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 introduce and explore a new statechart (sc) abstraction method and define a simplified statechart (ssc) model. We study the basic building blocks (so the term ‘DNA’) of UML sc models. Within this formal approach, we untangle the object-oriented concepts characterizing the UML. We treat triggers, guards and effects as related to each other, but make no reference to any explicit value (type) in the computation. This abstract approach allows us to introduce mathematical manipulations of UML sc, in line with the theory of regular automata. Statechart DNA has been applied in defining complexity metrics for UML sc models, the automatic generation of model test cases and behavior manipulations in CASE tool environments.

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 [1]. The language 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. We will study the benefits and applications of abstracting the UML action language for statecharts to a basic one, consisting only of event throwing actions, and memory reads and writes.

2 Statechart Definition

The UML sc is an evolution of the Harel sc [11]. It is a state machine model extended with constructs for hierarchical encapsulation and concurrent computation. The execution semantics are based on the queuing of events [16] and on the properties of some action language. We only keep the most essential constructs from the UML metamodel in our definition of simplified statecharts (ssc). In Sec. 3 we will show how UML statecharts can be converted to simplified statecharts through action abstraction. We assume that the reader has some familiarity with automata theory [13].

Definition 1 (Simplified Sc) A simplified statechart (ssc) $M$ is a tuple $M = (\Sigma, L, \delta, \delta', s_0, S, T)$, where $\Sigma$ 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 $L = eL \cup mL, eL \cap mL = \{\epsilon\}$. Here $\epsilon$ is the empty character. The functions $\delta$ and $\delta'$ define transitions between states. $\delta : \Sigma \times \Lambda \rightarrow \Sigma$ (intra-region transitions), $\delta' : \Sigma \times \Lambda \rightarrow 2^\Sigma$ (inter-region transitions), where $\Lambda = eL \times mL \times L$ is the set of all transition 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 $\Sigma$. The set $T$ consists of all terminate pseudostates of $\Sigma$.

An explicit construct for regions is lacking in Definition 1. We compose regions as collections of states. We build these collections from paths of states in the ssc model. We 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 $\Sigma$, such that there exists a list of labels $[l_1, l_2, \ldots, l_n]$ of $\Lambda$ for which $\sigma_0 \in S, \delta(\sigma_0, l_1) = \sigma_1, \delta(\sigma_1, l_2) = \sigma_2, \ldots, \delta(\sigma_{n-1}, l_n) = \sigma_n$. A composite path of an ssc model $M$, is a list $[\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n]$ of states of $\Sigma$, such that there exists a list of labels $[l_1, l_2, \ldots, l_n]$ of $\Lambda$ for which $\sigma_0 \in S, \delta(\sigma_0, l_1) = \sigma_1 \lor \sigma_1 \in \delta'(\sigma_0, l_1), \delta(\sigma_1, l_2) = \sigma_2 \lor \sigma_2 \in \delta'(\sigma_1, l_2), \ldots, \delta(\sigma_{n-1}, l_n) = \sigma_n \lor \sigma_n \in \delta'(\sigma_{n-1}, l_n)$.

Definition 3 (Region) A region of ssc $M$ from $s_i$, notation $\rho(M, s_i)$ 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 \( s_i <_h s_j \), 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 \( s_i \neq s_j \)).

Some properties are enforced on the UML sc meta-model through constraints, expressed in the Object Constraint Language (OCL). A detailed discussion of the OCL can be found in [22]. In [7] we reformulate these constraints as mathematical properties on the ssc model.

The different kinds of pseudostates are known to be shorthands for UML sc models built of more basic model elements of the UML metamodel [16]. With (global) memory access available (the set \( mL \) consists of memory locations, see Sec. 3), all pseudostates can be simulated by the basic model elements, or delegated to the action language. Since pseudostates can be converted this way, we close the section with following theorem, the proof of which matches all model elements of UML sc models with mathematical constructs in defining ssc models.

Theorem 5 (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.

