Management and Object Behavior of Statecharts through Statechart DNA

BENJAMIN DE LEEUW
Ghent University
Department of Pure Mathematics
and Computer Algebra
Gent B-9000, Belgium
benjamin.deleeuw@ugent.be

ALBERT HOOGEWIJS
Ghent University
Department of Pure Mathematics
and Computer Algebra
Gent B-9000, Belgium
albert.hoogewijs@ugent.be

Abstract: We propose composed strings called ”statechart DNA” as essential building blocks for a new statechart (sc) abstraction method. We define the simplified statechart (ssc) and show that our definition covers the UML 2.0 sc model, by matching it to all model elements of the StateMachine package of the UML 2.0 metamodel and to the OCL constraints on these model elements. A Model Driven Architecture (MDA) is defined, inspired by a PIM-to-PIM model transformation procedure between UML sc models and ssc models. We discuss the rationale behind action abstraction in ssc models. This framework is used to isolate sc DNA, first in ssc models, then in UML sc models. We show how sc DNA, a compaction of sc construction primitives, can be used to define behavior model metrics and more generally, to manage and maintain evolving object behavior. State machine versioning is an important application of statechart DNA to manage industrial model repositories.

Key–Words: Statecharts, UML, Model checking, State machine versioning

1 Introduction

The Unified Modeling Language (UML) is a visual language to specify all sorts of systems, on an abstract level [2]. UML offers several diagrams to model different aspects of a (software) system. Each kind of diagram shows its own *viewpoint* on this system. This paper studies one particular viewpoint on object oriented systems, namely the statechart (sc), which represents object behavior, similar to automata. In [14] Hunyadi et al. argue ”although UML facilitates software modeling, its object-oriented approach is arguably less than ideal for developing and validating conceptual data models with domain experts”. To overcome this problem, we will study the benefits of abstracting the UML action language for statecharts into a very basic one, consisting only of event throwing actions, and memory reads and writes. We define a homomorphic mapping (similar to Levendovszky et al. in [16]) which transforms standard UML statecharts to so-called simplified statecharts, statecharts with abstracted actions (Sec. 3). We show how this abstraction allows us to propose a mathematical definition of statecharts, similar to automata (Sec. 2) and introduce a grammar and language, called statechart DNA. Relying on graph rewriting principles as introduced in [16] we get a statechart construction process (Sec. 4) and define a formal as well as practical way to scalably manage complexity (Subsec. 5.1) and object behavior (Subsec. 5.3). The main motivation for this research was the generation of a repository of test-cases for verification tools (Subsec. 5.2). And last but not least we propose a versioning manegement tool for State Machine models (Sec. 6).

2 Statechart Definition

The UML sc is an evolution of the Harel sc [12], and has been incorporated in the UML standard since version 1.1. A precise description can be found in the UML 2.0 specification documentation [22]. It is a rich, hybrid model incorporating some influences that cater for different modeling preferences. A basic sc is a state machine model extended with constructs for hierarchical encapsulation and concurrent computation. The execution semantics are based on the queuing of events [22] and on the properties of some kind of action language. Fig. 1 displays part of the UML 2.0 metamodel, which defines the UML sc. Some model elements of UML statecharts, shown in Fig. 1 are redundant, others are convenient extensions to the basic state machine model. For a deeper explanation of the different diagram elements, we refer to the UML 2.0 superstructure document [22].

We only keep the most essential constructs from the UML metamodel in our definition of simplified

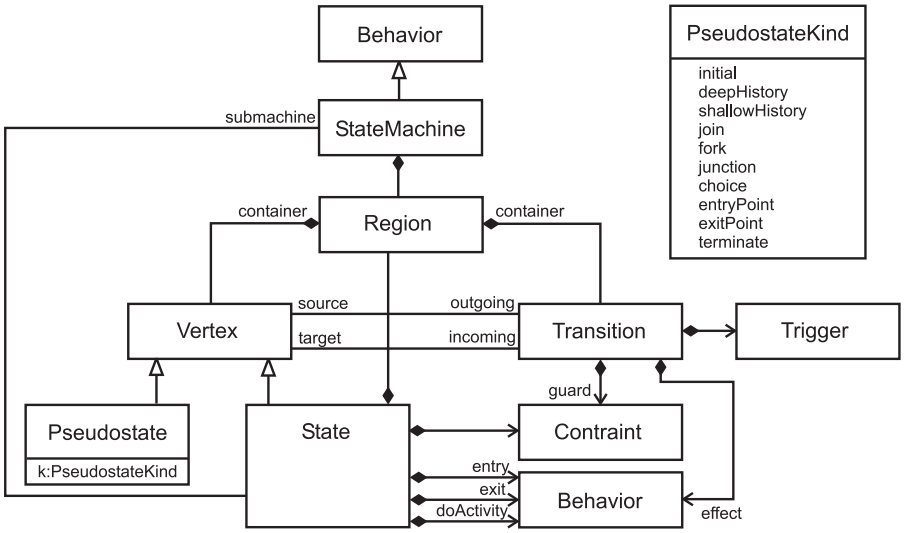

Figure 1: Partial UML 2.0 Metamodel Defining Statecharts

statecharts (ssc). In Sec. 3 we will show how UML statecharts can be converted to simplified statecharts through action abstraction. We use mathematical language to describe this restricted model for statecharts. We therefore assume the reader has some familiarity with automata theory (see for example [18]), as our definitions will make use of notions common in this theory.

Definition 1 (Simplified Sc) *A* simplified statechart *(ssc)* M *is a tuple*

$$M = \langle \Sigma, L, \delta, \delta', s_0, S, T \rangle,$$

where Σ *is a set of atomic objects, called* states. L *is a finite alphabet consisting of two sets of symbols* eL *(events) and* mL *(memory locations) with*

$$\begin{aligned} L &= eL \cup mL \\ eL &\cap mL = \{\epsilon\}. \end{aligned}$$

Here ϵ *is the empty character. The functions* δ *and* δ' *define* transitions *between states.*

$$\begin{aligned} \delta &: \Sigma \times \Lambda \to \Sigma \textit{ (intra-region transitions)} \\ \delta' &: \Sigma \times \Lambda \to 2^{\Sigma} \textit{ (inter-region transitions)}, \end{aligned}$$

where

$$\Lambda = eL \times mL \times L$$

is the set of all labels.*The first component of a label is called the* trigger *of the label, the second one is the* guard *and the third one is the* effect. *The state* s_0 *is the* root state. *It belongs to the set* S*, consisting of all* initial pseudostates *of* Σ*. The set* T *consists of all* terminate pseudostates *of* Σ.

