster with summary nodes $v_1, \ldots, v_n$, such that each v_i has exactly one non-summary neighbor. Then for every connected graph G with $\alpha(\{G\}) = \{\hat{C}\}$, there exist $q_1, \ldots, q_n > 1$ such that G is isomorphic to $\hat{C}$ with summary node v_i materialized q_i times (where the original summary node disappears in the materialization).*

4 Implementation

The technique is implemented in a tool called `hiralysis`. This tool consists of app. 3000 lines of C code. It reads a textual description of a graph transformation system

as input and generates two output files: `gts.out` and `result.out`. Both are graph description files in the `.gdl` format that can be visualized by the `aisee` graph viewer [8]. The first file gives a graphical representation of the specified GTS, the second visualizes the abstract semantics of the input GTS, *i.e.* it visualizes a set of abstract clusters. Sample inputs and outputs to the tool are given in Figure 7 and Figure 8, respectively.

Figure 7 implements the platoon merge case study presented in this work. The textual description requires the declaration of node and edge labels (in fact, the tool implements slightly more than presented here). The user may declare an initial graph, which is the empty graph in our case study. There is one keyword for each of the three types of rules: create, destroy, and edge rules. The latter are simply called `rule` in the specification. Create and destroy rules require a single graph, edge rules require two graphs. In the graphs, all nodes are given names, that allow to specify both the edges and the mapping that comes with each graph transformation rule. Node names are followed by a label. The `disjoint` keyword tells the tool, that this rule shall only be applied if the connected components of the left graph match two distinct connected components in the matching graph. The final rule displays an example of a partner constraint. The node named `x1` must have adjacent nodes, that all have the label `fldr`.

On the given input file, the tool computes the output in Figure 8 in less than 0.1 seconds.19 rules were applied in five iterations, where the iterations correspond to the i in the definition of $[\![\cdot]\!]_\alpha$. An interesting effect is achieved, when the tool is applied to the same input with the partner constraint left out. It then needs 3 seconds to apply 8913 rules in 100 iterations to obtain 96 abstract clusters. Most of these abstract clusters look rather strange. This demonstrates the huge impact of partner constraints.

Additionally, we have implemented and analyzed a combined merge/split protocol. During the design process the tool proved very helpful in discovering design errors, many of them related to missing/wrong partner constraints.

5 Hierarchical versus Canonical Abstraction

Certainly, Hierarchical Abstraction was inspired by the work on Shape Analysis [11]. In that approach concrete graphs are coded as logical structures. Abstraction works by summarizing objects indistinguishable under a set of abstraction predicates. One major difference to our abstraction is the way edges are abstracted. Edges in abstract topologies in our work correspond to 1/2-edges in Canonical Abstraction. Such edges denote that there *may* be such an edge in the concretization. Additionally, they offer 1- or *must*-edges that we do not consider. Put differently, Canonical Abstraction offers both $\forall\forall$ and $\exists\exists$ abstraction, whereas Hierarchical Abstraction only supports the latter. (Even though it may be extended to have both.) Summary nodes in [11] differ for ours, too. We distinguish 0, 1, and > 1, whereas Canonical Abstraction supports 0, 1, and ≥ 1.

It seems tedious to encode DCS transformation rules in the setting of [11], because it was designed for the analysis of heap-manipulating programs. Updates in such programs are much more restricted than in graph transformation systems. Hierarchical Abstraction in the so far un-parameterized form is very easy to specify. It does not

need complicated instrumentation predicates or update formulas. Rather, it comes with a stand-alone tool. Certainly, this comes at the price of parametricity.

A clear advantage of Canonical Abstraction over Hierarchical Abstraction is the precise statement about property preservation made by a so-called *embedding theorem*. Formulas evaluating to 0 or 1 in an abstract structure will evaluate to 0 or 1 in each concrete structure represented by it. In the next section, we try to give some hints on how Hierarchical Abstraction could be encoded in the other approach.