3 Action Abstraction

UML sc models describe effects of transitions in some action language. The UML specification [16] defines formal classes of actions. One language supporting these abstract classes of actions is the Java programming language [10]. Contrariwise, the ssc model only allows following sorts of actions:

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

We divide the different programming constructs of the Java language into seven classes, more or less resembling the classes of actions of the UML metamodel. Some constructs like control flow, concurrency and event reaction are already implicitly present in ssc models. They would therefore be redundant inclusions in our atomic action language for ssc models. UML class diagrams model memory control. Assignment of variables, object invocation and side-effect-free operation are the remaining classes of actions that should be present in ssc models. Since the ssc model definition doesn’t support sequences of actions on transitions, it also 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 \((e, g, a) \in \Lambda \) [16]. States or vertices of UML sc models can have actions on entry and exit. We put all entry and exit actions on all incoming, respectively outgoing transition labels, of these states. In-state reactions are modeled as reflexive transitions on states. We apply a conversion procedure to UML sc models to arrive at intermediary sc models with transition labels consisting of action sequences of variable assignments and set operations. Fig. 1 shows how such 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.

Side-effect-free (set) 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 effects, without reference to any actual value. When a guard \([g]\) appears on a transition labeled \( e[g]/a, \) this transition should be interpreted as ‘dependent on the value of memory location \( g, \) action \( a \) will happen’. Figure 2 shows the translation of a set function taken together with an assignment, \( e[g]/m = m + l; a. \) The most recent memory writes [17, 8], on memory locations \( m \) and \( l, \) are shown with dotted lines in Fig. 2. The guard \([m]\) on the second transition of the right hand side of Fig. 2, 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 \( l, \) dependent 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 not different from sef operations, and are therefore abstracted in a similar fashion. Fig. 3 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.

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

**Theorem 6 (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 2: Side-Effects-Free Operations in an Ssc Model

Figure 3: Decomposition of Boolean Connectives in an Ssc Model

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

**Definition 7** An extended simplified statechart (exssc) is a tuple \( (\Sigma, L, \delta, \delta', s_0, S, T, \iota) \) such that \( (\Sigma, L, \delta, \delta', s_0, S, T) \) is an ssc, and \( \iota \) is a functor \( \iota = \iota \cup \iota', \iota : \Sigma \times \Lambda \times \Sigma \rightarrow \lambda\text{-calc.expr.} \times \tau, \iota' : \Sigma \times \Lambda \times 2^{\Sigma} \rightarrow \lambda\text{-calc.expr.} \times \tau, \tau = \{ b \mid b : \text{var}(\lambda\text{-calc.expr.}) \rightarrow 2^{mL} \} \).

With \( \lambda\text{-calc.expr.} \) we mean a Turing computable function specification [13]. In the case of exssc models, all \( \lambda \)-expressions are of the form \( \lambda gf. \) \( gf \epsilon \) with \( g \) referring to the guard and \( f \) denoting a sef operation. The set \( \tau \) consists of all bindings. A binding \( b \) lays a connection between the variables in the \( \lambda \)-expression, and the known memory values at that point in the execution. We use exssc to define the execution semantics of ssc. This way \( \varphi \) becomes 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 \( \varphi \) 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, and the sc model limited to its most basic constructs. This simplification makes it easy to compose and manipulate ssc models. In this section we introduce a grammar rewrite system, inspired by the theory of scenario composition [2]. 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. In this paper, we use the same combination rules as in the work on scenario composition [23, 21, 14, 12], but we apply them on atomic ssc instead of on scenarios. An atomic ssc (assc) represents the simplest conceivable ssc.

**Definition 8** An atomic ssc is an ssc \( \langle \{ e, e, g, a \}, \{ s, \sigma, t \}, \delta, \delta', s, \{ s \}, \{ t \} \rangle \) with one region, initial pseudostate \( s \), normal state \( \sigma \) and
terminate pseudostate \( t \). Transition function \( \delta \) is empty and \( \delta \) is defined as follows: \( \delta(s, e, \epsilon, a) = \sigma \), \( \delta(\sigma, e, \epsilon, a) = t \). We use the notation \( (a|e|g) \).

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 \( L \) 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. 1. 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).

We call the language defined by the production system of Tab. 1 statechart DNA, because it consists of strings describing how ssc models (comparable to cells) are composed of assc (comparable to proteins). 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 \( M_0 = M_1 + M_2 \). 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 then composed into one label \( e|g\rightarrow a \), 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. 1 guarantees that this constraint holds.