Matching the definition of the ssc model to the UML sc model of Fig. 1 is a trivial exercise, since we used the same names for the components in Def. 1 as in Fig. 1. An explicit construct for *regions* is lacking in Def. 1. We compose regions as collections or containers of states. We build these collections from paths of states in the ssc model. We also define the region hierarchy of ssc M as an ordering relation on S, based on paths.

Definition 2 *A* simple path *of an ssc model* M*, is a list* $[\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n]$ *of states of* Σ*, such that there exists a list of labels* $[l_1, l_2, \ldots, l_n]$ *of* Λ *for which*

$$\begin{aligned} &\sigma_0 \in S \\ &\delta(\sigma_0, l_1) = \sigma_1 \\ &\delta(\sigma_1, l_2) = \sigma_2 \\ &\ldots \\ &\delta(\sigma_{n-1}, l_n) = \sigma_n \end{aligned}$$

A composite path *of an ssc model* M*, is a list* $[\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n]$ *of states of* Σ*, such that there exists a list of labels* $[l_1, l_2, \ldots, l_n]$ *of* Λ *for which*

$$\begin{aligned} &\sigma_0 \in S \\ &\delta(\sigma_0, l_1) = \sigma_1 \vee \sigma_1 \in \delta'(\sigma_0, l_1) \\ &\delta(\sigma_1, l_2) = \sigma_2 \vee \sigma_2 \in \delta'(\sigma_1, l_2) \\ &\ldots \\ &\delta(\sigma_{n-1}, l_n) = \sigma_n \vee \sigma_n \in \delta'(\sigma_{n-1}, l_n) \end{aligned}$$

Definition 3 (Region) *A* region *of ssc* M *from* s_i*, notation* $\rho(M, s_i)$ *or* $\rho(s_i)$ *for short, is a set consisting of the union of all simple paths in* M *headed by some initial pseudostate* s_i *of* S.

Definition 4 *The ordering relation $<_h$, called the* region hierarchy*, is such that $s_i <_h s_j$ iff s_j is on a composite path starting from s_i (with $s_i \neq s_j$).*

Figure 2 shows a "dummy" *ssc diagram*. It is a simplified version of the UML sc diagram. We refer to [22] where the semantics of the UML sc diagram are explained. Figure 2 translates to following ssc model:

$\Sigma = \{s_1, s_2, s_3, s_4, s_5, 1, 2, 3, 4, 5, 6, c_{12}, t_1, t_2, t_3, t_4, t_5\}$
$L = \{\epsilon, e, f, g, h, a, b\}$
with $eL = \{e, f, a\}$ and $mL = \{g, h, b\}$
$S = \{s_1, s_2, s_3, s_4, s_5\}$ and $T = \{t_1, t_2, t_3, t_4, t_5\}$
with root state s_1

$\delta(s_1, (\epsilon, \epsilon, a)) = 1$
$\delta(1, (e, g, a)) = c_{12}$
$\delta(c_{12}, (f, h, b)) = c_{12}$
$\delta(c_{12}, (e, g, a)) = 4$
$\delta(4, (e, g, \epsilon)) = t_3$
$\delta(s_2, (\epsilon, \epsilon, b)) = 2$
$\delta(2, (f, h, \epsilon)) = t_1$
$\delta(s_3, (\epsilon, \epsilon, b)) = 3$
$\delta(3, (f, h, \epsilon)) = t_2$
$\delta(s_4, (\epsilon, \epsilon, b)) = 5$
$\delta(5, (f, h, \epsilon)) = t_4$
$\delta(s_5, (\epsilon, \epsilon, b)) = 6$
$\delta(6, (f, h, \epsilon)) = t_5$

$\delta'(1, (e, g, a)) = \{s_2, s_3, s_5, s_6\}$
$\delta'(4, (e, g, a)) = \{2, 3\}$
$\delta'(t_1, (e, g, a)) = \{4\}$
$\delta'(t_2, (e, g, a)) = \{4\}$
$\delta'(t_4, (e, g, a)) = \{4\}$
$\delta'(t_5, (e, g, a)) = \{4\}$

$\rho(s_1) = \{s_1, 1, c_{12}, 4, t_3\}$
$\rho(s_2) = \{s_2, 2, t_1\}$
$\rho(s_3) = \{s_3, 3, t_2\}$
$\rho(s_4) = \{s_4, 5, t_4\}$
$\rho(s_5) = \{s_5, 6, t_5\}$
$s_1 <_h (s_2, s_3, s_4, s_5)$

$$M = \langle \Sigma, L, \delta, \delta', s_1, S, T \rangle$$

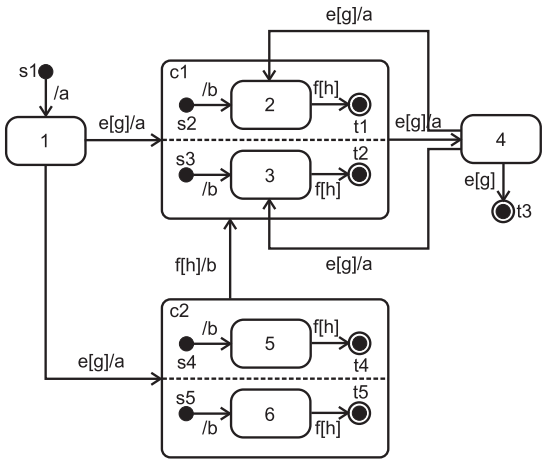

Figure 2: Example Ssc Diagram

Accompanying these definitions, there are some properties which further refine the ssc model. These properties are enforced on the UML sc metamodel through *constraints*, formulated in the Object Constraint Language (OCL). A detailed discussion of the OCL can be found in [28], but is beyond the scope of this paper.

Property 5 *Each region $\rho(s_i)$ in M is disjoint from the other regions in M:*

$$\rho(s_i) \cap \bigcup_{s_j \in S \setminus \{s_i\}} \rho(s_j) = \phi$$

Regions form a partition *of* Σ.

Property 6 *Each region $\rho(s_i)$ contains exactly one initial pseudostate s_i and one terminate pseudostate t_i:*

$$S \cap \rho(s_i) = \{s_i\}$$
$$T \cap \rho(s_i) = \{t_i\}$$

Property 7 *Any initial pseudostate s_i of S has no incoming intra-region transitions, and any terminate pseudostate t_i of T has no outgoing intra-region transitions:*

$$\forall \sigma \in \Sigma : \delta(\sigma, l) \neq s_i$$
$$\forall \sigma \in \Sigma : \delta(t_i, l) \neq \sigma$$

Property 8 *A state cannot be an initial pseudostate and a terminate pseudostate at the same time:*

$$T \cap S = \phi$$

Property 9 *There is exactly one transition from the initial pseudostate s_i of every region to some state $\sigma \in \Sigma$, belonging to that region. Such a transition only carries an effect.*

$$\exists \sigma \in \Sigma : \exists a \in L : \delta(s_i, (\epsilon, \epsilon, a)) = \sigma) \wedge (\forall \sigma' \in \Sigma : \delta(s_i, (e, g, b)) = \sigma' \Rightarrow \sigma = \sigma' \wedge e = \epsilon \wedge g = \epsilon \wedge b = a)$$

Property 10 *There is at least one transition from some state σ of every region to its final pseudostate t_i. These transitions only carry a trigger and a guard.*

$$\exists \sigma \in \Sigma : \exists e, g \in L : \delta(\sigma, (e, g, \epsilon)) = t_i) \wedge (\forall \sigma' \in \Sigma : \delta(\sigma', (f, h, a)) = t_i \Rightarrow a = \epsilon) \wedge (\forall \sigma' \in \Sigma : t_i \in \delta'(\sigma', (f, h, a)) \Rightarrow a = \epsilon)$$

Property 11 *The application $\delta'(t_i, l)$ always returns either the empty set or a singleton set for every $t_i \in T$.*

Property 12 *The transition functions δ and δ' are aligned to eachother:*

$$\forall \sigma \in \Sigma : \delta'(\sigma, l) \subseteq S \Rightarrow \exists \sigma' : \delta(\sigma, l) = \sigma'$$
$$\forall t_i \in T : \sigma \in \delta'(t_i, l) \Rightarrow \exists \sigma' : \delta(\sigma', l) = \sigma$$

Property 13 *The ordering relation $<_h$ of an ssc M is a tree-order (connected, anti-reflexive, anti-symmetric, transitive and at most one predecessor or parent for each element). The root of this tree is the root state s_0 and the children of s_i are those initial pseudostates s_j for which $s_i <_h s_j$.*

Property 14 *If for $\sigma_i \in \rho(s_i)$ and $\sigma_j \in \rho(s_j)$ holds that*

$$\neg(s_i <_h s_j) \wedge \neg(s_j <_h s_i)$$

then there can be no inter-region transitions from σ_i to σ_j.

The different pseudostate kinds are known to be shorthands for UML sc models built of more basic model elements of the UML metamodel [22]. With (global) memory access available (the set mL consists of *memory locations*, see Sec. 3), all pseudostate kinds can be simulated by the basic model elements of Fig. 1, or delegated to the action language. Since pseudostates can be converted this way, we conclude the section with following theorem, the proof of which matches all model elements of UML sc models with mathematical constructs in the definition of ssc models.

The information represented by a model has a tendency to be incomplete, informal, imprecise, and sometimes even inconsistent. For example a UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification [20]. The same applies to state machine behavior. There is a need to describe additional constraints about the objects in the model. The Object Constraint Language (OCL) [22] is a formal language that remains easy to read and write. It is a subset of standard UML that allows software developers to write constraints and queries over object models. We use techniques presented in [20] to prove the following theorem.

Theorem 15 (Ssc Expressivity 1) *Every UML sc model element of the UML metamodel, defining statecharts, is covered by the ssc model definition, as is every OCL constraint on the UML metamodel by properties of the ssc model.*

The platform independent model (PIM) to PIM transformation (see [21]), from UML sc models to ssc models is therefore well defined. We refer to the UML documentation [22] for a full description of the different OCL constraints on the UML metamodel. They are easily matched with the properties of ssc models. The advantages of our approach to sc models can be formulated as follows:

1. we refrain from object-oriented concepts because they are not ideally suited in a mathematical context,
2. each transition label has a fixed size, bringing regular automata and sc models closer together and allowing well-defined mathematical manipulations of sc models,
3. it allows easy translation to other mathematical frameworks such as model checking or assisted theorem provers and
4. it introduces two-tier transition semantics, distinguishing intra-region and inter-region transitions.

3 Action Abstraction

UML sc models describe *effects* of transitions in some kind of *action language*. The UML specification [22] defines formal classes of actions. One language supporting these abstract classes of actions is the Java programming language [11]. The ssc model allows following sorts of actions:

1. *event catches*, appearing as the first component of transition labels, $(\mathbf{e}, g, a) \in \Lambda$, $\mathbf{e} \in eL$.
2. *event throws*, specified as the third component of transition labels, also referred to as the *action* of transition labels, $(e, g, \mathbf{a}) \in \Lambda$, $\mathbf{a} \in eL$.
3. *memory reads*, defined as the second component of transition labels, $(e, \mathbf{g}, a) \in \Lambda$, $\mathbf{g} \in mL$.
4. *memory writes*, shown as the third component of transition labels, $(e, g, \mathbf{a}) \in \Lambda$, $\mathbf{a} \in mL$.

In Tab. 1, we divide the different programming constructs of the Java (action) language into seven classes, more or less resembling the classes of actions of the UML metamodel. All constructs mentioning "implicit" under the **Translation** table header are already present in the ssc model definition and are therefore redundant inclusions in the action language of ssc models. Memory control constructs belong to another UML viewpoint (diagram), and are abstracted away in the sc viewpoint. It remains to show the translation of the classes of constructs, marked "special" in Tab. 1 to ssc model actions.

Object method invocations are sequences of instructions, and can be replaced by a *sequence of actions* from some of the other six classes of programming constructs (see also [22] for *behavioral state machines*, supporting the same principle). They

Table 1: Programming Constructs and their Translation

Construct	Examples	Translation
memory control	int i **finalize** **volatile** **extends**	abstract/omd
assignment	$a = b$ System.println()	special
control flow	**if-else** **while** **for**	implicit
concurrency	Thread t t.start() t.stop()	implicit
event reaction	**throw** **try-catch** itemStateChanged() gen(**new** Event e)	implicit
object invocation	object.method(arg1,arg2)	special
sef operation	$a + b$ $n!$ $a <= b$	special

would therefore be redundant additions to the ssc action language. Side-effects-free (sef) operations also use a sequence of machine instructions to calculate a value. Since the ssc model definition doesn't support *sequences of actions* on transitions, it remains to show how these sequences are translated to the ssc model.

In a visual representation of the sc model, transition labels show the pattern $e[g]/a$, with $(e, g, a) \in \Lambda$ [22]. We put all entry and exit actions of states of the UML sc on all incoming, respectively outgoing transition labels, of these states. Subsequently applying the conversion described in the previous paragraph results in transition labels with action sequences consisting of assignments and sef operations. Figure 3 shows how a sequence $/a; b$ of actions is translated to ssc model transitions with single action component. In the general case, one or more new states are added in a sequential fashion, and the action list, is linearly decomposed into single actions between a sequence of states.

