Live and Let Die: LSC-based Verification of UML-Models

Werner Damm and Bernd Westphal

Carl von Ossietzky Universität Oldenburg,
Department für Informatik, PO Box 2503, 26111 Oldenburg, Germany
{damm,westphal}@informatik.uni-oldenburg.de

Abstract. We present a strategy for automatic formal verification of Live Sequence Chart (LSC) specifications against UML models in the semantics of [7] employing the symmetry-based technique of Query Reduction [18, 34, 44] and the abstraction technique Data-type Reduction [34]. Altogether this allows for automatic formal verification without providing finite bounds on the numbers of objects created during a run of the system.

Our presentation is grounded on a specific formal interpretation of LSCs for the UML domain in terms of [7] which is rich enough to in particular express properties about objects which are created only during activation of the LSC.

1 Introduction

The increasing use of UML or specialised sublanguages thereof in the domain of safety-critical systems design raises a need for formal verification techniques.

A necessary pre-requisite is on the one hand a formal semantics of a UML sublanguage sufficiently rich for behavioural description as provided by [7] and on the other hand a formal foundation of one of UML's specification languages for inter-object communication like Sequence or Collaboration Diagrams.

Previous efforts towards automatic formal verification of a significant sublanguage of UML concentrate on different subsets of the state-machine language and consider only a single object or an explicitly given finite set of objects, i.e. do not address dynamic creation or destruction of objects during runtime [9, 27, 4, 40, 41, 39, 28] or don't elaborate on this topic [12].

Recent achievements [43] implement object creation and destruction explicitly in the input language of the employed formal verification tool based on "switching on and off" objects like in [7] or translate the UML model into an intermediate language [37] which provides constructs for this kind of dynamics

s.t. the problem of choosing a finite representation is shifted to the translation-step from the intermediate language to the employed formal verification tool. Both approaches presuppose a finite bound on the number of objects alive in each snapshot of a run as long as the target is a finite-state formal verification tool.[1] In the following we present a technique which allows to overcome these limitations under certain premises even for finite-state methods.

The specification languages used in previous approaches range from temporal logic expressions over variable names on the level of the underlying model-checker's input language [41, 39, 28] to temporal logic (resp. patterns) on the UML model level [43, 12]. The tools presented in [27, 40] provide automatic solving of "drive to collaboration" tasks, i.e. to verify for UML with real-time information whether a given system is able to show the behaviour described by collaboration diagrams.

We propose to adopt the language of Live Sequence Diagrams [5] (LSC) for UML as the specification language for *inter-object communication*. The LSC language is a superset of UML's Sequence Diagram language – thus appealing for the UML designer – and explicitly designed to overcome the limitations in expressiveness of Sequence Diagrams as discussed in [5].

Our definition of LSCs for UML is in particular designed to express properties over instances which are created during a run of the system, and even during the activation of the LSC, by interpreting instance lines as universally or existentially quantified logical variables, thus a system satisfies an LSC if all runs satisfy all instantiations of the quantification resp. if there exists a run which satisfies an instantiation. Hence the whole specification can be discharged by carrying out numerous separate concrete verification tasks, each a binding of concrete object instances to instance lines [25, 42].

The observable communication comprises events and so called triggered operations, i.e. operations whose behaviour is defined by a state-machine, and is integrated into the fully abstract LSC semantics of [24, 26].

The authors of [30] present an alternative approach to explain binding of instances to instance lines with the same underlying intuition, but in addition allow to quantify single instance lines. The description is tailored for the application in their play-in/play-out tool [14], that is, for observing or "playing-out" a complete system. Our presentation is in contrast chosen to be able to apply the theory of symmetry reduction and data-type reduction to consider only a reduced system for automatic formal verification.

The most closely related approach for temporal logic patterns in the domain of object-oriented systems is the (textual) Bandera Specification Language [3] of the Bandera Java verification toolset, which allows to express similar quantifications.

---

[1] Note that these obstacles are raised by aiming at automatic formal verification, they naturally do not apply in general to proposals for formal semantics of UML, which in contrast use methods which intentionally do not require finite bounds and don't explicitly represent "not existing" objects, e.g. [15, 23, 36, 35].

The numerous verification tasks obtained by binding objects to instance lines in the system are a prominent application of a technique for which the authors of [44] coined the term "Query Reduction", since not the state-space of the verified transition system is reduced, but the number of concrete bindings to be proven. A small representative set implies the (possibly infinite) whole set of properties by symmetry.

This way to exploit symmetry was first demonstrated by the author of [31, 34] for general temporal-logic properties of the form of a universal quantification over a *symmetric type*. Then it is sufficient to prove only a representative set of concrete bindings since all other bindings are implied by symmetry.

This technique applies to systems where the state of replicated components is kept in an array data-structure indexed by a symmetric type and to properties which claim that for all indices i a property $\phi_0(i)$ holds, thus in particular to our interpretation of LSCs.

The idea to exploit in formal verification the symmetries of a system caused by replicated components, like processors in a cache-coherency protocol, actually dates back to 1993, when the authors of [10] and [18] independently discovered that symmetries of transition systems can be exploited to prove certain properties on the quotient graph by the equivalence relation induced by symmetry instead of on the full transition system. The authors of [18] even provided a set of criteria which allows to declare and syntactically check symmetric data types in the system description language of the MurΦ model-checker [19].

A disadvantage of the quotient graph approach is that the set of properties is restricted to safety properties independent from individual identities like "none of the symmetric components of the system runs into a deadlock" [18] or to properties which are itself symmetric, e.g. identical under permutation of indices [10].

The dSPIN [16] variant of the SPIN model-checker is an application of this kind of symmetry reduction in software-verification exploiting heap symmetries — system states which differ only in the allocation of objects into memory places on the heap are equivalent on the program level in languages like Java — and analogously process (allocation) symmetries [17]. The published results yet comprise only checking for absence of deadlock.

As mentioned before, the direct effect of query reduction is just *not* a reduction of the transition graph, although indirectly a reduction may be obtained by standard techniques like cone-of-influence reduction which apply more effectively on the concretely bound properties and may render tasks feasible, which are far too complex in the original form.

Yet cone-of-influence reduction alone does not address the problem that the state-space of a UML model is in general infinite if there are no finite bounds on the number of objects. Therefore we propose to apply the over-abstraction technique Data-type Reduction [34] which by heuristics abstracts away all objects which are not explicitly referenced in the concrete binding of object instances to LSC instance lines from the view of the bound objects s.t. these remaining objects can in particular not determine the actual number of objects in the system.

Thus there is no requirement for a finite bound on the number of simultaneously alive objects.

This technique also supports reasoning about parameterised systems: if a property can be proven for an abstraction constructed for an arbitrary concrete choice of parameters, than for any choice of *larger* parameters the abstraction also satisfies the property since it is also an abstraction for the larger parameters.

Note that the length of the event queue is a priori still unbounded in a UML design, thus for the scope of this paper we assume a finite upper bound on the length of event queues to reach the domain of automatic techniques for finite state verification. For the category of so called mode separated models, [6] presents an exact abstraction which eliminates queues from the model and yields a finite representation.

The remainder of the paper is organised as follows. In section 2 we introduce signatures, expressions, and interpretations to be able to define symbolic transition systems (STS) [29], our computational model, and to define linear temporal logic (LTL) over expressions which provides the ground for the presentation of the general semantics of LSCs and the specialisation of the LSC language for the context of UML in terms of [7] in Section 3. In section 4, we provide the theory of query reduction, contribute the yet missing proofs, show how it applies to LSCs in the context of UML, and briefly introduce a running example for the subsequent sections. Section 5 presents the theory of data-type-reduction together with yet missing proofs and discusses the common class of "interference" false-negatives caused by the data-type-reduction abstraction. Section 6 discusses briefly how interference could be avoided by separately proving and then assuming non-interference lemmata derived from information in the UML-model, based on the methodology of [34], and section 7 concludes.

2   Preliminaries

As our computational model, we take symbolic transition systems [29], which allow a purely syntactical description of a transition system by first-order-logic expressions over a signature.

Section 2.2 defines a symbolic transition systems as two first-order-logic predicates over a signature, so we first introduce signatures, predicate- and first-order-logic expressions and their interpretation in section 2.1.

Section 2.3 defines linear temporal logic and the satisfaction of LTL formulae by a symbolic transition system to be able to explain the semantics of live sequence charts in the following section and to provide the formal foundation of the proofs in Sections 4 and 5.

All definitions are standard, hence the reader may safely skip this section on the first reading.

2.1 First-order-logic Expressions

Definition 1 (Predicate-logic expressions). *Let V be a set of typed variables and Ω a set of typed constants. The pair $B = (V, \Omega)$ is called* signature. *The set $Expr(B)$ of typed expressions over B is defined inductively as follows:*

- *Let $v \in V$ be variable of type τ. Then v is an expression over B of type τ, $type(v) =_{df} \tau$.*
- *Let $f \in \Omega$ be a constant of type $\tau_1 \times \cdots \times \tau_n \to \tau$, $n \in \mathbb{N}_0$. Let $expr_i \in Expr(B)$, $type(expr_i) = \tau_i$, for $0 < i \le n$. Then $expr = f(expr_1, \ldots, expr_n)$ is an expression over B of* arity n, $type(expr) =_{df} \tau$.
  A constant with range type $\tau = \mathbb{B}$ is called predicate.

We use T_B to denote the set of types used by B. The elements of the set $Expr_{PL}(B) =_{df} \{expr \in Expr(B) \mid type(expr) = \mathbb{B}\}$ are called predicate-logic *expressions over B.* ∎

In the following, we assume $\Omega \supseteq \{true, \vee, \neg, false, \wedge, \dot\vee, \implies, \iff\}$ for each signature, where the symbols "*false*", "$\wedge$", "$\dot\vee$" (exclusive or), "$\implies$", and "$\iff$" are used as abbreviations with the conventional definition for brevity.

Definition 2 (First-order-logic expressions). *Let $B = (V, \Omega)$ be a signature. The set $Expr_{FO}(B)$ of* first-order-logic (FOL) *expressions over B is defined inductively as follows:*

- *Let $expr \in Expr_{PL}(B)$ be a predicate-logic formula. Then 'expr' is a first-order-logic expression.*
- *Let $expr \in Expr_{PL}(B)$ be a first-order-logic formula and τ a type. Then $\exists\, x \in \tau : expr$ is a first-order-logic expression of type $\mathbb{B}$. Every occurrence of the variable $x \in V$ in 'expr' is called* bound.

Let $expr \in Expr_{FO}(B)$. A variable $x \in V$ occurring in 'expr' is called free *in 'expr' if not all occurrences are bound.*
We call a first-order-logic expression over $B = (V \cup V', \Omega)$, that is, an expression referring to both unprimed and primed versions of the variables in V, a first-order logic transition predicate *over B.* ∎

In the following, we use the conventional definition of the symbol "$\forall$" in first-order-logic expressions for brevity.

Definition 3 (Interpretation). *Let $B = (V, \Omega)$ be a signature, $\mathcal{D} = \bigcup_{\tau \in T_B} \mathcal{D}_\tau$ a domain for all types used in B, and $\mathcal{I}$ an interpretation of the constants which assigns to each constant $f \in \Omega$ of type τ a value $\mathcal{I}(f) \in \mathcal{D}_\tau$. Then the tuple $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ is called a* structure *of B.*
A function $s : V \to \mathcal{D}$ is called (type-consistent) valuation *of V if it assigns each variable $v \in V$ a value $s(v) \in \mathcal{D}_\tau$. The set of valuations is called Σ. We use $\mathcal{M}[\![expr]\!](s)$ to denote the canonical interpretation of the first-order-logic expression 'expr' in the valuation s.* ∎

The interpretation $\mathcal{M}[\![expr]\!](s, s')$ of a transition predicate is defined analogously letting s provide the interpretation of unprimed and s' the interpretation of primed variables in *expr*.

In the following, we consider only interpretations $\mathcal{M}$ which gives the canonical interpretation to the constants "*true*", "$\vee$", and "$\neg$".

2.2 Symbolic Transition Systems

Definition 4 (STS). *A symbolic transition system (STS) $S = (B, \Theta, \rho)$ consists of $B = (V, \Omega)$, a signature with a finite set V of variables, $\Theta \in Expr_{FO}(B)$, and ρ, a FOL transition predicate over B. The set of variables $v \in V$ which are free in Θ or ρ are called* system variables *of S.* ∎

An STS *induces* a transition system on the set of valuations of its system variables as follows:

Definition 5 (Runs of an STS). *Let $S = (B, \Theta, \rho)$ be an STS and $\mathcal{M}$ a structure of B.*

(i) A valuation of the system variables of S is called snapshot *of S.*

(ii) A snapshot $s \in \Sigma$ of S is called initial, *iff $\mathcal{M}[\![\Theta]\!](s) = true$.*

(iii) Let $s, s' \in \Sigma$ be snapshots of S. Snapshot s' is called S-successor of s, iff $\mathcal{M}[\![\rho]\!](s, s') = true$.

