

# Specification and Verification of Dynamic Communication Systems\*

Jörg Bauer (joba@cs.uni-sb.de)

Universität des Saarlandes, 66041 Saarbrücken, Germany, Phone/Fax: +49-681-302-5583/3065

Ina Schaefer (inschaefer@mpi-sb.mpg.de)

Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany

Tobe Toben, Bernd Westphal ({toben,westphal}@informatik.uni-oldenburg.de)

Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, Germany

## Abstract

*Dynamic communication systems (DCS) are complex because of their unboundedness in several dimensions. They have an unbounded and changing number of objects, a dynamically changing communication topology and unbounded message queues for asynchronous communication. We present a specification language for DCS that captures these features but is still amenable for formal verification. The verification of relevant properties of DCS is demonstrated using a combination of model-checking and abstract interpretation. Our approach is illustrated using the application domain of car platoons.*

## 1. Introduction

Formal verification of dynamic communication systems (DCS) is difficult because they exhibit unboundedness and dynamics in three dimensions: (i) a dynamically changing and potentially unbounded number of objects, (ii) a dynamically changing communication topology, and (iii) unbounded message queues for asynchronous communication. Prominent examples of DCS are wireless ad-hoc networks, traffic configuration systems such as car platoons [11], our running example, or radio-based train control as well as recent concepts like mobile or ubiquitous computing.

In this paper, we address formal verification of DCS properties by first providing an elaborate modelling language, expressive enough to specify the three characteristic features of DCS. The modelling language we propose in Section 2 explicitly speaks about *local states* of individual processes, e.g. to represent the current role in a protocol situation, the *communication topology*, in which

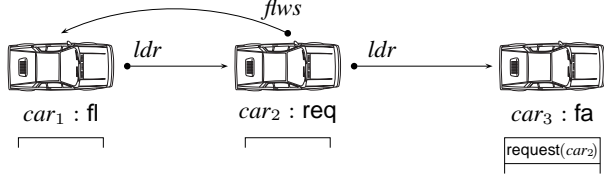
processes communicate, and the *message queues* buffering asynchronous communication. Crucial for the communication is the ability of processes to *send process identities*. A process can send its own identity or the identity of a process it knows to others as a means to change the communication topology. Physical behaviour, for example distance sensors announcing cars appearing in front, is not explicitly modelled but abstracted to a *non-deterministic* environment that triggers creation and deletion of processes (corresponding to cars appearing and disappearing from sight) and may send messages to processes (corresponding to announcements by the distance sensor). Although it would be possible to model DCS in low level process calculi we prefer a high level language in which all of the special features of DCS can be observed and reasoned about directly. If the three aspects named above are encoded into another formalism, high level, aspect specific properties are typically lost and cannot be exploited for tailored analyses.

In Section 3, the core of this work, we verify relevant properties of DCS using a combination of model-checking and abstract interpretation. Our technique can handle both the dynamically changing and unbounded number of objects and the dynamically changing communication topology. For this paper we do not handle unbounded message queues and impose a bound  $n$  on the length of all queues. First, we apply Query- and Data-Type-Reduction to a DCS specification to compute a bounded abstract model. This abstraction is often too coarse to establish desired properties by model-checking. Spurious counter-examples introduced by the abstraction can be eliminated and the abstraction refined using a novel topology analysis based on abstract interpretation. It computes an over-approximation of the topologies occurring for given DCS specifications.

Section 4 demonstrates the practical adequacy of DCS verification by translating a significant subset of UML into our modelling language, making it amenable to subsequent verification using our new techniques. Section 5 discusses

---

\*This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Centre "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).



**Figure 1. Snapshot of the merge protocol.**

related work and Section 6 gives directions of future work.

**Example: Car Platooning.** Our running example follows the “car platooning” system as studied by the PATH project [11]. The idea is to maximise traffic density on highways by merging autonomous cars into platoons. A car in this setting can either play the role of a *free agent* (not involved in a platoon), *leader* (the first car in the platoon), or *follower* (in a platoon in a follow-up position). Within a platoon each follower has a communication channel to its current leader and the leader maintains a channel to each of its current followers.

The key to platoon verification is the verification of the *merge manoeuvre*. It specifies how two platoons (or free agents) can merge building a larger platoon. A merge of two platoons is initiated if a sensor announces to the leader of a platoon that another platoon (or free-agent) is driving in front. Then the back platoon leader sends a request message to the leader of the front platoon asking for a merge. After that, the back leader sends a message to its followers introducing the front leader to be their new leader. The followers of the back platoon then announce themselves to their new leader, the front leader. Finally, the former back leader becomes itself a follower of the front leader.

Figure 1 depicts a snapshot of a merge manoeuvre. There are three cars where  $car_1$  and  $car_2$  form a platoon with  $car_2$  as the leader. Therefore  $car_2$  has a ‘*flws*’ communication channel to  $car_1$ , while  $car_1$  is a follower and has a ‘*ldr*’ channel to  $car_2$ .  $car_3$  is a free agent connected to  $car_1$  via a ‘*ldr*’ channel. In the message queue of  $car_3$  we see a message from  $car_2$  requesting a merge. In the course of the merge operation,  $car_2$  will hand over  $car_1$  to the new leader  $car_3$  and will itself become a follower of  $car_3$ .

Our framework allows to reason that the merge manoeuvre protocol is correct w.r.t. a certain specification, for instance, that no two cars believe to be each others’ leader at any point in time. We will show in this paper how we can establish the validity of such properties.

## 2. DCS Protocols

The behaviour of a DCS is formally defined by a *DCS protocol*  $\mathcal{P}$  specifying two aspects: the behaviour of processes and the behaviour of the environment. A protocol specifies the behaviour of all processes, i.e., each process is an instance of the protocol: one definition of process be-

haviour, but many processes may show this behaviour. Note that this assumption of homogeneity is only made to keep the presentation simple. The extension to a finite number of different protocols is straightforward.

Creation and destruction of processes, each equipped with a unique process identity, is modelled by an environment that abstracts from concrete physical behaviour. Freshly created processes will always be in one of the *initial states*  $A$ . A process may only be destroyed, if it is in one of the *fragile states*  $\Omega$ . Furthermore, the environment sends messages non-deterministically to arbitrary processes. The set  $\mathcal{E}_{\text{msg}}$  denotes the messages that the environment can send. They are a subset of the set  $\Sigma$  of all *messages*. Three components of a DCS protocol have not been mentioned, the set  $Q$  of all (local) *states* that a process can assume, a set  $\chi$  of *channel types* and a *successor relation* ‘*succ*’.

