Electronic Notes in Theoretical Computer Science 44 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume44.html 13 pages

Formal concepts for an integrated internal model of the UML

Institut für Kommunikations- und Softwaretechnik
TU Berlin, FB13, Sekr. FR 6–1, Franklinstr. 28/29
10587 Berlin, Germany

Martin Große–Rhode [1,2]

Abstract

In the Unified Modeling Language (UML) different views of software systems are specified by different models. The abstract syntax of the modelling languages is defined precisely in the UML standard, but the (dynamic) semantics up to now are only sketched in natural language descriptions. Moreover, the correspondences between the different models are not described precisely. In this paper I present an abstract semantic domain that has been defined independently of the UML and can be used to provide formal semantics for the different modelling languages. Since one common domain is employed also the integration of the different viewpoint models is supported by this approach.

1 Introduction

With its different modelling techniques the Unified Modeling Language (UML, see [13]) realizes two fundamental features of rational software systems development. Firstly a *model based development* is supported in that abstract models of systems and their constituent parts can be built and maintained as documents throughout the life cycle of the system. Secondly, separating concerns, different models are used to specify selected *viewpoints* of the system, where a single viewpoint model only yields a partial specification. The main views in the design stage of a system's development process are the ones on the static structure and the dynamic behaviour of the system. The former comprises for instance the structure of objects and the (class) architecture of the system, whereas the latter can be subdivided into functionality, use case scenarios, interactions, protocols, intra–object behaviour etc. The separation into different viewpoint models, however, immediately prompts the question

[1] `http://tfs.cs.tu-berlin.de/∼mgr`
[2] Email:`mgr@cs.tu-berlin.de`

for their integration to a (virtual) global model of the whole system. That means, it must be clarified how the collection of models can be considered as one specification of one system.

In the UML standard the abstract syntax of the language(s) is defined precisely using the meta model. The semantics, however, in particular the dynamic semantics of the behaviour models, are only informally described in natural language. Obviously, this leads to a certain lack of precision, and some important design decision are left open. Furthermore, the are no precise statements about the semantic interrelations and correspondences between the different models. That means the integration problem cannot be addressed satisfactorily yet.

In this paper an approach to the formalization and integration of UML models is sketched that is based on an internal model. That means, instead of exploiting the meta model—which is itself formulated using UML class diagrams—an independent semantic domain is used that serves to interpret the UML constructs. The semantic domain is given by *transformation systems* as formal models of single units (like objects for example), representing static structure and dynamic behaviour in one integrated mathematical structure. Transformation systems are extended transition systems, where states are labelled by structures representing the internal data state of a system, and transitions are labelled by sets of actions or other expressions. General schemes for composition operations and development relations are defined for transformation systems to represent different kinds of concrete composition operations and development and refinement relations for semantic models. Due to the formal mathematical approach general compositionality properties can be formulated and proven on a very general level. The domain of transformation systems has been developed and applied to formal specification techniques in [6,7]. The purpose of this paper to sketch the application of this approach to the integration of UML models. Note, however, that this is a report on work in progress; especially the formal semantics of the dynamic models have not yet been given completely, whence only the basic ideas are presented here.

In order to establish transformation systems as an internal model of the UML the different modelling techniques have to be mapped to this domain, i.e., their semantics have to be defined in terms of transformation systems. Thereby the meaning of the constructs can be made precise, and the correspondences between the different techniques can be formulated, since all languages are interpreted in the same domain. Due to the complexity of the UML languages these mappings should be defined incrementally of course, starting with kernel languages that are extended step by step.

As mentioned above the single viewpoint models only yield partial specifications of the system. Moreover, the semantics of certain UML constructs may be left open for different context-specific interpretations. The basic idea to deal with this partiality and under-specification here is to define the formal semantics of a UML model as a set of transformation systems, that contains all admissible interpretations as elements. Furthermore, different models may address different levels of abstraction and granularity. For example, some models specify single objects whereas others concern collections of objects. Thus mappings of transformation systems are needed, like projections, that allow us to relate such different models and formalize their correspondences. Sets of transformation systems as formal semantics of viewpoint models, together with the appropriate mappings reflecting their correspondences, then allow us to define the semantic integration of a collection of models by the intersection of their admissible interpretations. A collection of models is consistent if the intersection is not empty, and it is complete if it has exactly one element, i.e., if all models together specify the system completely.