5.1 Encoding of Hierarchical Abstraction as Canonical Abstraction

Two formulas need to be specified: ψ_c and ψ_t, The first formula defines the equivalence of nodes as in cluster abstraction, which is easy enough in first-order logic. The second formula describes that two nodes are in the same connected component. As [11] comes with transitive closure, this poses no problem, either. However, we cannot simply summarize objects v_1 and v_2 if both $\psi_t(v_1, v_2)$ and $\psi_c(v_1, v_2)$ hold in a concrete structure, because this would not result in a bounded abstraction. There can be arbitrarily many connected components. Rather v_1 and v_2 are summarized, if $\psi_c(v_1, v_2)$ holds *and* if the sets $V_i = \{v \mid \psi_t(v, v_i)\}$ are isomorphic after quotient building under the equivalence relation specified by ψ_c. This is not straightforward to formalize using Canonical Abstraction.

The only possible way of encoding seems to use an exponential number of additional predicates, where there is a predicate for each possible abstract cluster (exponentially many) being true for all nodes in such a cluster. There must be update formulas for all of these many additional predicates, which may make the analysis very hard.

6 Ongoing Work

6.1 Properties of the Abstraction

Results about the precision of Hierarchical Abstraction are scarce so far. We are currently investigating, whether there are logics whose formulae are preserved under Hierarchical Abstraction. Ideally, there would be some notion of *embedding* as defined in [11]. A suitable logic together with a refined transition system (see next section below) may allow for model-checking to be applied.

6.2 Refined Transition Systems

This paper presents the mere computation of all possible clusters. So far, Hierarchical Abstraction does not track relations between abstract clusters and can hence only be applied to prove some safety properties. The first step here is to track the information which and how many abstract clusters were involved in a rule application yielding a relation on multisets of abstract clusters. Consider the example of applying [SENSOR1] to two [FAGENT] objects. This results in the pair

$$\{\!\{C_1, C_1\}\!\} \mapsto \{\!\{C_2\}\!\}$$

where C_1 is the abstract cluster with a single $[\text{FAGENT}]$, and where C_2 is the abstract cluster with the two connected $[\text{BLDR}]$ and $[\text{FLDR}]$. Note that, the right hand side of $\mapsto$ can in general be a true multiset, too. Later it may even be advisable to track evolution node-wise to refine the transition system even further.

6.3 Parameterization

Hierarchical Abstraction as presented so far is not very flexible. However, there are several possibilities for parameterization – each presenting a further line of research.

Cluster abstraction can be performed using other equivalence relations than the one shown for Hierarchical Abstraction. A less precise one could be considering equally labeled nodes to be equivalent – regardless of their partners. This particular choice would reduce complexity by an exponential factor and might still be precise enough. Requirements for other equivalence relations are, that they must have a finite number of equivalence classes in every quotient graph. Furthermore, they must have the property "concrete match implies abstract match" as was shown for Hierarchical Abstraction. This is the case for all equivalence relations defining a (label-preserving) homomorphism. It implies that the "equally labeled nodes" relation is the least precise relation satisfying these requirements.

Partner constraints can be made more or less expressive depending on the application. However, in order to derive correctness from the soundness of Hierarchical Abstraction the satisfaction of a partner constraint should be invariant under abstraction, in particular invariant under graph homomorphisms.

Multiplicity is another possible parameter to gain either precision or speed. So far, Hierarchical Abstraction tracks only whether there are zero, one, or more than one concrete nodes represented by nodes in abstract clusters. This could be refined arbitrarily. Also, one may choose to gain speed and make the multiplicity information coarser – as is the case in [11].

Even more interesting is the second layer of multiplicity: cluster multiplicity. It was not explicitly mentioned, but due to the Cluster Multiplicity Principle only one multiplicity value is tracked for abstract clusters: ≥ 0. It is straightforward to be more precise there yielding a notion of *summary clusters*. Certainly, this comes at the price of a more expensive analysis.

Cluster definition Though not being arbitrary, the choice of connected components as clusters is not mandatory. It comes with some really nice properties, however. The Cluster Multiplicity Principle would most probably need to be dropped, if one decided on parameterizing on the definition of clusters. Also, there may be edges between different abstract clusters in the more general case, which inflicts technical overhead. Unless applications require, this is the last place where one should think of parameterization.