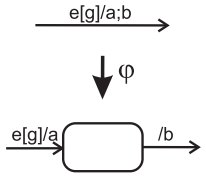

Figure 3: Decomposition of Action Lists in an Ssc Model

Sef operations compute a new value from available ones. In the ssc model, we denote the *read* of the available values, as *guards* on transitions, and the *store* operation of the new value, as memory write *actions*, without reference to any actual value. When a guard $[g]$ appears on a transition labeled $e[g]/a$, this transition should be interpreted as "depending on the value of memory location g, action a will happen". Figure 4 shows the translation of a sef operation taken together with an assignment, $e[g]/$**m = m + l**$; a$. The most recent memory writes [23, 9], on memory loca-

tions m and l, are shown with dotted lines in Fig. 4. The guard $[m]$ on the second transition of the right hand side of Fig. 4, fixes the value of memory location m in the next state. The next transition fixes the value of l in the same way. Given this fixed value for m and l, which is depending on the most recent memory writes for the respective locations, a new value for m is stored on the last transition with action $/m$.

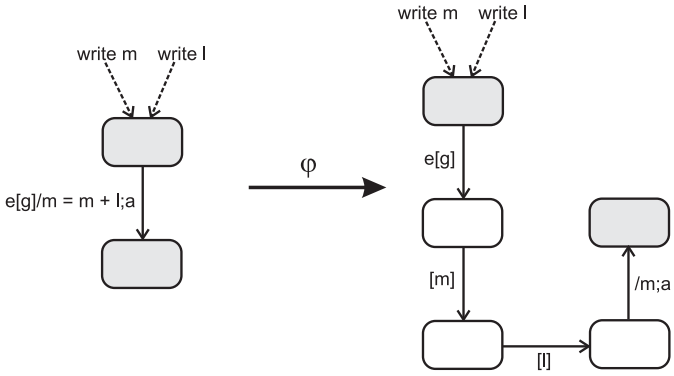

Figure 4: Side-Effects-Free Operations in an Ssc Model

In the UML sc model, guards denote conditions on variables, needed to be true, in order for certain transitions to fire. The UML sc model allows these guards to be compound guards using Boolean operators. The latter are no different from sef operations, and are therefore abstracted in a similar fashion. Figure 5 shows the translation of two well known operators. The *or* connective is decomposed on two distinct transitions, with same source and target. If one of the guards is true, the next state will be reached, and action $/a$ will be executed. The *and* operator is translated into a concurrent state with two regions, each of which checks one of the composite guards. If both guards are true, the concurrent state will be left, and action $/a$ executed. Further discussion of the Boolean operators within guards lies beyond the scope of this paper.

We propose following theorem, the proof of which can be composed using the information in this section.

Theorem 16 (Ssc Expressivity 2) *The ssc model covers the action semantics of the UML sc action language, except for sef operations and assignment, which are abstracted to their most basic forms as memory reads and writes.*

Theorem 15 and thm. 16 taken together allow us to construct a translation *morphism* φ between UML sc models and ssc models, which preserves all information, except for the sef operations of the action language. A trivial exercise shows that every UML sc

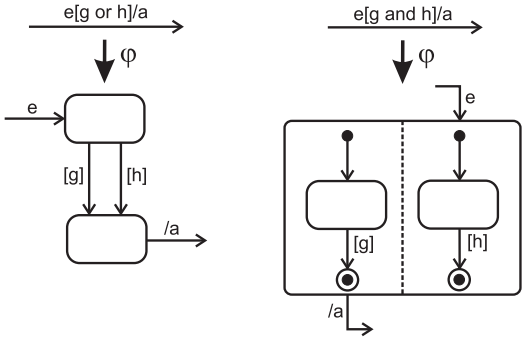

Figure 5: Decomposition of Boolean Connectives in an Ssc Model

model can be translated to one unique ssc model, but that one ssc model, may be the translation of several UML sc models with different sef operations, but the same memory accesses. φ is therefore an *epimorphism*. This morphism formally defines a PIM-to-PIM model transformation from UML sc models to ssc models.

Let us now extend the definition of the ssc model as follows:

Definition 17 *An extended simplified statechart (exssc) is a tuple*

$$\langle \Sigma, L, \delta, \delta', s_0, S, T, \iota \rangle$$

such that $\langle \Sigma, L, \delta, \delta', s_0, S, T \rangle$ *is an ssc, and* ι *is a functor*

$$\begin{aligned}
&\iota = \iota \cup \iota' \\
&\iota : \Sigma \times \Lambda \times \Sigma \rightarrow \lambda - \textit{calc. expr.} \times \tau \\
&\iota' : \Sigma \times \Lambda \times 2^{\Sigma} \rightarrow \lambda - \textit{calc. expr.} \times \tau \\
&\tau = \{b \mid b : \textit{var}(\lambda - \textit{calc. expr.}) \rightarrow 2^{mL}\}
\end{aligned}$$

With λ-*calc. expr.* we mean a Turing computable function specification. Many authors in computer science literature cover λ-calculus as a formal approach to recursive function specifications [1]. In the case of exssc models, all λ-expressions are of the form $\lambda gf.\ gf\epsilon$ with g referring to the guard and f denoting a sef operation. The set τ consists of all *bindings*. A *binding* b lays a connection between the variables in the λ-expression, and the known memory values at that point in the execution. We use exssc to define the execution semantics of ssc, but a discussion of executions lies beyond the scope of this paper. A trivial exercise now shows that with this definition of exssc we are able to make φ an *isomorphism* between UML sc models and exssc models such that one exssc model also translates back to one unique UML sc model.

4 Statechart DNA

With the translation morphism φ defined in Sec. 3, the action language for sc models can be reduced to its most basic form, consisting of memory reads and writes and throws and catches of events. The sc model is limited to its most basic constructs. This simplification makes it easy to compose and manipulate ssc models. We will postpone the discussion on manipulation to Sec. 5 and 6. In this section we introduce a grammar rewrite system, inspired by the theory of *scenario composition* [3]. This scheme allows us to identify the most important complexity determining factors of sc models. Each factor is formalized in this grammar as a composition construct. When identified, these factors can be counted and put onto a complexity scale for sc models (see Subsec. 5.1). The rewrite system itself will allow us to generate sc test cases, usable within sc analysis and development tools.

A detailed description of scenarios lies beyond the scope of this paper. We therefore refer to [3] for more information. Scenarios can easily be integrated in the industrial design of software [15], but in order to gain the ability to specify executable systems with scenarios, we have need of some typical programming constructs which go beyond a linear execution of coupled scenarios. Expert authors in the field have identified how multiple scenarios can be *composed* into one executable behavior [29, 27, 19, 13]. In order to make realistic behaviors, composed from scenarios, we need a construct that executes more than one scenario *sequentially*, one that allows conditional or *disjunctive* execution, another for parallel or *conjunctive* execution and a construct that *iterates* over a scenario a number of times [29, 27]. In this paper, we use the same combination rules as in the work on scenario composition, but we apply them on *atomic ssc* instead of on scenarios. An atomic ssc(assc) represents the simplest conceivable ssc.

Definition 18 *An* atomic ssc *(assc)* $(a|e[g])$ *is an ssc*

$$\langle\{\epsilon, e, g, a\}, \{s, \sigma, t\}, \delta, \delta', s, \{s\}, \{t\}\rangle$$

with one region, initial pseudostate s*, normal state* σ *and terminate pseudostate* t*. Transition function* δ' *is empty and* δ *is defined as follows:*

$$\delta(s, \epsilon, \epsilon, a) = \sigma$$
$$\delta(\sigma, e, g, \epsilon) = t$$

We show that every ssc model is composed of a finite number of assc. Only a finite number of different assc is available to ssc models, by the finite alphabet L of Def. 1. One single assc can however be repeated a finite number of times in an ssc model. We compose assc in an ssc model, guided by the rewrite system displayed in Tab. 2. The grammar defines two *composition* operators $+$ (4) and $\oplus$ (3,6), a *lifting* operator $\rightarrow$ (7,8), and a *wrapping* operator $[\ldots]_{(a|e[g])}$ (10,12).

Table 2: Rewrite Rules of Ssc Composition (Sc DNA)

(1)	$Start \hookrightarrow R$
(2)	$R \hookrightarrow \lambda \quad \mid \quad S$
(3)	$R \hookrightarrow S \oplus \langle_d\ R, R_d\ \rangle_d$
(4)	$S \hookrightarrow S + S$
(5)	$S \hookrightarrow (a\|e[g])$ for some $(e, g, a) \in \Lambda$
(6)	$S \hookrightarrow S \oplus \langle_d\ R, R_d\ \rangle_d \oplus S$
(7)	$S \hookrightarrow S \rightarrow \langle_i\ R_i\ \rangle_i$
(8)	$S \hookrightarrow S \rightarrow \langle_c\ R_c\ \rangle_c$
(9)	$R_d \hookrightarrow R_d, R_d$
(10)	$R_d \hookrightarrow [R]_{(a\|e[g])}$ for some $(e, g, a) \in \Lambda$
(11)	$R_i \hookrightarrow R_i, R_i$
(12)	$R_i \hookrightarrow [R]_{(a\|e[g])}$ for some $(e, g, a) \in \Lambda$
(13)	$R_c \hookrightarrow R_c, R_c$
(14)	$R_c \hookrightarrow R$

We call the *language* defined by the production system of Tab. 2 *statechart DNA*, because it consists of strings describing *how* ssc models are composed of assc. We use the symbol λ to denote the empty assc having no state and no transitions. The different operators of sc DNA are explained as follows:

1. *composition by* $+$ glues two ssc operands M_1 and M_2 together in one resulting ssc model $M_1 + M_2$. The terminate pseudostate of the root region of M_1 ($\rho(s_0)$) is removed from M_1, as is the initial pseudostate of the root region of M_2. Both "loose" transition labels $e[g]$ of M_1 and $/a$ of M_2 are then composed into one label $e[g]/a$, connecting $M_1 + M_2$. The first operand M_1 may therefore only have one edge to the terminate pseudostate of the root region, otherwise composition with $+$ is undefined. The rewrite system of Tab. 2 guarantees that this constraint holds. Composition by $+$ is associative, if the constraint also holds for the second operand M_2.

2. *composition by* $\oplus$ glues one or more ssc, to more than one ssc and results in composite ssc model M. The basic operation is analogous to composition by $+$, but in case of a lifted operator preceding or following the $\oplus$ operator, transitions are redistributed (change source or target states) over

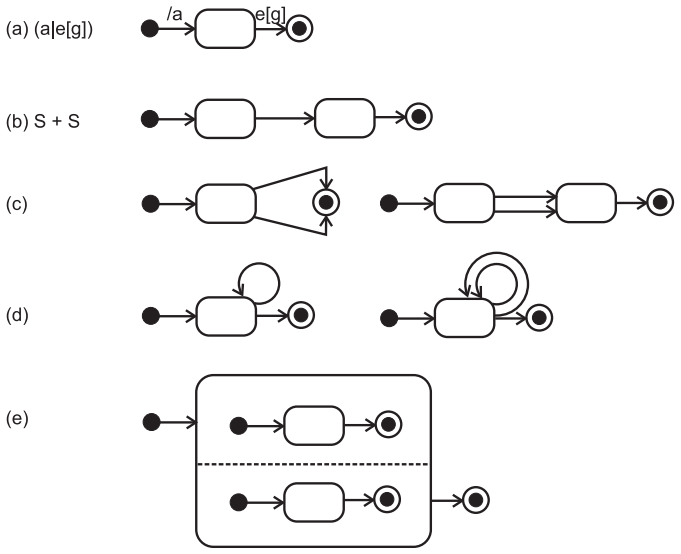

Figure 6: Corresponding Statecharts

the loop or concurrent sub regions that are lifted (see *lifting* below). We define this redistribution of transitions to be irrelevant to the model's purpose.

3. states *lifted* to encapsulated regions and loops denote where clusters of transitions might be redistributed to the states of those regions and loops respectively. *Lifting* therefore marks the ssc models M_i of which loops and encapsulated regions consist.

4. *wrapping* identifies certain label parts in the case of *disjunctive* and *iterative* composition by $+$ and $\oplus$.

Disjunctive composition is delimited by $\langle_d \ldots \rangle_d$ (Fig.6c,6d,6e), *iterative* composition by $\langle_i \ldots \rangle_i$ (Fig.6f,6g) and *conjunctive* composition is marked with $\langle_c \ldots \rangle_c$ (Fig. 6h). An example production string of the rewrite system in Table 2

$$
\begin{aligned}
&(c|a[b]) + (f|d[e]) \\
&\quad \oplus \langle_d \lambda, [(i|g[h])]_{(b|f[h])}, [(l|j[k])]_{(v|t[u])} \rangle_d \\
&\quad \oplus (o|m[n]) \rightarrow \langle_i [\lambda]_{(a|e[m])} \rangle_i \\
&\quad \oplus \langle_d \lambda, [\lambda]_{(z|x[y])} \rangle_d \\
&\quad \oplus (p|q[r]) \rightarrow \langle_c (u|s[t]) + (x|v[w]), (a'|y[z]) \\
&\qquad\qquad + (d'|b'[c']) + (g'|e'[f']) \rangle_c \\
&\quad + (j'|h'[i']) + (m'|k'[l']) + (p'|n'[o']),
\end{aligned}
$$

translates to a *class* of ssc models. One element of this class is shown in Fig. 7. It illustrates the translation of one sc DNA string to one of its possible ssc model representations. The *wrapper* operation is displayed as underlined transition label parts. Figure 7 shows one *conjunctive* site, one *iteration*, and two *disjunctive* sites, matching their counterparts in the example production string. Reading the example production string from left to right and the ssc diagram of Fig. 7 from the root state to the terminate pseudostate of the same region, we encounter

1. first a disjunctive site, consisting of three ssc models $\langle_d \lambda, [r_2]_{(b|f[h])}, [r_3]_{(v|t[u])} \rangle_d$, the first of which is empty,

2. an iterative site consisting of an (empty) ssc model connected through a reflexive transition $\langle_i [\lambda]_{(a|e[m])} \rangle_i$,

3. a second disjunctive site consisting of two empty ssc models $\langle_d \lambda, [\lambda]_{(z|x[y])} \rangle_d$ and

4. a conjunctive site consisting of two ssc models $\langle_c r_4, r_5 \rangle_c$. Incoming transitions from the second disjunctive site are redistributed to the two concurrent regions of the conjunctive site.

Because of the *redistribution* of incoming edges to the concurrent regions, shown in the example of Fig. 7, every sc DNA string represents a *class* of ssc models, with as many elements as there are possible redistributions of edges. In case of the given example, there are five states, two final pseudostates and one concurrent state, eligible for redistribution. This means 64 possible redistributions, hence the class of ssc models translated from this production string consists of 64 ssc models.

A relatively complex procedure based on model transformations and graph rewriting as explained in [16] allows us to construct for each class of ssc

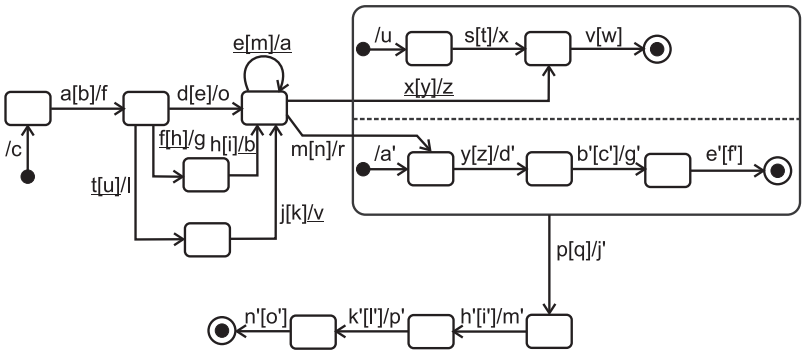

Figure 7: Translation of an Example Sc DNA

models, the sc DNA. The proof of the following theorem builds on this procedure. Both lie beyond the scope of this paper.

Theorem 19 (Sc DNA Translation) *Each element of the sc DNA language translates to a* class *of ssc models, each of which is disjoint from the classes translated from other sc DNA production strings.*

This theorem allows us to construct another translation *isomorphism* ψ between sc DNA production strings and ssc model classes. The complex procedure, mentioned above implements the inverse morphism ψ^{-1}. The morphism ψ defines a PIM-to-PIM model transformation from ssc models to sc DNA. The inverse morphism ψ^{-1} defines a PIM-to-PIM model transformation from sc DNA to ssc models.

Definition 20 *The isomorphism ψ returns a* complexity class *of ssc models, for each element of the sc DNA language.*

Following theorem is a consequence of thm. 19. We denote the set of all ssc models, allowed by Def. 1 as SSC, and the set of all sc DNA production strings with DNA.

Theorem 21 (Sc DNA Completeness)

$$\psi(DNA) = SSC$$
$$\psi^{-1}(SSC) = DNA$$

The composed morphism $\psi^{-1} \circ \varphi$, by thm. 15, thm. 16, thm. 19 and thm. 21, allows us to abstract any UML sc model to its complexity class, represented by a unique sc DNA production string.

5 Statechart DNA Experimentation

We have been working ([4], [7], [5], [6], [8]) on developing following applications :

5.1 Complexity Metric for UML sc models

Further compression of sc DNA on a numeric scale allows us to define a complexity metric for UML sc models, considering the *intenseness* of concurrent, iterative and disjunctive (conditional) execution and the label *density* of the composing assc. Engineering their development, different versions of UML sc models can be evaluated by such a metric.

5.2 Generation sc Model Test Cases

Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Extremely large state-spaces need special techniques such as slicing [25] in order to reduce the traversal time. A classical test case for such model checkers in the ”washing machine controler” as described in [17]. The rewrite system of Sec. 4 allows (automatic) generation sc model test cases, in different complexity classes. These are usable as a general purpose sc repository for benchmarking model checking tools, and in gathering empirical evidence for sc model theories ([26],[24]).

5.3 “Benign” Behavior Manipulations

We use the sc DNA framework, to define “benign” behavior manipulations, applicable to sc development in CASE tool environments. Given an sc DNA specification, dna, of a UML sc, replace all occurrences of $assc$, and of λ, except those occurring in *wrapper* operations, with a variable S, and call the resulting string dna'. A *conservative* sc modification is defined as any UML sc, which converts to an sc DNA string, obtained by rewriting *through* dnd', in the parse tree for the rewrite system of Tab. 2. A *mutative* sc modification is obtained by a rewriting *through* any dnd'

that is obtained by a permutation of two sequences of the form $+S \dots S+$ in dna'. Combined with the complexity metric of Subsec. 5.2 these patterns can be used to manage and control object behavior evolutions in industrial model repositories.

6 State Machine Versioning

In this section we highlight an important industrial application of sc DNA: versioning and automatically merging different state machine models. Software companies protect their source code from unwanted change by adding version control to the company's code repository. A large number of versioning systems exist (e.g. MS Team Foundation System, SourceSafe, Eclipse) all based on the same principle: starting from a base-lined text file only the changes to this file are stored (called a "diff" or "delta"). The history of such a text file can therefore be reconstructed up to the creation of the original version. Changes can also be undone by rolling back to any previous version of the text file. If the language of the text file is a compositional programming language, versioning can also be understood as preventing multiple developers from modifying the same source code inconsistently. This means that the versioning system will signal differences upon storing the code and force developers to merge their code into one consistent version. Storing text files in a versioning system is referred to as a check-in, changing and locking the code as check-out.