As mentioned above the domain of transformation systems beyond the single models also provides composition operations and development relations. As an added value of the approach presented here these can be reflected to the UML languages to discuss and make precise notions of object composition and refinement for example. This aspect, however, will not be discussed further in this paper.

2 Transformation Systems

Transformation systems are extended labelled transition systems, where both the transitions and the states are labelled. The unlabelled part of the system, given by a set of *control states* and *transitions*, is just a directed graph , called the *transition graph* of the system. It represents the skeleton of the dynamic behaviour, by stating the existence of control states and transitions, without showing their internal structure or contents. The labels for the transition graph are provided by the *data space* of the system, that contains all possible *data states* and *data state transformations* according to the given static structure.

To define the latter a *data space signature* $D\Sigma = (\Sigma, A)$ needs to be given, where $\Sigma = (S, F)$ is an algebraic signature of sorts names S and function names F respectively, and A is a family of sets action names, indexed by the lists of their parameter sorts. Each (partial) Σ–algebra is then considered as a possible data state of the system. For an object with attributes a_i of types s_i $(i = 1, \ldots, n)$ for example the algebraic signature Σ contains the sort names $s_1, \ldots, s_n$ and the constant function names $a_1 : s_1, \ldots, a_n : s_n$. A partial algebra A of this signature is given by a family of sets $A_{s_1}, \ldots, A_{s_n}$ (which will often be the same in all states) and, for each $a_i : s_i$ a designated element $a_i^A \in A_{s_i}$ representing the actual value of the attribute a_i. Partial algebras allows us for instance to leave attributes undefined in some object states and to use partial functions and predicates in the data types. Mutable carrier sets allow the representation of creation and deletion of elements.

The data state transformations $T : C \Rightarrow D$ are given by pairs of data states (= algebras), the input state C and the output state D, and action sets T, where each action $a(p_1, \ldots, p_n) \in T$ is given by an action name a and a list of actual parameters $(p_1, \ldots, p_n)$ from the input state C that conform to the arity of a. In the object example the action names are the methods, and the actions are method invocations with concrete parameters. Using sets of actions with more than one element allows us to model parallel executions of actions; the empty action set in a transformation represents an internal, invisible action.

Taking all partial Σ–algebras as data states and for each pair of data states all action sets as transformations yields the space of all syntactically possible interpretations. Of course, this is much larger than the intended data space. The selection of the data states and transformations that are actually realized by a system is obtained by defining a specific transition graph together with a mapping into the data space. On the one hand this defines the labels of the transition graph (i.e., the extended transition system), on the other hand it defines the subset of the data space used by this system, as the image of the mapping. To conclude, a $D\Sigma$–transformation system (of a given a data space signature $D\Sigma$) is given by a transition graph together with a mapping to the data space induced by $D\Sigma$.

In [8] a generalization of the definition of transformation systems is given that allows us to replace partial algebras and action sets by other models for data states and transformation respectively. For example, single algebras can be replaced by indexed families of algebras to represent data states of collections of objects, where each individual state is given by a single algebra and object links are represented by references. Furthermore action sets can be replaced by other structures on actions, such as formal strings of actions to represent the sequential execution of actions within a single step, as in state charts for example.

3 Class Diagrams

A class diagram defines the static structure of a software system, but (usually) does not contain information or constraints concerning the dynamic behaviour of the objects of the class. Concerning the semantic interpretation in terms of transformation systems this means that a class diagram only yields data space signatures, but no constraints on the transition graphs.

The static structure covers the static structure of the objects of the class, i.e., the number and types of their attributes and methods, and the relations (associations, generalizations etc.) between the classes. Consider first a single class, as shown for example in Figure 1. The corresponding data space signature $\Sigma Person$ can be derived immediately from the class presentation. Its sort names are given by the types that appear in the class, the methods yield the action names and their parameter sort lists, and the attributes yield the (constant) function names. (Note that using algebraic signatures also param-

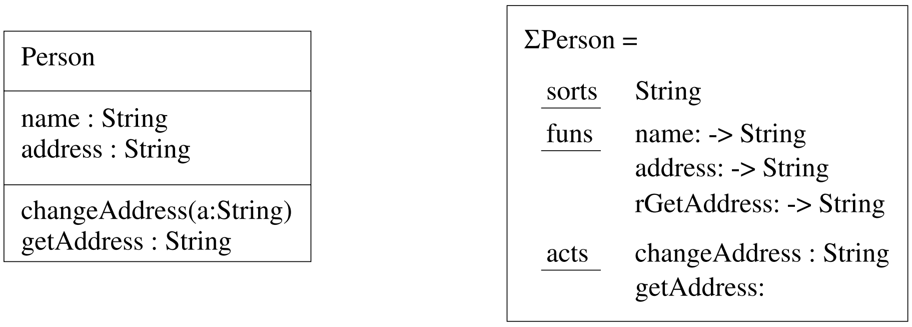

Fig. 1. Class *Person* and its induced data space signature $\Sigma Person$

eterized attributes are supported, as advocated for in [1].) Furthermore, for each method m that has a return type s a function $r\_m : s_1, \ldots, s_k \to s$ is added, where $s_1, \ldots, s_k$ is the list of parameter types of m. In a transformation $\{m(p_1, \ldots, p_k)\} : C \Rightarrow D$ the return value of m will then be bound to the function value $r\_m^D(p_1, \ldots, p_k)$ in the output state D. Each partial algebra of the signature $\Sigma Person$ then represents a state of a *Person*–object.

A system snapshot, given by the states of a collection of objects and their links, can be modelled by replacing algebras as data states by families of algebras, where each member represents the data state of one of the objects. In this way the projection to a single object is supported, and collections need not be encoded into single algebras artificially. Then object references can be represented by introducing designated *class sorts* and additional *reference* functions that map elements of class sorts (= object references) to members of the family of algebras (= object states). A state of a system is thus formally modelled by

- an index set I of abstract identifiers of the currently existing objects,
- a family $\mathcal{C} = (C_i)_{i \in I}$ of algebras as object states, and
- for each $i \in I$ and class sort cs a partial function $ref_{i,cs} : (C_i)_{cs} \to I$.

Consider for example the extension of the class *Person* by an attribute $marriedWith :\to Person$ that holds a reference to another *Person*–object, if any. A system snapshot is shown in Figure 2, where the class sort is *Person*, the person called *'Richard'* has three references to Persons, R, S and M, and *'Sabine'* has two references. The two reference functions are defined by $ref_0(R) = 0, ref_0(S) = 1, ref_1(R) = 0$, and $ref_1(S) = 1$. Thus in particular, *'Richard'*s reference M cannot be dereferenced in this state. To navigate in an object configuration the usual algebraic definition of terms is extended by a dot notation $o.t$, where o is a term of a class sort and t is any term. A dot term $o.t$ is evaluated in a snapshot, starting at a designated algebra C_0, as follows. Evaluation of o yields a reference $r = o^{C_0}$ in a class sort of C_0. Applying ref_0 yields the index $j = ref_0(r)$ of the algebra C_j, where finally t is evaluated. The term *marriedWith.name* evaluated at object C_0 for example yields $'Sabine'$, since $marriedWith^{C_0} = S$, $ref_0(S) = 1$, and $name^{C_1} = 'Sabine'$. (The concepts