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. Redistribution of transitions is necessary to cover all ssc models with sc DNA. A detailed description of the implications of composition by \( \oplus \) is beyond the scope of this paper.

3. lifting of encapsulated regions and loops is necessary to properly define where redistribution of transitions may take place. It also marks the ssc models \( M \) of which loops and encapsulated regions consist.

4. wrapping completes missing label parts in the case of disjunctive and iterative composition by + and \( \oplus \).

Disjunctive composition is delimited by \( (q_1 \ldots q_n) \), iterative composition by \( (\ldots j \ldots) \), and conjunctive composition is marked with \( (c_1 \ldots c_n) \). Sequential composition is implicit by the definition of the + and \( \oplus \) operators.

An example production string of the rewrite system in Tab. 1: \( (\epsilon|a|b) + (f|d|e) \)
\( \oplus \) \( (i|g|h) \)
\( \oplus \) \( (l|j|k) \)
\( \oplus \) \( (r|m|n) \)
\( \oplus \) \( \epsilon \rightarrow \) \( \lambda \) \( (a|e|g|) \) \( d \)
\( \oplus \) \( (p, q, r) \)
\( \rightarrow \) \( (l|j|k|) \)
\( \oplus \) \( (m'|k'|) \)
\( \oplus \) \( (p'|m'|) \)
translates to a class of ssc models, because of the different possibilities of redistributing transitions. Figure 4 shows the translation of this sc DNA string to one of its possible ssc model representations. The wrapper operation is displayed there as underlined transition label parts. Figure 4 shows one conjunctive group of transitions, one iteration, and two disjunctive transition groups, matching their counterparts in the example production string. Reading the example production string from left to right and the ssc diagram of Fig. 4 from the root state to the terminate pseudostate of the same region, we encounter (variables \( r_j \) are shorthands for non-empty constructs, \( \lambda \) represents empty transition groups)

1. disjunctive transition group, consisting of three ssc models \( (\lambda | r_2 | (b|f|h)) \), \( (r_3 | (e|t|u)) \), the first of which is empty.
implements the inverse morphism \( \psi \) from ssc model descriptions. This procedure allows us to distill ssc models. A complex procedure allows us to distill the class of ssc models translated from this production string consists of 64 ssc models. The translation of sc DNA to ssc models therefore defines a morphism \( \psi \) between sc DNA strings and classes of ssc models. A complex procedure allows us to distill sc DNA from ssc model descriptions. This procedure implements the inverse morphism \( \psi^{-1} \).

5 Statechart DNA Experimentation

We have been working ([3], [6], [4], [5], [7]) 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 intenseless 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

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, and in gathering empirical evidence for sc model theories ([20], [18], [19]).

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 \( \lambda \), 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 \( dna' \), in the parse tree for the rewrite system of Tab. 1. A mutable 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 \( dna'' \).

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 changes 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 SD 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. SD DNA represents state machines based on a formal grammar and therefore sensible automatic model merges of different state machine versions can be proposed as follows.

We showed how every state machine can be represented as an SSC (with morphism $\varphi$). An SSC can be compiled into an SD DNA string. This form can be made normal by ordering comma separated R-lists of SD DNA from smallest to biggest number of atoms and 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 [9], 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 SD DNA is now in 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. 5 shows two state machine model versions on which an insert (or dually, a delete) has been performed.

We use following technique to visualize and automatically merge both models. They are compiled into SD DNA. Applying the Diff-function on both normalized and serialized parse trees can therefore show additional and removed grammar steps in between SC DNA versions under comparison. Automatic merges can now be done if there is an ordering available on the SD DNA grammar rules (e.g. 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).

Edge 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. 5 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 SSCs matched to the Diff-sets of both XMI representations will show (and visualize) where redistributions have happened. Redistributions can again be assigned weights 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 be determined similarly 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.

7 Conclusion

In this paper, we confirm that the effects of UML sc models can be abstracted, and that they are but a secondary complexity determining construct of the rich UML model. The morphism $\varphi$, introduced in Sec. 3, translates UML sc models to ssc models. These reduce to sc DNA descriptions, by morphism $\psi^{-1}$ of Sec. 4. Sc DNA allows us on the one hand, to partition ssc models, and therefore also UML models, into complexity classes, which gives 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:


