Statechart Normalizations

BENJAMIN DE LEEUW AND ALBERT HOOGEWIJS

Ghent University
Department of Pure Mathematics
Gent B-9000,
BELGIUM
benjamin.deleeuw@ugent.be, albert.hoogewijs@ugent.be

Abstract: Simplified statecharts are derived by excluding all redundant constructs of the UML (Unified Modeling Language) metamodel on statecharts. In previous papers we introduced this basic concept and pointed out some interesting applications. We transformed and compacted state machines into serializable objects that clearly highlight their basic constructs and showed how a lot of sparse edges were created during this transformation. Sparse edges contain little or no information (e.g. empty transitions). From this compaction we derived a theory in which state machines are reduced to two distinct partial orderings on states. For the sake of convenience to read this paper, we briefly recapture part of this theory and show the exponentially growing complexity of calculating all possible values of variables that appear on state machine edges. In order to arrive at feasible algorithms leading to practical applications of the implicity complex partial ordering relations we need to reduce this complexity by formulating and proving reductions to state machine normal forms. Apart from aligning formal and behavioral equivalence, normal forms allow us to reduce the number of sparse edges and useless states thereby limiting calculational complexity.

In this paper, we extend our theory with *normal forms* and inject them into the theory of simplified statecharts presented in earlier work. Some statecharts are only seemingly different from others if one analyzes the different paths in those statecharts. The UML is designed to allow for this kind of (uncontrollable) flexibility but in mathematical descriptions it has adverse effects. We introduce an equivalence relation on simplified statecharts and derive a normalization procedure which converts a simplified sc to a normalized simplified sc equivalent to the original one. Our formalism allows us to unravel superficial differences between simplified statecharts.

Key–Words: Statecharts, Statechart Normalization, UML, Model Checking, State Machine Theory

1 Introduction

In [10] Hunyadi et al. argue “although UML [2] 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, in [6, 7, 8] we introduced the notion of simplified statecharts (ssc) which allows us to introduce mathematical manipulations of the UML sc. We defined a homomorphic mapping (similar to Levendovszky et al. in [11]) transforming standard UML statecharts into so-called simplified statecharts, statecharts with abstracted actions. This abstraction induces a mathematical definition of statecharts, similar to automata and a formal grammar and language, called statechart DNA. Relying on graph rewriting principles as introduced in [11] we get a statechart construction process and a formal as well as practical way to scalably manage complexity and object behavior. The main motivation for this research was the generation of a repository of test cases for verification tools and a versioning management tool for state machine models. We made strong abstractions of the action language, normally offered for UML statecharts, to induce sensible and strict definitions, and finitary properties. The normal UML statechart executable model has been matched to the executable model of simplified statecharts [8], and semantics was given to the simplified action language.

In this paper we propose an equivalence relation on simplified statecharts by deriving a normalization procedure which converts a simplified sc to a normalized simplified sc equivalent to the original one.

In order to make this paper self-consistent, we recall the basic results from [8]. We start with the definition of a simplified statechart:

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

$$M = \langle \Sigma, L, \delta, \delta', s_0, S, T \rangle, \tag{1}$$

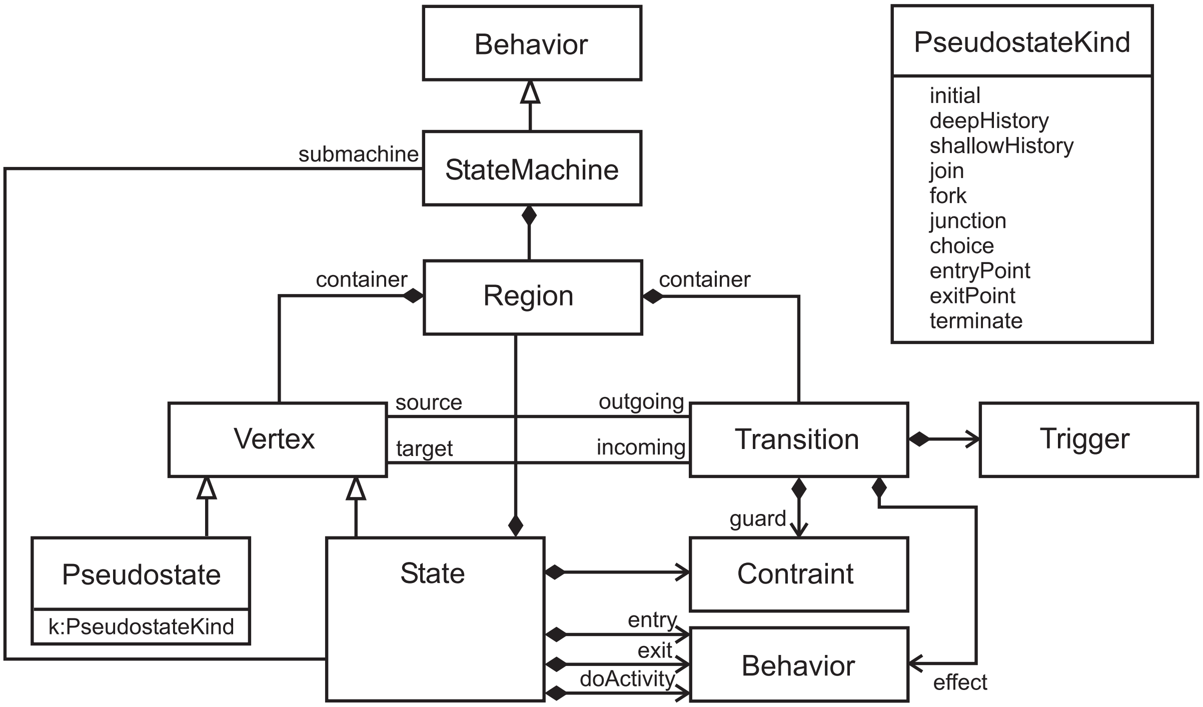

Figure 1: Partial UML 2.0 Metamodel Defining Statecharts

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*

$$\begin{aligned} L &= eL \cup mL && (2) \\ eL \cap mL &= \{\epsilon\}. && (3) \end{aligned}$$

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

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

where

$$\Lambda = eL \times mL \times L \qquad (4)$$

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.

2 Reducing Cycles within Simplified Statecharts

In [6], we have defined two different kinds of *paths* in simplified statecharts. Here we will introduce *executions* as an extension to complex paths or c-paths within state machines and try to enumerate all possible ones to arrive at a formal technique useful for model checking this kind of state machine.

Definition 2 *A* simple path *(s-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*

$$\begin{aligned} \delta(\sigma_0, l_1) &= \sigma_1 && (5) \\ \delta(\sigma_1, l_2) &= \sigma_2 && (6) \\ &\ldots \\ \delta(\sigma_{n-1}, l_n) &= \sigma_n && (7) \end{aligned}$$

A composite path *(c-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*

$$\begin{aligned} \delta(\sigma_0, l_1) &= \sigma_1 \vee \sigma_1 \in \delta'(\sigma_0, l_1) && (8) \\ \delta(\sigma_1, l_2) &= \sigma_2 \vee \sigma_2 \in \delta'(\sigma_1, l_2) && (9) \\ &\ldots \\ \delta(\sigma_{n-1}, l_n) &= \sigma_n \vee \sigma_n \in \delta'(\sigma_{n-1}, l_n) && (10) \end{aligned}$$

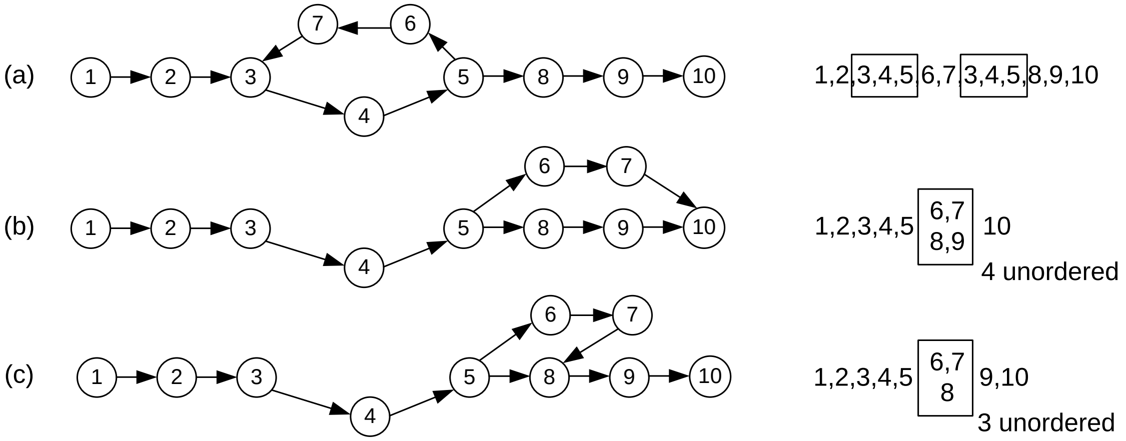

Figure 2: Maximal Partial Ordering on s-Paths

In this section we formulate a new definition of paths in statecharts, taking concurrency into account. We call these paths concurrent composite paths or cc-paths for short. An execution is partially ordered set of labels $\{l_1, l_2, \ldots, l_n\}$ of Λ corresponding to some cc-path from s_0 to t_0 of the root region $\rho(s_0)$ of the simplified sc. We use the known technique of partial orders [12, 9] (instead of linear orders) to arrive at a sound representation of concurrent paths and the means to reason about value propagation in simplified statecharts.