**DCS Protocols.** Formally, a DCS protocol is a seven-tuple  $\mathcal{P} = (Q, A, \Omega, \chi, \Sigma, \mathcal{E}_{\text{msg}}, \text{succ})$  with the following components:

1. a finite set of *states*,  $Q$ : The states a process can be in.
2. *initial states*  $A \subseteq Q$ : If a new process is created, it is in one of the states of  $A$ .
3. *fragile states*  $\Omega \subseteq Q$ : A processes in one of these states can be destroyed.
4. a finite set of *channel types*,  $\chi$ : Each channel type denotes a set of processes that a process may be connected to, i.e. each channel type stands for a potentially unbounded number of communication links between processes.
5. a finite set of *messages*,  $\Sigma$ : The identifiers of the messages that can be sent to processes.
6. *environment messages*  $\mathcal{E}_{\text{msg}} \subseteq \Sigma$
7. *successor relation* ‘*succ*’: See below.

**The Successor Relation.** Let  $\mathcal{P} = (Q, A, \Omega, \chi, \Sigma, \mathcal{E}_{\text{msg}}, \text{succ})$  be a DCS protocol and let  $Op$  be a set of operations working on sets, such as union or intersection. The successor relation *succ* reflects five different *actions* that processes can perform at run-time: sending and receiving of messages (with and without process identities) and local actions. To sum up,  $\text{succ} = \text{Send} \cup \text{SendId} \cup \text{Rec} \cup \text{RecId} \cup \text{Local}$ , where

- $(q, m, c, q') \in \text{Send} \subseteq Q \times \Sigma \times \chi \times Q$ : A process in state  $q$  sends a message  $m$  via channel  $c$  changing its state to  $q'$ .
- $(q, m, c_1, c_2, q') \in \text{SendId} \subseteq Q \times \Sigma \times \chi \times (\chi \dot{\cup} \{id\}) \times Q$ : Same as above. Additionally, the sender’s identity or an arbitrary process identity from another channel  $c_2$  is attached to the message.
- $(q, m, q') \in \text{Rec} \subseteq Q \times \Sigma \times Q$ : A process changes its state from  $q$  to  $q'$  by receiving a message  $m$ .

- $(q, m, c, op, q') \in RecId \subseteq Q \times \Sigma \times \chi \times Op \times Q$ : A process in state  $q$  receives message  $m$  with an attached identity. This identity is then processed by combining it with channel  $c$  using  $op$ .
- $(q, c_1, op, c_2, q') \in Local \subseteq Q \times \chi \times Op \times \chi \times Q$ : A process locally changes its state from  $q$  to  $q'$ . Additionally, it combines channels  $c_1$  and  $c_2$  using  $op$ .

**Configurations and Topologies.** In order to distinguish different processes, they are equipped with unique *process identities* from a countably infinite set  $Id$  of identities. Processes can be in different *configurations*. A configuration  $\sigma$  is a triple  $(q, C, M)$ , where  $q \in Q$  is the local state.  $C : \chi \rightarrow 2^{Id}$ , a total function mapping channels to sets of process identifiers, describes to which other processes the process is connected by its channels.  $M : Id \dot{\cup} \{env\} \rightarrow (\Sigma \times (Id \dot{\cup} \{\ominus\}))^*$ , a partial mapping from process identities to message queues, gives the contents of the message queues for messages sent by the environment or any other processes. Note, that each process has a message queue for each identity *and* one for the environment. Message queues are sequences of pairs of messages and parameter identities. Among the parameters there is a special parameter  $\ominus$  denoting that no identity was attached to the message. A configuration  $(q, C, M)$  is called *initial*, if  $q \in A$ ,  $C = \lambda x. \emptyset$ , and  $M$  is the empty mapping. The set of all configurations w.r.t. a given DCS protocol  $\mathcal{P}$  is written  $\mathcal{S}(\mathcal{P})$ .

A *topology* describes the global state of a DCS, i.e. all existing processes in their respective configurations. Formally, a topology  $\mathcal{N}$  is a partial mapping  $Id \rightarrow \mathcal{S}(\mathcal{P})$ . It is called *initial*, if all configurations in its range are initial.

**Topology Transitions.** This paragraph defines how topologies evolve. We write  $\mathcal{N} \rightarrow \mathcal{N}'$ , if topology  $\mathcal{N}$  evolves in one step into  $\mathcal{N}'$ . The relation  $\rightarrow$  is defined for a DCS protocol  $\mathcal{P} = (Q, A, \Omega, \chi, \Sigma, \mathcal{E}_{msg}, succ)$ . Assume a process with identity  $\iota \in dom(\mathcal{N})$ , such that  $\mathcal{N}(\iota) = (q, C, M)$ . Topology  $\mathcal{N}$  evolves into  $\mathcal{N}'$  if one of the following conditions is satisfied.

**send message** There is an action  $(q, m, c, q') \in succ$ , process  $\iota$  changes its local state to  $q'$ , and the rest of its configuration remains the same, i.e.  $\mathcal{N}'(\iota) = (q', C, M)$ . Furthermore, message  $m$  (without parameter identity) is attached to the queues of those processes that  $\iota$  is connected to via channel  $c$ , i.e.  $\mathcal{N}'(\iota_i) = (q_i, C_i, M_i.(m, \ominus))$  for each process  $\iota_i \in dom(\mathcal{N})$  with  $\iota_i \in C(c)$  and  $\mathcal{N}(\iota_i) = (q_i, C_i, M_i)$ . As no processes are created or disappear  $dom(\mathcal{N}') = dom(\mathcal{N})$  and processes other than  $\iota$  and the  $\iota_i$  remain unaffected.

**send identity** Analogous to the previous case for some  $(q, m, c_1, c_2, q') \in succ$ . The only difference is that a pair  $(m, \iota')$  is added to the message queues of the connected processes,  $\iota'$  non-deterministically chosen from  $C(c_2)$ . If  $c_2 = id$ , then the pair  $(m, \iota)$  is added to these

queues. This is our mechanism to make processes get to know each other: by sending identities.

**receive identity** There is an action  $(q, m, c, op, q') \in succ$  and a message  $(m, \iota')$  in front of a queue of process  $\iota$ , i.e. there is a  $\iota_0 \in Id$  with  $M(\iota_0) = (m, \iota').M'$ . The received identity  $\iota'$  is then combined with the identities from  $\iota$ 's channel  $c$  using the set operation  $op$ . Process  $\iota$  changes its local state to  $q'$ . Formally,  $\mathcal{N}' = \mathcal{N}[\iota \mapsto (q', C', M')]$ , where  $C' = C[c \mapsto op(C(c), \{\iota'\})]$ .