(iv) A computation or run of S is an infinite sequence of snapshots, $r = s_0\, s_1\, s_2 \ldots$, satisfying the following requirements:
- *Initiation: s_0 is initial.*
- *Consecution: Snapshot s_{j+1} is an S-successor of s_j, for each $j \in \mathbb{N}_0$.*

(v) The set of all computations of S is called runs(S). We use $r(i)$ to denote the i-th snapshot of a run $r \in runs(S)$ and

$$r/i =_{df} r(i)\, r(i+1)\, r(i+2) \ldots$$

to denote the infinite sequence starting at $r(i)$, $i \in \mathbb{N}_0$. ∎

2.3 Linear Time Logic

In section 3, we assume a formalisation of the temporal properties expressed within an LSC in the well-known temporal logic LTL (linear time logic):

Definition 6 (LTL). *Let B be a signature. An LTL formula over B is defined inductively as follows:*

(i) $expr \in Expr_{PL}(B)$ is an LTL formula.

(ii) $\neg f$ and $f \vee g$ are LTL formulae if f and g are LTL formulae, and

(iii) $\mathbf{X}\, f$ ("next f"), $\mathbf{G}\, f$ ("globally f"), and $f\, \mathbf{U}\, g$ ("f until g") are LTL formulae if f and g are LTL formulae. ∎

In the following we define what it means for a run to satisfy an LTL formula and in addition introduce the orthogonal notions of existential vs. universal and initial vs. invariant satisfaction of an LTL formula as the foundation for the semantics of life sequence charts.

Definition 7 (Satisfaction of an LTL formula). *Let $S = (B, \Theta, \rho)$ be an STS, ϕ an LTL formula over B, and $\mathcal{M}$ a structure of B.*

Let $r = s_0\, s_1\, s_2 \ldots$ be a (suffix of a) run of S. We say the run r satisfies ϕ wrt. $\mathcal{M}$, denoted by $r \models_{\mathcal{M}} \phi$, iff:

(i) $\phi \equiv expr$ *and* $\mathcal{M}[\![expr]\!](r(0)) = true$, *or*
(ii) $\phi \equiv f \vee g$ *and* $r \models_{\mathcal{M}} f$ *or* $r \models_{\mathcal{M}} g$, *or*
(iii) $\phi \equiv \neg f$ *and* $r \not\models_{\mathcal{M}} f$, *or*
(iv) $\phi \equiv \mathsf{X}\, f$ *and* $r/1 \models_{\mathcal{M}} f$, *or*
(v) $\phi \equiv \mathsf{G}\, f$ *and* $\forall\, i \in \mathbb{N}_0 : r/i \models_{\mathcal{M}} f$, *or*
(vi) $\phi \equiv f\, \mathsf{U}\, g$ *and* $\exists\, i \in \mathbb{N}_0 : r/i \models_{\mathcal{M}} g \wedge \forall\, 0 \leq j < i : r/j \models_{\mathcal{M}} f$.

The STS existentially satisfies ϕ invariantly, denoted by $S \models_{\mathcal{M}, \exists} \phi$, iff

$$\exists\, r \in runs(S)\ \exists\, i \in \mathbb{N}_0 : r/i \models_{\mathcal{M}} \phi$$

and initially, *denoted by $S \models_{\mathcal{M}, \exists, 0} \phi$, iff* $\exists\, r \in runs(S) : r/0 \models_{\mathcal{M}} \phi$

The STS universally satisfies ϕ invariantly, denoted by $S \models_{\mathcal{M}, \forall} \phi$, iff

$$\forall\, r \in runs(S)\ \forall\, i \in \mathbb{N}_0 : r/i \models_{\mathcal{M}} \phi$$

and initially, *denoted by $S \models_{\mathcal{M}, \forall, 0} \phi$, iff* $\forall\, r \in runs(S) : r/0 \models_{\mathcal{M}} \phi$ ∎

3   Live Sequence Charts

Live Sequence Charts (LSC) are an extension of Messsage Sequence Charts (MSC), introduced to overcome serious deficiencies of the MSC language wrt. formal verification, so we begin with a short overview of the MSC language and the MSC dialect of UML Sequence Diagrams. We recall the deficiencies of both formalisms, followed by a brief introduction of the subset of the LSC language which we propose to use as a specification language for formal verification of UML models.

3.1   From Message Sequence Charts and Sequence Diagrams to LSCs

The MSC language is a well-known visual formalism to describe behaviour of a system by visualising the *inter-entity* communication basically as arrows (representing asynchronous messages) between vertical instance lines (representing entities within the system). Intuitively, the semantics of MSCs is a (partial) ordering in time of the observations of messages which is derived from the relative positions of message arrows and their beginning or ending at instance lines.

The MSC language is standardised in different versions [20, 21, 22] which extend the core language by means to structure and compose MSCs, to express loops and branches, by different annotations for timers and timing-constraints, by means to explicitly state ordering informations, and by different kinds of messages, e.g. synchronous messages to express method calls and replies.

Although the MSC language was originally formalized in the telecommunication domain to match this domain's system specification language, it is not inherently bound to a particular domain, design-language, or paradigm, but the kind of entities represented by an instance line can be chosen when giving semantics for a particular domain. Typical kinds of entities are processes in the context of process-oriented languages and objects in the object-oriented domain.

The Sequence Diagram language [38] of UML is an adoption of MSCs for UML where instance lines are in fact restricted to represent objects and where concrete message types are provided to represent event based resp. method call communication.

The main deficiencies of MSCs and SDs wrt. their use in formal verification are that they are meant to show only a sample run of the system – one scenario – where one would rather like to express that the system *always* behaves as depicted in the MSC, and that MSCs do not allow to express *liveness* properties, i.e. to distinguish whether progress is enforced or not.

Furthermore, the MSC versions except for MSC-2000 do not allow to specify an activation time thus it is left open *when* a system has to show the behaviour described by the MSC in order to fulfil it. The intention of an MSC describing the behaviour in case of erroneous input, for example, is typically meant to be observed only after a particular error-condition holds. No MSC version allows

to express this *activation* in terms of a sequence of messages, for example to express that error handling takes place after a sequence of a particular number of error-events have been observed.

Other major drawbacks are the facts that conditions annotated to locations on instance lines (which are not even present in SDs) are merely comments up to MSC-2000, i.e. it is not possible to e.g. specify that the system should be in a particular state when sending an event, and that simultaneity of items like messages and conditions cannot be expressed. Only MSC-2000 provides simultaneity but restricted to pairs of messages and timers.

Aside these concerns of expressiveness, MSC-2000 and SD are not directly usable for formal verification since they are not provided with an official formal semantics. For a complete discussion of the sequence charts dialects and their shortcomings see [24].

Note that although LSCs also provide a more sophisticated semantical treatment of timers and time-annotations in comparison to MSCs or SDs, we don't consider timers and time-annotations at all in the following since the UML semantics of [7] which our presentation is based on, is an un-timed semantics.

LSC were introduced in [5] to overcome the deficiencies of MSCs and SDs named above employing the basic idea to distinguish *mandatory* and *possible* behaviour per LSC element and for the whole LSC.

Intuitively, a *possible* or *existentially quantified* LSC is meant as a scenario, just like MSCs, i.e. it expresses that there is a run of the system which complies to the LSC, while a *mandatory* or *universally quantified* LSC requires that, whenever the LSC is activated, the system shows the behaviour depicted in the LSC.

The activation point of an LSC can be specified by giving a boolean *activation condition* and a so called *pre-chart* which is itself a restricted LSC. The LSC is then activated whenever the activation condition holds *and then* the (possibly empty) behaviour depicted in the pre-chart is observed. Additionally, the activation of an LSC depends on the *activation mode* of "initial", "initial first", "invariant", or "iterative". In the following we only consider the activation modes which directly correspond to our Definition 7: "initial", i.e. the LSC is activated at most once per run and only if its pre-chart is observed from the initial step of a run on, and "invariant", i.e. the LSC may be activated multiple times during a run, there may even be overlapping activations.

Within the LSC, each location, i.e. each place of an element on an instance line, e.g. a message start or end, is equipped with a temperature. A *mandatory* or *hot* location enforces progress, that is, eventually the next location has to be reached. A *possible* or *cold* location allows to stay at the location forever, that is, the behaviour following a cold location need not be observed.

A *possible* or *cold* condition is a legal exit point of an LSC, i.e. if a run of the system adheres to the prefix of an LSC up to a cold condition and the condition does not hold, then the run is said to satisfy the LSC, since the LSC "exits" and is no longer activated. Reaching a location with a hot condition which does not hold is considered to be a violation of the specification. As an extension

of conditions, LSCs also provide (possible or mandatory) local invariants, i.e. conditions which are not bound to a single location but to a start and end location.

The concrete graphical representation of LSCs generally follows MSCs, but in the following we use a concrete syntax more similar to UML sequence diagrams. The mandatory elements are, as usual for LSCs, depicted by solid lines and possible elements by dashed lines (cf. Fig. 1). For a complete presentation of the LSC features, see [24].

In [5], the LSC language is introduced as a conservative extension of MSCs, thus LSCs are as domain, design-language, and paradigm independent as MSCs. In particular, [24, 26] give the formal semantics of LSCs independent from the *mapping*, abstracting from what "sending a message" actually means in a concrete system from a particular domain, only the ordering and temporal constraints expressed in the LSC are considered.

Thus for an application of the LSC language in the UML domain, we have to provide the concrete syntax for e.g. message and condition annotations and the derivation of a mapping, that is a characterisation of the points in time when we want to consider a message to be sent and received, resp., and we have to explain a *binding* of instance lines to entities in the system.

The topic of binding of instance lines goes beyond the presentation of a specialisation of LSCs for the domain of Statemate-designs as presented in [24] where the author requires an explicit static binding of instance lines to Statemate-activities, which is possible since Statemate designs have a static structure, i.e. there is no dynamic creation or destruction of "system entities" as there is in the UML domain.

The rest of this chapter is structured as follows. In section 3.2 we provide a definition of general (yet domain-independent) LSCs which abstracts from syntactical aspects and from the elements which don't need a mapping, e.g. simultaneous regions for simultaneity, and we briefly report their abstract formal semantics as given by [24, 26]. That is, we do not elaborate on the temporal properties induced by the *relative position* or *partial ordering* of the parts of an LSC but take for granted that [24, 26] (indirectly) provide us with an LTL formula which expresses just these temporal properties.

In section 3.3 we define LSCs for UML models (in the sense of [7]) by giving constraints on the annotations of LSC elements s.t. we can construct a so-called observer extension for a UML model. The satisfaction of an LSC by the UML model is then defined in terms of the model's observer extension, binding objects to instance lines, thus taking objects as the kind of entities to be bound.

3.2   Live Sequence Charts

In general and independent from the design-language domain, the intuition of an instance line within an LSC is the denotation of an entity of the system the LSC refers to, where it of course depends on the domain what is considered an entity.

If there are multiple instances of the same type of entity, then we take an LSC as an abbreviation for all possible *bindings* of concrete system entity instances to instance lines, thus instance lines can be seen as *free* or *logical* variables of the specification which are quantified over the entity type.

In the following, we technically formalise this intuition by relating instance-lines to 0-ary *constants* from the given signature. A concrete binding is then given by the structure which interprets the LSC's signature.

In addition to these constants for instance-lines, we allow to refer to a general set of constants called *specification variables* in the LSC which are also intended to be bound to concrete values.

Definition 8 (LSC).
Let $B = (V, \Omega)$ be a signature and Msg a set of message names. A live sequence chart $L = (\ell, ac, pch, m, X, actmode, quant)$ *over B and Msg consists of the following components:*

- *ℓ : The finite body of the LSC, comprising the following* body elements*: instance lines, synchronous and asynchronous message sending and reception, conditions, and local invariants.*
- *ac: The* activation condition*.*
- *pch: The possibly empty body of the* pre-chart*.*
- *m: The* annotation *of body elements as defined below.*
- *$X = \{x_1, \ldots, x_n\} \subseteq \Omega$: A finite set of 0-ary* logical variables*.*
- *actmode: The* activation mode *from $\{initial, invariant\}$.*
- *quant: The* (chart-)quantification *from $\{existential, universal\}$.*

The bodies ℓ and pch of L together define the sets $inst(L)$ of instance lines, $send(L)$ and $recv(L)$ of synchronous and asynchronous message sendings resp. receptions, and $cond(L)$ of conditions and local invariants including the activation condition. Message sendings and receptions are required to be pairwise related, i.e. there exists a bijection between $send(L)$ and $recv(L)$, and to be uniquely related to an instance line.

The annotation m is a partial function which maps instance lines, messages, and conditions of L to an expression obeying the following restrictions:

(i) *If $p \in inst(L)$, then $m(p) = x : \tau$ where $x \in X$ is a 0-ary constant of type τ.*

(ii) *If $p \in send(L) \cup recv(L)$, then $m(p) = msg(expr_1, \ldots, expr_n)$, $n \in \mathbb{N}_0$, where $msg \in Msg$ and $expr_i \in Expr(B)$.*

(iii) *If $p \in cond(L)$, then $m(p) = expr \in Expr(B)$ of boolean type or $m(p) = \neg dest.msg(expr_1, \ldots, expr_n)$, $n \in \mathbb{N}_0$, where $dest \in X$, $msg \in Msg$, and $expr_i \in Expr(B)$.*
    The latter case is used to assume the absence of messages in a local invariant.