Before we can talk about partial orderings we need to get rid of any cycles in the state machine model M. In [6] we introduced the notion of *regions* as a function ρ on the set of initial pseudostates S. For each region $\rho(s_i)$ we construct the relation R_{s_i} from δ of M as follows:

Definition 3 *For each $s_i \in S$ and $\sigma, \sigma' \in \rho(s_i)$*

$$\sigma R_{s_i} \sigma' \Leftrightarrow \exists l \in \Lambda \quad \delta(\sigma, l) = \sigma' \tag{11}$$

In order to arrive at a valid partial order relationship between states in a simplified sc we introduce a well known procedure to remove cycles from an arbitrary relation R.

Procedure 4 (reduce cycles)

```
Determine the minimal elements $M_n$ of $R$,
determine the maximal elements $M_x$ of $R$ and
stack $(0, M_n)$.
While stack not empty
    pop the expression $(e, O)$.
    For each element $f$ of $O$
        if $f$ does not appear on the stack as $(f, X)$
        then push $(f, \{x \mid R(f, x)\})$ on the stack
        else remove $(e, f)$ from $R$.
Determine the minimal elements $M'_n$ of red. $R$,
determine the maximal elements $M'_x$ of red. $R$,
$R = R \cup (M_n \times M'_n \setminus M_n)$ and
$R = R \cup (M'_x \setminus M_x \times M_x)$.
```

Applying Proc. 4 to all R_{s_i} for each s_i in S, returns the partial orderings R'_{s_i}. The union of all R'_{s_i} for each s_i in S is denoted R'_S. From the enumerable set of all s-paths we can now construct the finite set of partially ordered and non-repeating simple path states sets or the finite set of *nrs-posets* for short. To define nrs-posets we need following trivial property.

Property 5 *If partial orderings $R_0, R_1, R_2, \ldots$ on the same set can themselves be linearly ordered according to the $\subseteq$ relationship we can always define a maximum and minimum of this relationship ordering. The maximum has the least unordered elements and the minimum has the most unordered elements.*

Applying Prop. 5 with minimal element R'_{s_i} (with most unordered elements) allows us to define some

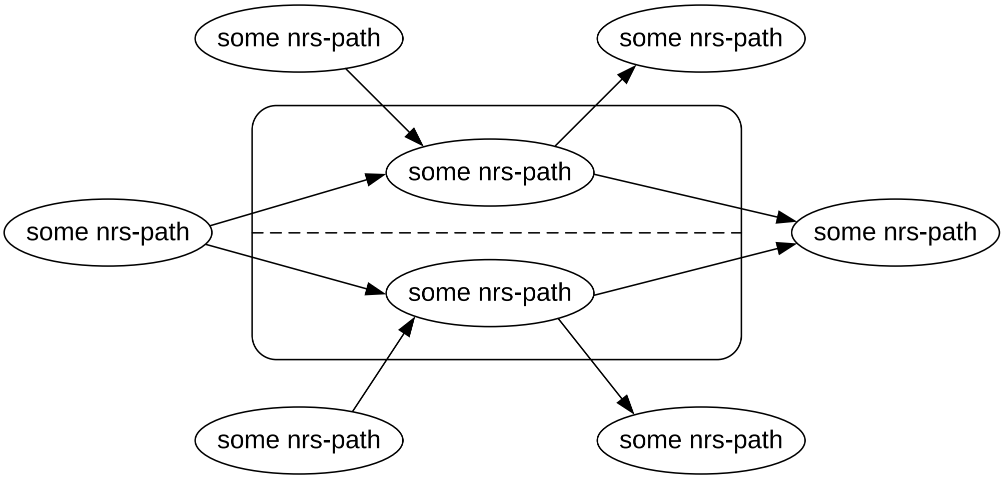

Figure 3: R_c Ordering nrs-Posets in between State Machine Regions

relationship R''_{s_i} for which the number of unordered elements is minimal. Refer to Fig. 2 for how this relation would manifest itself if applied on some s-path. Fig. 2a shows an s-path with repeating states, Fig. 2b displays the reduced relationship R'_{s_i} and Fig. 2c shows the maximal partially ordered relationship R''_{s_i}. The union of all R''_{s_i} relations, for all s_i of S, is denoted R''_S.

$$R''_S = \bigcup R''_{s_i} \qquad (12)$$

Once the relation R''_S is determined we can construct a finite set of equivalence classes of the enumerable set of s-paths as follows:

Definition 6 *Two or more s-paths belong to the same equivalence class if R''_S transforms them into the same partially ordered set of states. Each equivalence class that can be constructed in this way is called an* nrs-poset.

We call each path within an nrs-poset, a non-repeating s-path or *nrs-path* for short. Remark that the set of nrs-paths is a subset of the s-path set by Def. 6. In the remainder of this paper we will refer to the R''_S relation (12) as $<_s$.

Each s-path is also a c-path and those c-paths in M that are not s-paths, define a relation on s-paths by δ' applications (see [6]). In order to reduce all cycles within M and come to executions we need to reduce the cycles within c-paths as well. For this we define the relation R_c:

Definition 7

$$\sigma R_c \sigma' \Leftrightarrow \sigma <_s \sigma' \vee \exists l \in \Lambda \quad \sigma' \in \delta'(\sigma, l) \qquad (13)$$

From the definition of ssc and from $<_s$ we know that we now have a situation as displayed in Fig. 3 with the oval regions, denoting equivalence classes of s-paths according to $<_s$ and the arrows denoting applications of the relation R_c that are not in $<_s$. R_c can therefore conveniently be imagined as a relation between nrs-posets.

The relationship R_c can have cycles as well but before reducing these we first introduce a derived function δ'' of M and construct the relationship R_C. In this definition we use the partial ordering relation $<_h$ of the region hierarchy of M introduced in [6].

Definition 8 *Function δ'' is a derived function of M*

$$\delta'' : \Sigma \times \Lambda \rightarrow 2^{\Sigma} \textit{ (inter-region transitions).}$$

For each σ in $\rho(s_i)$ that is not in T and with s_i an element of S, if σ' belongs to the set $\delta'(\sigma, l)$ for some label l of Λ and σ' is part of some $\rho(s_j)$ with $s_j <_h s_i$ then for each s_k of S for which $\neg(s_k <_h s_i)$ and $\neg(s_i <_h s_k)$ and for each σ'' that is part of the set $\rho(s_k)$ add the application $\delta''(\sigma'', l) = \{\sigma'\}$.

