

LSC Verification for UML Models with Unbounded Creation and Destruction

Bernd Westphal^{1,2}

*Department für Informatik, Fk II,
Carl von Ossietzky Universität Oldenburg,
26129 Oldenburg, Germany*

Abstract

The approaches to automatic formal verification of UML models known up to now require a finite bound on the number of objects existing at each point in time. In [4] we have observed that the class of hardware systems with replicated components studied by McMillan [10] is equivalent to the class of systems with unbounded creation and destruction and all other data-types finite. Exploiting the symmetry of UML models induced by objects being instances of classes, the restriction to finite bounds can be overcome applying [10].

In this paper we report on experiences from an evaluation of this approach within the Rhapsody UML Verification Environment, a state-of-the-art tool for formal verification of UML models using Live Sequence Charts for requirements specification.

Key words: Formal Verification, Infinite-state, UML, LSC.

1 Introduction

Approaches to automatic formal verification of executable UML models range from early works considering only the communication behaviour of an isolated state machine in an open environment to newer works considering a collaboration of objects with fixed extension and topology, e.g. [8], i.e. they don't address dynamic creation and destruction of objects at all.

Newer tools for UML verification like ObjectCheck [13] and the Rhapsody UML Verification Environment (RUVE) [12] support a substantially larger subset of UML and in particular provide for dynamic creation and destruction of objects. But both tools require the user to provide an upper bound on the

¹ This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Centre "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

² Email: westphal@informatik.uni-oldenburg.de

number of objects alive at each point in time (and both flag an overflow if the model does not remain within these bounds during model-checking).

This restriction can be overcome by employing some of the techniques presented under the name of “Compositional Model Checking” in [10] as part of a larger framework since the class of models that these techniques applies to is actually equivalent to the class of models with dynamic creation and destruction of objects given all data-types of attributes have finite domain, as we have shown in [4]. The approach is based on the observation that UML models are typically symmetric in the type of object references (or pointers). As the requirements specification we consider the full Live Sequence Charts [2,7] with dynamic binding.

In this paper we report from first experiences of putting these results to practice by extending the state-of-the-art UML verification tool RUVE.

The remainder of the paper is structured as follows. Sec. 2 outlines the considered (subset of) UML and the underlying formal semantics. The foundations of the approach from [10] is recalled in Sec. 3 and Sec. 4 provides an overview of the RUVE tool. The main contribution is Sec. 5. It discusses practical problems of the transfer of the theoretical results to practice and provides results obtained for the running example. Sec. 6 concludes and points out further work.

2 UML Models and Live Sequence Charts

We consider the subset of UML as supported by the RUVE [12]. It basically comprises classes and their relations, i.e. inheritance, association, and aggregation, as given by class diagrams and the classes’ behaviour as defined by state machines.

Fig. 1 provides an example that lies in this subset of UML. It models a part of the approach and departure procedure of the Automated Rail Cars System (ARCS) [5]. The class diagram in Fig. 1(a) introduces four classes, the *ARCSystem*, the *Car*, the *Terminal*, and the *CarHandler*. The *ARCSystem* only serves as the owner of the *Cars* and *Terminals* in the system. It has no behaviour except for the creation of its parts in the initial step of the model.

The state machine of a *Car* is given in Fig. 1(c). A car is *cruising* until it receives an event *alert* from the environment which announces a terminal ahead. The event carries the terminal’s identity as a parameter. The car reacts by sending an *arrivReq* event to ask the terminal to reserve a platform and arrange the switches (both not part of the model).