7 Related Work

Specifying DCS One of the earliest approaches to specify communicating systems are Communicating Finite State Machines [9]. They are not suited for the specification of

DCS, though, because they lack dynamics and deal with a fixed number of communicating processes. The π-calculus [10] may be the most prominent approach of describing communicating processes. Undoubtedly, one is able to find an encoding of DCS graph transformation systems in the π-calculus. But we are sure, that DCS are easier to specify, understand, and analyze, if they are modeled directly using graphs. Furthermore, we are not interested in concrete messages that are communicated, but only in the effects on the communication topology. Hence, graph transformation systems [1] seem like the first choice to specify DCS. Our formalization is closest to the single pushout approach of graph transformations, but slightly adapted. Relabeling of nodes is hard to handle in the standard approach [6], but easy to use here. Negative application conditions are common in graph transformation systems. Partner constraints can in fact be coded as such. Though less expressive than full negative application conditions, they reflect nicely the Transition Principle and are crucial for the correctness of the Cluster Multiplicity Principle. Finally, injective matches are not required in general transformation systems. Again, they suffice for specifying DCS and simplify the formal treatment in this work.

Graph Abstractions Finding a bounded abstraction of potentially unbounded graphs is a widespread task in computer science. Our abstraction was originally motivated by Canonical Abstraction [11, 12]. The relation to this approach was clarified in Section 6.3. The graph abstraction presented in the work on Canonical Graph Shapes [13] works by building quotient graphs – like we do – and thus lacks the expressiveness of 1 and 1/2 edges of [11]. On the other hand, it offers a more sophisticated generic approach to multiplicities: multiplicities algebras. Apart from that it can be straightforwardly coded as Canonical Abstraction. Compared to our work it does not distinguish different connected components at all.

Verification There are numerous papers on the verification of infinite-state transition systems. To name only a rather recent one, there is Abstract Regular Model Checking [14]. It is not suited for the verification of DCS in general, because configurations are stored as finite automata that are not expressive enough to encode arbitrary graphs.

[15] presents an approach to verification of graph transformation systems that does not use abstraction. Rather it deduces properties of a complete system by decomposing the system into views and verifying properties for the views.

[16] observes two major lines in verification of graph transformation systems by abstraction: the unfolding approach and the partitioning approach. The unfolding approach [17, 18] is based on the unfolding semantics of the given graph grammar [19] and approximates its behavior by means of finite Petri net-like structures. Our approach rather falls in the class of what is called partitioning approach in [16] due to the way abstract topologies are constructed. Among this approach there is a pragmatic comparison of concrete implementations of verification algorithms [20] that do not employ abstraction on the one hand. On the other hand there is the definition of a graph abstraction without the construction of an abstract transition system [13]. In contrast to our work so far, both [18] and [13] are much concerned about logic and how to define a logic such that properties expressed in it are maintained by the abstraction.

8 Conclusion and Future Work

We have presented Hierarchical Abstraction, a way of computing a finite approximate set of abstract topologies from a set of lightweight graph transformation rules. The abstraction is proven sound, *i.e.* all possible topologies of a DCS are conservatively approximated. It is one of the first works combining abstraction and (non-unfolding based) verification of graph transformation systems in a promising way. Its usefulness was demonstrated by a complex case study earlier approaches failed to verify. The technique is implemented and the tool showed useful in a first set of examples.

The next theoretical problem to be investigated is finding a logic that is suited to express interesting DCS properties, such that properties are preserved under Hierarchical Abstraction. Together with a refined transition system, this may enable us to model-check specifications. Several lines of ongoing and future work were presented in Section 6. The next immediate step to take is applying the tool to larger and more practically relevant examples.

References

1. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific (1997)
2. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: Symp. on Princ. of Prog. Lang., New York, NY, ACM Press (1977) 238–252
3. PATH: California partners for advanced transport and highway (1986-2003) http://www.path.berkeley.edu/.
4. Hsu, A., Eskafi, F., Sachs, S., Varaiya, P.: The design of platoon maneuver protocols for IVHS. Technical Report UCB-ITS-PRR-91-6, University of California, Berkley (1991)
5. Har'El, Z., Kurshan, P.: COSPAN user's guide. AT&T Bell Laboratories, Murray Hill, NJ (1987)
6. Habel, A., Plump, D.: Relabelling in graph transformation. In Corradini, A., Ehrig, H., Kreowski, H.J., Rozenberg, G., eds.: ICGT. Volume 2505 of Lecture Notes in Computer Science., Springer (2002) 135–147
7. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26** (1996) 287–313
8. AbsInt: Angewandte Informatik GmbH (1998-2005) http://www.aisee.com.
9. Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of the Association for Computing Machinery **30** (1983) 323–342
10. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press (2001)
11. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems **24** (2002) 217–298
12. Reps, T.W., Sagiv, S., Wilhelm, R.: Static program analysis via 3-valued logic. [22] 15–30
13. Rensink, A.: Canonical graph shapes. In Schmidt, D.A., ed.: ESOP. Volume 2986 of Lecture Notes in Computer Science., Springer (2004) 401–415
14. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. [22] 372–386
15. Heckel, R.: Compositional verification of reactive systems specified by graph transformation. In: FASE. (1998) 138–153