(iv) *If $p = ac$, then $m(p) = expr$ or $m(p) = dest.msg(expr_1, \ldots, expr_n) \wedge expr$, $n \in \mathbb{N}_0$, where $dest \in X$, $msg \in Msg$, and $expr_i \in Expr(B)$, and $expr \in Expr(B)$ is of boolean type.*
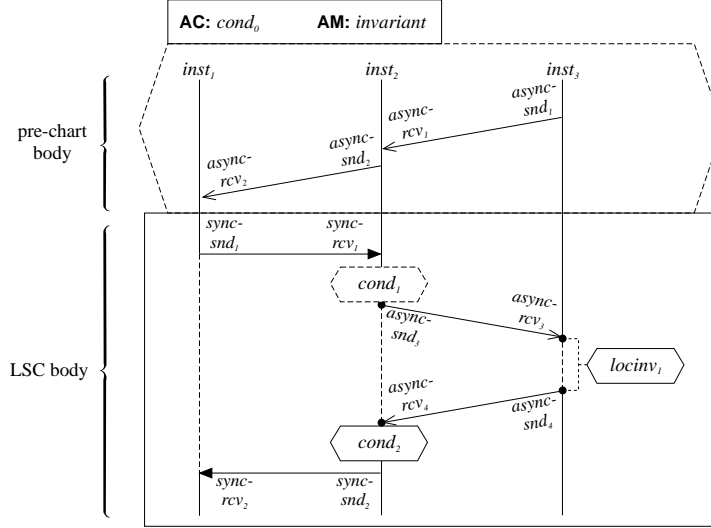   *The latter case is used to activate on messages.*   ∎

Fig. 1. LSC: A graphical representation of the LSC L with body $\ell = \{inst_{1,2,3},$ $async\_snd_{3,4}, async\_rcv_{3,4}, sync\_snd_{1,2}, sync\_rcv_{1,2}, cond_{1,2}, locinv_1\}$, activation condition $ac = cond_0$, pre-chart $pch = \{inst_{1,2,3}, async\_snd_{1,2}, async\_rcv_{1,2}\}$, $actmode = invariant$, and quantification $quant = universal$ as indicated by the solid line around the body of the LSC. The LSC is yet *unmapped*, i.e. the annotation m is empty.

Independent from the mapping, the LSC L is satisfied by all runs in which, any time after the two asynchronous messages in the pre-chart have been observed, a synchronous communication takes place between $inst_1$ and $inst_2$, and eventually – since the location between $sync\_rcv_1$ and $async\_snd_3$ is hot – an asynchronous communication takes place between $inst_2$ and $inst_3$ with the restriction that at the *same* point in time, when $async\_snd_3$ is observed, $cond_1$ is supposed to hold. $cond_1$ is a cold condition as indicated by the dashed border, thus if $cond_1$ does not hold, then the LSC is "exited successfully", i.e. the run satisfies the LSC.

Below $async\_rcv_3$, there is a *cold cut*, i.e. the current position on each instance line lies on a cold location hence the following communication need not take place as long as the local invariant $locinv_1$ holds. $locinv_1$ is mandatory (as indicated by the solid line), thus if the condition $locinv_1$ is violated after $async\_rcv_3$ but before $async\_snd_4$, the whole LSC is not satisfied.

The condition $cond_2$ is a mandatory condition, i.e. if $async\_rcv_4$ is observed and $cond_2$ does not hold at the same point in time, then the run does not satisfy L.

Since the subsequent locations are hot, both $sync\_snd_2$ and $sync\_rcv_2$ have to be observed in order to exit the LSC successfully.

Note that L may be activated multiple times in a run and even overlapping. The run satisfies the LSC only if it is not violated in any activation.

The semantics of an LSC over signature $B = (V, \Omega)$ and message set *Msg* is explained *symbolically* by [24, 26] in terms of a Timed Büchi Automaton (TBA). Using the annotation m, the TBA can be translated into an LTL formula $\Phi(L)$ over B using the constant function symbols *send* and *receive* which act as placeholders for the domain-dependent definition of message sending and reception.

By definition, the TBA and hence the formula $\Phi(L)$ depend on the chart-quantification of L: for an existential L it is principally the *sequential composition* of pre- and main-chart, while for an universal L it states that an observation of the prechart *implies* the main-chart.

For example consider the message arrow $(async\_snd_3, async\_rcv_3)$ in the LSC body in Fig. 1 between instance lines $inst_2$ and $inst_3$. An annotation

$$m(async\_snd_3) = m(async\_rcv_3) = msg(expr_1, \ldots, expr_n)$$

results in both, $send(msg, m(inst_2), m(inst_3), expr_1, \ldots, expr_n)$ and $receive(msg, m(inst_2), m(inst_3), expr_1, \ldots, expr_n)$, occurring in $\Phi(L)$, the former observing the sending and the latter observing the reception of *msg*.

Note that synchronous and asynchronous messages are not distinguished on this level of *predicates* but the distinction is incorporated into the LTL formula: for synchronous messages, sending and reception is observed *in the same* snapshot whereas for asynchronous messages, reception has to be observed *at least* one snapshot later than sending.

When explaining LSCs for a particular application domain, it is often a matter of choice which of the domain's "observable events" are better mapped to synchronous and which to asynchronous messages of the LSC.

Definition 9 (Satisfaction of an LSC).

Let $L = (\ell, ac, pch, m, X, actmode, quant)$ be an LSC over signature $B = (V_B, \Omega_B)$ and messages Msg. Let $\Phi'(L)$ be the LTL formula representation of L over B with all occurrences of 'send' and 'receive' replaced by boolean predicate-logic expressions over B. Let $S = ((V, \Omega), \Theta, \rho)$ be an STS with $V = V_B$ and $\Omega \supseteq \Omega_B \setminus X$, and $\mathcal{M}$ a structure of B.
Then the model S satisfies the LSC wrt. $\mathcal{M}$, $S \models_{\mathcal{M}} L$, iff

- *quant = existential and*
  - *actmode = initial and*

    $$\exists\, x_{0_1} : \mathcal{D}_{type(x_1)}, \ldots, x_{0_n} : \mathcal{D}_{type(x_n)} : S \models_{\mathcal{M}', \exists, 0} ac \wedge \mathbf{X}\, \Phi'(L), \text{ or}$$

  - *actmode = invariant and*

    $$\exists\, x_{0_1} : \mathcal{D}_{type(x_1)}, \ldots, x_{0_n} : \mathcal{D}_{type(x_n)} : S \models_{\mathcal{M}', \exists} ac \wedge \mathbf{X}\, \Phi'(L), \text{ or}$$

- *quant = universal and*
  - *actmode = initial and*

    $$\forall\, x_{0_1} : \mathcal{D}_{type(x_1)}, \ldots, x_{0_n} : \mathcal{D}_{type(x_n)} : S \models_{\mathcal{M}', \forall, 0} ac \implies \mathbf{X}\, \Phi'(L), \text{ or}$$

  - *actmode = invariant and*

    $$\forall\, x_{0_1} : \mathcal{D}_{type(x_1)}, \ldots, x_{0_n} : \mathcal{D}_{type(x_n)} : S \models_{\mathcal{M}', \forall} ac \implies \mathbf{X}\, \Phi'(L),$$

  where $\mathcal{M}' = (\mathcal{D}, \mathcal{I} \cup \{x_i \mapsto x_{0_i} \mid 1 \leq i \leq n\})$. ∎

3.3   LSCs for UML

In the following we elaborate on ideas already outlined in [25]. We refer to UML models in the definition of [7], i.e. a UML model is a tuple

$$M = (T, F, Sig, <, C, c_{root}, A),$$

with T a set of basic types, F a set of predefined primitive functions, e.g. arithmetic operations on T, Sig a finite set of signals, $< \subset Sig \times Sig$ a generalisation relation on signals, C a finite non-empty set of (further structured) classes, $c_{root} \in C$ the class of the root object, and $A \subset C$ the set of active classes (for the details the reader is referred to the companion paper [7]). In order to explain syntactical transformations on the transition predicate of $STS(()M)$ in Section 5, in the following we assume F to contain $=: \tau \times \tau \to \mathbb{B}$, the comparison for equality on all types, and $(\cdot ? \cdot : \cdot) : \mathbb{B} \times \tau^2 \to \tau$, the if-then-else function.

We denote by T_c the *type of references* to objects of class $c \in C$ and by T_C the set of all T_c. For each class $c \in C$, O_c denotes the semantic type or domain of T_c and O_C the union of all O_c.

An LSC over M is basically an LSC over a signature derived from M and the set of events and triggered operations in M as set of messages Msg together with a number of well-formedness rules:

Definition 10 (LSC over UML model).
Let $M = (T, F, Sig, <, C, c_{root}, A)$ be a UML model. An LSC over M is an LSC over $B = (\emptyset, F \cup X \cup \{.\})$, where each $x \in X$ is of a type from $T \cup T_C$ and "." is the binary navigation operator, *and the message set*

$$Msg = Sig \cup \{create_c \mid c \in C\} \cup \{destroy\} \cup \{reply_\tau \mid \tau \in T \cup T_C\} \cup \bigcup_{c \in C} c.ops$$

which obeys the following well-formedness rules:

(i) If $p \in inst(L)$, then $m(p) \in X$ is of a reference type T_c for $c \in C$. All $m(p) \in inst(L)$ are pairwise different.

(ii) If $p \in send(L)$ is an asynchronous message sending or reception and $m(p) = msg(expr_1, \ldots, expr_n)$, then $msg = ev \in Sig$ and either $n = 0$ or n matches the number of parameters of ev and $type(expr_i)$ matches the type of the i-th parameter of ev.[2]
An $ev \in Sig$ may not occur more than once in the whole LSC since the used underlying semantics does not provide identities of events.

(iii) If $p \in send(L)$ is a synchronous message sending or reception from instance line i_1 to i_2 and $m(p) = msg(expr_1, \ldots, expr_n)$, then $msg \in Msg \setminus Sig$. If p_1 and p_2 are the related synchronous message sending and reception, then $m(p_1) = m(p_2)$.

---

[2] providing only means to restrict either *all* or *none* of the parameters in the LSC is a matter of choice for brevity. The generalisation to restriction of only *some* parameters is straightforward in case practical evaluation reveals a demand.

If $msg = op$ and $m(i_2)$ is of type T_c, $c \in C$, then $op \in c.ops$ and either $n = 0$ or n matches the number of parameters of op and $type(expr_i)$ matches the type of the i-th parameter of msg.

If $msg = reply_\tau$, then $n \leq 1$ and there is a uniquely identified synchronous message sending p' from i_2 to i_1, i.e. in the opposite direction, with $msg(p') \in c.ops$ for a $c \in C$ and $\tau = type_r(msg(p'))$. That is, a reply has to be related to an operation call.

If $msg = create_c$, then $n = 0$ and p is the first message or condition of the destination instance line and p is the only message annotated by creation.

If $msg = destroy$, then $n = 0$ and p is the last message or condition of the destination instance line and p is the only message annotated by destruction.

(iv) For each creation $p \in send(L)$ to instance line i, there exists a cold condition $q \in cond(L)$ with $m(q) \equiv justcreated(i)$, yet another placeholder which will allow us to legally exit the LSC in each run, where the creation operation did not create the object bound to i.

(v) If $p \in cond(L)$ is not the activation condition and not a local invariant, then $m(p) \in Expr$ of boolean type.

(vi) If $p \in cond(L)$ is the activation condition or a local invariant and $m(p) = dest.msg(expr_1, \dots, expr_n)$ resp. $m(p) = \neg dest.msg(expr_1, \dots, expr_n)$, then $dest$ is of type $T_c \in T_C$ and $msg \in c.ops \cup Sig$ and either $n = 0$ or n matches the number of parameters of msg and $type(expr_i)$ matches the type of the i-th parameter of msg. ∎

Note that $c.ops$ comprises only *triggered operations* of class $c \in C$, i.e. operations whose behaviour is defined by c's state-machine. So called *primitive operations* which are defined by a method are no longer visible on the semantics level of [7].

As outlined in section 3.2, we obtain an LTL formula $\Phi(L)$ for an LSC over a UML model which uses for example for an asynchronous message sending from instance i_1 to i_2 the placeholder $send(ev, m(i_1), m(i_2), expr_1, \dots, expr_n)$.

To explain what it means for a UML model M to satisfy an LSC L, we use the STS semantics of M, $STS(M)$, according to [7]. The placeholders for the message send and receive are replaced by predicates over system variables including new system variables which are introduced to explicitly *observe* events and triggered operation based communications

We need to introduce new system variables, since predicates over the unchanged model can only refer to the valuation of a *single* snapshot while we want to observe e.g. sending of an event $E \in Sig$ from object o_1 to object o_2 in a snapshot $r(i+1)$ of a run $r \in runs(STS(M))$ only if the transition from $r(i)$ to $r(i+1)$ in $STS(M)$ corresponds to o_1 taking a transition which is annotated by an event sending action which enters an E into the event queue of o_2's active object.