**receive message** Analogous to the previous case.

**modify channel** There is an action  $(q, c_1, op, c_2, q') \in succ$ . Just like when receiving messages, only process  $\iota$  is affected changing its local state to  $q'$  and combining the sets of identities in channels  $c_1$  and  $c_2$  using  $op$ . Formally,  $\mathcal{N}' = \mathcal{N}[\iota \mapsto (q', C[c_1 \mapsto op(C(c_1), C(c_2))], M)]$ .

**environment message** The update in this case is like in the sending cases, except that the sent message is in  $\mathcal{E}_{msg}$  and entered into the receiver's 'env' queue. This case is always enabled.

**appearance** A new process can always be created starting in one of its initial states and not being connected to anyone else, i.e.  $\mathcal{N}' = \mathcal{N}[\iota' \mapsto \sigma]$  where  $dom(\mathcal{N}') \setminus dom(\mathcal{N}) = \{\iota'\}$  and  $\sigma$  is initial.

**disappearance** Process  $\iota$  is in a fragile state, i.e.  $q \in \Omega$ , and destroyed. Formally,  $\mathcal{N}' = \mathcal{N} \upharpoonright_{dom(\mathcal{N}) \setminus \{\iota\}}$ .

The above conditions are not disjoint, i.e.  $\rightarrow$  is not deterministic. The first five actions correspond to the transitions specified by *succ* and happen without environment interference. The remaining actions are always enabled and reflect the more non-deterministic behaviour of a DCS – modelled by the environment.

**Runs and Semantics.** A *run* of a DCS protocol  $\mathcal{P}$  is a sequence  $\mathcal{N}_0 \mathcal{N}_1 \dots$  of topologies such that for all  $i \geq 0$   $\mathcal{N}_i \rightarrow \mathcal{N}_{i+1}$ . The set of all runs of  $\mathcal{P}$  starting in an initial topology is called the *semantics* of  $\mathcal{P}$ , written  $\llbracket \mathcal{P} \rrbracket$ .

## 2.1. Example: Merge Protocol

To illustrate the formal definitions on a practical example, we define the merge manoeuvre as introduced in Section 1 in terms of the DCS protocol  $\mathcal{P}_M = (Q, A, \Omega, \chi, \Sigma, \mathcal{E}_{msg}, succ)$  with

- states  $Q = \{fa, ld, req, hnd, clr, fl, ann\}$ ,
- initial state  $A = \{fa\}$ ,
- fragile states  $\Omega = \{fa, fl\}$ ,
- channels  $\chi = \{ldr, flws\}$ ,
- messages  $\Sigma = \{car\_ahead, request, new\_ldr, new\_flw\}$ ,
- and environment message  $\mathcal{E}_{msg} = \{car\_ahead\}$ .

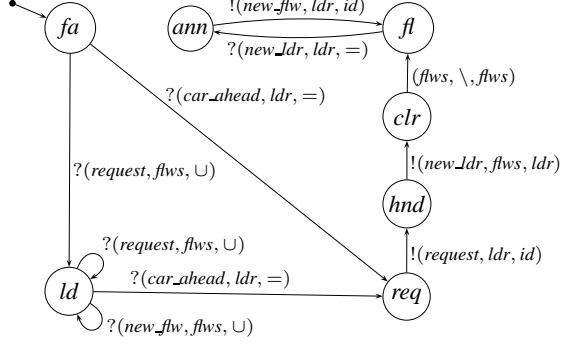


Figure 2. DCS Protocol for merge.

The successor relation  $succ$  is graphically represented in Figure 2. For each element  $s = (q, \dots, q') \in succ$  with source state  $q$  and target state  $q'$ , a transition between  $q$  and  $q'$  is drawn and annotated with the remaining components of  $s$ . For the sake of readability, elements of the action sets  $Send$  and  $SendId$  are prefixed with ‘!’, and elements of  $Rec$  and  $RecId$  are prefixed with ‘?’.

The protocol works as follows. The car’s sensor that detects other cars driving in front is modelled as a ‘ $car\_ahead$ ’ message sent by the environment. This message carries the identity of the newly detected car. The protocol reacts to this message if the car is either a free-agent, i.e. in state ‘ $fa$ ’, or if it is the leader of a platoon, i.e. in state ‘ $ld$ ’. In both cases, the ‘ $ldr$ ’ channel is modified by the assignment operator  $= \in Op$  to comprise the received identity. The car moves to state ‘ $req$ ’ from which it will send a ‘ $request$ ’ message carrying its own identity ‘ $id$ ’ to its new leader.

The transition starting at state ‘ $hnd$ ’ now hands over the set of follower cars to the new leader in which a ‘ $new\_ldr$ ’ message containing the identity stored in the ‘ $ldr$ ’ channel is sent to all identities that are stored in the ‘ $flws$ ’ channel. A car that is in state ‘ $fl$ ’ reacts to this ‘ $new\_ldr$ ’ message by storing the received identity in the ‘ $ldr$ ’ channel and entering state ‘ $ann$ ’. On the transition back to the ‘ $fl$ ’ state, the follower car announces itself to its new leader by sending a ‘ $new\_flw$ ’ message carrying its own identity as parameter. The former leader in state ‘ $clr$ ’ now clears its ‘ $flws$ ’ channel and becomes itself a follower car by entering state ‘ $fl$ ’.

It remains to explain how a car reacts to ‘ $request$ ’ and ‘ $new\_flw$ ’ messages. These messages are handled by the self-loops of the ‘ $ld$ ’ state in which the identity that comes with the corresponding message is added to the ‘ $flws$ ’ channel. The same applies to the transition from state ‘ $fa$ ’ to ‘ $ld$ ’ triggered by a ‘ $request$ ’ message.

To see how different instances of the DCS protocol  $\mathcal{P}_M$  interact, we exemplarily sketch a topology evolution leading to a platoon of size three. The messages that are exchanged during the manoeuvre are shown in the sequence diagram in Figure 3. In the following run, the channel contents are given as tuple  $\langle C(ldr), C(flws) \rangle$  and configurations

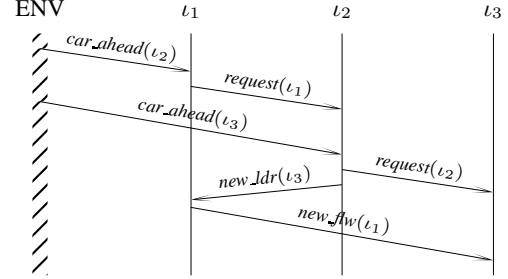


Figure 3. Sequence Diagram.

are only shown if one of its components has changed.