For each σ in $\rho(s_i)$ with s_i an element of S, if σ' belongs to the set $\delta'(\sigma, l)$ for some label l of Λ and σ' is part of some $\rho(s_j)$ with $s_i <_h s_j$ and σ' is not an element of S then for each s_k of S for which there exists an s_l in S with $s_l <_h s_i$ and with s_k a direct

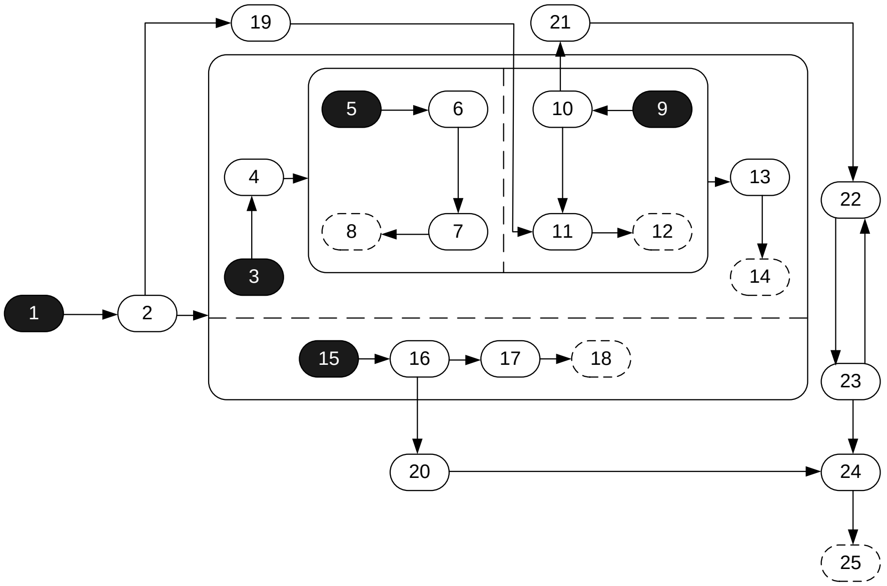

Figure 4: A Complex Statechart M, without Labels

child of s_l in the tree-order $<_h$ then add the application $\delta''(\sigma, l) = \{s_k\}$. If s_{k_1}, s_{k_2} and s_{k_3} all fulfill this constraint, add the $\delta''(\sigma, l) = \{s_{k_1}, s_{k_2}, s_{k_3}\}$ application instead.

We need the derived δ'' function in our path orderings, because it models the activation of concurrent state machine regions due to inter-region transitions that are not on initial or terminate pseudostates. The UML state machine semantics [2] allows this kind of activation, hence we need to account for it in our model of executions. Note that the size of the set of δ'' applications for one ssc model M might be quite large. Mainly due to the intrinsic complexity of this kind of region activation, we need disentanglement of statecharts. With the introduction of δ'' we can define the relationship R_C which is an extension of R_c.

Definition 9

$$\begin{aligned} \sigma R_C \sigma' \Leftrightarrow & \ \sigma <_s \sigma' \vee \\ & \exists l \in \Lambda \quad \sigma' \in \delta'(\sigma, l) \vee \\ & \exists l \in \Lambda \quad \sigma' \in \delta''(\sigma, l) \end{aligned}$$

We apply Proc. 4 to the relationship R_C between nrs-posets and in doing so construct the relationship R'_C. Again we apply Prop. 5 on R'_C to construct the relationship R''_C between nrs-posets in exactly the same way as we did for R''_S. This way we are able to construct equivalence classes of nrs-posets.

Definition 10 *Two or more paths consisting of nrs-posets ordered by R_C belong to the same equivalence class if R''_C transforms them into the same partially ordered set of nrs-posets. Each equivalence class constructed in this way is called an nrc-poset.*

Bringing $<_s$ and R''_C together results in a partial ordering on states of M constrained by both relationships: first constrained to nrs-posets and second to nrc-posets. We call this ordering $<_c$ and nrc-paths are c-paths that are constrained by this ordering. Note that the set of all nrs-paths is a subset of the set of all nrc-paths just like s-paths and c-paths respectively and that $<_s$ is a subset of $<_c$.

The remaining discussion in the next section will be about characterizing executions of a simplified sc in terms of the partial order $<_c$.

3 Value Propagation through Simplified Statecharts

Before talking about executions in a ssc model M we need to introduce more vocabulary.

Definition 11 *A* concurrent set *of state σ of M is a subset of the set $(\delta' \cup \delta'')(\sigma)$. If σ' belongs to the concurrent set of σ and σ' belongs to $\rho(s_i)$ for some s_i of S then all s_j of S that are also part of $(\delta' \cup \delta'')(\sigma)$, for which there exists an s_k of S with $s_k <_h s_i$ and for* *which s_j is a direct child of s_k in the tree-order $<_h$ are also in the concurrent set. We denote a concurrent set of σ with $cset(\sigma, M)$. The set of all concurrent sets for some σ of M is called $Cset(\sigma, M)$.*

Definition 12 *The set $nrs_{min}(\sigma, M)$ is the set of all nrs-paths of M containing σ as minimal element.*

The set $nrs_{con}(N, M)$ is the set of all nrs-paths of M containing at least all states of the set N.

The set $nrs_{max}(\sigma, M)$ is the set of all nrs-paths of M containing σ as maximal element.

We now have introduced all mathematical material to construct all executions or cc-paths of an ssc model M. Before tackling the procedure to construct the set E of all cc-paths of model M we explain the mechanics. The set E is a powerset of Σ of M and each set in E is partially ordered by $<_c$. Initially the set E is empty and at the end of the procedure application it will contain all calculated cc-paths of M.

We denote the maximal element of an nrs-path p with $max(p)$. To know which nrs-path we are on in each step of the procedure we have to store the current nrs-path together with the complete set of states of M that constitutes to the cc-path we are constructing as a pair $< p, q >$ with p the current path we are evaluating and q the growing set of states (starting from initial pseudostate s_0, and implicitly ordered by $<_c$) at that point in the execution of the procedure.

Since an nrs-path is also a (linearly ordered) set, we conveniently use the notation $p \cup q$ to show the addition of the states of path p to the set q. Dropping the order of p when adding to q doesn't matter since we know that p's ordering is implicitly present in the ordering $<_c$ of M. We use a set P_σ of pairs $< p, q >$ to group different paths starting from the same state σ. We need to group the sets P_σ when we apply $cset(\sigma, M)$ in the procedure and use the notation Q'_σ for this purpose, with each σ' representing a different state in the concurrent set $cset(\sigma, M)$. Because the $Cset(\sigma, M)$ of one state σ might have more than one element (see Def. 11) we are in need of yet another level of grouping (of sets Q'_σ this time) and use the sets R_i for this purpose, with each i representing a different set of concurrent sets in $Cset(\sigma, M)$.

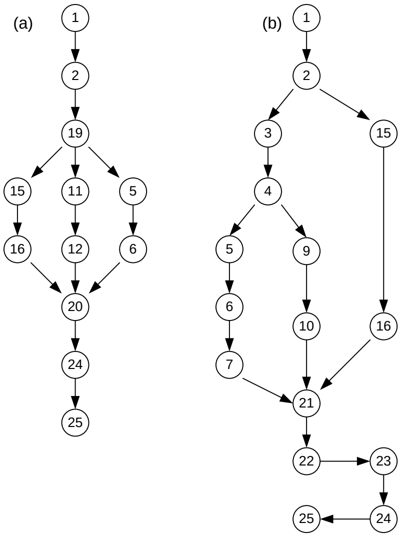

Figure 5: Two reductions of the example in Fig. 4

Procedure 13 (construct cc-paths)

```
for each path $p$ of $nrs_{min}(s_0, M)$
  add $<p, p>$ to the set $P_{s_0}$,
  add $P_{s_0}$ to $Q_{s_0}$,
  add $Q_{s_0}$ to $R_0$ and
  stack $R_0$.
While stack not empty pop $R_i$.
  For each $Q_\sigma$ in $R_i$,
    for each $P_\sigma$ in $Q_\sigma$ and
      for each $<p, q>$ in $P_\sigma$,
        if $max(p) \in \rho(s_0) \cap T$
        then add $q$ to the set $E$,
        else for each c-set $c$ of $Cset(max(p), M)$,
          for each $\sigma'$ of c,
            for each path $p'$ of $nrs_{min}(\sigma', M)$
              add $<p', q \cup p'>$ to the set $P_{\sigma'}$
            add non-empty $P_{\sigma'}$ to $Q_{\sigma'}$,
          add non-empty $Q_{\sigma'}$ to $R_j$ and
          stack non-empty $R_j$.
  For each $Q_\sigma$ in $R_i$,
    for each choice $g$ of one $<p, q>$ of each $P_{\sigma'}$ of $Q_\sigma$,
      for each label $l$ that is applicable with
        $N_l = nrs_{con}(\bigcup_g \delta' \cup \delta''(max(p), l), M)$,
        for each path $p' \in N_l$,
          add $<p', q \cup p'>$ to $P_{\sigma''}$,
      add non-empty $P_{\sigma''}$ to $Q''_\sigma$,
    add non-empty $Q''_\sigma$ to $R_j$ and
    stack $R_j$.
```

The set E contains all partially ordered cc-paths of M.

Proposition 14 *By Proc.13 every well formed simplified statechart model M can be transformed into a set of partially ordered cc-path, called CCP_M. This set is always unique and has a finite number of elements.*

Fig. 4 shows a simplified sc model M, and Fig. 5 shows two possible reductions of $e \in CCP_M$.

