FORMAL MANAGEMENT OF OBJECT BEHAVIOR WITH STATECHART DNA

Benjamin De Leeuw* **and Albert Hoogewijs†**

* *Dept. of Pure Mathematics and Computer Algebra, Universiteit Gent, Galglaan 2 - 9000 Gent, Belgium*

† *Dept. of Pure Mathematics and Computer Algebra, Universiteit Gent, Galglaan 2 - 9000 Gent, Belgium*

Abstract: We introduce and explore a new statechart (sc) abstraction method. We define simplified statecharts (ssc) and discuss the use of action abstraction in ssc models. We isolate sc DNA from UML sc models, and show how this sc DNA can be used to define behavior model metrics and more generally, to manage object behavior.

Keywords: State Machine Metrics; Unified Modeling Language; Behavior Modeling and Design; Mathematical Abstraction of Statecharts.

1. INTRODUCTION

The Unified Modeling Language (UML) is a visual language to specify all sorts of systems, on an abstract level [1]. The language offers a number of diagrams to specify different parts 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. We will investigate the benefits of abstracting the UML action language for statecharts to a very basic one, consisting only of event throwing actions, and memory reads and writes. We define a morphism which transforms standard UML statecharts to so-called simplified statecharts, statecharts with abstracted actions (Sect. 3.). We show how this abstraction allows us to propose a mathematical definition for statecharts, similar to automata (Sect. 2.), introduce a grammar and language, called statechart DNA, which summarizes the statechart construction process (Sect. 4.) and define a formal as well as practical way of scalable managing object behavior (Sect. 5.).

2. STATECHART DEFINITION

The UML sc is an evolution of the Harel sc [2], 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 [3]. It is a rich, hybrid model incorporating a number of influences that cater for different modeling preferences. A basic sc is a state machine model, extended with constructs for hierarchical encapsulation and for the denotation of concurrent computation. The execution semantics are based on queuing of events [3] and on the properties of some kind of action language. Fig. 1 displays part of the UML 2.0 metamodel, which defines the

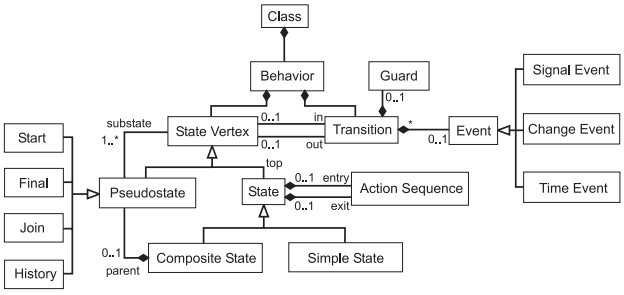

Figure 1: Partial UML 2.0 Metamodel Defining Statecharts

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 an explanation of the different diagram elements, refer to the UML 2.0 superstructure document [3].

We only retain the most essential constructs from the UML metamodel in our definition of simplified statecharts (ssc). In Sect. 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 the mathematical theories of automata (see for example [4]), as our definitions will make use of notions common in these theories. We abbreviate $n \in r, r \in R$, with R a powerset of any set of objects n, as $n \in^* R$, if it is clear or doesn't matter which $r \in R$ we are referring to.

[Simplified Sc] A *simplified statechart* (ssc) is a tuple

$$\langle \Sigma, R, \delta, \delta', \rho, r_0, S, T \rangle$$

Σ is a finite alphabet consisting of two sets of symbols eL (event locations) and mL (memory locations)

$$\Sigma = eL \cup mL$$
$$eL \cap mL = \{\lambda\}$$

λ is the empty symbol. R is a set of regions. Each region $r \in R$ is a set of atomic objects (states). δ and δ' are transition functions, ρ is an encapsulation function. S is the set of start pseudostates, of any region $r \in R$. T is the set of final pseudostates, of any region $r \in R$. r_0 is one designated element of R (root region).
Accompanying this definition, there are a number of properties, which further refine the ssc model. These properties are usually enforced on the UML sc model through constraints, formulated in the Object Constraint Language (OCL). A detailed discussion of the OCL can be found in [5].
Each set $r \in R$ is disjoint from the other sets in R:

$$\forall r \in R \cdot r \cap \bigcup_{r_i \in R \setminus \{r\}} r_i = \phi$$

Each region $r \in R$ contains exactly one start pseudostate (named s_r) and one final pseudostate (named t_r):

$$\forall r \in R \cdot S \cap r = \{s_r\}$$
$$\forall r \in R \cdot T \cap r = \{t_r\}$$

A state cannot be a start pseudostate and a final pseudostate at the same time:

$$T \cap S = \phi$$

We define *labels* for the transitions, and the signature of the δ,δ' and ρ functions. In order to define these as total functions, we make use of an *error state* $*$, which is implicitly present in any ssc. [Label] An element of the set L is a *label*.

$$L = eL \times mL \times (eL \cup mL)$$

[Labelled Transition]

$$\delta : \bigcup_{r \in R} \delta_r$$
$$\delta_r : r \times L \to r \cup \{*\} \qquad \text{(for any } r \in R\text{)}$$

$$\delta' : \bigcup_{r \in R} \delta'_r$$
$$\delta'_r : r \setminus \{s_r\} \times L \to \bigcup_{r_i \in R \setminus \{r\}} r_i \cup \{*\} \quad \text{(for any } r \in R\text{)}$$

$$\rho : \bigcup_{r \in R} \rho_r$$
$$\rho_r : r \to 2^{S \setminus \{s_r\}} \qquad \text{(for any } r \in R\text{)}$$

Following properties further constrain the definition of labelled transition. There is exactly one transition from the start pseudostate s_r of every region $r \in R$ to some state $n \in r$:

$$\forall r \in R \cdot (\exists n \in r \cdot \exists a \in \Sigma \cdot \delta(s_r, (\lambda, \lambda, a)) = n) \wedge$$
$$(\forall n' \in r \cdot \delta(s_r, (e, g, b)) = n' \Rightarrow n = n' \wedge e = \lambda \wedge g = \lambda \wedge b = a)$$

There is at least one transition from some state $n \in r$, for every region $r \in R$, to the final pseudostate t_r:

$$\forall r \in R \cdot (\exists n \in r \cdot \exists e, g \in \Sigma \cdot \delta(n, (e, g, \lambda)) = t_r) \wedge$$
$$(\forall n' \in r \cdot \delta(n', (f, h, a)) = t_r \Rightarrow a = \lambda) \wedge$$
$$(\forall n' \in^* R \cdot \delta'(n', (f, h, a)) = t_r \Rightarrow a = \lambda)$$

There exist no transitions from any start pseudostate, which cross regions:

$$\forall r \in R \cdot \forall (e, g, a) \in L \cdot \delta'(s_r, (e, g, a)) = *$$

If for some regions $r_1, r_2 \in R$, and for some state $n \in^* R$ holds that r_1 and r_2 are subregions of n, then there can be no region crossing transitions between the states of regions r_1 and r_2:

$$\forall r_1, r_2 \in R \cdot \forall n \in^* R \cdot s_{r_1}, s_{r_2} \in \rho(n) \Rightarrow$$
$$\neg(\exists n_1 \in r_1, n_2 \in r_2 \cdot \delta'(n_1, (e, g, a)) = n_2 \vee \delta'(n_2, (e, g, a)) = n_1)$$