Starting from the empty topology, two processes  $\iota_1, \iota_2$  are successively created by the environment:

$$[] \rightarrow [\iota_1 \mapsto (fa, \langle \emptyset, \emptyset \rangle, [])] \rightarrow [\iota_1, \iota_2 \mapsto (fa, \langle \emptyset, \emptyset \rangle, [])]$$

We assume the car with identity  $\iota_2$  is driving in front and car  $\iota_1$  is notified about car  $\iota_2$  via a ‘ $car\_ahead$ ’ message. This triggers a sequence of transitions in car  $\iota_1$  leading to state ‘ $fl$ ’:

$$\begin{aligned} &\rightarrow [\iota_1 \mapsto (fa, \langle \emptyset, \emptyset \rangle, [env \mapsto (car\_ahead, \iota_2)]), \iota_2] \\ &\rightarrow [\iota_1 \mapsto (req, \langle \{\iota_2\}, \emptyset \rangle, []), \iota_2] \\ &\rightarrow [\iota_1 \mapsto (hnd, \langle \{\iota_2\}, \emptyset \rangle, []), \\ &\quad \iota_2 \mapsto (fa, \langle \emptyset, \emptyset \rangle, [\iota_1 \mapsto (request, \iota_1)])] \\ &\rightarrow [\iota_1 \mapsto (clr, \langle \{\iota_2\}, \emptyset \rangle, []), \iota_2] \\ &\rightarrow [\iota_1 \mapsto (fl, \langle \{\iota_2\}, \emptyset \rangle, []), \iota_2] \end{aligned}$$

The ‘ $request$ ’ message that has been sent by car  $\iota_1$  is now received by car  $\iota_2$ , leading to state ‘ $ld$ ’:

$$\rightarrow [\iota_1 \mapsto (fl, \langle \{\iota_2\}, \emptyset \rangle, []), \iota_2 \mapsto (ld, \langle \emptyset, \{\iota_1\} \rangle, [])]$$

If now a third car  $\iota_3$  enters the scene in front of this platoon, car  $\iota_2$  will initiate a merge after the reception of the corresponding ‘ $car\_ahead$ ’ message. This corresponds to Figure 1 (with  $car_i = \iota_i$ ) where car  $\iota_3$  is just about to receive the merge request of car  $\iota_2$ . During this manoeuvre, car  $\iota_2$  hands over his follower  $\iota_1$  to the new leader of the platoon, car  $\iota_3$  (cf. Figure 3). After this second merge manoeuvre, the resulting topology is the following:

$$\begin{aligned} &[\iota_1 \mapsto (fl, \langle \{\iota_3\}, \emptyset \rangle, []), \iota_2 \mapsto (fl, \langle \{\iota_3\}, \emptyset \rangle, []), \\ &\quad \iota_3 \mapsto (ld, \langle \emptyset, \{\iota_1, \iota_2\} \rangle, [])] \end{aligned}$$

### 3. Protocol Verification

Besides formally capturing the behaviour of a DCS like car platooning in form of a DCS protocol, our overall aim is to provide automated analyses of DCS properties. To this end, Section 3.1 introduces METT, a variant of temporal logic that provides means to refer to all aspects of a

DCS, i.e. process identities, creation and destruction, process states, queue-based communication, and topologies.

Our approach to verification combines two techniques. Assuming finite bounds on the length of queues, we can directly apply state-of-the-art techniques for the abstraction of infinite-state systems into finite-state model-checking problems, if infiniteness stems from unbounded creation and destruction of processes [9]. Section 3.2 uses the car platooning running example to describe the verification strategy.

This abstraction may produce spurious counter-examples. To rule them out, the abstraction has to be refined, e.g. by adding so called non-interference lemmata that state that certain spurious behaviour doesn't happen in the concrete model, or by representing more objects exactly (cf. Section 3.2).

In Section 3.3, additional invariants about DCS protocols are obtained by an abstract interpretation [5] based topology analysis [1], yielding a fairly precise over-approximation of all topologies that may occur during the run of a DCS protocol. The invariants can be used in the model-checking process or in proving non-interference lemmata furthering the automation of DCS verification.

### 3.1. A Logic for Reasoning about DCS

Our property specification language METT is an extension of the well-known temporal logic LTL with first-order quantification of so called *anonymous objects* since there are no global names in a topology. In addition, there are constructs to refer to communication between processes and to the topology. As explained in Section 5, METT resembles ETL [25].

The syntax of METT is given by the following grammar where  $c$  denotes a channel and  $m$  a message.

$$\begin{aligned} \phi ::= & p_1 = p_2 \mid \text{instate}[q](p) \mid \text{conn}[c](p_1, p_2) \\ & \mid \text{pend}[m](p_1, p_2, p) \mid \text{snd}[m](p_1, p_2, p) \mid \text{rcv}[m](p_1, p_2, p) \\ & \mid \odot p \mid \otimes p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \forall p. \phi \mid \mathbf{X} \phi \mid \phi_1 \mathbf{U} \phi_2 \end{aligned}$$

Semantically, the logical variables  $p$  denote process identities, i.e. the formula  $p_1 = p_2$  compares process identities. Subformulas referring to processes' local states, the topology, and pending messages are evaluated over a single topology  $\mathcal{N}$ . The local state formula  $\text{instate}[q](p)$  is satisfied iff the process with identity denoted by  $p$  is in  $\mathcal{N}$  and its local state is  $q$ . The topology formula  $\text{conn}[c](p_1, p_2)$  is satisfied iff the processes  $p_1$  and  $p_2$  are in  $\mathcal{N}$ ,  $p_1$  is in configuration  $(q, C, M)$ , and  $p_2 \in C(c)$ . The pending message formula  $\text{pend}[m](p_1, p_2, p)$  is satisfied iff the event  $(m, p)$  occurs in  $M(p_2)$ .

Communication, i.e. sending and receiving messages, and creation and destruction of processes involve two successive topologies  $\mathcal{N}$  and  $\mathcal{N}'$ . The formula

$\text{snd}[m](p_1, p_2, p)$  is satisfied in  $\mathcal{N}$  iff  $p_1$  is in both topologies and an  $(m, p)$  message appears at the end of the queue  $M(p_2)$  in  $\mathcal{N}'$ . Analogously, reception of a message is observed iff a message disappears from the front of  $M(p_2)$  in  $\mathcal{N}'$ . The notion of appearance and disappearance of messages in a queue is well-defined since our semantics is a strict interleaving semantics.

We consider a process  $p$  to be created in  $\mathcal{N}$ , denoted by  $\odot p$ , iff  $p$  appears freshly in  $\mathcal{N}'$ . A process  $p$  is destroyed  $\otimes p$  in  $\mathcal{N}$  iff it is present in  $\mathcal{N}$  and disappears in  $\mathcal{N}'$ .