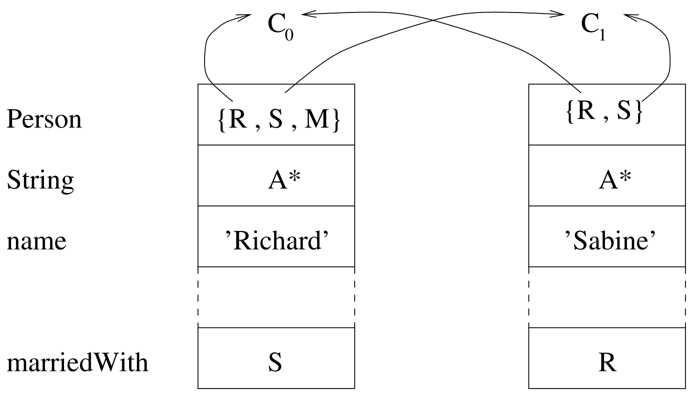

Fig. 2. State of an object configuration of the extended *Person* class

of snapshots, object states and references are worked out precisely in [11].)

A class diagram can now be translated into a diagram of data space signatures. Each association thereby yields a relation of the corresponding class data space signatures, represented by an additional association signature and a span of signature morphisms. Moreover, the local class data space signatures are extended by constants $role :\to set(c)$, given by the role name of the associated class. Since several objects may be linked its type is set of elements of the corresponding class sort. The relation models the links as instances of the association in the actual state, whereas the constants make possible to navigate along the association. They can be restricted accordingly to represent uni-directional associations and multiplicities. Compositions and aggregations are represented analogously; concerning the static viewpoint they are just associations.

Finally inheritance relations give rise to extensions of the local class data space signatures, where the signature corresponding to the super class is added to the one of the subclass. Note that the extensions of the local class data space signatures w.r.t. the relations must proceed in the right order. For example, associations are inherited, whence the association attribute *role* has to be added before the inheritance relation is translated. More details on the translation from class diagrams to diagrams of transformation specifications can be found in [12].

4 Specification of Properties of Transformation Systems

As mentioned above a data space usually contains much more data states and transformations than intended. Although finally the mapping from the transition graph selects the right parts, the data space can also be reduced in advance by adding further constraints or axioms to the data space signature. According to the structure of transformation systems and data spaces (and signatures) they can be grouped as follows.

Static properties

The set of Σ–algebras as data states can be constrained by equations, or more general logic formulas. They may be used to define data invariants, derived static functions of the built–in or defined static data types, or derived attributes. Equations are the canonical choice when using partial algebras. The extension mechanism for transformation systems, however, also allows the usage of other kinds of formulas. Higher order constraints for example can be used to require certain parts of the data states to have a fixed immutable interpretation (like built–ins) or to represent class cardinalities. The static properties reduce the data space in that only the algebras that satisfy the equations and other formulas are taken as data states.

Pre and post conditions

Data state transformations have been defined as pairs of data states together with an action set. According to this structure they can be specified by *transformation rules* that consist of two sets of equations L and R for the specification of the input and the output state of the transformation respectively, and a formal action expression. Then for a given action the sets L and R specify the pre and post condition respectively. This reduces the transformations of the data space to the ones satisfying the pre and post conditions. Note that transformation rules only constrain the transformational behaviour of methods/actions in single steps. In addition, a precondition contains a global behaviour condition: *whenever* it is satisfied the method must be applicable. This aspect has to be translated into a formalization of global behaviour properties.

Further dynamic properties

Pre and post conditions constrain single transformation steps. In order to constrain the set of transition graphs, i.e., the overall behaviour, other specification means have to be considered. As opposed to the static properties and pre and post conditions, there is no canonical choice in this case. In [8] the corresponding specification means are treated generically, including both descriptive (logical) and constructive (automata etc.) techniques. For the application to UML models concrete instances given by state charts and sequence diagrams for the specification of dynamic behaviour are sketched below.

The discussion of the specification means should have shown the intended application to the UML already. In a UML model object constraints specified in the object constraints language OCL [14] are used to reduce the set of data states and transformations. The most important features, data invariants, navigation, and pre and post conditions have been discussed above, indicating the intended mapping of the OCL. Other behaviour specifications are specified next.