A *Terminal* reacts on an *arrivReq* event by creating a new *CarHandler* and making this *CarHandler* acquainted with the car who sent the request by setting the *CarHandler*’s attribute *handledCar* (cf. Fig. 1(d)). From this point on, the transaction of arrival (and following departure) is completely controlled by the *CarHandler*. It is this object which actually communicates with the *Terminal* to set up the switches and afterwards sends an acknowledge

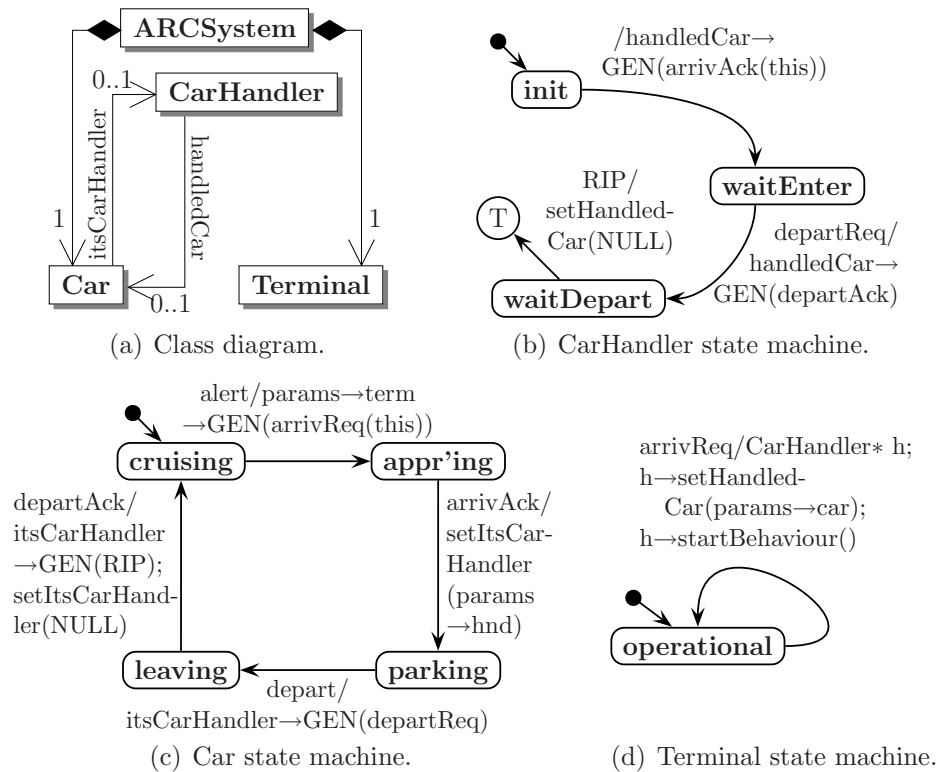


Fig. 1. Automated Rail Cars System class and state chart diagram.

to the *Car*. In the model, the *CarHandler* sends the acknowledge immediately, passing its own identity as a parameter of this event (cf. Fig. 1(b)).

On receiving the *arrivAck* event, the car stores the identity of the *CarHandler* responsible for itself and enters state *parking* (cf. Fig. 1(c)). Then the system is stable until the car receives an event *depart* from the environment which causes it to send a *departReq* to its *CarHandler*. If it has received the corresponding *departAck*, it sends an *RIP* event to its *CarHandler* causing the *CarHandler* to reach the termination connector, hence destroying itself.

More formally, a UML model is a finite set C of classes, each possibly equipped with attributes and a state machine without any do-actions, and a finite set E of events, some of them designated as *external*, i.e. they may be sent by the environment. Within a state machine, modification of attributes, creation and destruction of objects, and sending of events may be used as actions. Note that the relations association, aggregation, and even inheritance are attributes, their type is a set of references to objects [3].

The formal semantics of a UML model can be given as a Symbolic Transition System (STS) [9] over state variables with infinite domain [3]. A symbolic transition system (STS) is a triple (B, Θ, ρ) where $B = (V, \Omega)$ is a signature comprising a finite set V of (typed) variables and a set Ω of (typed) constants. Θ is a first-order predicate over B and ρ is a *transition predicate* over B , i.e. a first-order predicate over B extended by the variables from V in primed