16. Baldan, P., König, B., Rensink, A.: Graph grammar verification through abstraction. Dagstuhl Seminar Proceedings 04241 (2004)
17. Baldan, P., Corradini, A., König, B.: Verifying finite-state graph grammars: An unfolding-based approach. In Gardner, P., Yoshida, N., eds.: CONCUR. Volume 3170 of Lecture Notes in Computer Science., Springer (2004) 83–98
18. Baldan, P., König, B., König, B.: A logic for analyzing abstractions of graph transformation systems. In Cousot, R., ed.: SAS. Volume 2694 of Lecture Notes in Computer Science., Springer (2003) 255–272
19. Baldan, P., Corradini, A., Montanari, U.: Unfolding and event structure semantics for graph grammars. In: FoSSaCS. (1999) 73–89
20. Rensink, A., Schmidt, Á., Varró, D.: Model checking graph transformations: A comparison of two approaches. In Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G., eds.: ICGT. Volume 3256 of Lecture Notes in Computer Science., Springer (2004) 226–241
21. Lev-Ami, T., Sagiv, M.: TVLA: A framework for Kleene based static analysis. In: Static Analysis Symposium, Springer (2000) http://www.math.tau.ac.il/∼ rumster.
22. Alur, R., Peled, D., eds.: Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings. In Alur, R., Peled, D., eds.: Computer Aided Verification. Volume 3114 of Lecture Notes in Computer Science., Springer (2004)