5 State Machine Diagrams

State machine diagrams are one of the UML modeling techniques for dynamic behaviour, specifying the behaviour of a single object of a given class. In principle a state machine is a constructive model, i.e., it defines one fixed behaviour schema. However, usually the concrete information on the semantics of the actions is missing. Furthermore, the behaviour of other objects (of other classes) may have an impact on the behaviour when their methods are invoked. Whereas the latter aspect can be handled as a local non–determinism, the first aspect is more adequately represented via a set of admissible interpretations as in the case of class diagrams. The reason is again that the state machine is just a partial (viewpoint) specification.

In [5] a transition system semantics for UML state machines has been defined, based on a corresponding construction in [4] for STATEMATE state charts. Since it is very close to the aspired transformation system semantics it only remains to rephrase this construction w.r.t. the terminology of transformation systems and to point out which items are under-specified and lead to sets of admissible interpretations. The main difference with [5] is in fact that the action semantics are assumed to be specified completely there, whereas in the transformation system approach all admissible interpretations of the given information are considered.

The transformation systems representing the formal semantics of a UML state machine are constructed as follows. The *control states* of a transition graph consist of 1. a configuration, i.e., a maximal consistent set of state machine states, 2. history information, which can be modelled by redefining the *initial-substate* function, 3. the set of events that still have to be consumed, 4. global clock and local timer states (see [5] for details). Furthermore, since a unique data state has to be attached to the control state, sufficient data state information has to be added which the complete data state can be recovered from. This additional data information is represented in the formal object model introduced in [15] for example by taking the cartesian product of (pure) control states and data states. Considering the corresponding pairs immediately as control states is just a formal variant that is needed here to obtain the label mappings as total functions.

To define the *data states* the attribute declaration of the corresponding class, including the concerned data types, has to be given at least. (In fact, state machines cannot be built without a corresponding class.) It yields the algebraic part of the data space signature. The corresponding algebras may have a fixed part for the static data types. The values of the independent attributes in an algebra representing a data state must be obtained from the control state it is attached to. That means, these values must be contained in the control state. Further derived parts, like derived attributes, can then be constructed to obtain the complete data state. Different interpretations are admissible at this point if static data types are declared but not completely

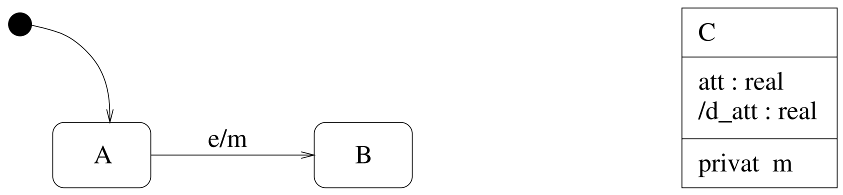

Fig. 3. A very simple state machine

defined, or if attributes are declared as derived ones, but the dependency from the other attributes is not (fully) specified.

Consider for example the more or less trivial state machine shown in Figure 3. It does not pose problems w.r.t. its behavioural semantics, but nevertheless does not yield a single transformation system, since the structural information given in the corresponding class is incomplete. The type *real* could be interpreted as the set $\mathbb{R}$ of real numbers, or a finite interval of real numbers, or double precision reals, etc. It might be resolved by fixing one interpretation for reals once, but with user defined types several interpretations become possible anyway. Secondly, if the value of the attribute *att* is given the value of the derived attribute *d_att* is determined, but the class does not provide the relation between *att* and *d_att*. So yet all interpretations are admissible and yield different mappings of the control states A and B (that happen to coincide with the state machine's state here) to data states.

The *transitions* for the transition graph are given by sets of state machine transitions that are enabled (w.r.t. a given event e and a condition), consistent, maximal, and have highest possible priority (see [5,9]). Their input and output control states in the transition graph are given as defined in the state machines semantics. Note, however, that these now contain also data state information. In the example mentioned above for example at least the value of the attribute *att* must be given in the control state in order to obtain its data state. Accordingly, each transition also determines the data state transformation corresponding to it. This yields further choices for admissible interpretations, since the effect of the method m is not specified. Corresponding OCL constraints in the class could reduce the number of interpretations, but need not determine exactly one.

According to the semantics of state machines each transition (of the transition graph) contains exactly one event. This fact will be used for the composition of state machines represented by the corresponding composition operation applied to the transformation systems associated with the state machines. It is based on a synchronization of steps, where in this case a method call (event) can be synchronized with the execution of a method (action). (Concerning the definition and further discussion of composition operations for transformation system see [6,8].)

The action set of the *transformation* associated with a transition is given by the action set of the state machine's transitions contained in this step. (Using the transformation system extension mechanisms again also other action structures like sequential compositions for example can be employed as labels. Cf. the definition of actions in the UML meta model.)

In the semantics definitions in [5] the possible behaviour of the environment of an object is modelled by non–deterministic state changes of the concerned object, that are not induced by its own actions. When combining it with other objects this non–determinism is reduced in that the other objects realize some of the foreseen external actions, discarding the other ones. This representation of communication is taken over from process calculi and can be represented faithfully by the composition operations of transformation systems, as shown in [7]. As mentioned above, the main issue here is to synchronize events of one system with actions of the other one. An advantage of the transformation system approach in this case is that single state machines are interpreted by sets of single transformation systems, and their composition is defined by the corresponding composition operation of transformation systems. In [5] no composition operation is defined. Instead, the behaviour of a collection of state machines is modelled directly as a single transition system without reference to the individual ones.

6 Sequence Diagrams

The interpretation of sequence diagrams as transformation systems proceeds analogously to the mapping of state machines. On the one hand the basic idea is to map a sequence diagram to a set of transformation systems as its admissible interpretations. The degrees of freedom in the interpretation are basically given by the under-specification of the effects of methods and the underlying data types, as for state machines. Again, this has to be distinguished from the non–determinism specified within a sequence diagram, which is reflected in the branching within the transition graph of each associated transformation system. On the other hand, the semantics definition is not constructed from scratch, but existing formalizations are reconstructed in terms of transformation systems. The main reference for this purpose is [3], where *life sequence charts* (LSCs) are introduced as an extension of sequence charts. The main novelty of LSCs is the possibility to designate mandatory (hot) parts of sequence charts, as opposed to the usual provisional (cold) interpretation of the scenarios. This supports the passage from requirements specifications to design specifications. The semantics of LSCs are defined in terms of transition systems, close to the structure required here, and the restriction to sequence diagrams in the sense of message sequence charts (MSCs) yields the semantics defined in [10], i.e., LSCs are a conservative extension.

The control states of a transformation system defining the formal semantics of a sequence diagram are given by maximal sets of locations on the life lines of the instances that are mutually independent w.r.t. the partial order induced by the sequence diagram (called cuts in [3]). These represent states of the system without inconsistencies w.r.t the causal and temporal order of actions specified by the sequence diagram. The attachment of data states to these control states yields again a set of admissible interpretations, since the relevant information about the structure and the concrete object states is not given in the sequence diagram, but in the class diagram or the object constraints (if at all). The transitions of the corresponding transition graphs are the ones induced by the arrows (messages and internal actions) and the life lines of the sequence diagram. The actions sets of the transformation system are accordingly given by send, receive, and internal actions. Due to the under-specification of their effects several input and output data states of the internal actions are possible again. The sending or reception of a message does not change a data state directly. Conditions need not be reported in the transitions or action sets, since they can be evaluated in the underlying data states already, as in the transformation system semantics of state machines.

Since sequence diagrams specify collections of objects of different classes their semantics cannot be compared directly with the semantics of state machines by taking the intersection of the sets of admissible interpretations. First a projection mapping has to be defined that yields the behaviour of a single object of a designated class. This is induced by a data space signature morphism that embeds the signature of a single object into the data space signature of the sequence diagram, given by the union of the signatures of the interacting objects. Data space signature morphisms and the induced projections (forgetful functors) are defined formally in [6]. Applying this projection to the set of sequence diagram interpretations yields a set of transformation systems of the right signature, the one corresponding to the class the concerned state machine is associated with. Taking the intersections of the images of the sequence diagram's semantics under the projections with the local state machines semantics then shows whether the sequence diagram is consistent with the state machines. That means, it shows whether the objects have the capacity—specified by the state machine—to fulfill the requirements specified by the sequence diagram.

7 Conclusion

Transformation systems constitute a formal semantic domain for the interpretation of different software specification techniques. By mapping a language to this domain its semantics can be made precise and formalized. Moreover, relations between different languages can be investigated on the semantic level. Reflecting these back to the syntactic level correspondences between different models are obtained. Furthermore, also consistency of heterogeneous specifications can be checked due to the interpretation in one common domain. This yields a very flexible and general integration framework for software specification techniques.

In this paper an application of the transformation system approach to the integration of UML models has been sketched. Mapping the UML languages incrementally to the semantic domain yields an internal model with the above mentioned benefits. An important aspect thereby is to reflect the partiality of the viewpoint specifications and the under-specification of the UML semantics by sets of transformation systems as admissible interpretations. Integration of different models is then obtained by appropriate mappings between sets of transformation systems, and taking intersections of admissible interpretations.

In this paper class diagrams, constraints (OCL), state machines, and sequence diagrams have been considered. The results can be carried over to collaboration diagrams and activity diagrams directly, since the former are semantically equivalent to sequence diagrams and the latter can be considered (semantically at least) as special cases of state machine diagrams. Thus, the main modelling techniques for the design stage of software systems development using the UML are covered.

Due to the development of the domain of transformation systems with general schemes for composition operations and development relations further investigations on software specification techniques can be based on these results immediately. A particular instance of the development relations, projection w.r.t. a smaller signature, has already been used in the comparison of sequence diagrams and state machines. Analogously, other development relations between state machines can be defined precisely on the semantic level now, like refinement or specialization to model inheritance from the behaviour viewpoint. The semantic definitions can then be reflected back to the syntactic level and be expressed directly in terms of the model elements. Correspondingly, object (state machine) composition can be defined using an appropriate instance of the composition operations for transformation systems, and investigated w.r.t. its compositionality with the specialization relation.

References

[1] J. Cheesman and J. Daniels. *UML Components: A simple process for specifying component-based software.* Addison–Wesley, October 2000.

[2] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. Technical Report CS98–09, The Weizmann Institute of Science, Rehovot, Israel, 1998.

[3] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object–Based Distributed Systems (FMOODS'99)*, pages 293–312. Kluwer Academic Publishers, 1999. Abridged version of [2].

[4] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Proc. COMPOS'97*, volume 1536 of *LNCS*, 1997.

[5] R. Eshuis and R. Wieringa. Requirements-level semantics for UML statecharts.

In S.F. Smith and C.L. Talcott, editors, *Proc. FMOODS 2000, IFIP TC6/WG6.1*, pages 121–140. Kluwer Academic Publishers, 2000.

[6] M. Große–Rhode. Algebra transformation systems and their composition. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering (FASE'98)*, pages 107–122. Springer LNCS 1382, 1998.

[7] M. Große–Rhode. A compositional comparison of specifications of the alternating bit protocol in CCS and UNITY based on algebra transformation systems. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proc. of the 1st International Conference on Integrated Formal Methods (IFM'99), York, UK, 28–29 June 1999*, pages 253–272. Springer Verlag, 1999.

[8] M. Große–Rhode. Semantic integration of heterogeneous formal specifications via transformation systems. Technical report, TU Berlin, 2001. To appear, see `http://www.cs.tu-berlin.de/~mgr`.

[9] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. Technical report, i-Logix, Inc., 1995.

[10] ITU recommendation Z.120: Message Sequence Charts.

[11] D. Parnitzke. Formal semantics of object systems with data– and object attributes. Master's thesis, TU Berlin, Fachbereich Informatik, January 2001.

[12] J. Tenzer. Translation of UML class diagrams into diagrams of transformation specifications. Technical Report 2000/15, TU Berlin, FB Informatik, November 2000.

[13] *Rational Software Cooperation. Unified Modeling Language – version 1.3*, 1999. Available at `http://www.rational.com`.

[14] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[15] M. Weber. Abstract object systems – a three-layer model of concurrent real-time object systems. Technical Report 97/12, TU Berlin, 1997.