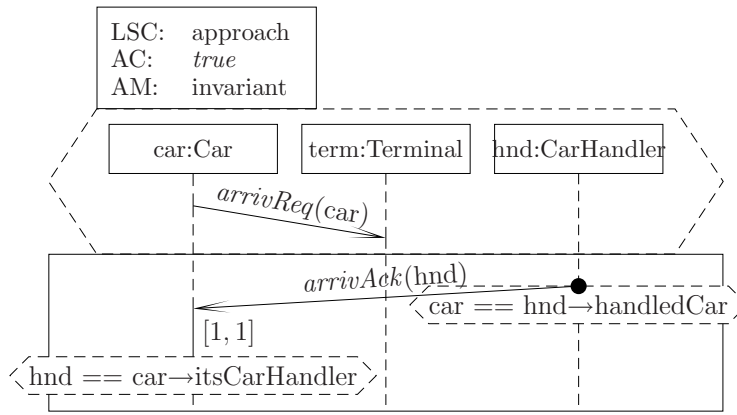


Fig. 2. **Live Sequence Chart.** If *car* contacts terminal *term*, then it is granted access by the *CarHandler* responsible for it and establishes the link *itsCarHandler*.

form. An infinite sequence $r = s_0 s_1 s_2 \dots$ of valuations of the variables in V is called *run* of S iff $\mathcal{M}[\Theta](s_0) = true$ and $\mathcal{M}[\rho](s_i, s_{i+1}) = true$ for all $i \in \mathbb{N}_0$, where s_i provides the valuation for unprimed and s_{i+1} provides the valuation for primed variables in ρ .

The STS for a UML model has one system variable in V for each class in C . The type of each of these system variables is an array over a designated index type, one for each class. The entries of the arrays are further structured and comprise fields for the classes’ attributes, the current state machine state, the FIFO to store events (may be shared between multiple objects), and a boolean field that indicates whether the object represented by a particular field is currently *alive*, i.e. has been created but not yet destroyed again. The domain of the index types is chosen countably infinite thereby representing general UML models with unbounded creation and destruction of objects.

As specification language for requirements on the model we use Live Sequence Charts (LSCs) [2,7]. As a conservative extension of Message Sequence Charts and UML Sequence Diagrams and equipped with a formal semantics they are a natural choice to formalise requirements on the *inter-object* behaviour in the terms of a UML model.

Figure 2 shows the LSC requirement ‘approach’ on the ARCS model. The LSC header identifies it as invariant with activation condition *true* and the solid border around the main chart indicates that it is universal. Thus a system only satisfies the LSC if in each run of the system, whenever the pre-chart (the part inside the large, dashed hexagon) is observed from any snapshot of the run on, then the remainder of the run adheres to the main-chart. In the example, the pre-chart comprises only the communication between an arbitrary *Car car* and an arbitrary *Terminal term*. The main-chart requires that if any *CarHandler hnd* sends an acknowledge to *car*, then it is the one responsible for *car* and that *car* remembers this *CarHandler* in the next snapshot after receiving the acknowledge.

For a more thorough introduction of LSCs the reader is referred to the literature [2,7]. With Car , $Terminal$, and $CarHandler$ denoting the sets of object identities, the semantics of the example is structurally of the form

$$\begin{aligned} &\forall car \in Car, term \in Terminal, hnd \in CarHandler \\ &\forall r \in runs(S) \forall i \in \mathbb{N}_0, j \geq i : r^i \models ac \wedge r^i..r^j \models pre-chart(car, term, hnd) \\ &\implies r/j \models main-chart(car, term, hnd) \end{aligned}$$

where r^i denotes the i -th snapshot of the run r , $r^i..r^j$ the sequence of snapshots from r^i to r^j , and r/j the suffix of r starting with r^j . Read out, it requires that for all $Cars$, $Terminals$, and $CarHandlers$ identities and for all system runs r , if the i -th snapshot satisfies the activation condition and the sequence of snapshots from r^i to r^j satisfies the pre-chart, then the suffix starting with the j -th snapshot satisfies the main chart.

Note that the quantification of objects appears *outermost*, this fact is essential for the applicability of query reduction [10] (cf. Sec. 3).

3 Query Reduction and Data-Type Reduction

The query reduction technique reduces quantified verification tasks to small finite representative sets [10] if the model is symmetric in the quantified variable's type. For example, verifying that all $Cars$, $Terminals$, and $CarHandlers$ in the running example adhere to the LSC 'approach' requires to establish this property for the infinitely many $CarHandlers$ who may be created during a run of the system. For the example it is in fact sufficient to establish only a single case, e.g. $\{car \mapsto 1, term \mapsto 1, hnd \mapsto 1\}$, since the system is symmetric in the identities of $Cars$, $Terminals$, and $CarHandlers$. If the model fails to satisfy the requirements for the chosen case, then we obtain by permutation of identities in the counter-example a counter-example for any other case.