The semantics of all other constructs of METT is standard and we shall use the abbreviations ' $\wedge$ ', ' $\rightarrow$ ', ' $\exists$ ', ' $\mathbf{G}$ ', and ' $\mathbf{F}$ '. The satisfaction relation between METT formulae and DCS runs is then obtained by structural induction.

**Examples.** The property "A process is a follower, whenever its leader channel is non-empty" is formalised in METT as follows:

$$\mathbf{G} \forall p_1, p_2. \text{conn}[\text{ldr}](p_1, p_2) \rightarrow \text{instate}[\text{fl}](p_1) \quad (1)$$

"A follower's request is pending until a sane leadership is established" is an example of a temporal METT property:

$$\begin{aligned} \mathbf{G} \forall p_1, p_2. & \text{instate}[\text{fl}](p_1) \wedge \text{conn}[\text{ldr}](p_1, p_2) \\ & \rightarrow (\text{pend}[\text{req}](p_1, p_2, p_1) \\ & \mathbf{U} (\text{instate}[\text{ld}](p_2) \wedge \text{conn}[\text{flws}](p_2, p_1))) \end{aligned} \quad (2)$$

Finally, the following is a slightly more elaborate topological example. It requires that a pathological case doesn't occur, namely that two followers mutually consider each other to be the leader:

$$\begin{aligned} \mathbf{G} \forall p_1, p_2. & \text{instate}[\text{fl}](p_1) \wedge \text{instate}[\text{fl}](p_2) \wedge \\ & p_1 \neq p_2 \rightarrow \neg(\text{conn}[\text{ldr}](p_1, p_2) \wedge \text{conn}[\text{ldr}](p_2, p_1)) \end{aligned} \quad (3)$$

### 3.2. DCS Model-Checking

There are three reasons for DCS being infinite-state systems. Firstly, there is no finite bound on the number of processes that exist in a topology, thus, secondly, also not on the number of channels. Thirdly, there is no finite upper bound on the length of message queues. For this paper we sidestep the last issue by assuming a finite bound  $n$  on the length of all queues and focus on the first issues. They can be treated with a combination of Query- and Data-Type Reduction [9] because DCS lie in the class of systems where the processes are instances of finitely many templates or classes.

**Query Reduction.** First of all, we can establish that DCS are symmetric in the process identities by applying syntactical rules [12]. Intuitively, the reason is that the literal value of process identities is never referred to in the actions of a DCS protocol  $\mathcal{P}$ . Thus if there is a run in  $\llbracket \mathcal{P} \rrbracket$  where a number of processes with identities  $\iota_1, \dots, \iota_k$  interact, then

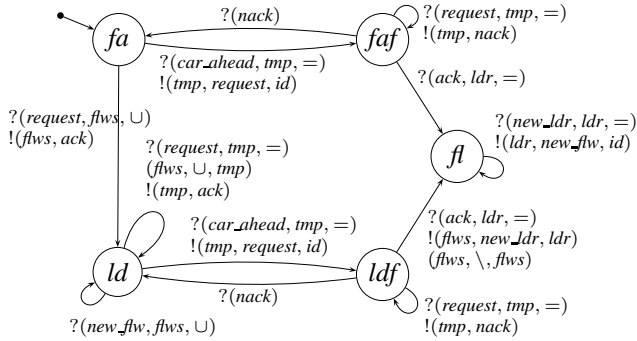


Figure 4. Fixed DCS protocol for merge.

by symmetry there is a run where processes with identities  $l'_1, \dots, l'_k$  go through the same scenario.

For example, property (3) excluding some invalid topologies, can be verified by considering only two representative cases (instead of checking all possible combinations), one where  $p_1 = \iota = p_2$  and one where  $p_1 = \iota_1 \neq \iota_2 = p_2$ ,  $\iota, \iota_1, \iota_2 \in Id$ . In this particular example the second case is already sufficient, since (3) explicitly considers different cars only. Note that Query Reduction allows to focus on finitely many representative cases, but provides no reduction of the model.

**Under-approximation as Falsification** checks whether a property holds in those runs of  $\llbracket P \rrbracket$  where at most  $m$  processes exist simultaneously and where the length of the queues doesn't exceed  $n$ . Model-checking property (3) for the DCS protocol from Sect. 2.1 with  $m = 3$  and  $n = 2$  already unveils a serious flaw in the DCS protocol as it doesn't negotiate the merge. If two cars happen to see each other in front, for instance since they drive on parallel lanes, they simultaneously execute the protocol, and both end up as followers considering each other as their leader. The DCS protocol shown in Figure 4 introduces positive and negative acknowledgements to fix this problem. To avoid the representation of dispensable intermediate states in Figure 4, we merged successive states with one outgoing transition into *one* transition labelled by the corresponding *sequence* of actions. Checking the accordingly changed model confirms satisfaction of property (3).

A similar approach to formal verification of the car platooning case study has been taken by the PATH project [16]. They modelled an instance of the merge protocol comprising two cars. Model-checking confirmed that the two cars merge model complies to an observer automaton requiring that the merge operation completes.

But from such results about an under-approximation, i.e. the results of the PATH project and ours as reported so far, we can only conclude (by Query Reduction) that all symmetric cases obey the property. However, we cannot conclude the correctness of the protocol in general because most runs of the system, in particular those with possible

interferences among multiple cars requesting a merge simultaneously, are excluded.

Under-approximation is an adequate approach for falsification since in this case model-checking is efficient. It does not suffice, however, to establish correctness in general.

**Over-approximation: Data-Type Reduction.** Considering the actions a DCS protocol can take, we observe that the operations on process identities can be seen as accesses to an unbounded array-like data-structure with process identities as indices. For example, the local state of each process  $\iota$  can be stored as  $q(\iota)$  with  $q : Id \rightarrow Q$ . The growing and shrinking extension of topologies within a run can be represented by explicitly keeping track of whether a process identity is active or not. Thus DCS protocol runs are represented as sequences of valuations of finitely many unbounded data-structures.

In this setting we apply a particular abstract interpretation called Data-Type Reduction [14] (DTR) to the DCS protocol yielding a finite over-approximation of the original DCS protocol (following [9]). Given a finite set of process identities whose configurations shall be represented exactly, DTR maps all other identities to a special identity  $\perp$ . The resulting set of identities is finite. If the content of channels is not counted, like in the running example, the size of channels also becomes bounded.

The special identity  $\perp$  still occurs in messages and channels, e.g., if we represent two cars exactly then the first car may receive a 'car\_ahead' environment message with  $\perp$  as parameter and subsequently send a request to  $\perp$ .