Any one-dimensional compositional string of symbols (e.g. Java programs) can be versioned but multi-dimensional structures like graphs (and state machines) don't allow this. This is why versioning and merging of UML models in general doesn't belong to the possibilities of versioning systems because they most commonly store the models as XMI serializations. With sc DNA we introduce an abstraction of state machines that can be used as a compositional serialization subject to versioning mechanics just as normal source code can. The XMI serialization of a state machine represents it as a hierarchical object oriented structure of meta-model elements. On this representation an additional calculation step is needed. This paper contains a simple and elegant way to achieve this. Sc DNA represents state machines based on a formal grammar and therefore sensible automatic model merges of different state machine versions can be proposed.

We showed how every state machine can be represented as an ssc (with morphism φ). An ssc can be compiled into an sc DNA string. This form can be made normal by ordering comma separated R-lists of sc DNA from smallest to biggest number of atoms and

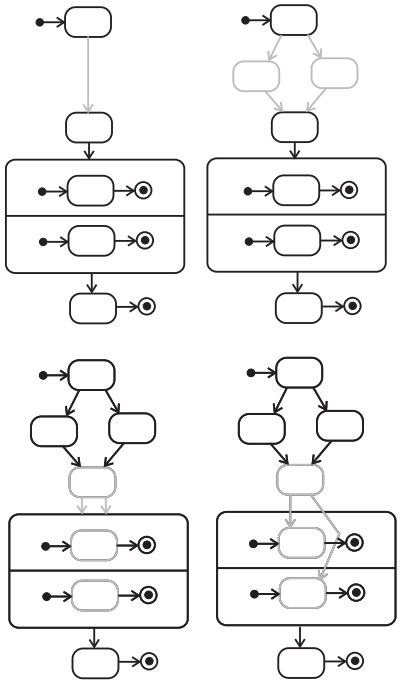

Figure 8: Merging State Machine models

then alphabetically by atom labels (see Sec. 4). We use a function called "Diff" calculating the smallest set of differentiating substrings of two text files [10], with the smallest total number of characters appearing in both sets of substrings. We also use the XMI description of both state machines to be compared (for state and transition identification, see below).

Because the sc DNA is now in a normal form, there is one unique parsing tree for it. This tree can be serialized breadth first with nodes ordered by parse step length or weight (see 5.1). This serialization is again normalized and results in another text file on which the "Diff" calculation can apply. The top row of Fig. 8 shows two state machine model versions on which an insert (or dually, a delete) has been performed.

Consider for example the sc DNA compilation of the upper left and right sc. Their respective compilation trees are compared as shown in Tab. 3. We determine where and how two new sc (represented as R) were added to the existing state machine.

From Tab. 3 we see that the compilation tree differs in step four where a disjunctive pattern was used instead of a sequential one. This transformation can now automatically be assigned to the Diff set of the sc.

We use this observation to visualize and automatically merge both models. They are compiled into sc DNA. Applying the Diff-function on both normalized and serialized parse trees can therefore show additional and removed grammar steps in between sc

Table 3: Determining The "Diff" Set of Fig.8

(1)	$Start$	$Start$
(2)	R	R
(3)	S	S
(4)	$S + S$	$S \oplus \langle_d R, R_d \rangle_d \oplus S$
(5)	$S + S + S$	$S \oplus \langle_d R, R_d \rangle_d \oplus S + S$
(6)	$S + S + S + S$	$S \oplus \langle_d R, R_d \rangle_d \oplus S + S + S$
(7)	$S + S + S \rightarrow \langle_c R_c \rangle_c + S$	$S \oplus \langle_d R, R_d \rangle_d \oplus S + S \rightarrow \langle_c R_c \rangle_c + S$
(8)	$S + S + S \rightarrow \langle_c R_c, R_c \rangle_c + S$	$S \oplus \langle_d R, R_d \rangle_d \oplus S + S \rightarrow \langle_c R_c, R_c \rangle_c + S$
(9)	...	...

DNA versions under comparison. Automatic merges can now be done if there is an ordering available on the sc DNA grammar rules (e.g. Subsec. 5.1). Dependent on the weight of the Diff-set and if it is an insert or removal it can then be decided to be allowed automatically or rolled-back (e.g. normal inserts merge automatically, similar to code merges).

Transition redistribution can also be detected. After compilation to sc DNA it is known where redistributions can happen (everywhere the $\oplus$-sign appears, see Sec. 4). The lower row of Fig. 8 shows a redistribution of edges over two concurrent regions. Although compiling both models to sc DNA will show the same serialized compilation tree, compiling the sc DNA strings back to ssc and matching the Diff-sets of both XMI representations will show (and visualize) where redistributions have happened. Redistributions can again be ordered in preference and therefore be merged automatically together with other changes.

More powerful changes can be detected and merged with the described technique. Transition label changes can consitently and automatically be determined and matched to a memory model (of read-write access). An impact analysis (or model check) on memory states can then determine to allow or discard certain transition label manipulations automatically.

Figure 9 shows three versions of the branching transitions in Fig. 8 belonging to the "Diff" set. From left to right their labels are detailed as follows:

$$(\epsilon|e_1[g_1]) \; + \; (a_1|e_2[g_2])$$
$$(\epsilon|e_1[g_1]) \; \oplus \; \langle_d \, (x_1|y_1[z_1]), [(x_2|y_2[z_2])]_{(a_1|e_1[g1])} \, \rangle_d \oplus (a_1|e_2[g_2])$$
$$(\epsilon|e_1[g_1]) \; + \; (a_1|y_1[z_1]) \; \oplus \; \langle_d \, \lambda, [\lambda]_{(x_2|y_2[z_2])} \, \rangle_d \oplus (x_1|e_2[g_2])$$

Suppose the leftmost version of Fig. 8 was originally checked-in. The middle and rightmost version might both have been developed by separate analysts. An automatically deducible merge between the two versions of the right is to keep the rightmost one since it has the least intricate disjunctive site but does appear to have the same memory access.

Depending on the intended power of automatic merges, calculations get gradually more involved. Automatically merging the state machine structure is the least involved. Next comes the detection of redistributed transitions and lastly perceiving patterns of label changes. State machine versioning systems should restrict the complex calculations according to this ordering. Starting from the XMI description of a state machine check if the XMI descriptions of parallelly checked-in versions are indeed different. If this is the case calculate the sc DNA to check for structural changes. If you find mutated sites detect for each of these if and where transition redistributions were applied. After this step determine transition label patterns.