The start and final pseudostates of every region $r \in R$, have empty ρ:

$$\forall r \in R \cdot \rho(s_r) = \phi \wedge \rho(t_r) = \phi$$

We define the region hierarchy as an ordering relation on R. The ordering relation $<_h$ is such that $r <_h r'$ ($r, r' \in R$) if and only if

$$\exists n \in r \cdot s_{r'} \in \rho(n)$$

The ordering relation $<_h$ of an ssc must be a tree-order (connected, antireflexive, antisymmetric, transitive and at most one predecessor or parent for each element), on all regions in R. The root of this tree is the region r_0 and the children of region $r \in R$ are those regions $r' \in R$ for which $r <_h r'$.
The basic building blocks of ssc models are start (s_r) and termination (t_r) pseudostates, simple states ($\rho(n) = \phi$), compositional states ($\rho(n) \neq \phi$), transitions (δ, δ') and regions ($r \in R$), all of which are easily matched to the UML metamodel of Fig. 1. We encounter event locations, which match events, and memory locations, matching guards. Action sequences are replaced by actions on the transitions themselves, represented by the third component of transition labels ($(e, g, \mathbf{a}) \in L$). Join pseudostates are shorthands for concurrency with region crossing edges, and are therefore implicitly present in the definition of the ssc model. We don't introduce history pseudostates, because they can also be considered syntactic sugar. With (global) memory access available (the set mL consists of *memory locations*), this memory can be used to simulate the history pseudostate.
According to the UML semantics [3], the history pseudostate *stores* the current state of affairs, when a region

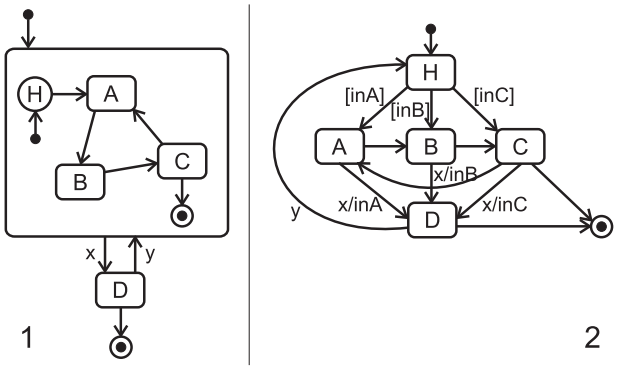

Figure 2: Conversion of a Region with UML History Pseudostate

is left. Upon reentry of this region, the history state returns the region to this state. Fig. 2.1 shows an example (UML) statechart with a history pseudostate. It shows a region with a loop of transitions. Each time event x is thrown, the history pseudostate stores the current state the loop is in. When from state D, event y is thrown, the loop is reentered at that same state, and continues from there. Fig. 2.2 displays a translation of this loop, using (global) memory access. Each time, event x is thrown from some state pertaining to the loop (A, B, C), a memory write action stores the state that was left. Before the loop can be reentered, when event y is thrown, control passes through the H state, from which three guards check the memory for the last known state, the loop was in. This behavior matches the history state semantics.
We conclude this section with the following theorem:

Theorem 1 (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.*
We refer to the UML documentation [3] for a full description of the different OCL constraints on the UML metamodel.

3. ACTION ABSTRACTION

The ssc model definition gives access to following actions:

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

In Table 1, we divide the different programming constructs of the Java (action) language into seven classes. All

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

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. Memory control constructs belong to the omd viewpoint, and are abstracted away in the sc viewpoint. It remains to show the translation of the classes of constructs, marked "special" in Tab. 1.

Object method invocations are sequences of (smaller) instructions, and can be replaced by a *sequence of actions* from some of the other six classes of programming constructs. They 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 $(e, g, a) \in L$ [3]. Actions on entry and exit of each state of the UML sc, are put on all incoming, respectively outgoing transition labels, of that state. Subsequently applying the conversion described in the previous paragraphs 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.