We introduce an extended technique to further reduce the cc-paths in the set CCP_M, for some simplified sc model M. This technique will be very similar to the one used in memory model specification [12], and is based on partial ordering information on the control flow in statecharts. The technique that is described there can easily be transfered to our system now that we possess a way of reducing a complex state machine to all of its partially ordered cc-paths and executions.

The transitive reduction of $<_c$ is called $\prec_c$ and can easily be constructed for every cc-path in CCP_M. By the definition of $<_c$ we know that for each $\sigma \prec_c \sigma'$ there exists a unique application of either δ or δ' for a certain label l of Λ for which $\delta(\sigma, l) = \sigma'$ or $\sigma' \in \delta'(\sigma, l)$ respectively.

Definition 15 *The* trace *of M for any cc-path e of CCP_M, denoted as $trace(e, M)$, is an ssc model $M' = \langle \Sigma_e, L, \delta_e, \delta'_e, s_0, S_e, T_e \rangle$ for which Σ_e is a subset of Σ consisting of the set of states in cc-path e, S_e is a subset of S composed of the initial pseudostates in e and T_e is a subset of T of the terminate pseudostates in e.*
$\delta_e \ : \Sigma_e \times \Lambda \to \Sigma_e$

$$\delta_e(\sigma, l) = \sigma' \Leftrightarrow \sigma \prec_c \sigma' \wedge \delta(\sigma, l) = \sigma' \tag{14}$$

$\delta'_e : \Sigma_e \times \Lambda \to 2^{\Sigma_e}$

$$\sigma' \in \delta'_e(\sigma, l) \Leftrightarrow \sigma \prec_c \sigma' \wedge \sigma' \in \delta'(\sigma, l) \tag{15}$$

and $\Lambda = eL \times mL \times L$ remains unchanged from M.

All cc-paths e of M have a single trace and if we calculate all cc-paths of the ssc $trace(e, M)$ we retain a single cc-path e. Each application of $\prec_c$ is called an *edge* of $trace(e, M)$. The set of all edges of cc-path e is called $edge(e, M)$, the set of edges for any nrs-path p is denoted with $edge(p, M)$ and the set of all edges of all cc-paths of CCP_M is denoted with $edge(M)$. Remark that the ordering relation $<_c$ can in a trivial way be transfered to the elements of $edge(p, M)$ and $edge(M)$.

There are two kinds of guards in (simplified) statecharts: *constraining* guards, and *non-constraining* guards. Constraining guards freeze the evolution to the next stable configuration, on some path in the sc, until a certain memory condition is met. This condition is dependent on the random access writes which already have been executed on previous transitions. In terms of our morphism φ of [6], constraining guards are the result of the translation of normal UML guards like $[x < 5]$. Only a limited number of memory actions will satisfy such a guard (for example $/x = 7$ will satisfy this guard, while $/x = 1$ will not). Non-constraining guards are normal reads of variables of which we need the content. The theory of simplified state machines treats both kinds of guards equivalently in the sense that if no value is found for a non-constraining guard the state machine halts its execution. Therefore each guard is considered constraining in a simplified sc model M.

We divide the set $edge(M)$ into (not necessarily disjoint) sets $edge_w(M)$ and $edge_r(M)$ of edges that write and read some value respectively. First we define an accesor function for the labels:

$$lbl : edge(M) \to \Lambda$$

which returns for each edge its label. We now introduce a new relation $gsat$ between guards and memory write actions.

Definition 16

$$gsat : edge_w(M) \times edge_r(M)$$
$$gsat(e_1, e_2) \Leftrightarrow$$

$$\begin{aligned} lbl(e_1) &= (t, m, a), & (16) \\ lbl(e_2) &= (t', a, a') & (17) \end{aligned}$$

and action a of e_1 satisfies the guard a of e_2.

Non-constraining guards are always satisfied by any memory write action to the same location a of mL. For constraining guards, the $gsat$ relation becomes an essential part of the information to calculate legal executions. It can be decided from the original UML statechart or can be considered an execution property of some simplified sc.

We introduce another relation over memory actions and guards, namely the *most recent write* relation ([9], mrw), for every guarded edge of cc-path e in CCP_M we construct the mrw relation as follows:

Definition 17

$$mrw : edge_r(M) \times edge_w(M)$$
$$mrw(e_1, e_2) \Leftrightarrow$$

$$\begin{aligned} lbl(e_1) &= (t, m, a) & (18) \\ lbl(e_2) &= (t', m', m), & (19) \end{aligned}$$

$$\neg(e_1 <_c e_2)$$

and no $e_3 \in edge(M)$ *exists with*

$$\begin{aligned} lbl(e_3) &= (t'', m'', m) \quad (20) \\ e_2 <_c & \, e_3 <_c e_1 \end{aligned}$$

The mrw relation points out which write actions on a certain variable can happen before some guard on that variable is encountered. The most recent write relation was first introduced in [9] and later adapted for the Java programming language in [12]. In a similar way we compose the $mrwgsat$ relation:

Definition 18

$$\begin{aligned} & mrwgsat : edge_r(M) \times edge_w(M) \\ & mrwgsat(e_1, e_2) \Leftrightarrow mrw(e_2, e_1) \wedge gsat(e_1, e_2) \end{aligned}$$

If we leave out event semantics of simplified sc we can now constrain the set of cc-paths, CCP_M, by defining that every legal cc-path e must possess for every guard on every edge e_i in $edge(e, M)$ at least one edge e_j of $edge(e, M)$ satisfying the relation $mrwgsat$.

In order to consider events in this schema, more work needs to be done.

Definition 19 *An* interleaving *of a set of linear ordered sets*

$$N = \{(L_1, <_i), \ldots, (L_n, <_n)\} \quad (21)$$

is a linear ordered set $(L, <)$ *consisting of all elements of*

$$L_1 \cup \ldots L_n$$

and with $<$ *constrained as follows:*

$$\forall l_1, l_2 \in L_i \quad l_1 <_i l_2 \Rightarrow l_1 < l_2$$

The set of all interleavings *of set* N *of linearly ordered sets is denoted with* $il(N)$*.*

To construct an interleaving of a partial order, we need to isolate the linear ordered strings in that order, and apply the definition of il recursively. This can only be done if the partial order is *connected*.

Definition 20 *A partially ordered set* $(L, <)$ *is* connected *if* L *has a unique minimal (*min*) and maximal (*max*) element and for each other element* $l \in L$ *holds that* $min < l < max$*.*

In particular, we need to identify the "deepest" choice points (defined below as *sync-in* and *sync-out*) in the connected partial order, and start the recursion of il on linear ordered paths, between these identified states. We will use the notion of an *interval* again, in a partially ordered set, to access the elements which lie between two choice points.

Definition 21 *Given a partially ordered set* $(L, <)$*, for each* $l_1, l_2 \in L$*, the interval* $]l_1, l_2[$ *is defined as a set of elements*

$$]l_1, l_2[\; = \{l \in L \setminus \{l_1, l_2\} \mid l_1 < l < l_2\} \quad (22)$$

and for each $l, l' \in \;]l_1, l_2[$ *holds that* $l < l'$ *or* $l' < l$*.*

Definition 22 *A* sync-in point *of a partially ordered connected set* $(L, <)$ *is an element* l *of* L *for which there exist* $l_1, \; l_2$ *in* L *with* $l < l_1$*,* $l < l_2$*,* $\neg(l_1 < l_2)$ *and* $\neg(l_2 < l_1)$*.*

A sync-out point *of a partially ordered connected set* $(L, <)$ *is an element* l *of* L *for which there exist* l_1, l_2 *in* L *with* $l_1 < l$*,* $l_2 < l$*,* $\neg(l_1 < l_2)$ *and* $\neg(l_2 < l_1)$*.*

The closest sync-out point *for every* sync-in point l *of* L *is a* sync-out point l' *for which holds that* $l < l'$ *and there is no* l'' *in* L *for which* $l < l'' < l'$ *and* l'' *is a* sync-out point *hold at the same time.*

The closest sync-in point *for every* sync-out point l *of* L *is a* sync-in point l' *for which holds that* $l' < l$ *and there is no* l'' *in* L *for which* $l' < l'' < l$ *and* l'' *is a* sync-in point *hold at the same time.*

A deepest sync-in point *of a partially ordered set* $(L, <)$ *is a* sync-in point l *of* L *for which also holds that, given its* closest sync-out point l' *there doesn't exist an* l'' *in* $]l, l'[$ *with* l'' *a* sync-in point.