The process with identity  $\perp$  is also regularly scheduled. In this case determining the current state means accessing  $q(\perp)$  which is where the abstract interpretation comes into play. The expression  $q(\perp)$  yields an upper bound of all possible states a DCS process can be in, or equivalently, non-deterministically any of the possible states. The protocol description then yields a transition and a successor state for the obtained state, resulting for example, in sending a new-leader message to one of the current followers. Determining the current followers means accessing  $flws(\perp)$  which in turn non-deterministically yields any possible set of process identities that may comprise the exactly represented ones as well as  $\perp$ . For more details we refer to [9].

A DTR of the manually obtained model can automatically be computed [23]. Model-checking the over-approximation yields a counter-example which is, by close examination, spurious. It was caused by scheduling  $\perp$  in a topology where (exact) car  $\iota_1$  is a follower of (exact) car  $\iota_2$ . The configuration of  $\perp$  was chosen such that it considers car  $\iota_2$  to be its follower and announces  $\iota_1$  to be the new leader using a 'new\_ldr' message. A topology where  $\iota_1$  and  $\iota_2$  form a platoon and a third (different) car has  $\iota_2$  among its followers is not possible in the exact model. We effectively refine the abstraction by explicitly excluding the spurious

behaviour using the assumption:

$$\mathbf{G} \forall p_1, p_2, p_3. \quad \text{snd}[\text{new\_ldr}](p_1, p_2, p_3) \rightarrow \text{conn}[\text{flws}](p_3, p_1) \quad (4)$$

Now model-checking the implication ‘(4)  $\rightarrow$  (3)’ succeeds.

Assumptions like (4) are called *non-interference lemma*, as they state that certain kinds of interference between processes do not take place. These lemmas have to be established for each system separately. The non-interference lemma (4) can in fact be automatically proven using the analysis presented in Section 3.3.

**Open Ends.** The previous paragraphs presented how far state-of-the-art techniques help in the analysis of DCS. Leaving aside the unboundedness of queues, we are facing two issues that currently require manual intervention: First, identifying counter-examples as spurious is in general not decidable because on the one hand guessing the current state of the special process  $\perp$  makes  $\perp$  react in one step where an exact process had to take a number of steps. On the other hand  $\perp$  stands for arbitrarily many processes. Each occurrence of  $\perp$  in a spurious counter-example could be backed up by a different instance of the DCS protocol.

Second, deriving non-interference lemmata from counter-examples is a creative act that involves deep understanding of the system. Also, once found, non-interference lemmata must be proven.

Topology analysis as introduced in Section 3.3 can help to automate these two issues. As it computes a superset of all possible topologies, it may be possible to automatically deduce from this set a number of non-interference lemmata. Counter-examples can be rejected, if they rely on a topology that does not occur in this superset. Certainly, this approach is not complete, so we may fall back to manual intervention.

### 3.3. Topology Analysis

Knowing the topologies that can occur during the runs of a DCS protocol is a crucial step in DCS verification. An abstract interpretation based solution of this problem is presented in [1]. There, DCS topologies are represented by directed, node-labelled graphs. Evolution of topologies is specified in terms of graph transformation systems inducing an infinite-state transition system of possibly unbounded graphs. This transition system is over-approximated to a finite-state transition system of *abstract graphs*, where each abstract graph denotes a (possibly infinite) set of concrete graphs. The abstract system is computed automatically from a set of graph transformation rules. Consider [1] for more details about the abstract interpretation of graph transformation systems and [17] for an overview of graph transformation in general.

This algorithm is now utilised to compute a superset of all topologies occurring during the run of a DCS protocol.

We show how a set of graph transformation rules serving as input to this algorithm can be derived automatically from a DCS protocol. The result of the algorithm is then used to prove the spuriousness of certain error traces discovered in the previous section.

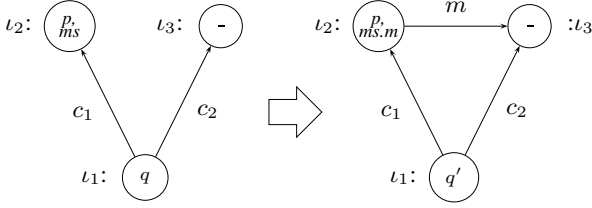
**DCS as Graphs.** It remains to show, how DCS protocols are encoded as graph transformation system. First, we show how a single topology is encoded, later we shall see, how a DCS protocol yields a set of transformation rules. Bounded message queues and local states of objects are encoded in a finite set of node labels. Communication channels are directed, labelled edges.

Formally, let  $\mathcal{P} = (Q, A, \Omega, \chi, \Sigma, \mathcal{E}_{\text{msg}}, \text{succ})$  be a DCS protocol. A topology of  $\mathcal{P}$  is coded as a directed, edge- and node-labelled graph as depicted in Figure 1. The only difference is that node-labels additionally encode bounded length message queues meaning, that the set of node labels will be  $Q \times \Sigma^n$ . A bound  $n$  on the length of message queues is needed in order to guarantee a finite set of node labels as required in [1]. Edge-labels are from the set  $\chi \cup \Sigma$ , where edges with a label from the set  $\Sigma$  of message names are used to connect a message parameter to a process that has received such a message (see below).

**Actions as Graph Transformation Rules.** A graph transformation rule consists of two graphs, a *left graph*  $L$  and a *right graph*  $R$ , and a relation between them, i.e. which nodes and edges in  $L$  and  $R$  correspond to each other. In Figure 5 this correspondence is given implicitly by the position of the nodes and edges. A rule can be *applied* to a graph  $G$ , if  $L$  is a subgraph of  $G$  (formally, if there is an injective graph morphism from  $L$  to  $G$ ). The result of the application is the replacement of  $L$  with  $R$  in  $G$ . Replacement is formalised in categorical terms. Again, cf. [17] for formal details.

Each element of  $\text{succ}$  is translated into a set of graph transformation rules. An example of a graph transformation rule resulting from the send identity element  $(q, m, c_1, c_2, q') \in \text{SendId}$  is shown in Figure 5. The left graph shows a situation where a process  $\iota_1$  in state  $q$  is connected to processes  $\iota_2$  and  $\iota_3$  via channels  $c_1$  and  $c_2$ . The result of sending the identity of process  $\iota_3$  present in channel  $c_2$  attached to message  $m$  to process  $\iota_2$  is exactly the right graph. Process  $\iota_1$  has changed its state to  $q'$ ,  $m$  is attached to the message queue of process  $\iota_2$ , and process  $\iota_2$  is connected to process  $\iota_3$  by an  $m$ -labelled edge. This denotes, that identity  $\iota_3$  was sent to  $\iota_2$  attached to message  $m$ . Similar translations are applied to the remaining elements of  $\text{succ}$ .