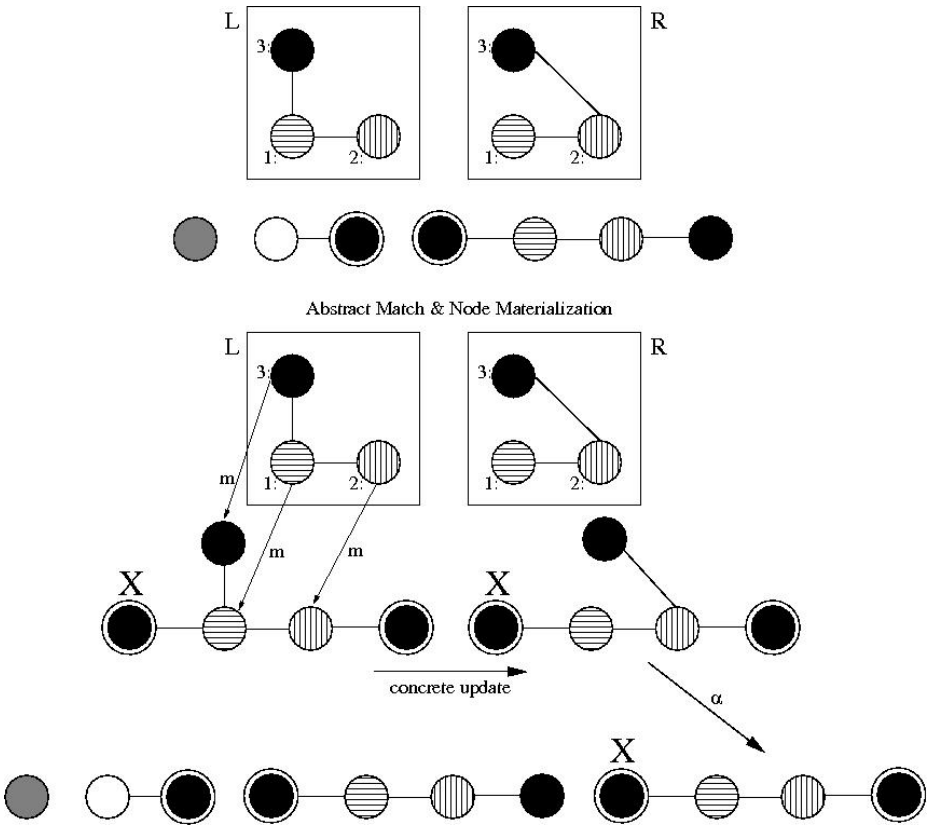


Fig. 6. Abstract Application of [PASS] to abstract topology T: The left graph of the rule matches the abstract topology T by instantiating M' in Definition 6 to the multiset $\{\!\{C\}\!\}$. Only one cluster is thus materialized. Node 3 in [PASS] matches to the summary node X. A possible node materialization according to Definition 7 is shown, where X is maintained. The materialization is then updated like in the concrete case and finally abstracted again. In fact, this figure shows only one of three possible applications of [PASS] to T. The other applications would result in the disappearance of X or in X becoming a non-summary node.

```
nodelabels ldr, fa, flw, bldr, fldr;

edgelabels l;

empty; // initial graph

create [{x:fa},{}];

destroy [{x:fa},{}];

rule [{x1:fa,x2:fa}, {}], disjoint,
     [{x1:bldr,x2:fldr}, {(x1,x2):l} ];
// sensor between two free agents

rule [{x1:ldr,x2:ldr}, {}], disjoint,
     [{x1:bldr,x2:fldr}, {(x1,x2):l} ];
// sensor between two leaders

rule [{x1:fa,x2:ldr}, {}], disjoint,
     [{x1:bldr,x2:fldr}, {(x1,x2):l} ];
// sensor between free agent and leader

rule [{x1:ldr,x2:fa}, {}], disjoint,
     [{x1:bldr,x2:fldr}, {(x1,x2):l} ];
// sensor between leader and free agent

rule [{x1:bldr, x2:fldr, x3: flw}, {(x1,x2):l, (x1,x3):l}],
     [{x1:bldr, x2:fldr, x3: flw}, {(x1,x2):l, (x2,x3):l}];
  // passing a follower from back to front leader

rule [{x1:bldr,x2:fldr}, {(x1,x2):l}, partner(x1)={fldr} ],
     [{x1:flw, x2:ldr},{(x2,x1):l}];
// re-establish platoon after all followers are handed over;
// an example of a "partner constraint"
```

Fig. 7. The implementation of our case study as input to the `hiralysis` tool.

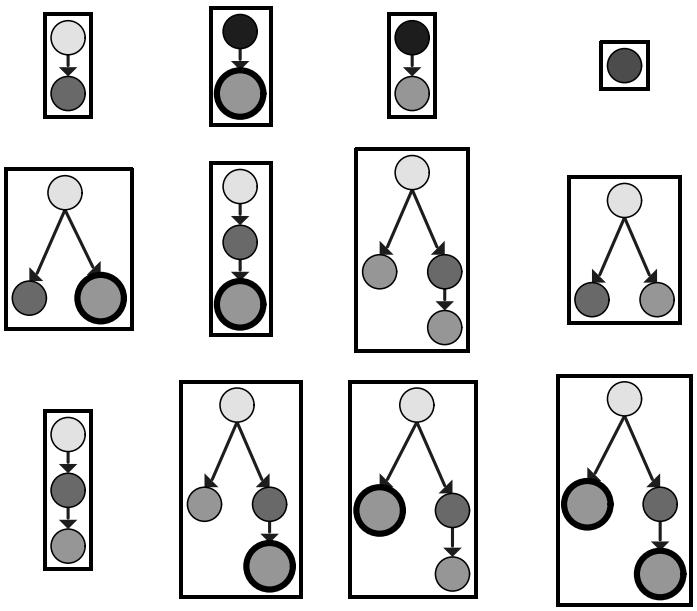


Fig. 8. Sample output from the `hiralysis` tool. Twelve abstract clusters are computed for our case study. The cluster on the top right shows the [FAGENT] cluster. Then, the next two to the left denote platoons of two, respectively more than two cars. All other abstract clusters stand for various intermediate steps occurring during a merge. Circles with thick borders are summary nodes.