A deepest sync-out point *of a partially ordered connected set* $(L, <)$ *is a* sync-out point l *of* L *for which also holds that, given its* closest sync-in point l' *there doesn't exist an* l'' *in* $]l', l[$ *with* l'' *a* sync-out point.

A deepest sync pair *is a pair* (l_i, l_o) *of* deepest sync-in point l_i *and* deepest sync-out point l_o *with* $l_i < l_o$*.*

We now define a procedure returning the interleaving of any connected partially ordered set $(L, <)$.

Procedure 23 (interleaving)

```
If no deepest sync pair can be found in $(L, <)$
then store the ordered set $(L, <)$ in $il(L, <)$
For each deepest sync pair $(l_i, l_o)$ of $(L, <)$
    determine all intervals $]l_i, l_o[$ in $(L, <)$,
    store all $]l_i, l_o[$ in the set $N_{l_i,l_o}$,
    remove all elements of all $]l_i, l_o[$ from $(L, <)$ and
    store $il(N_{l_i,l_o})$ in $I$.
For each choice $c$, one of each set $il(N_{l_i,l_o})$ in $I$
    $(L_i, <) = (L, <) \cup \bigcup_c ]l_i, l_o[$ and
    store $(L_i, <)$ in $J$.
For each $(L_i, <)$ in $J$
    apply the procedure interleaving.
```

The set of all possible interleavings determined by this procedure is called $il(L, <)$. Following proposition can easily be verified:

Proposition 24 *The partially ordered set $edge(e, M)$ for each cc-path e of M is connected.*

With the definitions of the accessor functions for triggered and event generating action edges we will be able to construct the set of all legal cc-paths for any ssc model M.

Definition 25 *The* trigger list *of a linearly ordered set of edges is a linearly ordered set of events (e in $eL \setminus \{\epsilon\}$ of M) such that for each edge e_i, with $lbl(e_i) = (e, m, a)$, there is a corresponding element e in tl such that the label function is order preserving. The* trigger list *of a linearly ordered set E of edges is accessed with $tl(E)$.*

The generator list *of a linearly ordered set of edges is a linearly ordered set of events (e in $eL \setminus \{\epsilon\}$ of M) such that for each edge e_i, with $lbl(e_i) = (e, m, a)$ and a of $eL \setminus \{\epsilon\}$, there is a corresponding element a in gl such that the label function is order preserving. The* generator list *of a linear ordered set E of edges is accessed with $gl(E)$.*

For each $trace(e, M)$ and concordant connected partially ordered set $edge(e, M)$ it is possible to remove all elements between any sync-in point and its closest sync-out point as well as all elements that are ordered to any element of this path to arrive at a partially ordered set of edges consisting of one or more connected partially ordered sets. We denote the set of these connected partially ordered sets with $edge(e, M) \setminus p$ for any nrs-path p between any sync-in point and closest sync-out point.

Definition 26 *Any cc-path e of M and concordant simplified state machine $trace(e, M)$ is* triggerable *if there exists a connected partially ordered set $N \subseteq edge(e, M) \setminus edge(p, M)$ for any nrs-path p in $trace(e, M)$ between any sync-in point and its closest sync-out point for which $tl(edge(p, M))$ is a subset of $gl(il(N))$. Any* triggerable *cc-path (and concordant $trace(e, M)$) e for which for each guarded edge e_i of $edge(e, M)$ there exists an e_j in $edge(e, M)$ for which $mrwgsat(e_i, e_j)$ holds is a* legal *cc-path. The set of all legal cc-paths of simplified sc model M is called $LCCP_M$.*

Given a $gsat$ relation, the set $LCCP_M$ now only contains those cc-paths that reaches termination.

Proposition 27 *The set $LCCP_M$ is well defined and finite. Reachability in simplified statecharts is therefore decidable, given a $gsat$ relation.*

The *most recent write* relation that we introduced earlier, can now be used to study the propagation of values throughout any cc-path in $LCCP_M$. Every guard reads a value, dependent on his mrw write action. Each state of $trace(e, M)$ of any cc-path e of $LCCP_M$ knows the possible values of some memory locations ($m \in mL$). These values are the result of guarded edges lying on any path from the initial pseudostate to the current state.

Definition 28 *For each state σ of Σ_e of $trace(e, M)$ the* data *of that state σ is the data of the previous states (the source states σ' of the edges e_i for which $\sigma' \prec_c \sigma$, $lbl(e_i) = (t, m, a)$) plus the guards m of mL that are read on the incoming edges e_i minus any memory location that is written to in the actions a in mL of each e_i. We denote the data present within a state σ with $data(\sigma, e, M)$*

A state can only have data on locations that appear as guard on every path to σ and that are not written to after they are read. If a location is written to it is removed from the *data* of successor states. *Sync-out points* of the partially ordered cc-paths only have the intersection of the *data* of all source states of its incoming edges. Both constraints on the *data* of states are the result of our abstraction of functional calculation in simplified state machines. The only things that can happen on an edge are a *read* (guard) which is added to the known values of the destination state and subsequent memory writes of a *transformation* of the data of the previous states plus the read value on the edge under consideration. In analogy to the mrw relation we now introduce the mrr relation pointing out for every state and for every location known in that state which are the *most recent reads* for that location (e.g. the reads that influence the written result value).

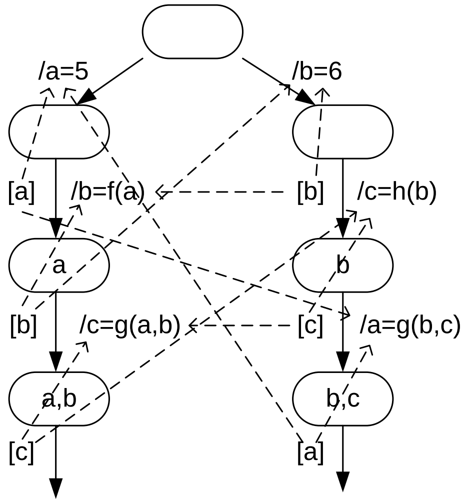

Figure 6: An Incomplete Example of the Most Recent Write relation

Definition 29 *For every cc-path e of M and concordant ssc $trace(e, M)$ we define*

$$mrr : \Sigma_e \times mL \times edge(e, M)$$
$$mrr(\sigma, m, e_i) \Leftrightarrow lbl(e_i) = (t, m, a)$$

with m in $data(\sigma, e, M)$ and there exists a σ' for which $\sigma' \prec_c \sigma$ induced edge e_i.

Fig.6 shows two unordered paths in the partial order of some cc-path in $LCCP_M$, for some simplified sc model M. The gray arrows point out the mrw relationship between reads and writes. Note that the assignment $/a = 5$ for example, is the application of a function, without any arguments. This kind of function is most commonly known as a constant function.

In the literature on memory models the likelihood of causal loops between memory writes and reads is illustrated [12]. Such causal loops between mrw and mrr relations can still be present in legal cc-paths of $LCCP_M$. A causal loop is created when some most recent write w_1 of some most recent read r_1 has in its calculation (or transformation) need of a reference to some other read value r_2, which has as one of its most recent writes, w_1. Fig. 6 shows a couple of causal loops. Memory location a for example, is read into the write on b. This read sees both writes of a, one of which is depending on the value of b and c. The value of b in its turn, is depending on the value that was written to b, which was depending on the value of a. Every legal cc-path needs a *resolution* of these causal loops. To guide this *resolution* we need a function for each edge with a memory write on it. In this definition, we will represent the set of all primitive recursive functions as $\mathcal{F}$.

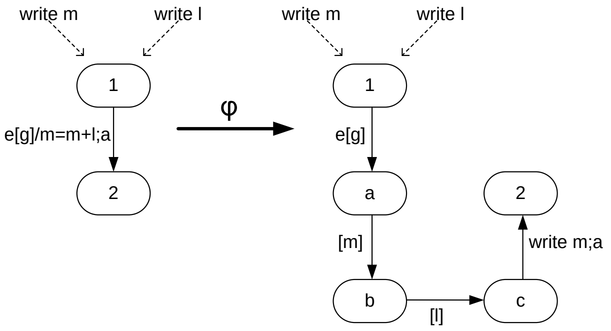

Figure 7: Side-Effects-Free Operations in an Ssc Model

Definition 30 *For every execution e of M and concordant ssc $trace(e, M)$ we define*

$$\chi : edge(e, M) \rightarrow ((F) \times 2^{mL})$$

is a function which returns for each edge with a memory write in its label the function which is applied before the transformed value is written. It also points out which variables are needed for its calculation. For each result pair of function χ and for each relevant subset of mL each one of the elements in the second component must be present in the data *of the state of which the edge origins, in all executions of $LCCP_M$ with a path through that state.*

If for example the function χ is partially defined for edge e_i as $\chi(e_i) = (g, \{a, b, c\})$ it means that the write operation in $lbl(e_i)$ is depending on the values of a, b and c, and that some function g is applied to them.