So far, the resulting graph transformation system does not take care of environment messages, creation, and destruction of processes. Environment messages are coded in the obvious way. For creation of a process, there is a graph



**Figure 5. Transform. rule: send message.**

transformation rule for each  $q \in A$  with an empty left graph and a single node labelled  $(q, \epsilon)$  being the right graph. Note, that these creation rules are always enabled because of the empty left graph. With respect to destruction of a process for each  $q \in \Omega$  there is a similar rule with left and right graph interchanged.

Note that this translation of a DCS protocol into graph transformation rules can be conducted fully automatically, because it works on syntax only. As before, a bound on the length of message queues is required. The result of the topology analysis applied to these graph transformation rules presents valuable assumptions helping to prove the non-interference lemmata from the previous section. We are currently working on integrating these two techniques into an automatic tool chain.

For our case study, property (4) can be coded easily into a graph that resembles the right graph of Figure 5 with  $p_i \mapsto \iota_i$ ,  $m$  replaced with  $new\_ldr$ , and arbitrary  $p$ ,  $q$ ,  $q'$ , and  $ms$ . All such graphs are then also required to have a  $flws$  edge from  $\iota_3$  to  $\iota_1$ . Running the topology analysis on the platoon DCS protocol reveals that all graphs occurring for the protocol meet this requirement, thus proving the non-interference lemma stated as property (4).

## 4. UML and DCS

A domain where DCS naturally appear is the large body of UML [15] models. The following three fundamental principles of UML relate to characteristic features of DCS.

Firstly, following the object-oriented approach underlying the definition of the Unified Modelling Language, systems are defined in terms of classes of which objects are instantiated at runtime. In general there are no finite bounds given on the number of objects alive during system runs, similar to DCS protocols. Secondly, UML explicitly provides asynchronous, message-queue-based communication by events as primitive without a priori imposing finite bounds on queue lengths. And thirdly, UML has the concept of associations (in different flavours) that are instantiated to links, basically directed connections between objects, which dynamically change during a system run, that is, UML models have a dynamic topology.

Formal analysis of UML models is a topic of growing interest. It is one advantage of a model-based development process that in early development phases a model of

a system under design can be tested for whether it satisfies its requirements, at best fully automatically. But even the most elaborate approaches to UML verification available today [24, 20] exclude, for example, the aspect of unbounded creation and destruction of objects in general and expect the user to provide finite bounds.

The DCS language is designed to study the three aspects of unboundedness in a clean, focused formalisation. In the following we sketch an embedding of an executable UML core into DCS, which enables us to finally transfer analysis procedures obtained for DCS to the domain of UML.

**Embedding Core UML into DCS.** We focus on an executable UML subset, e.g. following [8] and [24], i.e. we consider UML models where the intra-object behaviour is explicitly defined by classes and state-machines. In particular, we leave out semantically unclear sub-languages of UML like Use-case diagrams or Deployment diagrams.

A UML model in this sense basically comprises a set of basic, finite domain, non-object types, a finite set of signals (or events), and a finite set of classes with an inheritance relation on them and each of them comprising object-reference typed associations and a state-machine. The action language of state-machines is restricted to (local) attribute manipulation and event sending, guards are sensitive to event reception. Some of the classes may be designated to be actors, i.e. to represent the environment.

The representation of classes with basic type attributes and state-machine as a DCS is obvious: State-machines are unfolded into non-hierarchical state-machines according to their semantics and the current configuration of attributes becomes part of the DCS protocol states, i.e. for each combination of attributes and state-machine states there is a state in the DCS protocol for the class. The extension of DCS protocols from a single kind to finitely many ones is straightforward. The associations directly become channels and events are represented as messages in the DCS protocol.

Class inheritance can be established by splitting an object of a subclass into one object of the superclass and one object that provides the attributes added by the subclass, both linked by an additional, designated *uplink* channel in the subclass [20].

A fundamental concept of UML state-machines is the run-to-completion step, i.e. once triggered, an object takes transitions in its state-machine as long as it is possible to take a transition without dispatching an event. As the UML standard doesn't require a run-to-completion step to be atomic, it is faithfully represented by the full interleaving of DCS protocols.

The UML standard [15] doesn't enforce a particular queue model, but considers event transport and management to be an implementation detail. However, some UML semantics, e.g. [8], use a single event queue per object, i.e. keep the order of events sent by different other objects. This

is only representable in DCS with a modified DCS semantics as it currently uses an event queue *per sender* in each object following the approach of [3] (cf. Section 5).

## 5. Related Work

**Modelling DCS.** There are many models useful for describing dynamic topologies, for instance process calculi like  $\pi$ -calculus [19], ambient calculus [4] or graph rewriting [17]. Why should one choose DCS protocols?

Certainly, it is possible to encode DCS protocols in much lower level process calculi. However, we chose a more practical approach, where we want to see high-level features like roles in protocols (represented by process states) or asynchronous communication via queues in their *explicit natural* instead of their *coded* form as necessary for all lower level specification mechanisms. In practice, communication is asynchronous and needs some sort of message queues. We stress the practical relevance of DCS protocols by encoding UML [10] into DCS protocols in Section 4.

Originally, our work was inspired by Communicating Finite State Machines as introduced by Brand and Zafiropolo in [3]. However, they deal with a completely static scenario with a fixed number of processes and a fixed communication topology. We extended Communicating Finite State Machines conservatively to incorporate the features necessary for modelling interesting systems.

Additionally, we provide a strong verification framework for DCS protocols: the specification language METT and the techniques in Section 3 combining techniques from model-checking (Section 3.2) and abstract interpretation [5] (Section 3.3). Also for these verification purposes, it is convenient to reason about high level features in their natural instead of their coded form.

**Specification Languages.** Section 3.1 presented METT as a specification language tailored for DCS protocols. There are many other specification languages for evolving graph structures to be found in the literature. Most prominent among them is ETL [25]. The main difference to ETL is the lack of communication primitives in ETL because it is tailored to verifying multi-threaded Java programs. METT, on the other hand, lacks general transitive closure. Live Sequence Charts [6] with dynamic binding [13] are ideally suited for specifying communication behaviour, but lack the ability to naturally reason about graph structures. Furthermore they are restricted in expressiveness to a proper sublanguage of CTL\* [21]. Universal LSCs have an equivalent representation in first-order LTL [21], thus are actually a sublanguage of METT.