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

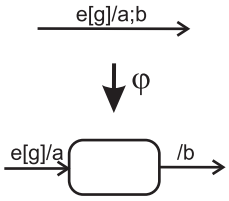

Figure 3: Decomposition of Action Lists in a Simplified Statechart Model

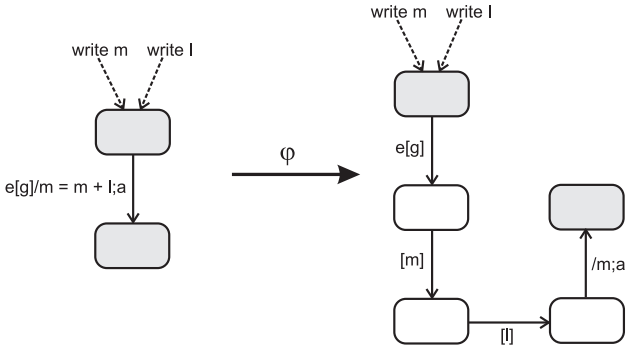

Figure 4: Side-Effects-Free Operations in a Simplified Statechart Model

new value, as memory write *actions*, without reference to any actual value. Figure 4 shows the translation of a sef operation taken together with an assignment, $e[g]/\mathbf{m = m + l}; a$. The most recent memory writes [6, 7], on memory locations 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 $/m$.

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 2 (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.*

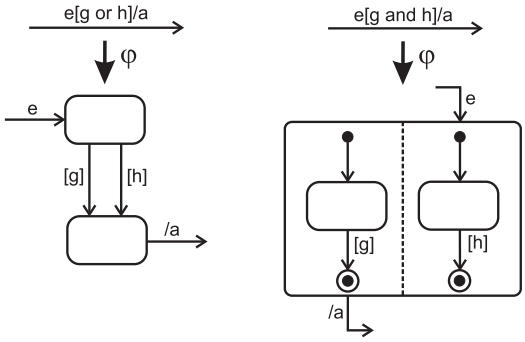

Figure 5: Decomposition of Boolean Connectives in a Simplified Statechart

Theorem 1 and thm. 2 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 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*. Refer to [8] for more information on category theory.

Let us now extend the definition of the ssc model as follows: An *extended simplified statechart*(exssc) is a tuple

$$\langle \Sigma, R, \delta, \delta', \rho, r_0, S, T, \iota \rangle$$

such that $\langle \Sigma, R, \delta, \delta', \rho, r_0, S, T \rangle$ is an ssc, and ι is a function

$$\begin{aligned}
&\iota = \textstyle\bigcup_{r \in R} \iota_r \cup \iota'_r \\
&\iota_r : r \times L \times r \to \lambda - \text{calculus expr.} \times \tau \\
&\iota'_r : r \times L \times \textstyle\bigcup_{r_i \in R \setminus \{r\}} r_i \to \lambda - \text{calculus expr.} \times \tau \\
&\tau = \{\sigma \mid \sigma : var(\lambda - \text{calculus expr.}) \to 2^{mL}\}
\end{aligned}$$

With *λ-calculus 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 [9]. τ is the set of all *bindings*. A *binding* σ lays a connection between the variables in the λ-expression, and the known memory values at that point in the execution (a thorough 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 translation morphism φ, defined in Sect. 3., the action language for sc models can be reduced to its most basic form, consisting of reads and writes, and the sc model 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 Sect. 5.. In this section we introduce a grammar rewrite system, inspired by the theory of *scenario composition*. 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 measured and put onto a complexity scale for sc models (see Sect. 5.). The rewrite system itself will allow us to generate random 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 [10] for more information. Scenarios can easily be integrated in the industrial design of software [11], but in order to gain the ability to specify executable systems with scenarios, we have need of a number of 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 [12, 13, 14, 15]. 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 *repeats* a scenario a number of times [12, 13]. 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. An *atomic ssc* $assc(e, g, a)$ is an ssc

$$\langle \{e, g, a\}, \{\{s, n, t\}\}, \delta, \delta', \rho, \{s, n, t\}, \{s\}, \{t\} \rangle$$

with one region, start pseudostate s, normal state n and final pseudostate t. δ' and ρ are empty, δ is defined as follows:

$$\delta(s, \lambda, \lambda, a) = n$$
$$\delta(n, e, g, \lambda) = t$$

We show that every ssc model is composed of a finite number of assc. Every ssc model has access to a finite number of different assc, by the finite alphabet Σ of Def. 2.. 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 binary *composition* operators $+$ and $\oplus$, a *lifting* operator $\rightarrow$, and a *wrapping* operator $[\ldots]_{assc(e,g,a)}$.