To observe the intended relation between two subsequent snapshots, we construct an *observer extension* of $STS(M)$ by introducing five new system variables *justsend*, *justrecv* and *justcall*, *justret*, and *justcreated*. whose value has to

be defined by the transition relation s.t. for example *justsend* becomes valid in snapshot $r(i+1)$ and holds the type and parameter values of the event sent when taking the transition from $r(i)$ to $r(i + 1)$. The first component of the former variables is a boolean flag which indicates that the variable's value is valid. A single flag is sufficient due to the strictly interleaving and atomic nature of the underlying semantics [7].

All of the first four variables carry sender, destination, and all parameters since e.g. the return value of a triggered operation is actually no longer visible in $r(i + 1)$ in the pending-request-table. The variable *justcreated* is just an object reference which, if non-nil, contains the identity of the object created in the transition to the current state.

Definition 11 (Observer extension). *Let $M = (T, F, Sig, <, C, c_{root}, A)$ be a UML model, and $S = STS(M) = (B, \Theta, \rho)$ its semantics according to [7].*
The observer extension of S, $S_o = (B_o, \Theta_o, \rho_o)$, with $B_o = (V_o, \Omega_o)$ is obtained from S as follows:

(i) V is extended by variables to observe events

$$justsend, justrecv : \mathbb{B} \times Sig \times O_C \times O_C \times \bigcup_{ev \in Sig} \mathcal{T}_{type_{par}(ev)},$$

to observe triggered operation calls

$$justcall : \mathbb{B} \times (\bigcup_{c \in C} c.ops) \times O_C \times O_C \times \bigcup_{\substack{c \in C \\ op \in c.ops}} \mathcal{T}_{type_{par}(op)},$$

to observe completion of triggered operations

$$justret : \mathbb{B} \times (\bigcup_{c \in C} c.ops) \times O_C \times O_C \times \bigcup_{\substack{c \in C \\ op \in c.ops}} \mathcal{T}_{type_r(op)},$$

and to observer object creation justcreated : O_C.

(ii) Θ is changed s.t. the first four variables' first components get the value false initially.

(iii) $\rho_{non\_op\_action}$ which formalises taking a transition annotated with a non-operation call action is conjoined with the following predicate:

$$(\gamma \equiv \text{``}r.send(ev, expr_1, \ldots, expr_n)\text{''}$$
$$\wedge \neg sysfail' \implies justsend' := (true, o, o.r, ev, (expr_1, \ldots, expr_n)))$$

where γ denotes the considered transition and o the object taking the transition (cf. [7] for the full set of used abbreviations).
Effectively, "justsend" observes the enqueueing of an event of type ev with destination o.r when object o takes a transition. It holds a valid value in the first snapshot where ev shows up in the queue.

(iv) ρ_{get_event} and $\rho_{discard\_event}$ *which formalise dispatching resp. discarding of an event are conjoined with*

$$(\neg sysfail' \implies justrecv' := (true, \mathsf{nil}, o, head(o.my\_ac.eq).ev, o.ev'_p))$$

where nil *is used as the sender, since the sender is not retained with the event, and o is the destination object. head denotes the first entry in the event queue of o's active object and $o.ev'_p$ the attributes of o holding copies of the event parameters.*
"justrecv" observes the dequeuing of an event of type ev with destination o. It holds in the first snapshot where ev has disappeared from the queue.

(v) $\rho_{init\_opcall\_or\_create}$ *which formalises calling a triggered operation is conjoined with*

$$(\neg sysfail' \implies justcall' := (true, o, r, prt(o).op', prt(o).op'_p))$$

where o is the object initiating the call and r the destination.
"justcall" observes the operation call op when the caller changes status from executing to suspended and writes op with receiver r into its pending request table entry.
It holds in the first snapshot where o is suspended due to the call.

(vi) $\rho_{pick\_up\_result}$ *which formalises picking up the result of a triggered operation call by the caller is conjoined with*

$$(\neg sysfail' \implies justret' := (true, o, prt(o).dest, prt(o).op, prt(o).op'_p))$$

"justret" observes return from operation call op when the caller o changes status from suspended back to executing or idle. It holds in the first snapshot where o is no longer suspended due to the call.

(vii) $\rho_{non\_op\_action}$ *is also conjoined with the following first-order predicate:*

$$(\gamma \equiv \text{"destroy}(expr)\text{"} \wedge \neg sysfail' \implies justcall' := (true, o, expr))$$

Thus destruction is observed just like a triggered operation call.

(viii) ρ *is finally changed s.t.*

$$(\neg sysfail' \implies$$
$$([\forall\ o \in O_C, o \neq \mathsf{nil}:$$
$$(o.status = dormant \wedge o.status' = executing) \implies justcreated = o]$$
$$\vee\ justcreated = \mathsf{nil}))$$

and s.t. for each other observer variable the first component gets the value false if the observer variable is not "assigned" to in a step. ∎

Note that in the above definition we chose to consider triggered operation calls as synchronous and observe only the call and picking up the result, although

they are actually *asynchronous* since a call can soonest be accepted one step after the call. It is still to be assessed whether it is a better choice to consider triggered operation calls as asynchronous (in the sense of LSCs).

The following definition builds the predicates characterising message sending and reception from a system extended with observer variables and thereby defines the semantics of LSCs for UML.

Definition 12 (Satisfaction of an LSC for UML).
Let $M = (T, F, Sig, <, C, c_{root}, A)$ be a UML model, and $S_o = (B_o, \Theta_o, \rho_o)$ the observer extension of its semantics. Let L be an LSC over M, $\Phi(L)$ the LTL formula representation of L and $\mathcal{M}$ a structure of B_o.
The UML model M satisfies the LSC L wrt. $\mathcal{M}$, $M \models_{\mathcal{M}} L$, iff $STS(M) \models_{\mathcal{M}} L$ where $\Phi'(L)$ is obtained as follows:

(i) *For an event $ev \in Sig$, $o_1, o_2 \in O_C$, and expressions $expr_1, \ldots, expr_N$, $N = 0$ or $N = n$, we set:*

$$send(ev, o_1, o_2, \ldots) \equiv_{df} \bigvee_{ev \leq \hat{ev}} justsend = (true, \hat{ev}, o_1, o_2, \ldots),$$

$$receive(ev, o_1, o_2, \ldots) \equiv_{df} \bigvee_{ev \leq \hat{ev}} justrecv = (true, \hat{ev}, \mathsf{nil}, o_2, \ldots).$$

If send resp. receive do not refer to expressions, then the parameter values of justsend resp. justrecv are not considered. Otherwise the i-th parameter value of justsend resp. justrecv is to be compared with the i-th parameter expression of send resp. receive.

(ii) *For a triggered operation, creation, or destruction,*
$op \in c.ops \cup \{create_c, destroy\}$, $c \in C$, $o_1, o_2 \in O_C$, we set:

$$send(op, o_1, o_2, \ldots) \equiv_{df} receive(op, o_1, o_2, \ldots)$$
$$\equiv_{df} justcall = (true, op, o_1, o_2, \ldots).$$

Parameter expressions in send resp. receive are treated as explained above. Creation and destruction don't have parameters.

(iii) *For a reply $op = reply_\tau$, $o_1, o_2 \in O_C$ we set:*

$$send(op, o_1, o_2, \ldots) \equiv_{df} receive(op, o_1, o_2) \equiv_{df} justret = (true, op, o_1, o_2, \ldots).$$

The optional parameter expression in send resp. receive is treated as explained above.

(iv) *Each occurence of justcreated(i) is replaced by*

$$justcreated = m(i).$$

(v) *And or each o_i whose instance line does not begin with a creation, the activation condition is conjoined with*

$$o_i.status \in \{idle, executing, suspended\},$$

to require that the object denoted by o_i is alive at the time of activation. ∎

4 Query Reduction

Consider the LSC specification of the $ARCS$ system [13] depicted in figure 2 (for brevity we don't present the UML-model of the $ARCS$, but only implicitly introduce the classes relevant for our discussion; for details the reader is referred to the description in [13]).

By the LSC semantics of Section 3 it can be checked whether the system satisfies the specification by checking *all* concrete bindings of *Car* and *Terminal* objects to instance lines. But intuitively, it should be sufficient to check *a single* concrete binding for the *Car* identity car_0 since if the instance with this identity *always* behaves as required, then *every* *Car* behaves like that, since they are all instances of the same class with the same behaviour. The reason is that new objects are chosen non-deterministically at creation time, thus if an object car_1 would violate the specification in a run of the system, then there existed a run which choses car_0 instead of car_1 at creation time and thus there existed a run where car_0 violates the property, too.

In the following, we provide a formal basis for the just outlined intuition in full generality referring to the work of Ip and Dill [18] and McMillan [34] in Sections 4.1–4.3. In Section 4.4 we demonstrate the application of these results to the UML domain and in particular the example of figure 2.
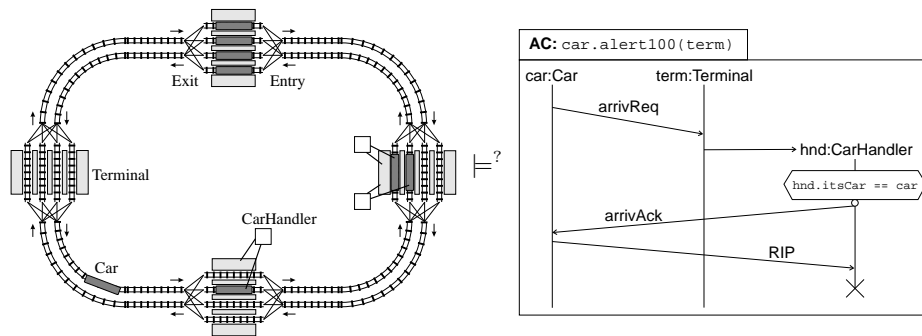

Fig. 2. LSC over $ARCS$: Whenever *car* (an instance of class *Car*) is 100 units ahead of a *Terminal term*, then it starts the entering protocol by sending an *arrivReq* event to *term* whose identity it obtained from one of its sensors.

The terminal then creates an instance of a class *CarHandler* which subsequently manages the whole entering and leaving procedure, i.e. it reserves and frees platforms and exits within *term* and sets the switches.

Once the car-handler obtained a platform and set the switch, it sends an *arrivAck* event back to the car which then enters the terminal. The car stores the identity of the sending car-handler for the further communication.

When the car is about to leave the terminal, it sends another request to its car-handler which sends back a granting event once the switches of the desired exit are set and free (not shown in the LSC).

After having left the terminal, *car* sends an event *RIP* to its *CarHandler* which causes *hnd* to free the reserved platform and exit and finally to destroy itself.

4.1   Introduction

By a result of [34], queries over quantified variables of a "symmetric" or scalar-set [18] type are implied by a finite set of representative cases.

If the representative cases are proven separately, there is not only an anticipated benefit from the smaller size of the formulae compared to the original quantification. The representative formulae are also more specialised than the original in that they refer to only a concrete binding of the quantified variables, thus standard model-reduction techniques like cone-of-influence reduction [2] can be applied more effectively.

Hence the reduction is at first not at all a model reduction – which would also be possible based on symmetric types by building the quotient of the transition relation wrt. the equivalence relation induced on the snapshots by symmetry [18, 10, 1, 19] as discussed in the introduction in Section 1 – but only a decomposition of formulae s.t. standard model-reduction techniques yield better results. The split into seperate tasks for the representative formulae may yet render proofs feasible, for which proving the whole property is not possible due to space or time complexity. be feasible.

The quotient graph approach to symmetry-based model-reduction is in general not applicable for LSCs since it applies only to a subset of LTL [10] which is not expressive enough for LSCs.

In this section we first introduce the general theory for special LTL formulae over STSs and then demonstrate that the semantics of LSCs for a UML model according to Def. 12 is a formula of the form the theory applies to.

The remainder of this section is structured as follows. In Section 4.2 we briefly provide the concept of automorphisms and the observation of [18] that permutations on a certain type – called scalarset – induce automorphisms on the state-space, thus allow to infer properties of the shape of the state-space. Section 4.3 concludes that LTL formulae which are quantifications over scalarset types can be proven by considering only a finite, representative set, yielding the general theory of query reduction. Section 4.4 demonstrates LSCs in the interpretation of Section 3 over the STS-semantics of UML [7] as a prominent application domain for these results.

4.2   Permutations and Automorphisms

Definition 13 (Permutation and Automorphism). *Let A be a finite set. A bijection $\pi : A \to A$ is called* permutation *on A. The set of all permutations on A is called $Sym(A)$.*

Let $S = (B, \Theta, \rho)$ be an STS and $\mathcal{M}$ a structure of B. A permutation $\pi \in Sym(\Sigma)$ is called an automorphism *of S iff*

(i) $\forall \, s \in \Sigma : \mathcal{M}[\![\Theta]\!](s) \implies \mathcal{M}[\![\Theta]\!](\pi(s))$

(ii) $\forall \, s, s' \in \Sigma : \mathcal{M}[\![\rho]\!](s, s') \implies \mathcal{M}[\![\rho]\!](\pi(s), \pi(s'))$ ∎

A central ingredient for the theory of query-reduction is the following notion of a relation between a permutation on the *domain* of an STS's system variable's type and the STS's state-space.

Definition 14 (Induced Permutation). *Let $B = (V, \Omega)$ be a signature and $\mathcal{M}$ a structure of B. Let $S = (B, \Theta, \rho)$ be an STS. A permutation $\pi : \mathcal{D}_{\tau_s} \to \mathcal{D}_{\tau_s}$ on the domain of type τ_s induces the permutation $\tilde{\pi} \in Sym(\Sigma)$ defined inductively pointwise for each snapshot $s \in \Sigma$ of STS by*

- $\mathcal{M}[\![v]\!](\tilde{\pi}(s)) = \hat{\pi}(\mathcal{M}[\![v]\!](s))$,
- $\mathcal{M}[\![a[expr]]\!](\tilde{\pi}(s)) = \hat{\pi}(\mathcal{M}[\![a]\!](s)(\mathcal{M}[\![expr]\!](\tilde{\pi}(s))))$,
- $\mathcal{M}[\![expr.x]\!](\tilde{\pi}(s)) = \hat{\pi}(\mathcal{M}[\![expr]\!](s).x)$,
- $\mathcal{M}[\![expr_0.a[expr_1]]\!](\tilde{\pi}(s)) = \hat{\pi}(\mathcal{M}[\![expr_0.a]\!](s)(\mathcal{M}[\![expr_1]\!](\tilde{\pi}(s))))$,
- $\mathcal{M}[\![expr]\!](\tilde{\pi}(s)) = \mathcal{M}[\![expr]\!](s)$, *otherwise,*

where $\hat{\pi}$ is π on $\mathcal{D}_{\tau_s}$, $\hat{\pi}|_{\mathcal{D}_{\tau_s}} = \pi$, and the identity otherwise, $\hat{\pi}|_{\complement \mathcal{D}_{\tau_s}} = id$.   ∎

In the following we are in particular interested in so called *scalarset* types, i.e. types s.t. every permutation on their domain induces an automorphism.

Note that [18] use the term scalarset for a type which obeys the set of syntactical rules given in Lemma 1 below, but the rules are sufficient but obviously not *necessary* criteria for scalarsets in the sense of the following definition:

Definition 15 (Scalarset). *A type τ with at most one special element $\mathsf{nil} \in \mathcal{D}_\tau$ is called* scalarset *iff for every permutation $\pi \in Sym(\mathcal{D}_\tau)$ with $\pi(\mathsf{nil}) = \mathsf{nil}$ the induced permutation $\tilde{\pi} \in Sym(\Sigma)$ is an automorphism.*   ∎

In the following, we simply translate the results of [18] for abstract transition programs – that a particular set of syntactic criteria is sufficient for the scalarset property – into the domain of STSs:

Lemma 1 (Automorphism). *Let $S = (B, \Theta, \rho)$ be an STS, τ a type of variables in B. The type τ is a scalarset if the predicates Θ and ρ obey the following syntactical rules [18]:*

S1 *Scalarset values are not used literally, except for the special element* nil.

S2 *Scalarset terms may be compared for equality. In a comparison, both sides must be terms of exactly the same scalarset type.*

S3 *If the left hand side of an assignment is of scalarset type, then the right hand side must be of exactly the same type.*

S4 *A scalarset type τ_s may be used as the index-type of arrays. Such arrays are only indexed by terms of type τ_s.*

S5 *A variable of scalarset type τ_s may be used as the running index of a for-loop if the body of the loop is independent from the order of the iterations.*

S6 *Other operations are not allowed, in particular may scalarsets not be used as operands of "+" or "casted" into an integer type.*   ∎

Proof. Analogous to [18].   □

The following Lemma 2 states that the truth-value of a property over scalar-set constants o_i in a π-permuted interpretation and evaluated in snapshot $\pi(s)$ can be obtained by evaluating the property in the *original* interpretation and *original* snapshot s.

Lemma 2 (Substitution). *Let $B = (V, \Omega)$ be a signature with $o_1, \ldots, o_n \in \Omega$, $n \in \mathbb{N}_0$, 0-ary constants of scalarset type τ_s. Let $\phi \in Expr_{PL}(B)$ be an expression over B.*
Let $o_{1_i}, o_{2_i} \in \mathcal{D}_{\tau_s} \setminus \{\mathsf{nil}\}$ for $1 \le i \le n$. Let $\mathcal{M}_1 = (\mathcal{D}_1, \mathcal{I}_1)$ be a structure of B with $\mathcal{I}_1(o_i) = o_{1_i}$ and $\mathcal{M}_2 = (\mathcal{D}_2, \mathcal{I}_2)$ a structure of B with $\mathcal{I}_2(o_i) = o_{2_i}$, and $\mathcal{I}_1(f) = \mathcal{I}_2(f)$, $f \in \Omega$, otherwise. Set

$$\pi = \{ o_{1_i} \mapsto o_{2_i}, o_{2_i} \mapsto o_{1_i} \mid 1 \le i \le n \} \in Sym(\mathcal{D}_{\tau_s})$$

and let $s \in \Sigma$ be a valuation of V. If ϕ obeys the scalarset rules (S1)–(S6), then

$$\mathcal{M}_1[\![\phi]\!](s) = \mathcal{M}_2[\![\phi]\!](\tilde{\pi}(s)).$$ ■

Proof. (By induction over the structure of ϕ.)
Since ϕ obeys the scalarset rules, a constant $o_i, 1 \le i \le n$, can without loss of generality only appear in in the following places:

- Comparison against another constant:
  $\phi \equiv o_i = o_j$:

$$\mathcal{M}_1[\![\phi]\!](s) = \mathcal{M}_1[\![o_i = o_j]\!](s) = eq(o_{1_i}, o_{1_j})$$
$$\stackrel{!}{=} eq(o_{2_i}, o_{2_j}) = \mathcal{M}_2[\![o_i = o_j]\!](\tilde{\pi}(s)) = \mathcal{M}_2[\![\phi]\!](\tilde{\pi}(s)),$$

  since $o_{1_i} = o_{1_j} \iff o_{2_i} = o_{2_j}$ and $o_{1_i} \ne o_{1_j} \iff o_{2_i} \ne o_{2_j}$ by definition of π.

- Comparison against a variable x of type τ_s:
  $\phi \equiv x = o_i$:

$$\mathcal{M}_1[\![\phi]\!](s) = \mathcal{M}_1[\![x = o_i]\!](s) = eq(s(x), o_{1_i})$$
$$\stackrel{!}{=} eq(\pi(s(x)), o_{2_i}) \underset{\text{Def.14}}{=} \mathcal{M}_2[\![x = o_i]\!](\tilde{\pi}(s)) = \mathcal{M}_2[\![\phi]\!](\tilde{\pi}(s)),$$

  since $s(x) = o_{1_i} \iff \pi(s(x)) = o_{2_i}$ and $s(x) \ne o_{1_i} \iff \pi(s(x)) \ne o_{2_i}$ by definition of π.

- Array index and comparison against indexed value:
  $\phi \equiv a[o_i] = o_j$:

$$\mathcal{M}_1[\![\phi]\!](s) = eq(s(a)(o_{1_i}), o_{1_j})$$
$$\stackrel{!}{=} eq(\pi(s(a)(o_{1_i})), o_{2_j}) = eq(\pi(s(a)(\pi(o_{2_i}))), o_{2_j})$$
$$\underset{\text{Def.14}}{=} \mathcal{M}_2[\![a[o_i] = o_j]\!](\tilde{\pi}(s)) = \mathcal{M}_2[\![\phi]\!](\tilde{\pi}(s)),$$

  since $\pi(s(a)(s(o_{1_i}))) = o_{2_j}$ iff $s(a)(o_{1_i}) = o_{1_j}$ by definition of π.

- Comparison against structure component of type τ_s:
  $\phi \equiv expr.x = o_i$:

$$\mathcal{M}_1\llbracket\phi\rrbracket(s) = eq(\mathcal{M}_1\llbracket expr\rrbracket(s).x, o_{1_i}) \overset{!}{=} eq(\pi(\mathcal{M}_1\llbracket expr\rrbracket(s).x), o_{2_i})$$
$$\underset{\text{ind.}}{=} eq(\pi(\mathcal{M}_2\llbracket expr\rrbracket(\tilde{\pi}(s)).x), o_{2_i}) \underset{\text{Def.14}}{=} \mathcal{M}_2\llbracket expr.x = o_i\rrbracket(\tilde{\pi}(s))$$
$$= \mathcal{M}_2\llbracket\phi\rrbracket(\tilde{\pi}(s)),$$

since $\pi(\mathcal{M}_1\llbracket expr\rrbracket(s).x) = o_{2_i}$ iff $\mathcal{M}_1\llbracket expr\rrbracket(s).x = o_{1_i}$ by definition of π.

- Comparison against a general array expression of type τ_s:
  $\phi \equiv expr_0.a[expr_1] = o_i$:

$$\mathcal{M}_1\llbracket\phi\rrbracket(s) = eq(\mathcal{M}_1\llbracket expr_0.a[expr_1]\rrbracket(s), o_{1_i})$$
$$\overset{!}{=} eq(\pi(\mathcal{M}_1\llbracket expr_0.a[expr_1]\rrbracket(s)), o_{2_i})$$
$$= eq(\pi(\mathcal{M}_1\llbracket expr_0.a\rrbracket(s)(\mathcal{M}_1\llbracket expr_1\rrbracket(s))), o_{2_i})$$
$$\underset{\text{ind.}}{=} eq(\pi(\mathcal{M}_2\llbracket expr_0.a\rrbracket(\tilde{\pi}(s))(\mathcal{M}_2\llbracket expr_1\rrbracket(\tilde{\pi}(s)))), o_{2_i})$$
$$\underset{\text{Def.14}}{=} \mathcal{M}_2\llbracket expr_0.a[expr_1] = o_i\rrbracket(\tilde{\pi}(s)) = \mathcal{M}_2\llbracket\phi\rrbracket(\tilde{\pi}(s))$$

by definition of π.   □

4.3   Query Reduction

If a quantified property uses only a single quantification constant of scalarset type τ_s, then it is sufficient to prove one particular binding. But if there are two quantification constants in the property, we need at least two concrete bindings: one which represents all bindings which bind the same value to both constants and one which represents the bindings with different values.

Def. 16 introduces the concept of a representative set and Lemma 3 claims that a finite representative set exists for every finite property over a scalarset type.

Definition 16 (Representative Set).
Let τ_s be a scalarset-type. A set $R \subset (\mathcal{D}_{\tau_s} \setminus \{\mathsf{nil}\})^n$, $n \geq 1$, is called representative set *for τ_s^n iff*

$$\forall \, (o_1, \ldots, o_n) \in \mathcal{D}_{\tau_s}^n \; \exists \, r_0 = (o_{0_1}, \ldots, o_{0_n}) \in R, \pi \in Sym(\tau_s):$$
$$(\pi(o_{0_1}), \ldots, \pi(o_{0_n})) = (o_1, \ldots, o_n).$$

R is called minimal representative set *if all proper subsets $R' \subsetneq R$ are not representative.*   ∎

Lemma 3 (Representative Set). *Let τ_s be a scalarset-type and $n \geq 1$. Then there exists a finite representative set R for τ_s^n.*   ∎

Proof. Choose $o_1, \ldots, o_n \in \mathcal{D}_{\tau_s} \setminus \mathsf{nil}$ pairwise different.
Set $R = \{o_1\} \times \{o_1, o_2\} \times \cdots \times \{o_1, \ldots, o_n\}$.   □

Note that the R constructed in Lemma 3 is in general not minimal, since e.g. (o_1, o_1, o_2) and (o_1, o_1, o_3) are equivalent in case of $n = 3$ but both are in R.

Lemma 4 (Query Reduction). *Let ϕ be an LTL expression over signature $B = (V_B, \Omega_B)$ with 0-ary constants $o_1, \ldots, o_n \in \Omega_B$, $n \in \mathbb{N}_0$, of type τ_s, and 0-ary constants $x_1, \ldots, x_m \in \Omega_B$, $m \in \mathbb{N}_0$, not of type τ_s. Let $S = ((V, \Omega), \Theta, \rho)$ be an STS with $V = V_B$ and $\Omega_B \setminus \{o_i, x_j \mid 1 \le i \le n, 1 \le j \le m\} \subseteq \Omega$. Let $\mathcal{M}$ be a structure of B. If R is a representative set for τ_s^n then*

$$(\forall \, (o_{0_1}, \ldots, o_{0_n}) \in R \tag{1}$$
$$\forall \, x_{0_1} \in \mathcal{D}_{type(x_1)}, \ldots, x_{0_m} \in \mathcal{D}_{type(x_m)} : S \models_{\mathcal{M}', q} \phi)$$

$$\iff (\forall \, o_{0_1} \in \mathcal{D}_{type(o_1)}, \ldots, o_{0_n} \in \mathcal{D}_{type(o_n)} \tag{2}$$
$$\forall \, x_{0_1} \in \mathcal{D}_{type(x_1)}, \ldots, x_{0_m} \in \mathcal{D}_{type(x_m)} : S \models_{\mathcal{M}', q} \phi).$$

for $q \in \{\exists; \exists, 0; \forall; \forall, 0\}$ and $\mathcal{M}'$ constructed as in Def. 12.   ∎

Proof. Let $q = \exists$.
In order to prove direction (1) $\implies$ (2), choose $o_1', \ldots, o_n' \in \mathcal{D}_{\tau_s}$. Since R is representative, there exists $(o_{r_1}, \ldots, o_{r_n}) \in R$ and $\pi \in Sym(\tau_s)$ s.t. $o_i' = \pi(o_{r_i})$.
By premise, $S \models_{\mathcal{M}', q} \phi$, where $\mathcal{M}' = \mathcal{M} \cup \{o_i \mapsto o_{r_i} \mid 1 \le i \le n\}$, i.e. $\exists \, r \in runs(S) \, \exists \, i \in \mathbb{N} : r/i \models_{\mathcal{M}', q} \phi$.
Set $r_\pi = \pi(r(0)) \, \pi(r(1)) \ldots$ and $\mathcal{M}' = \mathcal{M} \cup \{o_i \mapsto o_i' \mid 1 \le i \le n\}$. Then $r_\pi \in runs(S)$ by Lemma 1 and $r_\pi/i \models_{\mathcal{M}', q} \phi$ by induction over the structure of ϕ:

- $\phi \equiv expr$: $\mathcal{M}' [\![\phi]\!](r_\pi(i)) \underset{\text{Lem.2}}{=} \mathcal{M}'[\![\phi]\!](r(i)) = true$ by premise.

- $\phi \equiv f \vee g$: Then $r \models_{\mathcal{M}', q} f$ or $r \models_{\mathcal{M}', q} g$.
  Thus by induction hypothesis $r_\pi \models_{\mathcal{M}', q} f$ or $r_\pi \models_{\mathcal{M}', q} g$.

- $\phi \equiv \neg f$: analogously to the previous case.

- $\phi \equiv \mathbf{X} f$: Then $r/i + 1 \models_{\mathcal{M}', q} f$, thus $r_\pi/i + 1 \models_{\mathcal{M}', q} f$ by induction hypothesis.

- $\phi \equiv \mathbf{G} f$: Then for all $j \ge i$, $r/j \models_{\mathcal{M}', q} f$, thus also $r_\pi/j \models_{\mathcal{M}', q} f$ by induction hypothesis.

- $\phi \equiv f \mathbf{U} g$: Then exists $k \ge i$ s.t. $r/k \models_{\mathcal{M}', q} g$, thus also $r_\pi/k \models_{\mathcal{M}', q} g$ by induction hypothesis. For all $i \le j < k$, $r/j \models_{\mathcal{M}', q} f$ thus also $r_\pi/j \models_{\mathcal{M}', q} f$ by induction hypothesis.

The case $q = \exists, 0$ is obtained analogous, the cases $q = \forall$ and $q = \forall, 0$ similar by contradiction. The direction (2) $\implies$ (1) holds trivially.   □

4.4   Verifying LSCs against UML Models

The basis for query reduction in the domain of UML is the following observation that in the UML semantics of [7] the object reference types O_c are in fact scalarset types:

Lemma 5 (Scalarsets in UML).
Let $M = (T, F, Sig, <, C, c_{root}, A)$ be a UML-Model and $STS(M) = (B, \Theta, \rho)$ its semantics according to [7].

(i) All object reference types T_c, $c \in C$ are scalarsets with special element nil.
(ii) For all unordered association ends a, the index type τ_a is a scalarset without special element.
(iii) For all unordered behavioral features f, the index type τ_f is a scalarset without special element. ∎

The proof of 5.(i) is by syntactically checking Θ and ρ against the rules (S1)–(S6).[3] The proof of 5.(ii) and (iii) cannot be obtained as directly since iterators over associations and behavioural features are not present in Θ and ρ because iteration is supposed to take multiple steps of the transition system in the semantics. Although, they are visible on a higher language level, thus the property of rule (S5) has to be checked on this higher level and then to be preserved by the preprocessing steps of [7].

According to Section 3, the semantics of an LSC wrt. a UML model is an LTL formula quantified over constants of types O_c, hence Lemma 4 directly applies.

A representative set may be obtained as demonstrated in the proof of Lemma 3. By taking into account the activation condition, some of the representative cases may already be found to trivially fulfil the requirement. For example if the specification contains two instances of the same type which the activation condition requires to be different at activation time.

From a technical point of view, all verifications of representative cases form *completely* independent tasks thus may be carried out fully parallel thus allow for further reductions of completion time for the whole task.

Back to the example from the beginning of the section, we find that we refer to only one instance of classes *Car*, *Terminal*, and *CarHandler* in the LSC, thus the set

$$R =_{df} \{(\textit{Car}, 0), (\textit{Terminal}, 0), (\textit{CarHandler}, 0)\} \subset C \times \mathbb{N}_0$$

is a (minimal) representative set (assuming $(c, 0) \neq \text{nil}_c$ for classes $c \in C$). Thus it is sufficient to verify only this single case of concrete bindings (see figure 3).

5   Model Abstraction by Data-Type-Reduction

As suggested by figure 3, there is a priori no model reduction due to the application of query reduction. Unfortunately, the standard technique of cone-of-influence reduction may not work well for UML models due to the indirect

---

[3] The typing in the presentation of [7] and even in Section 3 is actually too weak to fulfil the syntactical criteria "as is": for brevity both refer to O_C which is the union of all object reference types. It is straightforward to obtain a representation which separates the object reference types s.t. we can obtain their scalarset property directly by syntactical criteria.
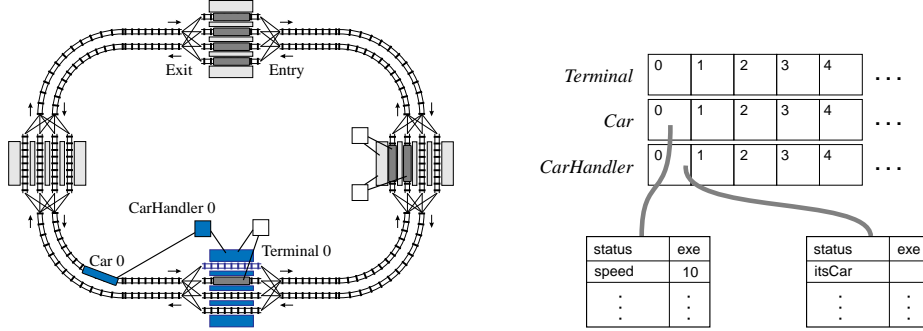
Fig. 3. Query Reduction: The focus is on only the single *Car* 0, *Terminal* 0, and *CarHandler* 0. The model itself is a priori not reduced.

addressing of array places. Thus intuitively, we want to refer to "all other cars" by a single object reference $(Car, \bot)$ and over-approximate "their" behaviour.

It is obviously not sufficient to only change the unbounded array to a width of two in the example s.t. $(Car, \bot)$ denotes the second entry whose values are freed, i.e. may take all possible values during a run, since an expression may refer to two "other cars" at once, for example $p.speed + q.speed$ would always yield the even value $2 \cdot p.speed$ if p and q refer to "an other car", which is not necessarily the case in the original implementation where p and q might refer to *different* other cars.

An abstraction technique, which yields the desired result, is the *data-type reduction* introduced by McMillan [34]. It can be made explicit by a syntactical transformation of the system description which modifies the "places" of objects referring to an other object in the sense that every reference which possibly reads a value of an other object is modified. For the above example expression it would yield

$$(p = \bot \,?\, guess_1(\tau) : p.speed) + (q = \bot \,?\, guess_2(\tau) : q.speed)$$

where τ is the type of attribute *speed* and '$guess_i$' are new free variables (unrestricted system inputs) of type τ.

This expression can evaluate to every possible value allowed by the typing of attribute *speed*, thus it over-approximates the valuations observed in the original model.

The following definitions of section 5.1 introduce the notion of data-type reduction and how the initial snapshot and transition predicate have to be changed to implement a data-type reduction. Lemma 6 claims, based on the notion of a projection, that for every predicate holding in a snapshot of the original system there exists a snapshot of the abstracted system in which the predicate holds.

In section 5.2 we show that the data-type reduced system simulates the original system and thus is in fact an over-approximation of the original system.

5.1   Data-Type-Reduction

Definition 17 (Data-type reduction). *Let τ_s be a scalarset type. A subset $dtr(\tau_s) \subseteq \mathcal{D}_{\tau_s}$ of its domain is called a* data-type reduction *(DTR) on τ_s.*

Let $t = \{T \mid T \text{ type}\} \supseteq \{\tau_s\}$ be a set of types. The data-type reduction dtr induces a data-type reduction d on the domains of the (structured) types constructed from t inductively as follows:

(i) If $\tau \in t$ is an unstructured type,
  then $d(\mathcal{D}_\tau) = dtr(\tau_s) \cup \{\bot\}$, if $\tau = \tau_s$, and $d(\mathcal{D}_\tau) = \mathcal{D}_\tau$ otherwise.

(ii) If $\tau = \tau_1 \times \cdots \times \tau_n$ is a record type, then $d(\mathcal{D}_\tau) = d(\mathcal{D}_{\tau_1}) \times \cdots \times d(\mathcal{D}_{\tau_n})$.

(iii) If $\tau = \tau_1 \to \tau_2$ is an array type, then $d(\mathcal{D}_\tau) = d(\mathcal{D}_{\tau_1}) \to d(\mathcal{D}_{\tau_2})$. ∎

   If multiple data-type reductions are applied, multiple distinct symbols $\bot$ representing "all other values", one for each type, have to be introduced. In the following it is clear by context, of which type $\bot$ is.

   The following definition describes how we obtain a "data-type reduced expression" which implements the over-approximation as a prerequisite for data-type reduction of STSs.

Definition 18 (Data-type reduction for Expressions). *Let 'expr' be a FOL expression over signature B. The* data-type reduced expression *$d(expr)$ (or $expr_d$) over B_d (as defined below) is obtained from 'expr' by applying the following syntactical transformations:*

(i) Indexing an array a with scalarset index-type τ_s at index 'expr' not on the left hand side of an "assignment" is changed s.t. it yields non-deterministically a value from the component-type T of a if the index expression has value $\bot$:

$$a[expr] \rightsquigarrow (expr = \bot \,?\, guess(T) : a[expr]),$$

Every $guess(T)$ stands for a fresh system variable $g \in V_d \setminus V$ which is not restricted by Θ or ρ and thus for each snapshot s and each value $x \in T$ there exists a snapshot s' which coincides with s on all other variables and evaluates $guess(T)$ to x.
We set $expr_g \equiv_{df} a[expr]$, i.e. $expr_g$ denotes the expression g was introduced for.

(ii) Two transformed expressions $expr_1$, $expr_2$ of the data-type reduced scalarset type τ_s which are compared for equality are changed s.t. the comparison yields a non-deterministic truth-value if both expressions have value $\bot$:

$$expr_1 = expr_2 \rightsquigarrow (expr_1 = expr_2 = \bot \,?\, guess(\mathbb{B}) : expr_1 = expr_2).$$

Set $expr_g \equiv_{df} expr_1 = expr_2$.

(iii) Indexing an array a with scalarset index-type τ_s at index $expr_1$ on the left hand side of an "assignment" (cf. [7]) is changed s.t. it is considered only if the index expression does not have value $\bot$:

$$a[expr_1]sel' := expr_2 \rightsquigarrow (expr_1 \neq \bot \implies a[expr_1]sel' := expr_2)$$

Here 'sel' denotes a possibly empty selection *if the values of a are again of a structured type.*

The signature B_d is

$$B_d = (V_d, \Omega_d) = (V \cup G, \Omega)$$

where $v \in V_d$ is associated with domain $d(\mathcal{D}_{type(v)})$. The set G denotes all fresh system variables introduced into $expr_d$ above. ∎

Note that in the general case of Def. 20, a value from the component domain of an array is "guessed" which might be a large structure like in the UML semantics. If parts of the structure are subsequently selected in the expression, a trivial optimisation consists of "guessing" only a value of the selected part's type.

Furthermore, there need not be an actual storage-place for the $\perp$-th entry of a data-type reduced array a. If there would be an actual place addressed by $\perp$, then its value would never be visible in any expression, since every expression using a is changed according to rule (i) which does not actually read $a[\perp]$ but provides that *any* value can be taken.

The following Lemma 6 states, based on the notion of a projection from Def. 19, that the data-type reduced expression can evaluate to every value observable for the original expression if the valuation is chosen appropriately. This is the main building block for the claim of Section 5.2, that the data-type reduced STS *simulates* the original one.

Definition 19 (Projection). *Let $B = (V, \Omega)$ be a signature, $dtr(\tau_s)$ a data-type reduction on the scalarset type τ_s, and $s \in \Sigma$ a valuation of the variables in V.*
The projection of s onto d, $\pi_d(s)$, is defined inductively as follows:

(i) If $v \in V$ is a variable of basic type T, then

$$\pi_d(s)(v) = \begin{cases} s(v) & , \text{ if } s(v) \in d \text{ or } T \neq \tau_s \\ \perp & , \text{ otherwise} \end{cases}$$

(ii) If $v \in V$ is a variable of record type $T_1 \times \cdots \times T_n$ and $s(v) = (x_1, \ldots, x_n)$ then $\pi_d(s)(v) = (\pi_d(s)(x_1), \ldots, \pi_d(s)(x_n))$.