**DCS Verification.** There is numerous work on analysis and verification of certain single aspects of DCS. In [22] topologies are defined for  $\pi$ -calculus expressions, and an abstract interpretation is employed to compute a superset

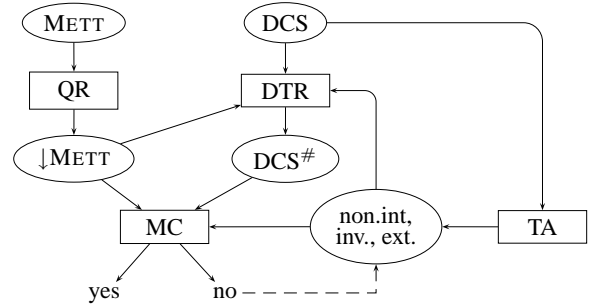


Figure 6. Tool Architecture.<sup>1</sup>

of the occurring topologies. This is somewhat similar to the topology analysis [1] as employed here, except that the latter uses more explicit graph transformation systems to model evolving topologies. [25] comes with a verification method based on Shape Analysis [18], but, as mentioned earlier, aims at a different application domain, where communication is not crucial. Closest to our work is the work in [9]. There, verification of properties specified as Live Sequence Charts is performed directly on UML models. Moreover, the verification algorithms themselves resemble those of Section 3.2 without the enhancement gained through the analysis of Section 3.3. Finally, DCS protocols provide a more concise formalisation compared to UML semantics, while at the same time being equally expressive (see Section 4).

**The PATH Project.** Our running example is motivated by the work of [11]. The authors try to deal with formal verification of their models using model-checking while suffering from severe drawbacks. The number of modelled objects is static and has to be predefined, hence there is no dynamic creation and deletion of objects. Moreover, as communication in the system is modelled via shared memory in contrast to message passing, there is no communication topology, not to mention a dynamically changing one. During their model-checking process only static scenarios with two interacting, statically selected objects are considered. All checked properties are very close to the implementation, for instance whether a certain flag is set at those points in time it is supposed to be set, whereas the considered properties should be motivated by the application scenario rather than its implementation.

## 6 Conclusion and Outlook

Dynamic communication systems are widely used but also highly complex and difficult for formal verification due to their inherent features such as a dynamically changing unbounded number of objects, a dynamically changing communication topology and unbounded message queues

<sup>1</sup>Ovals represent models information, boxes denote tools or procedures, solid lines denote automatic data-flow, dashed lines manual intervention.

for asynchronous communication. In this paper, we have presented a high-level specification language for DCS especially aiming at directly modelling these three aspects of DCS. Furthermore, we have shown how to specify properties over these models and how to verify them using model-checking and abstract interpretation based techniques. In order to demonstrate practical applicability, we have shown how to translate certain UML models into DCS.

For future work, we aim at an integration of existing techniques to handle unbounded queues [7, 2]. Furthermore, we want to automatise the verification of DCS within a unified tool chain as depicted in Figure 6, where the single tools already exist. Here, the topology analysis (TA) establishes non-interference lemmata (non.int) as well as general topology invariants (inv.) and gives hints about the extension (ext.) of the DCS model. These facts are used for a refined model-checking (MC) of the abstracted model (DCS<sup>#</sup>) w.r.t. the query-reduced (QR) requirement specifications ( $\downarrow$ METT).

**Acknowledgements.** The authors want to express their gratitude to the researchers of AVACS subproject S2, in particular A. Podelski, R. Wilhelm, and W. Damm, for fruitful discussions at an early stage of this work.

## References

- [1] J. Bauer and R. Wilhelm. Analysis of Dynamic Communicating Systems by Hierarchical Abstraction. Dagstuhl Seminar Proceedings 06081, 2006.
- [2] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs (extended abstract). In *Proc. SAS*, pages 172–186. Springer, 1997.
- [3] D. Brand and P. Zafiropulo. On communicating finite-state machines. *JACM*, 30(2):323–342, Apr. 1983.
- [4] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Proc. FoSSaCS*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.
- [5] P. Cousot and R. Cousot. Abstract interpretation. In *Proc. PoPL*, pages 238–252, New York, NY, 1977. ACM Press.
- [6] W. Damm and D. Harel. LSCs: Breathing life into MSCs. In *FMSD*, volume 19(1), pages 45–80, 2001.
- [7] W. Damm and B. Jonsson. Eliminating queues from RT UML model representations. In *Proc. FTRTFT*, pages 375–394. Springer, 2002.
- [8] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. A discrete-time UML semantics for concurrency and communication in safety-critical applications. *SCP*, 55(1–3):81–115, Mar. 2005.
- [9] W. Damm and B. Westphal. Live and let die: LSC-based verif. of UML-models. *SCP*, 55(1–3):117–159, Mar. 2005.
- [10] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Application with UML*. Addison-Wesley, 2000.
- [11] A. Hsu, F. Eskafi, S. Sachs, and P. Varaiya. The Design of Platoon Maneuver Protocols for IVHS. Technical report, UCB-ITS-PRR-91-06, 1991.
- [12] C. N. Ip and D. L. Dill. Better verification through symmetry. *FMSD*, 9(1/2):41–75, 1996.
- [13] J. Klose and B. Westphal. Relating LSC specifications to UML models. In H. Ehrig and M. Grosse-Rhode, editors, *Proc. INT*, pages 130–137, Apr. 2002.
- [14] K. L. McMillan. A methodology for hardware verif. using compositional model checking. *SCP*, 37:279–309, 2000.
- [15] OMG. OMG unified modeling language specification (1.4-UML-01-09-67), Sept. 2001.
- [16] PATH. California partners for advanced transport and highway, 1986-2003.
- [17] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
- [18] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Tr. Prog. Lang. and Sys.*, 24(3):217–298, 2002.
- [19] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge U. Press, 2001.
- [20] I. Schinz, T. Toben, C. Mrugalla, and B. Westphal. The Rhapsody UML Verification Environment. In J. R. Cuelar and Z. Liu, editors, *Proc. SEFM*, pages 174–183. IEEE, Sept. 2004.
- [21] T. Toben and B. Westphal. On the expressive power of LSCs. In *Proc. SofSem*, volume 2. Matfyz Press, 2006.
- [22] A. Venet. Automatic determination of communication topologies in mobile systems. In G. Levi, editor, *Proc. SAS*, volume 1503 of *LNCS*, pages 152–167. Springer, 1998.
- [23] B. Westphal. LSC verification for UML models with unbounded creation and destruction. In B. Cook et al., editors, *Proc. SoftMC*, ENTCS. Elsevier, July 2005.
- [24] F. Xie. *Integration of Model Checking into Software Development Processes*. PhD thesis, U. Texas, Austin, Aug. 2004.
- [25] E. Yahav, T. Reps, S. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In P. Degano, editor, *Proc. ESOP*, number 2618 in *LNCS*, pages 204–222. Springer, 2003.