In Fig. 7 we see that while translating the sc with morphism φ, and with e_i the edge under transformation, we would have to log in χ that $\chi(e_i)$ equals $(+, \{m, l\})$. The function χ thus can be derived from the translation of normal UML statecharts, while applying translation morphism φ described in [6].

With the partially ordered executions in the set $LCCP_M$, χ and $gsat$, we can deterministically trace the possible values of each memory location in any simplified sc model M. We already discussed how the $gsat$ relation allows us to limit our attention to legal cc-paths and executions. As we will see below the function χ can now guarantee that we only have to verify legal cc-paths without causal loops.

Let us give an extended example of the whole process, before we describe the procedure for causal loop detection and resolution. Fig. 8 shows us an example without events. Two of the reduced orders are shown in Fig. 9. Table 1 defines function χ for Fig. 8. Fig. 10 shows the *data* in each state of the reduced orders of Fig. 8.

Table 1: Definition of χ for Fig. 8

$(1, 2)$	$\rightarrow$	**5**	$(10, 21)$	$\rightarrow$	$f_{11}(b)$
$(2, c_1)$	$\rightarrow$	$f_6(a)$	$(15, 16)$	$\rightarrow$	$f_3(a)$
$(2, 19)$	$\rightarrow$	$f_1(a)$	$(16, 17)$	$\rightarrow$	$f_{12}(b)$
$(3, 4)$	$\rightarrow$	$f_7(a)$	$(16, 20)$	$\rightarrow$	$f_4(b)$
$(4, c_2)$	$\rightarrow$	$f_8(b, a)$	$(19, 11)$	$\rightarrow$	$f_2(a, b)$
$(5, 6)$	$\rightarrow$	**7**	$(20, 24)$	$\rightarrow$	$f_5(b)$
$(6, 7)$	$\rightarrow$	$f_9(c, b)$	$(21, 22)$	$\rightarrow$	$f_{13}(a, b)$
$(9, 10)$	$\rightarrow$	$f_{10}(b)$	$(23, 24)$	$\rightarrow$	$f_{14}(b, a, c)$

Using Fig. 9,10 and Table 1, we can verify causality, remove causal loops and determine the outcome value for each memory location in each state of cc-path and its concordant execution, and this for each legal cc-path. The cc-path of Fig. 9 left, has no causal loops, so it is rather easy to determine the possible values of each variable a, b and c for example, upon the termination of the execution. Table 2 shows these values. We give the example for the termination state in Tab. 2 but it may be clear that the same technique can be applied to any state of the sc that is present in some cc-path e of $LCCP_M$. The values for c upon termination are dependent on the write on edge $(20, 24)$, and is therefore determined by $f_5(b)$. The most recent write of b seen from edge $(20, 24)$ is determined by edge $(2, 19)$, where $f_1(a)$ is written to b. The most recent write, for the read seen on edge $(2, 19)$ is on edge $(1, 2)$, which assigns **5** to a. Conclusion: in state 24, c equals $f_5(f_1(5))$. We adopt similar reasoning for variables (memory locations) a and b. Notice that in the case of a we took the last value (the *maximal element* that writes a) in the $<_c$ order that was written. If there would have been more candidates, a would have had *all* of those values.

Table 2: Value Propagation of Fig. 9, Left Side

a	$\rightarrow$	$f_3(5)$
b	$\rightarrow$	$f_1(5)$
c	$\rightarrow$	$f_5(f_1(b))$

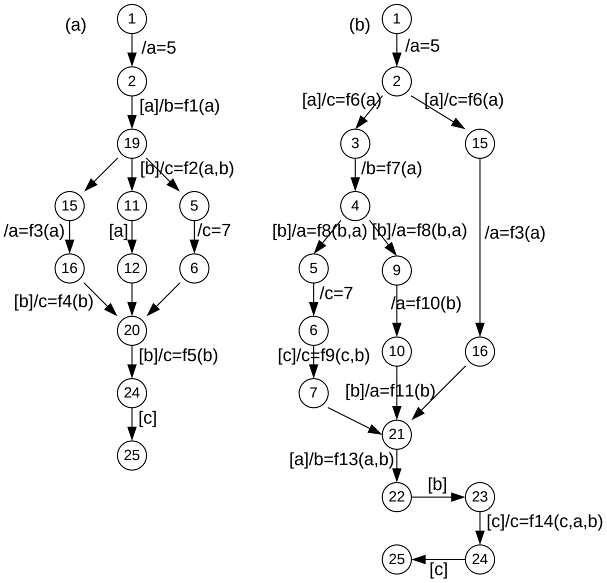

Figure 9: Complex Example Revisited, with Labels

If we now analyze Fig. 9, right, in the same manner we get a more complex causality pattern. For variable c in Fig. 9, right, we get causal dependencies (with a causal loop) as shown in Fig. 11. This example shows that it might be useful to introduce a more algorithmic and systematic approach to the determination of the values for each variable in each state. (This way, we can get a computer to do all the work for us.)

With x a memory location of the *data* of a certain state σ, in a certain execution e of $LCCP_M$ we outline the procedure (Proc. 31) to determine all values.

In this procedure we construct a tree of dependent writes and we close leaves if they either appear twice in the path to the root of the tree (which means we have a causal loop) or if they are a constant value (which needs no more arguments). This way we combine the resolution of causality with the determination of all possible values, in each state and for each memory location.

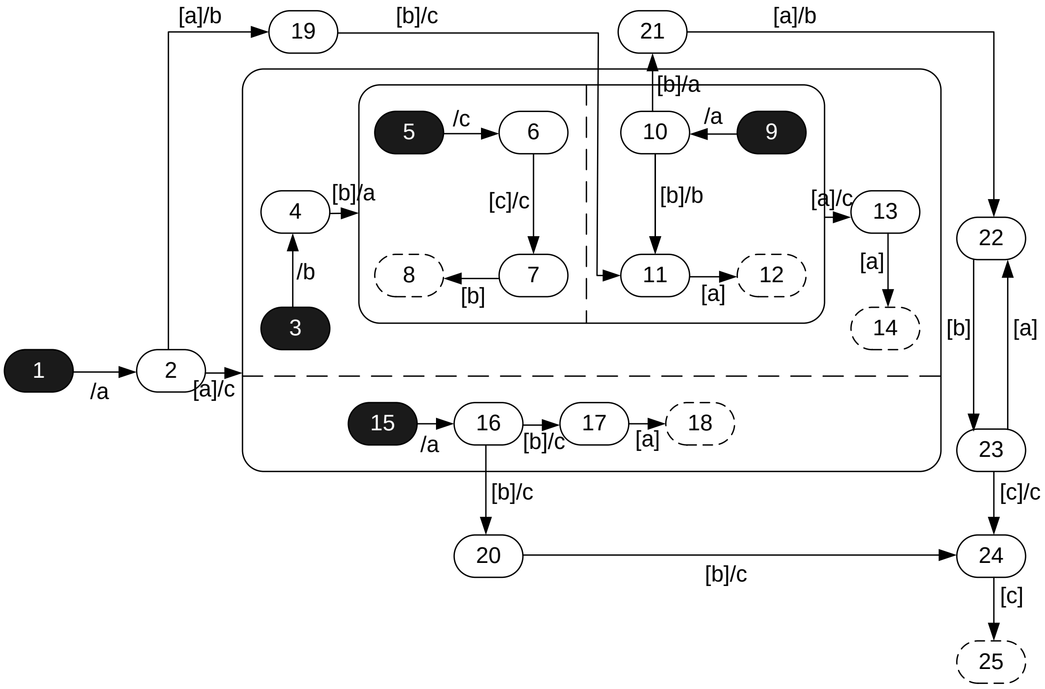

Figure 8: Complex Example Revisited, with Labels

Procedure 31 (determine values)

```
Search for $mrr(\sigma, x, e_i)$,
determine $mrw(e_i, e_j)$ and
construct a tree $T$ with root $\chi(e_j)$.
While not all leaves of $T$ closed,
    for each var $v$ used in $\chi(e_j)$ and in $data(\sigma, e, M)$
        search for $mrr(\sigma', v, e_k)$
        store $mrr(\sigma', v, e_k)$ in $W$ and
        make all elements of $W$, children of $v$.
        If leaves appear twice on a path to the root of $T$
        then close these leaves.
        If a new leaf is a constant
        then close this leaf.
    Redo this recursively.
Store the leaves of $T$ in $L$ and
remove all closed, non-constant leaves from $T$.
Recursively remove leaves not in $L$.
Recursively construct
        all values on any path to the root of $T$.
```