(iii) If $v \in V$ is a variable of array type $T_1 \rightarrow T_2$ and $s(v)(i) = x, i \in T_1$ then $\pi_d(s(v))(i) = \pi_d(x)$. ∎

Lemma 6 (Satifiability). *Let $B = (V, \Omega)$ be a signature, d a data-type reduction on the scalarset type τ_s, $\mathcal{M}$ a structure of B, and s a valuation of variables in V. Let 'expr' be a predicate-logic expression over B which obeys the scalarset rules (S1)–(S6).*

Let V_d and $expr_d$ be the result of the operations of Def. 18 applied. Let the valuation $\bar{s}$ of variables in V_d be defined pointwise as follows:

$$\bar{s}(v) = \begin{cases} \pi_d(s(v)) & , \text{ if } v \in V \\ \pi_d(\mathcal{M}[\![expr_g]\!](s)) & , \text{ if } v = g \in G \end{cases}$$

Then $\mathcal{M}[\![expr]\!](s) = \mathcal{M}[\![expr_d]\!](\bar{s})$. ∎

Proof. (By induction over the structure of *expr*.)
Let s be a valuation of variables in V. The data-type reduction affects only the following cases:

- $expr \equiv a[expr_1]$, with $expr_1$ an expression of type τ_s and a with components of boolean type:
  Then *expr* has been changed to

  $$expr_d \equiv (expr_1 = \bot \,?\, guess(\mathbb{B}) : a[expr_1]).$$

  where $guess(\mathbb{B})$ denotes a variable $g \in V_d$, thus

  $$\begin{aligned} \mathcal{M}[\![expr_d]\!](\bar{s}) &= \mathcal{M}[\![(expr_1 = \bot \,?\, g : a[expr_1])]\!](\bar{s}) \\ &= \mathcal{I}(?\!:)(\mathcal{M}[\![expr_1]\!](\bar{s}) = \bot, \bar{s}(g), \mathcal{M}[\![a[expr_1]]\!](\bar{s})) \\ &= \mathcal{M}[\![a[expr_1]]\!](s). \end{aligned}$$

  since $\mathcal{M}[\![expr_1]\!](\bar{s}) = \bot$ implies $\bar{s}(g) = \mathcal{M}[\![a[expr_1]]\!](s)$ by construction of $\bar{s}$.
  Analogously for structured array values from which a component of boolean type is selected.

- $expr \equiv expr_1 = expr_2$, both $expr_1, expr_2$ of type τ_s:
  Then *expr* has been changed to

  $$expr_d \equiv (expr_1 = expr_2 = \bot \,?\, guess(\mathbb{B}) : expr_1 = expr_2).$$

  where $guess(\mathbb{B})$ denotes a variable $g \in V_d$, thus

  $$\begin{aligned} \mathcal{M}[\![expr_d]\!](\bar{s}) &= \mathcal{M}[\![(expr_1 = expr_2 = \bot \,?\, g : expr_1 = expr_2)]\!](\bar{s}) \\ &= \mathcal{M}[\![expr_1 = expr_2]\!](s). \end{aligned}$$
  □

5.2 Simulation

Given an STS over a signature with a scalarset type, the data-type reduced STS is obtained as follows:

Definition 20 (Data-type reduced STS). *Let $S = (B, \Theta, \rho)$ be an STS. The data-type reduced STS is*

$$d(S) =_{df} S_d =_{df} (B_d, \Theta_d, \rho_d)$$

where B_d, Θ_d, and ρ_d are obtained from Θ and ρ by applying the transformations of Def. 18. ∎

The following lemma claims, that the data-type reduced system is in fact a simulation of the original system in the sense of Def. 21 below, i.e. for every run in the original system there is a related run in the abstract system.

Definition 21 (Simulation). *Let $S_1 = (B_1, \Theta_1, \rho_1)$ and $S_2 = (B_2, \Theta_2, \rho_2)$ be STSs, $\mathcal{M}_1$ and $\mathcal{M}_2$ structures of their signatures, and Σ_1 and Σ_2 the sets of their snapshots.*
We say S_1 simulates S_2, $S_1 \rightsquigarrow S_2$, iff there exists a relation $\varrho \subseteq \Sigma_1 \times \Sigma_2$ s.t.

(i) $\forall\, s_2 \in \Sigma_2 : \mathcal{M}_2[\![\Theta_2]\!](s_2) \implies \exists\, s_1 \in \Sigma_1 : \mathcal{M}_1[\![\Theta_1]\!](s_1) \wedge (s_1, s_2) \in \varrho$

(ii) $\forall\, (s_1, s_2) \in \varrho\, \forall\, s_2' \in \Sigma_2 :$

$$\mathcal{M}_2[\![\rho_2]\!](s_2, s_2') \implies \exists\, s_1' \in \Sigma_1 : \mathcal{M}_1[\![\rho_1]\!](s_1, s_1') \wedge (s_1', s_2') \in \varrho$$

The relation ϱ is called simulation relation. ∎

Lemma 7 (DTR Simulation). *Let $S = (B, \Theta, \rho)$ be an STS and d a data-type reduction on the scalarset type τ_s and S_d the data-type reduced STS. Let $\mathcal{M}$ be a structure of S and S_d. Then $S_d \rightsquigarrow S$.* ∎

Proof. Define $\varrho \subseteq \Sigma \times \Sigma_d$ for snapshots s, s' s.t. s' is the projection of s onto d for variables in B and assigns the value $\mathcal{M}[\![expr_g]\!](s)$ to every variable $g \in G$ introduced during construction of S_d as in Lemma 6:

$$\varrho =_{df} \{(s, \bar{s}) \mid s \in \Sigma\}.$$

ϱ is a simulation relation:

(i) Let $s_2 \in \Sigma$ s.t. $\mathcal{M}[\![\Theta]\!](s_2) = true$.
   Choose $s_1 = \bar{s}_2$ as in Lemma 6. Then $(s_1, s_2) \in \varrho$ and $\mathcal{M}[\![\Theta_d]\!](s_1) = true$ by Lemma 6, since Θ_d is obtained from the boolean expression Θ by applying the transformations of Def. 20.
(ii) Let $(s_1, s_2) \in \varrho$ and $s_2' \in \Sigma$ s.t. $\mathcal{M}[\![\rho]\!](s_2, s_2') = true$. Choose $s_1' = \bar{s}_2'$ as in Lemma 6. Then $(s_1, s_2) \in \varrho$ and $\mathcal{M}[\![\rho_d]\!](s_1, s_1') = true$ by Lemma 6. □

Putting it all together, the following Lemma shows that the relation chosen in the proof of Lemma 7 provides us with the snapshots required as premise of Lemma 6, hence yielding the desired result for LTL formulae.

Lemma 8 (Data-Type Reduction). *Let $S = (B, \Theta, \rho)$ be an STS and $dtr(\tau_s)$ a data-type reduction on the scalarset type τ_s and S_d the data-type reduced STS. Let $\mathcal{M}$ be a structure of S and S_d. Let ϕ be an LTL formula wrt. S which obeys the scalarset rules (S1)–(S6) and where no subterm except for o_i is of scalarset type τ_s. Then*

$$S_d \models_{\mathcal{M}, \forall(,0)} \phi \implies S \models_{\mathcal{M}, \forall(,0)} \phi \text{ and } S \models_{\mathcal{M}, \exists(,0)} \phi \implies S_d \models_{\mathcal{M}, \exists(,0)} \phi. \quad ∎$$

Proof. (By contraposition.)

(i) Let $S \not\models_{\mathcal{M},\forall} \phi$, i.e. $\exists\, r \in runs(S)\ \exists\, i \in \mathbb{N} : r/i \not\models_{\mathcal{M}} \phi$. By (the proof of) Lemma 7 exists a run $r_d \in runs(S_d)$ s.t. $\forall\, i \in \mathbb{N} : r_d(i) = r(i)$.

Then $S_d \not\models_{\mathcal{M},\forall} \phi$ follows by induction over the structure of ϕ:

- $\phi \equiv expr$:
  Then $\mathcal{M}[\![expr]\!](r(i)) = false$. By Lemma 6, $\mathcal{M}[\![expr]\!](r_d(i)) = false$, thus $r_d/i \not\models_{\mathcal{M}} \phi$.

- $\phi \equiv f \vee g$: Then $r/i \not\models_{\mathcal{M}} f$ and $r/i \not\models_{\mathcal{M}} g$.
  By induction hypothesis, $r_d/i \not\models_{\mathcal{M}} f$ and $r_d/i \not\models_{\mathcal{M}} g$.

- $\phi \equiv \neg f$: Then $r/i \models_{\mathcal{M}} f$. By induction hypothesis $r_d/i \models_{\mathcal{M}} f$.

- $\phi \equiv \mathbf{X}\, f$: Then $r/i + 1 \not\models_{\mathcal{M}} f$. By induction hypothesis $r_d/i + 1 \not\models_{\mathcal{M}} f$.

- $\phi \equiv \mathbf{G}\, f$: Then $\exists\, j \in \mathbb{N}_0 : r/i + j \not\models_{\mathcal{M}} f$.
  By induction hypothesis $r_d/i + j \not\models_{\mathcal{M}} f$, thus $r_d/i \not\models_{\mathcal{M}} \phi$.

- $\phi \equiv f\, \mathbf{U}\, g$: Differentiate between two cases:
  a) $\forall\, j \in \mathbb{N}_0 : r/i + j \not\models_{\mathcal{M}} g$. Then by induction hypothesis $\forall\, j \in \mathbb{N}_0 : r_d/i + j \not\models_{\mathcal{M}} g$, thus $r_d/i \not\models_{\mathcal{M}} \phi$.

  b) For every $j \in \mathbb{N}_0$ s.t. $r/i + j \models_{\mathcal{M}} g$, exists $0 \le k < j$ s.t. $r/i + k \not\models_{\mathcal{M}} f$. Let $j' \in \mathbb{N}_0$ s.t. $r_d/i + j' \models_{\mathcal{M}} g$. Then by induction hypothesis exists $k' < j'$ s.t. $r_d/i + k' \not\models_{\mathcal{M}} f$, thus $r_d/i \not\models_{\mathcal{M}} \phi$.

  Similar for $\models_{\mathcal{M},\forall,0}$.

(ii) The cases $\models_{\mathcal{M},\exists}$ and $\models_{\mathcal{M},\exists,0}$ are obtained directly by construction of S_d and Lemma 6.   □

The requirement on ϕ is not as strong as it seems since a boolean expression with a subterm $expr_0$ of type τ_s can easily be integrated into the model as an auxiliary (or observer) variable, i.e. a boolean variable which is assigned the value of $expr_0$. Then ϕ references the auxiliary variable instead of $expr_0$. Within the model, $expr_0$ undergoes the changes of Def. 18.

Note that for formal verification, only the former implication of Lemma 8 is of practical relevance: if we are able to prove the property for the abstract system, then it holds in the original system. But if we are seeking for an example run, there is no guarantee that a run found in the abstract system is also a run of the concrete system.

5.3 Parameterised Designs

A direct corollary of the previous Section 5.2 is the following [34]:

Corollary 1. *Let $S = (B, \Theta, \rho)$ be an STS and $dtr(\tau_s)$ a data-type reduction on the scalarset type τ_s and S_d the data-type reduced STS. Let ϕ be an LTL formula over B which obeys the scalarset rules (S1)–(S6). Then*

$$S_d \models_{\forall,(,0)} \phi \implies \forall d' \supseteq d : S_{d'} \models_{\forall,(,0)} \phi \qquad \blacksquare$$

That is, if it can be proven that a property ϕ holds for *some* data-type reduced system, then it holds for *every* larger system. This re-formulation of Lemma 8 is in particular relevant for parameterised systems like the *ARCS*, which is parameterised in the number of terminals, platforms per terminal, and cars.

5.4   Data-type reduction for UML

In the domain of LSC verification for UML, data-type reduction is applied depending on the specification analogous to the methodology proposed by McMillan [34]:

For an LSC with instance lines annotated by N different types, we apply N (orthogonal) data-type reductions as follows. Let $i_{k_1}, \ldots, i_{k_n}$, $1 \leq k \leq N$, be the chosen concrete objects of type O_{c_k} and $d_k = \{o_{k_1}, \ldots, o_{k_n}\}$ a data-type reduction. Then $S^k =_{df} d_k(S^{k-1})$, where $S^0 =_{df} S$. The overall data-type reduced system is $d(S) =_{df} S^N$.

This yields a very coarse abstraction. For example in the $ARCS$, a specification which requires that two cars don't collide may refer to only two concrete objects at first: two cars.

Since the position of a car depends on its speed which is measured by its cruiser, there will be a false-negative if all cruiser objects are abstracted according to the heuristic data-type reduction.

An iteration of the specification would introduce a cruiser instance for every car relating them in the activation condition but not showing any communication between cars and cruisers.[4] Then the heuristics would yield a system with as much concrete cruisers as needed for the cars s.t. the property might hold unless there are further iterations necessary.