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. The different operators of sc DNA are explained as follows: *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 $\rho(s_0)$ of M_1 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

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

$Start$	$\rightarrow$	R	
R	$\rightarrow$	S	
R	$\rightarrow$	λ	
S	$\rightarrow$	$S + S$	
S	$\rightarrow$	$assc(e, m, a)$	for some $(e, m, a) \in L$
S	$\rightarrow$	$S \oplus \langle_d R, R_d \rangle_d \oplus S$	
S	$\rightarrow$	$S \rightarrow \langle_i [R]_{assc(e,m,a)} \rangle_i$	for some $(e, m, a) \in L$
S	$\rightarrow$	$S \rightarrow \langle_c R_c \rangle_c$	
R_d	$\rightarrow$	$[R]_{assc(e,m,a)}$	for some $(e, m, a) \in L$
R_d	$\rightarrow$	R_d, R_d	
R_c	$\rightarrow$	R_c, R_c	
R_c	$\rightarrow$	R	

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 this constraint also holds for the second operand M_2.

Composition by $\oplus$ glues one or more ssc, to more than one ssc and results in a 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 the loop or concurrent subregions that are lifted (see *lifting* below). We define this redistribution of transitions to be non-deterministic. Each sc DNA string therefore translates to a *class* of ssc models.

Lifted regions and loops point out where redistribution of transitions must take place. It also marks the ssc models M of which loops and encapsulated regions consist.

Wrapping completes missing label parts in the case of *disjunctive* and *iterative* composition by $+$ and $\oplus$.

Disjunctive composition is delimited by $\langle_d \ldots \rangle_d$, *iterative* composition by $\langle_i \ldots \rangle_i$ and *conjunctive* composition is marked with $\langle_c \ldots \rangle_c$. *Sequential* composition is implicitly present in the definition of the $+$ and $\oplus$ operators.

An example production string of the rewrite system in Table 2 ($assc$ is omitted for brevity):

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

translates to a *class* of ssc models. One of them is displayed in Fig. 6. The *wrapper* operation is displayed there as underlined transition label parts. Fig. 6 shows one *concurrency* site, one *iteration*, and two *disjunctive* sites, matching their counterparts in the example production string. Reading from the start pseudostate of the root region, to its final pseudostate, together with the example

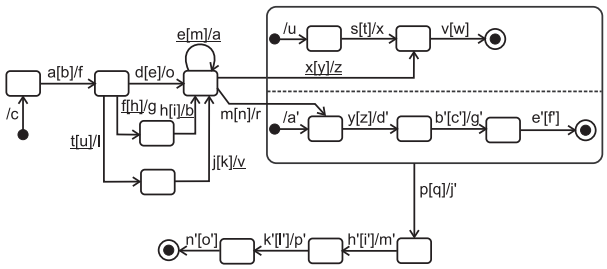

Figure 6: Translation of an Example Sc DNA

production string from left to right, we encounter:

1. first disjunctive site, consisting of three ssc models $\langle_d\ \lambda, [r_2]_{(f,h,b)}, [r_3]_{(t,u,v)}\ \rangle_d$, the first of which is empty;
2. iterative site, consisting of an (empty) ssc model, connected through a reflexive transition $\langle_i\ [\lambda]_{(e,m,a)}\ \rangle_i$;
3. second disjunctive site, consisting of two empty ssc models $\langle_d\ \lambda, [\lambda]_{(x,y,z)}\ \rangle_d$;
4. 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. 6, 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 elements.
A relatively complex procedure allows us to construct for each class of ssc models, the sc DNA. The proof of the following theorem builds on this procedure. Both will appear in full detail in [16].