More formally, we exploit that each permutation $\pi(s) := \{x \mapsto \{i \mapsto s(x)(\pi_0(i)) \mid i \in \tau_x\} \mid x \in V\}$ induced by a permutation π_0 on the semantic domains of the object identity types τ_x is an automorphism of the STS, that is, it has no effect on the evaluation of the initial state and transition predicates, $\mathcal{M}[\Theta](s) \iff \mathcal{M}[\Theta](\pi(s))$ and $\mathcal{M}[\rho](s, s') \iff \mathcal{M}[\rho](\pi(s), \pi(s'))$ for all valuations s, s' . Designated object identities, like NULL for an uninitialised reference, can be supported by considering only those permutations which map each designated identity to itself [4].

The problem to decide which types used by an STS are symmetric can be reduced to a type-checking problem on the STS' predicates. Each type adhering to the rules given in Fig. 3 is a scalarset [6], where the last rule is relaxed to support designated identities. Given a finite requirements specification outermost quantified over scalarsets (and not violating the rules of Fig. 3 either), there is a finite representative set of cases which implies the complete specification [10,4].

- (i) two variables of type τ may be *compared for equality*
- (ii) a variables of type τ may be *assigned* an expression of type τ
- (iii) τ may be the *index type* of arrays,
- (iv) a variable of type τ may be the *running index of a loop*, if the outcome of the loop is independent
- (v) nothing else, in particular *no literal values* of type τ , e.g. no explicit reference to *InLink* with identity ‘3’

Fig. 3. **Syntactical criteria** for symmetric data-types.

$$\begin{aligned}
 == (x_1, x_2) &= \begin{cases} x_1 == x_2 & , \text{ if } x_1 \neq \perp \vee x_2 \neq \perp \\ ?_{\mathbb{B}} & , \text{ otherwise} \end{cases} & a[x] &= \begin{cases} a[x] & , \text{ if } x \neq \perp \\ ?_{\tau_0} & , \text{ if } x = \perp \end{cases} \\
 \text{(a) Comparison for equality.} & & \text{(b) Array access.} &
 \end{aligned}$$

Fig. 4. **Abstract Interpretation.** Comparing \perp with itself yields $?_{\mathbb{B}}$, the least upper-bound of 0 and 1; accessing an array $a : \tau \rightarrow \tau_0$ at position \perp yields \perp_{τ_0} the least upper-bound of the set τ_0 .

Considering the concrete cases may render the cone-of-influence abstraction more effective, but in general the model remains infinite-state. In [10], McMillan proposes to apply the abstraction *data type reduction* to obtain a finite over-approximation of the original model. Strictly speaking it does not depend on symmetry in the model, but happens to be defined just for the legal operators on scalarset types, hence is usually applied to symmetric systems.

Data-type reduction is basically an abstract interpretation in the sense that concrete operators are given an interpretation on an abstract domain. The abstract domain of a scalarset type τ with domain $\mathcal{D} = \{d_1, d_2, \dots\}$ is of the form $\{\{d_1\}, \{d_2\}, \dots, \{d_n\}, \perp\}$, i.e. a set of singletons and their complement $\perp_{\mathcal{D}} := \mathcal{D} \setminus \{d_1, \dots, d_n\}$, abbreviated $\{d_1, \dots, d_n, \perp_{\mathcal{D}}\}$. For scalarsets only the comparison, assignment, array access operators, and loops have to be considered. The interpretation of comparison and array (read) access are shown in Fig. 4. Modifying an array with a scalarset index type does not change for indices different from $\perp_{\mathcal{D}}$ and for $\perp_{\mathcal{D}}$ the array is not changed at all since the next read access yields $?_{\tau_0}$, i.e. an upper-bound on all values of the array’s value type. A loop iteration over a scalarset only considers the concrete values (the singletons) and a commutative boolean operator applied to, e.g. an array indexed by a scalarset with boolean value type, considers the concrete values and once $?_{\mathbb{B}}$ over-approximating the result for the other values. We obtain an over-approximation of the original model, i.e. if the requirement holds for the abstract model, then it also holds for the concrete model, but not vice versa. All variables of type τ become finite, of arrays with index-type τ only the first n entries have to be represented.

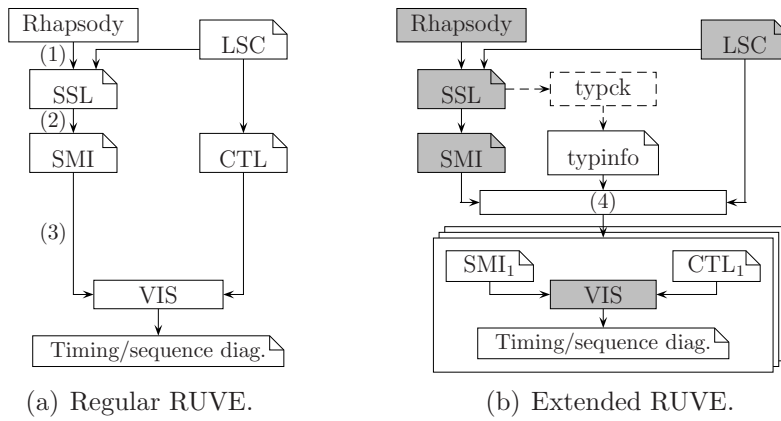


Fig. 5. **RUVE architecture.** Arrows represent data-flows, typically through multiple tools or stages. Only the most important tools are shown.

4 The Rhapsody UML Verification Environment

To assess the practical applicability of query and data-type reduction to UML models we integrated both into a state-of-the-art UML verification tool. The choice fell on RUVE [12] that stands in a history of increasingly sophisticated tools for automated formal verification of UML models. The aim of RUVE is to further the supported subset of UML, to be fully integrated into a schematic entry tool, and to provide requirements specification and counter-example presentation in terms of the UML model, not in that of the underlying tools like model-checkers. To this end it has exemplarily been integrated with “Rhapsody in C++” by i-Logix Inc.

In the following we briefly sketch the architecture of RUVE to prepare the discussion of the integration of query and data-type reduction in Sec. 5. For a more comprehensive description of RUVE the reader is referred to [12,11]. The approach of RUVE is translational, i.e. the UML model is translated into an equivalent model in the input language of a model-checker. The class structure is obtained from an API of the Rhapsody tool or from an XMI representation of the model. The UML model’s behaviour is obtained in form of C++ code from the Rhapsody code-generator thereby ensuring that counter-examples can be retraced using an animation of the UML model. Class structure and behaviour are translated into the proprietary high-level intermediate language SSL (cf. Fig. 5(a).(1)) and successively transformed into an SSL representation without scopes, functions, or classes which translates directly into another proprietary intermediate language, SMI, that is basically Dijkstra’s guarded commands with rich type system and expression language (cf. Fig. 5(a).(2)). SMI is translated to a VIS model-checker [1] input (cf. Fig. 5(a).(3)). The approach of RUVE is specification driven since C++ expressions from the specification, e.g. conditions in an LSC, are processed as parts of the model. As Fig. 5(a) indicates, the LSC is finally translated into a CTL expression.

An obtained errorpath is translated back into a Timing Diagram showing

the values of all objects' attributes over time and a Sequence Diagram showing the event communication, both on the level of the UML model.

5 Putting it all together

In [4] we observed that query and data-type reduction apply to UML models with unbounded object creation and destruction and LSCs. High-level UML models typically don't employ pointer arithmetics but only assign and compare identities and use them to access objects' attributes. Object identities are scalarsets then. LSCs are outermost quantified in object identities.