Fig. 12 shows a trace tree for variable c in state 24 ($x = c$ and $\sigma = 24$ in Proc. 31), and Fig. 11 shows how we got to it. Non-constant functions which are leaves, are removed in Fig. 12, and the possible values can now systematically and unambiguously be determined, by following the leaves bottom up. One of the 25 possible values of c upon termination of the executed statechart in Fig. 9, right, is

$$c = f_{14}(\ f_9(7, f_7(5)), f_{11}(f_7(5)), \\ f_{13}(f_3(f_8(f_7(5), 5)), f_7(5)))$$

Remark that one can parse this expression, using the trace tree of Fig. 12. We call the expression for c, a *footprint* of c in state σ and each variable can have any but an infinite number of footprints in each state of each cc-path of a simplified sc model M. The tree that was used to construct these values, is called a *trace tree*. Each variable or memory location has exactly one trace tree in every state of a simplified sc cc-path. If after removal of causally looping leaves, the tree becomes empty, then there aren't any legal values for that variable in that state. If the variable in question belongs to the *data* of that state, and is used in write functions, then we say that χ is illegal, instead of the cc-path e of M. We then remove this illegal part $\chi(e)$ but only if both conditions of above are met. If χ becomes an empty function in doing so for a certain cc-path e of M only then will we remove e from the set $LCCP_M$ and say that it is an illegal cc-path and execution.

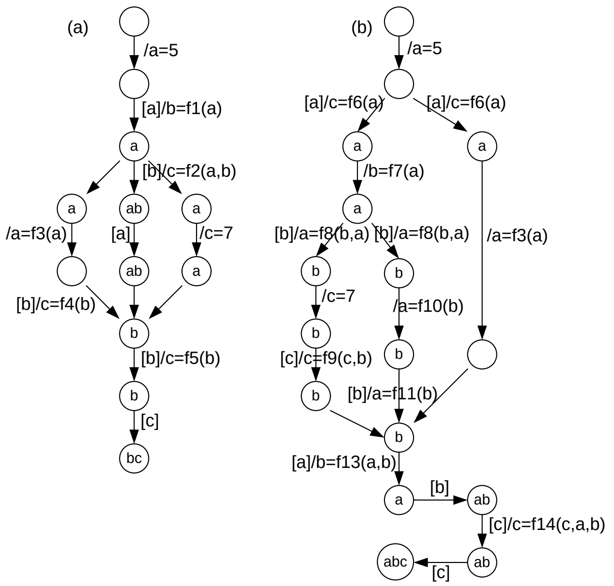

Figure 10: Complex Example Reduction Revisited, with Labels

Remark that it will almost always be necessary to calculate all values of all variables in all states of the model M. The calculations can be done much more efficiently if a certain order is followed, namely top down, as we can keep the calculated values stored for every state and reuse them in a recursive application of Proc. 31. To verify the model with model checking techniques for example, we do need to know all possible values in all reachable states for all possible executions of $LCCP_M$.

Proposition 32 *For each cc-path e of $LCCP_M$ each possible value of any memory location (of mL) in use in any state of a simplified sc, can unambiguously be determined by Proc. 31.*

4 Normalization

Simplified state machines, when derived from UML sc, can, due to the transformations that we apply to them (see [6]), have many sparse edges. An edge of an ssc is called sparse when it is missing a trigger, guard or action (or any combination of the three). Having many sparse edges in an ssc is not very useful due to this reason: in the ssc formalism a state can only be influenced by triggers, guards and actions and is only useful if any of the possible values of the variables available in that state differ with the previous and next states in any path containing that state or if the trigger list of a certain path becomes different by accessing that state. We therefore derive a *normal form* for simplified statecharts that will allow us to reduce sparse edges and unnecessary states. Having a normal form of simplified state machines will prove to be invaluable when defining complexity metrics on statecharts. Some statecharts are only seemingly different from others if one analyzes the different paths in those statecharts. The UML is designed to allow for this kind of (uncontrollable) flexibility but in mathematical descriptions it has adverse effects. Our formalism would only be useful if it allows us to unravel superficial differences. Also remark that the procedures to disentangle state machines are very complex and that any reduction of the number of states or edges can have tremendous positive effect on the calculational complexity of the formalism.

We start by defining what we mean by a normalized state machine, proceed with explaining equivalent simplified statecharts and derive a normalization procedure which converts a simplified sc to a normalized simplified sc equivalent to the original one.

Definition 33 *The set of all* values *of some variable v of the data of some state σ of some cc-path e of ssc model M is denoted with $val(v, \sigma, e, M)$.*

Definition 34 *A state σ of M is* distinguishable *if on some cc-path e of M for some σ_1 and σ_2, states of M different from σ with $\sigma_1 \prec_c \sigma \prec_c \sigma_2$ it hold that either*

$$data(\sigma, e, M) \neq data(\sigma_1, e, M) \text{ and}$$
$$data(\sigma, e, M) \neq data(\sigma_2, e, M)$$

or there exists a variable v of $data(\sigma, e, M)$ for which

$$val(v, \sigma, e, M) \neq val(v, \sigma_1, e, M) \text{ and}$$
$$val(v, \sigma, e, M) \neq val(v, \sigma_2, e, M).$$

Definition 35 *A state σ of M is* trigger-contributing *if by removing it from M and attaching incoming and outgoing edges to other states such that the transitive $<_c$-order is preserved on all states of M except σ, there exists a trigger list in M that doesn't appear on the model with σ removed.*

Definition 36 *A* normalized ssc *is a simplified statechart of which all states are distinguishable and trigger-contributing.*

From the properties of a normalized ssc we derive that a state machine can be called label-equivalent to another state machine if the order of guards is preserved, the order of triggers is preserved, the order of actions is preserved and if a guard on a certain variable v is ordered to an action on v that order is also preserved. To write this down formally we need to split up the definition of the label-function on edges.

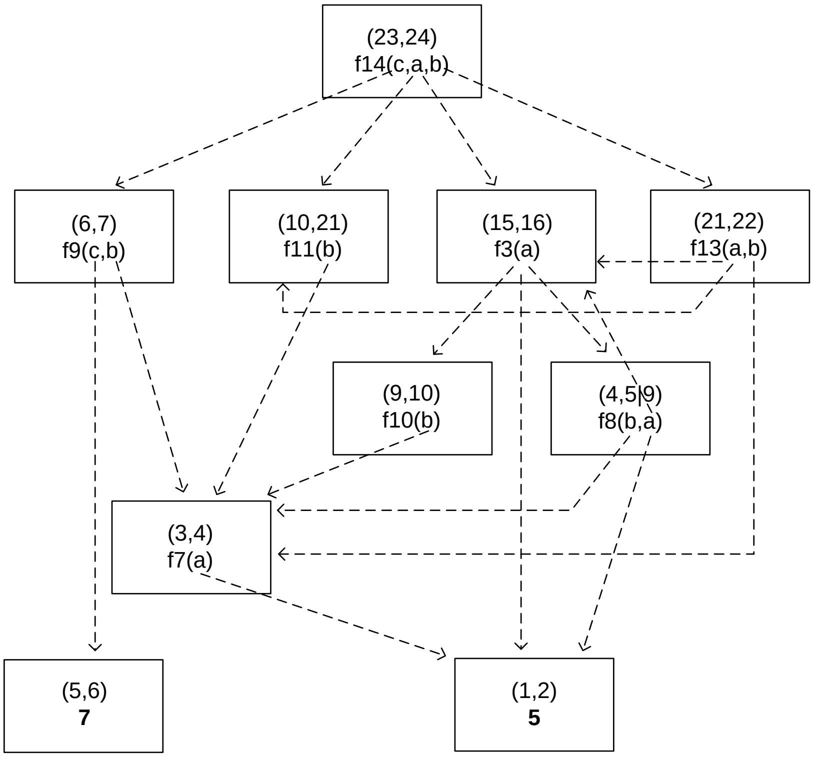

Figure 11: Dependencies of Variables and Values for Location c in State 25 of Fig. 9

Definition 37 *A* guard-label *of an edge e is the label $lbl(e)$ limited to the guard (of mL) and is denoted with $lbl_g(e)$.*
A trigger-label *of an edge e is the label $lbl(e)$ limited to the trigger (of eL) and is denoted with $lbl_t(e)$.*
An action-label *of an edge e is the label $lbl(e)$ limited to the action (of $eL \cup mL$) and is denoted with $lbl_a(e)$.*
A variable-label *of an edge e is the label $lbl(e)$ limited to the indexed guards and actions (of mL) on the same variable v and is denoted with $lbl_x(v, e)$. If there is no variable v appearing in the label, the* variable-label *returns ϵ.*

Extending the different label functions on sets we arrive at the following concise definition of label-equivalence.