Theorem 3 (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}. An ssc *complexity class* is the ssc model class returned by isomorphism ψ for some specific sc DNA string.
Following theorem is a consequence of thm. 3. We denote the set of all ssc models, allowed by Def. 2. as SSC, and the set of all sc DNA production strings with DNA.

Theorem 4 (Sc DNA Completeness)

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

The composed morphism $\psi^{-1} \circ \varphi$, by thm. 1, thm. 2, thm. 3 and thm. 4, allows us to abstract any UML sc model to its complexity class, represented by a unique sc DNA production string.

5. SCALABLE DEVELOPMENT

Further compression of sc DNA on a numeric scale allows us to define a complexity metric for UML sc models, taking into account the extensiveness of concurrent, iterative and disjunctive (conditional) executions, and the label *density* of the composing assc. Engineering their development, different versions of UML sc models can be evaluated by such a metric.
The rewrite system of Sect. 4. allows (automatic) generation sc model test cases, in different complexity classes. These are usable as a general purpose sc repository, and in gathering empiric evidence for sc model theories.
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 $assc(\ldots)$, and of λ, except those occurring in *wrapper* operations, with the 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* dna', in the parse tree for the rewrite system of Tab. 2. A *mutative* sc modification is obtained by a rewriting *through* any dna'' that is obtained by a permutation of two sequences of the form $+S \ldots S+$ in dna'. More invasive manipulations can also be constrained by the sc DNA framework. They are to appear in [16] together with a full explanation of the sc DNA framework.

6. CONCLUSION

Are action semantics a less essential part of UML statecharts? This paper argues that, if we abstract the action language to only its side effects (parallel memory operation), we retain enough information in the form of ssc models, to confirm this statement. The morphism φ, introduced in Sect. 3., translates UML sc models to ssc models. These reduce to sc DNA descriptions, by morphism ψ^{-1} of Sect. 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 and refactoring.

REFERENCES

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

[2] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987. [Online]. Available: citeseer.ist.psu.edu/harel87statecharts.html

[3] OMG, "Unified Modeling Language: Superstructure, version 2.1.1 (formal/2007-02-03)," February 2007. [Online]. Available: http://www.omg.org/docs/formal/07-02-03.pdf

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

[5] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition)*. Addison-Wesley Professional, August 2003.

[6] W. Pugh and T. Lindholm, "JSR-133: Java Memory Model and Thread Specification, final release," September 2004. [Online]. Available: http://www.cs.umd.edu/~~pugh/java/memoryModel/jsr133.pdf

[7] G. R. Gao and V. Sarkar, "Location Consistency-A New Memory Model and Cache Consistency Protocol," *IEEE Trans. Comput.*, vol. 49, no. 8, pp. 798–813, 2000.

[8] B. C. Pierce, *Basic Category Theory for Computer Scientists (Foundations of Computing Series)*. The MIT Press, 1991.

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

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

[11] R. Kelapure, M. A. Gonçalves, and E. A. Fox, "Scenario-Based Generation of Digital Library Services," in *ECDL*, ser. Lecture Notes in Computer Science, T. Koch and I. Sølvberg, Eds., vol. 2769. Springer, 2003, pp. 263–275.

[12] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios," in *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. New York, USA: ACM Press, 2000, pp. 314–323.

[13] S. Vasilache and J. Tanaka, "Synthesizing Statecharts from Multiple Interrelated Scenarios," Zheng Zhou, China, 2001. [Online]. Available: citeseer.ist.psu.edu/vasilache01synthesizing.html

[14] E. Makinen and T. Systa, "An Interactive Approach for Synthesizing UML Statechart Diagrams from Sequence Diagrams," in *OOP-SLA2000: Proceedings of the 10th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 2000.

[15] 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*, 2004, pp. 330–335.

[16] B. De Leeuw, "Statechart DNA: Formal and Practical Investigation in a Statechart Abstraction Method," Ph.D. Thesis UGent, December 2007.