Property 22 *Three (almost) ortogonal information axes are available to control automatic state machine merges. With XMI, the serialization into XML, DNA the serialization into sc DNA and MM the intended memory model to be applied to the label behavior of state machines we define following ordering on the calculation complexity of state machine merges:*
$XMI <$
$DNA <$
$DNA \times XMI <$
$DNA \times XMI \times MM$

7 Conclusion

In this paper, we confirm that the *effects* of UML sc models can be abstracted, and that they are but a secondary construct of the rich UML model. With a formal approach we build a strictly mathematical

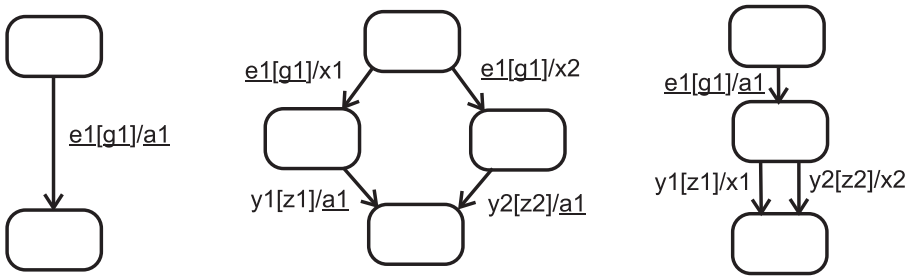

Figure 9: Tracking Label Changes

model of the UML sc and untangle the object-oriented concepts characterizing the UML. We treat triggers, guards and effects as related to eachother, but make no reference to any explicit value (type) in the computation. This abstract approach allows us introduce involved mathematical manipulations of UML sc, in line with the theory of regular automata. The morphism φ, introduced in Sec. 3, translates UML sc models to ssc models. These reduce to sc DNA descriptions by morphism ψ^{-1} of Sec. 4. Sc DNA allows us on the one hand to partition ssc models, and therefore also UML models, into complexity classes, which give us an indication of how *difficult* a behavioral model is. On the other hand sc DNA strings can be manipulated thereby allowing formal behavioral model management, refactoring and versioning.

References:

[1] H P Barendregt, editor. *The Lambda Calculus (Studies in Logic and the Foundations of Mathematics Series)*. Elsevier, 2006.

[2] G Booch, J Rumbaugh, and I Jacobson. *The Unified Modeling Language User Guide (2nd edition)*. Addison-Wesley Professional, 2005.

[3] J M Carroll, editor. *Scenario-Based Design: Envisioning Work and Technology in System Development*. John Wiley and Sons, 1995.

[4] Benjamin De Leeuw. *Statechart DNA: Formal and Practical Investigation in a Statechart Abstraction Method*. PhD thesis, Gent University, December 2009.

[5] Benjamin De Leeuw and Albert Hoogewijs. Complexity Metrics with Generated Statechart Instrumentation. Internal report, September 2006.

[6] Benjamin De Leeuw and Albert Hoogewijs. Random Test Generation and Instrumentation of Statecharts. Internal report, July 2006.

[7] Benjamin De Leeuw and Albert Hoogewijs. Scenario-Based Analysis of Statechart Construction. In P Schobbens and J Fiadeiro, editors, *Abstracts of the 18th International Workshop on Algebraic Development Techniques, 1-3 June 2006, La Roche en Ardenne, Belgium.*, pages 28–28, 2006.

[8] Benjamin De Leeuw and Albert Hoogewijs. Formal management of object behavior with statechart dna. In *10.1109/AFRICON.2007.4401522 / INSPEC Accession Number: 9857309*, pages 1–7. IEEE Xplore, september 2007.

[9] Guang R Gao and Vivek Sarkar. Location Consistency-A New Memory Model and Cache Consistency Protocol. *IEEE Trans. Comput.*, 49(8):798–813, 2000.

[10] Free Software Foundation Gnu. Comparing and merging files. *http://www.gnu.org/ software/ diffutils/manual/html_mono/diff.html*, 4 2002.

[11] D Harel and H Kugler. The RHAPSODY Semantics of Statecharts. *Lecture Notes in Computer Science*, 3147(0):325–354, June 2004.

[12] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[13] R L Hobbs. Using a Scenario Specification Language to Add Context to Design Patterns. In *SEKE '04: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 330–335, 2004.

[14] Daniel Ioan Hunyadi and Mircea Adrian Musan. Uml data models from an orm (object-role modeling) perspective: data modeling at conceptual level. *WSEAS Transactions on Information Science and Applications*, 5(5):796–805, 2008.

[15] Rohit Kelapure, Marcos André Gonçalves, and Edward A. Fox. Scenario-Based Generation of

Digital Library Services. In Traugott Koch and Ingeborg Sølvberg, editors, *ECDL*, volume 2769 of *Lecture Notes in Computer Science*, pages 263–275. Springer, 2003.

[16] Tihamr Levendovszky, Lszl Lengyel, and Hassan Charaf. Extending the dpo approach for topological validation of metamodel-level graph rewriting rules. *WSEAS Transactions on Information Science and Applications*, 2(2):226–231, 2005.

[17] Xin Ben Li and Feng Xia Zhao. Formal development of a washing machine controller model based on formal design patterns. *WTOS*, 7(12):1463–1472, 2008.

[18] Peter Linz. *An Introduction to Formal Languages and Automata (3rd edition)*. Jones and Bartlett Publishers, 2001.

[19] E Makinen and T Systa. An Interactive Approach for Synthesizing UML Statechart Diagrams from Sequence Diagrams. In *OOPSLA2000: Proceedings of the 10th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 2000.

[20] Gergely Mezei, László Lengyel, Tihamér Levendovszky, and Hassan Charaf. Minimizing the traversing steps in the code generated by OCL 2.0 compilers. *WSEAS Transactions on Information Science and Applications*, 3:818–824, April 2006.

[21] OMG. MDA Guide Version 1.0.1 (omg/2003-06-01), June 2003.

[22] OMG. Unified Modeling Language: Superstructure, v2.0 (formal/05-07-04), August 2005.

[23] W Pugh and T Lindholm. JSR-133: Java Memory Model and Thread Specification, final release, September 2004.

[24] Sara Van Langenhove. Protocal conformance through refinement mapings in cadence smv. *Bull. Belg. Math. Soc.-Simon Stevin*, 13:905–915, 2006.

[25] Sara Van Langenhove and Albert Hoogewijs. SVtL: System verification through logic tool support for verifying sliced hierarchical statecharts. *Lecture Notes in Computer Science*, 4409:142–155, 2007.

[26] Sara Van Langenhove, Albert Hoogewijs, and Benjamin De Leeuw. Uml based verification of software. In *Proceedings of the 32nd Spring School in Theoretical Computer Science, Concurrency theory and Applications*, page 1, Luminy, France, 4 2004.

[27] S Vasilache and J Tanaka. Synthesizing Statecharts from Multiple Interrelated Scenarios, 2001.

[28] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition*. Addison-Wesley Professional, August 2003.

[29] Jon Whittle and Johann Schumann. Generating Statechart Designs from Scenarios. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 314–323, New York, USA, 2000. ACM Press.