To study the practicability of this approach we prototypically integrated query and data-type reduction into the RUVE. As C++, the action language of RUVE, allows pointer arithmetics and comparison of pointers to literal numbers (both not supported by RUVE), we cannot prove once and for all that all object identities are scalarsets but have to determine the set of scalarsets ('typinfo' in Fig. 5(b)). The dashed line in Fig. 5(b) indicates that this check should be applied to a high-level representation. The prototype only checks the lower-level SMI representation and considers manually supplied hints.

As indicated in Fig. 5(b), we add to RUVE a component that determines a minimal finite representative set of verification tasks for a given LSC, i.e. a concrete valuation of the instance annotations. The component generates an initial abstraction *per task*, and controls the execution of the tasks, possibly in parallel since the tasks are completely independent. The set of representative tasks derives directly from the instance line annotation in the LSC, in the running example a single task is representative.

The initial data-type reduction for a verification task is (automatically) determined by the following *heuristics*. For each class, all identities not assigned to an instance line are collapsed to \perp , i.e. only the objects participating in the task are represented concretely. Thus there is not a single concrete object in the abstraction for classes not referred to in the requirements specification. If object identities are the only infinite data-type in the UML model, i.e. if all queues and relations are bounded and all attribute types (except for object references) are finite (prerequisite to apply RUVE), the resulting abstraction is finite-state and over-approximates the UML model. If the abstraction is too coarse, i.e. yields a counter-example not possible in the concrete UML model, it can be refined following two *strategies*. First, the number of concrete objects can be increased. Candidates are objects of classes being in composition relation with the classes referred to in the requirements specification. Second, the abstraction can be refined more selectively by using assumptions that exclude certain behaviour of the abstract model (called non-interference lemmata in [10]). They have to be established separately but are typically properties that are local to classes.

As the model-checker of RUVE, the VIS, does not support data-type reduction natively, the SMI description of the model is transformed into an SMI

equivalent to the abstract model. Consider the running example and assume we choose to use the instance $\{car \mapsto 1_c, term \mapsto 1_t, hnd \mapsto 1_h\}$ as the representative case (we write 1_c , 1_t , and 1_h to indicate values of different types). The heuristics yields the abstract domains $\{\{1_h\}, \perp_h\}$ for *CarHandlers*, $\{\{1_c\}, \perp_c\}$ for *Cars*, and $\{\{1_t\}, \perp_t\}$ for *Terminals*. In the SMI representation of the abstract model, each comparison of references to, e.g., *Cars* of the form ‘ $p == q$ ’ have been replaced by the statement ‘ $(p == \perp_c \wedge q == \perp_c ? i_B : p == q)$ ’ where ‘ \perp_c ’ is an identity different from the singletons in the abstract domain, e.g. $\perp_c = 2_c$, and ‘ i_B ’ is a fresh boolean input. Thus if p and q do *not* refer to one of the concrete objects, as indicated by them having value \perp_c , the model non-deterministically considers both possible outcomes of the comparison.

Array access is treated similarly to obtain the interpretation shown in Fig. 4(b). In a naïve approach, each expression of the form ‘ $o[p].x$ ’ would be replaced by ‘ $(p == \perp_o ? i_\tau : o[p]).x$ ’ where i_τ is a fresh input of the field type of array ‘ o ’. That is, talking about UML models a *whole* object with values for all attributes becomes an input. This domain for the input is typically far too large since array access in the SMI representations of UML models typically occurs as subexpression of a record field access. Back to UML models, the intuition is that objects are typically not accessed as a whole but only by one attribute at a time. Hence the type of the fresh input is better chosen according to the maximal expression influenced by the object identity; in the example this is the type of the record component ‘ x ’. The prototype already considers attribute access but does not yet determine maximal expressions. The number of inputs can be further reduced using common subexpressions. Note that these replacements allows to use standard tools like VIS as long as resulting counter examples are interpreted correctly, i.e. being aware that a particular identity has been chosen to represent \perp_c .

The integration into RUVE is not straightforward since RUVE already exploits particularities of UML models to reduce model-checking time [12] in a way that breaks symmetry. Firstly, RUVE *crystallises* relations as far as possible, for example it establishes a fixed relation between the parts of a composition relation and the whole whereby the actual references become constant. This is also justified by symmetry, the crystallised model is no longer symmetric in object identities. Secondly, RUVE pre-computes the initial step. It is deterministic for the UML models considered by RUVE but not for the abstract model. In the example considered for this paper we established manually that our premises still apply. In general the former obstacle should be treatable by casting the crystallisation as a *case-split* [10], leading to a larger set of representative tasks. The latter obstacle is overcome by pre-computing all possible initial steps, yielding multiple verification tasks per abstract model.

Figure 6 shows verification times³ for the running example. The first line includes determining and executing the single representative task; the

³ Sun Blade 2000, 900 MHz UltraSparc III+, 2 GByte.

model	inst.	model-gen.	model-checking	errorpath prep.
ARCSsystem	DTR	0:02:18	4:45:50	(prop. holds)
	Witness	0:02:12	0:00:06	0:00:05

Fig. 6. **Verification times.** Model-generation, -checking, and translation of the counter-example back to UML terms (hours:minutes:seconds).

property holds in the initial abstraction. The second line, for comparison, refers to verification that there exists a run of the (not abstracted) ARCS which satisfies the LSC. The reason for the long verification times (nearly five hours) is not completely understood. We conjecture that the main source are the FIFO queues used for communication even when reduced to small bounds. Furthermore the prototype is suboptimal in the introduction of inputs as noted in Sec. 5.

6 Conclusion and Further Work

We have reported experiences from a practical evaluation of the query and data-type reduction-based [10] approach [4] to automated verification of UML models with unbounded creation and destruction. To this end a finite abstraction is automatically derived from the model guided by the requirements specification and manual hints for refinement.

An evaluation of the heuristics to determine initial and refined abstractions and the strategies to obtain non-interference lemmata is further work. We conjecture that there are cases where unbounded associations are also treatable with our approach. Viewing them as arrays, the index types are also scalarsets as long as iteration over all objects in the association does not have countable effects like sending an event to all objects in the association.

References

- [1] Brayton, R. K., G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy and T. Villa, *Vis: A system for verification and synthesis.*, in: R. Alur and T. A. Henzinger, editors, *CAV*, Lecture Notes in Computer Science **1102** (1996), pp. 428–432.
- [2] Damm, W. and D. Harel, *LSCs: Breathing Life into Message Sequence Charts*, *Formal Methods in System Design* **19** (2001), pp. 45–80.
- [3] Damm, W., B. Josko, A. Pnueli and A. Votintseva, *A discrete-time UML semantics for concurrency and communication in safety-critical applications*, *Science of Computer Programming* **55** (2005), pp. 81–115.
- [4] Damm, W. and B. Westphal, *Live and let die: LSC-based verification of UML-models*, *Science of of Computer Programming* **55** (2005), pp. 117–159.

- [5] Harel, D. and E. Gery, *Executable object modeling with statecharts*, IEEE Computer **30** (1997), pp. 31–42.
- [6] Ip, C. N. and D. L. Dill, *Better verification through symmetry*, Formal Methods in System Design **9** (1996), pp. 41–75.
- [7] Klose, J., “Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior,” Ph.D. thesis, C. v. O. Universität Oldenburg (2003).
- [8] Lilius, J. and I. Porres, *vUML: a tool for verifying UML models*, in: *Proceedings ASE’99* (1999), pp. 255–258.
- [9] Manna, Z. and A. Pnueli, “The Temporal Logic of Reactive and Concurrent Systems: Specification,” Springer-Verlag, New York, 1991.
- [10] McMillan, K. L., *A methodology for hardware verification using compositional model checking*, Science of Computer Programming **37** (2000), pp. 279–309.
- [11] OFFIS, “UVE User Documentation (Guide, Tutorial, Restrictions),” (2004).
URL <http://www-omega.imag.fr/tools/UVE/UVE.php>
- [12] Schinz, I., T. Toben, C. Mrugalla and B. Westphal, *The Rhapsody UML Verification Environment*, in: *Proceedings SEFM 2004* (2004), pp. 174–183.
- [13] Xie, F., “Integration of Model Checking into Software Development Processes,” Ph.D. thesis, The University of Texas at Austin (2004), UTCS TR-04-29.