Another main source of false-negatives, so called interference, is discussed in Section 6.

For the running example, we would apply the data-type-reductions,

$$dtr(\textit{Car}) =_{df} \{(\textit{Car}, 0)\}, \, dtr(\textit{Terminal}) =_{df} \{(\textit{Terminal}, 0)\},$$
$$dtr(\textit{CarHandler}) =_{df} \{(\textit{CarHandler}, 0)\},$$

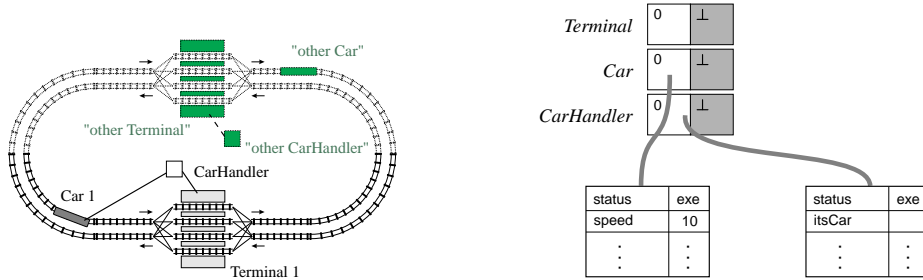and yield a system as illustrated by figure 4.

Fig. 4. Data-type Reduction: In the reduced system, the *Car* with identity 0 can distinguish only itself and "some other car" at the *Terminal* with identity 0 or at "some other terminal".

---

[4] This is the LSC equivalent to *case splitting* in the methodology of McMillan [34].

Note that actually drawing the objects with index $\perp$ is kind of misleading since there is in fact no place to store attribute values of the object with $\perp$. But from the point of view of someone holding a reference to $\perp$, it is a valid reference value and behaves in terms of types as expected, i.e. it offers the same set of attributes as every object of this class. It just behaves "strange" in the sense that reading the same attribute over the same, unchanged reference may yield different values due to the introduced over-approximation.

To illustrate the effect of the abstraction on the observable dynamic behaviour, in the following we briefly "play through" event sending and operation calls; the effect on expressions has already been discussed above.

Consider class *Car* in the *ARCS* which has an association to a *CarHandler* (cf. figure 6). The *CarHandler* is created by a *Terminal* and its identity is then passed over to the *Car*.

By executing the create action, the *Terminal* may get a reference to a concrete object or to "an other *CarHandler*", $\perp_{CarHandler}$.

In the following, assume it got the $\perp_{CarHandler}$, passed it to the *Car* and the *Car* now sends an event to this *CarHandler*.

The event is entered into the event queue responsible for the *CarHandler*, which is in fact guessed, s.t. the event may end up in *any* event queue. In the event queue of a concrete active object, the event will move to the top of the queue and thus become ready to be dispatched.

The changed transition predicate causes the current state of the destination $\perp_{CarHandler}$ to be *guessed*. Thus the event may be discarded or accepted. If the choice is for acceptance, the corresponding active object notes $\perp_{CarHandler}$ as the currently processing object and as long as the predicate $stable(\perp_{CarHandler})$ is not evaluated to *true*, all possible actions of the state-machine of class *CarHandler* may be executed [7].

Whenever during the execution of actions the associations and attributes of "$\perp_{CarHandler}$" are evaluated, the value is in fact guessed, hence if there is a transition which for example sends an event back to an object of class *Car*, then the execution might choose any object of class *Car* in the system as destination (cf. Sec. 6).

Analogously, when a *Car* calls a triggered operation of $\perp_{CarHandler}$, the object reference $\perp_{CarHandler}$ is entered into the *Car*'s pending-request table as the receiver.

The transformed transition predicate again allows to execute arbitrary transitions or becoming stable. The pending-request table entry is then changed to '*completed*' and the caller continues. If there is a reply action on a transition of the callee, then the caller may find any possible value as the reply; otherwise the default will remain.

Note that the "other callee" in particular needs not become stable, thus we can observe any number of steps between the call of the triggered operation and its completion.

6   Interference and Non-Interference Lemmata

Fig. 5. False negative: When checking the property "two *Cars* will not crash when entering the *Terminal* since their *CarHandler*s set the switches safely", the heuristics of Section 5 yields the depicted system. "An other *CarHandler*" sending an event *arrivAck* to the concrete *Car* awaiting this event could cause the *Car* to enter the terminal although all platforms are already occupied.

The discussion of event sending at the end of the previous section already named a typical reason for false-negatives: "an other *CarHandler*" may send an event to a *Car* although it not *the CarHandler* known by the *Car* (cf. figure 5).

In the original system, only a single *CarHandler* actually knows a *Car* and sends events only when appropriate.

6.1   Non-interference Lemmata

In his SMV-tutorial [32], K.L. McMillan demonstrates how to address this problem by so called *non-interference lemmata*.

A non-interference lemma is a property of the following general form:

"*If some entity sends something to me, then it is allowed to do so*".

The lemma is verified in a separate proof and taken as an *assumption* in the proof of the main property to rule out unwanted interferences.

In general, it might be necessary to explicitly introduce *auxiliary variables* which keep track of "the ones" who are allowed to send.

But in UML models, a binary associations a with association ends e_1, e_2, each of multiplicity 0..1 or 1, are often intended to be "bi-directional"[5], i.e. the navigation forth and back along the association yields the identity: $self.e_1.e_2 = self$. If furthermore communication in the model is closely related to associations in

---

[5] although this interpretation is (reasonably) not enforced by the UML 2.0 proposals.

the sense that an event's destination is always given in terms of an association, and if the modeller provides annotations of all such associations with a set of signals, which are intended to be sent "along" this association, we can heuristically derive the non-interference lemma: sending an event to e_2 implies $e_2.e_1 = self$.
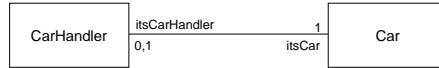

Fig. 6. Class diagram: Relation between *Car* and *CarHandler*.

In the above example, there exists only the single association between class *Car* and class *CarHandler* depicted in figure 6. Thus we would derive

$$\forall\ o_0 \in O_{CarHandler}\ \forall\ o_1 \in O_{Car}:$$
$$justsend.snd = o_0 \land justsend.dest = o_1 \implies o_0.itsCarHandler = o_1$$

which has to be proven separately. Note that again all symmetry reduction and abstraction techniques apply to this property.

7   Conclusion

We have provided a formal semantics for LSCs in the domain of UML in terms of the STS semantics of [7] and shown that LSCs in our interpretation are a prominent application domain for query-reduction, since UML models span an inherently symmetric state-space and LSCs are interpreted as quantifications over object identifiers, and provided yet missing proofs.

We formally described the abstraction technique of data-type reduction, discussed its advantages of an anticipated significant reduction of complexity and the possibility to prove properties without the need to provide finite bounds on the number of objects created during a run. The methodology applies in particular to parameterised systems. Its disadvantages comprise the extensive introduction of new free variables (inputs) and the introduction of false-negatives, which are possibly avoidable by automatically deriving non-interference lemmata from information in the UML model.

References

[1] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. *Formal Methods in System Design*, 9(1/2):77–104, August 1996.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

[3] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. A Language Framework for Expressing Checkable Properties of Dynamic Software. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, volume 1885 of *LNCS*, pages 205–223. Springer-Verlag, 2000.

[4] Gy. Csertn, G. Huszerl, I. Majzik, Zs. Pap, A. Pataricza, and D. Varr. Viatra - visual automated transformations for formal verification of uml models. In *Proceedings International Conference on Automated Software Engineering (ASE) 2002*, 2002.

[5] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):121–141, 2001.

[6] W. Damm and B. Jonsson. Eliminating Queues from RT UML Model Representations. In Damm and Olderog [8], pages 375–394.

[7] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In *Proceedings FMCO'02*, 2003.

[8] Werner Damm and Ernst-Rüdiger Olderog, editors. *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9-12, 2002, Proceedings*, volume 2469 of *LNCS*. Springer-Verlag, 2002.

[9] A. David, M. O. Möller, and W. Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE'2002)*, volume 2306 of *LNCS*, pages 218–232. Springer-Verlag, April 2002.

[10] E. A. Emerson and A. P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9(1/2):105–131, August 1996.

[11] M. Feather and M. Goedicke, editors. *Proceedings of ASE2001 (16th IEEE International Conference on Automated Software Engineering)*. IEEE CS Press, nov 2001.

[12] Alain Le Guennec. *Genie Logiciel et Methodes Formelles avec UML - Specification, Validation et Generation de Tests*. PhD thesis, Université de Rennes 1, 2001.

[13] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[14] D. Harel and R. Marelly. Specifying and executing behavioral requirements: The play-in/ play-out approach. Technical Report MCS01-15, The Weizmann Institute of Science, 2001.

[15] Heinrich Hussmann. Loose semantics for uml,ocl. In *Proceedings 6th World Conference on Integrated Design & Process Technology (IDPT 2002)*. Society for Design and Process Science, June 2002.

[16] Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In M. Feather and M. Goedicke, editors, *Proceedings of ASE-2001: The 16th IEEE Conference on Automated Software Engineering*. IEEE CS Press, nov 2001.

[17] Radu Iosif. Symmetry reduction criteria for software model checking. In D. Bosnacki and S. Leue, editors, *9th International SPIN Workshop, Grenoble, France, April 11-13, 2002. Proceedings*, number 2318 in LNCS, pages 22–41. Springer-Verlag, 2002.

[18] C. N. Ip and D. L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[19] C. N. Ip and D. L. Dill. Verifying Systems with Replicated Components in Murϕ. *Formal Methods in System Design*, 14(3):273–310, 1999.

[20] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, 1993.

[21] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, 1996.

[22] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, 1999.

[23] Kleppe and Warmer. Unification of static and dynamic semantics of uml. Technical report, Klasse Objecten, Soest, Netherlands, 2001.

[24] J. Klose. *Syntax and Semantics of Live Sequence Charts*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2003. To appear.

[25] J. Klose and B. Westphal. Relating LSC Specifications to UML Models. In Hartmut Ehrig and Martin Grosse-Rhode, editors, *Proceedings INT2002- International Workshop on Integration of Specification Techniques for Applications in Engineering*, April 2002.

[26] J. Klose and H. Wittke. An Automata Based Interpretation of Live Sequence Charts. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, number 2031 in LNCS, pages 512–527. Springer-Verlag, 2001.

[27] A. Knapp, S. Merz, and C. Rauh. Model Checking Timed UML State Machines and Collaborations. In Damm and Olderog [8], pages 395–416.

[28] D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioral Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[29] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[30] Rami Marelly, David Harel, and Hillel Kugler. Multiple instances and symbolic variables in executable sequence charts. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, November 4-8, 2002, Seattle, Washington, USA*, number 37(11) in SIGPLAN Notices, pages 83–100. ACM, November 2002.

[31] K. L. McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver*, number 1427 in LNCS, pages 110–121. Springer-Verlag, 1998.

[32] K. L. McMillan. Getting Started with SMV. Technical report, Cadence Berkeley Labs, March 1999.
`http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial.ps`.

[33] K. L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME'99*, number 1703 in LNCS, pages 219–233. Springer-Verlag, 1999.

[34] K. L. McMillan. A Methodology for Hardware Verification using Compositional Model Checking. *Science of Computer Programming*, 37:279–309, 2000.

[35] Ileana Ober. *Harmonizing Design Languages with Object-Oriented Extensions and an Executable Semantics*. PhD thesis, Institut National Polytechnique de Toulouse, April 2001.

[36] Ileana Ober. An asm semantics of uml derived from the meta-model and incorporating actions. In Egon Boerger, Angelo Gargantini, and Elvinia Riccobene, editors, *Abstract State Machines - Advances in Theory and Applications, 10th International Workshop, ASM 2003 Taormina, Italy, March 2003*, number 2589 in LNCS. Springer-Verlag, March 2003.

[37] Iulian Ober and Marius Bozga. Adapting and optimizing existing timed model checking tools to uml tools. Technical Report IST/33522/WP2.1/D2.1.2, Verimag, December 2002.

[38] OMG. OMG Unified Modeling Language Specification, September 2001. Version 1.4.

[39] I. Paltor and J. Lilius. Formalising uml state machines for model checking. In Robert B. France and Bernhard Rumpe, editors, *UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, volume 1723 of *LNCS*, pages 430–445. Springer-Verlag, 1999.

[40] T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3), 2001.

[41] W. Shen, K. Compton, and J. K. Huggins. A toolset for supporting uml static and dynamic model checking. In Feather and Goedicke [11], pages 315–318.

[42] B. Westphal. Exploiting Object Symmetry in Verification of UML-Designs. Master's thesis, Carl von Ossietzky Universität Oldenburg, April 2001.

[43] Xie, Levin, and Browne. Model Checking for an Executable Subset of UML. In Feather and Goedicke [11].

[44] F. Xie and J. Browne. Integrated State Space Reduction for Model Checking Executable Object-oriented Software System Designs. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2306 of *LNCS*. Springer-Verlag, 2002.