Definition 38 *Two simplified statechart models M_1 and M_2 are* label-equivalent *if there exist four isomorphisms $\varphi_i : P_i \to Q_i$, for $i = g, t, a, v$*

$$P_i = \{\alpha \mid p\, nrs - path, lbl_i(edge(p, M_1)) = \alpha\} \quad (23)$$
$$Q_i = \{\alpha \mid q\, nrs - path, lbl_i(edge(q, M_2) = \alpha\} \quad (24)$$

such that

$$\varphi_i(p) = q \Leftrightarrow p \equiv_l q \quad (25)$$

with $p \equiv_l q$ if and only if the two posets are equal when removing all ϵ's thereby preserving the poset order.

Remark the similarity to the definition of a *bisimulation* in process algebras. With this definition we are given enough room for a normalization procedure, consisting of these two steps: label normalization and state normalization. In the first step we try to fill out the transition labels as densely as possible by replacing ϵ's in the labels. We also put all triggers, guards and actions as early as possible on any path. The second step will use a transformed minimalisation procedure for regular automata, to remove unnecessary states. We will prove that the result of these transformations is a label-equivalent normalized simplified sc. We specify the label chord normalization transformation rules in Table 3 and define the λ-transition.

Definition 39 *A λ-transition is an edge with label $(\epsilon, \epsilon, \epsilon)$.*

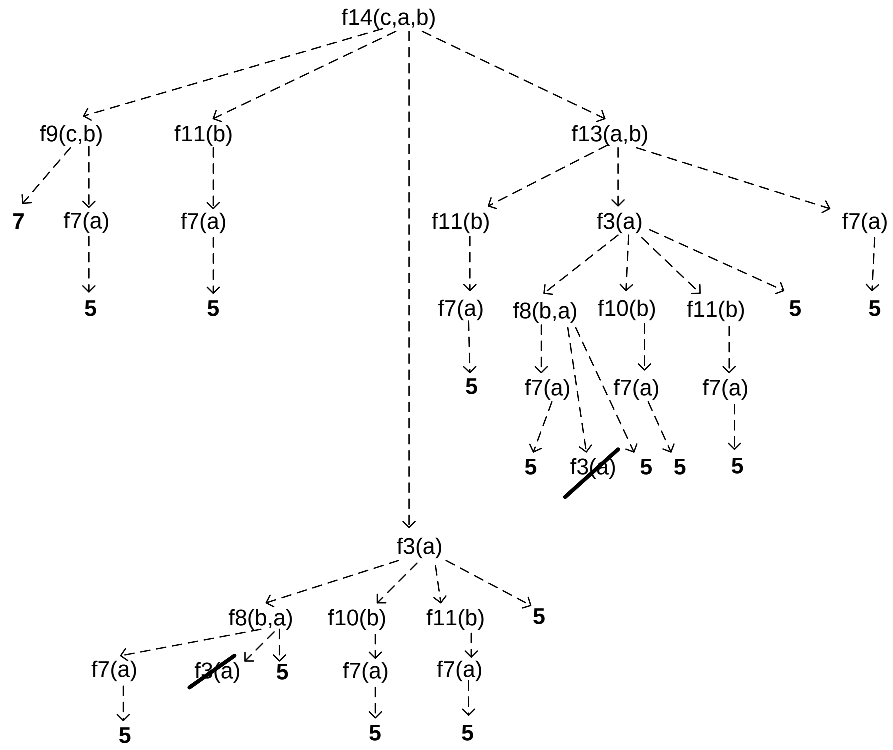

Figure 12: Trace Tree for Location c in State 25 of Fig. 9

Procedure 40

```
For each $s \in S$
repeat until nothing changes anymore:
  for each $\sigma$ in region $\rho(s)$,
    remove all (reflexive) $\lambda$-transitions on $\sigma$,
  for each $\sigma$ in region $\rho(s)$,
    apply the maximal set of lcn rules on $\sigma$.
```

A maximal rule links up as many parts of the labels as possible, but never all parts at once (see Tab. 3).

First part of Proc. 40 is to remove all (reflexive) λ-transitions in the simplified sc under consideration. The following proposition states that this doesn't influence label-equivalence.

Proposition 41 *Each λ-transition between σ_i and σ_j, of any region can be removed by assuming $\sigma_i = \sigma_j$, and results in a label-equivalent simplified sc.*
Each reflexive λ-transition on σ_i, of any region can be removed from δ and results in a label-equivalent simplified sc.

The proof of this proposition is rather trivial by the definition of $\equiv_l$.

The labels of a simplified sc can be seen as a bead chord. The beads are the three parts of the transition labels (triggers, guards and actions), and the chord that binds them together in a certain order, is formed by the paths that bear these labels. The transformation we realize in the second part of Proc. 40 can be imagined as follows: if we put a bead chord in our one hand, and hold the start of the chord between thumb and index, we can link up the beads relative to the start position with our other hand (draw them nearer to the start position, for example). Fig. 13 shows this process. Dependent on the number of beads, relative to the length of the chord (the number of ϵ-parts in labels), this linking has more or less effect. The result of this linking should be that actions are executed as early as possible without violating path causality (or order), guards are tested as early as possible and triggers are waited for as early as possible. 'Early' signifies close to the start state of some region. We do not touch transitions of δ' and leave the region hierarchy intact in our normalization procedure.

Table 3 shows an overview of the six possible transformations for every state which has incoming *and* outgoing edges (at least one of each). Table 3 shows six rules: the initial configuration is at the left hand side of the arrows, and the result is at the right hand side. The columns left of the double vertical line display the three parts of the different incoming labels (in the order trigger, guard, action), and the columns

Table 3: Label Chord Normalization **lcn**

e_1	g_1	ϵ	f_1	h_1	b		e_1	g_1	b	f_1	h_1	ϵ
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\rightarrow$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
e_n	g_n	ϵ	f_m	h_m	b		e_n	g_n	b	f_m	h_m	ϵ
						$\rightarrow$						
e_1	ϵ	ϵ	f_1	h	b_1		e_1	h	ϵ	f_1	ϵ	b_1
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\rightarrow$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
e_n	ϵ	ϵ	f_m	h	b_m		e_n	h	ϵ	f_m	ϵ	b_m
						$\rightarrow$						
ϵ	g_1	ϵ	f	h_1	b_1		f	g_1	ϵ	ϵ	h_1	b_1
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\rightarrow$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
ϵ	g_n	ϵ	f	h_m	b_m		f	g_n	ϵ	ϵ	h_m	b_m
						$\rightarrow$						
e_1	ϵ	ϵ	f_1	h	b		e_1	h	b	f_1	ϵ	ϵ
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\rightarrow$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
e_n	ϵ	ϵ	f_m	h	b		e_n	h	b	f_m	ϵ	ϵ
						$\rightarrow$						
ϵ	ϵ	ϵ	f	h	b_1		f	h	ϵ	ϵ	ϵ	b_1
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\rightarrow$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
ϵ	ϵ	ϵ	f	h	b_m		f	h	ϵ	ϵ	ϵ	b_m
						$\rightarrow$						
ϵ	g_1	ϵ	f	h_1	b		f	g_1	b	ϵ	h_1	ϵ
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\rightarrow$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
ϵ	g_n	ϵ	f	h_m	b		f	g_n	b	ϵ	h_m	ϵ

at the right side display the three parts of the outgoing labels in the same order. Everywhere in Table 3 we assume that the state σ under transformation has n incoming edges and m outgoing edges. The first rule states for example that if for a certain state, in a certain region, all outgoing edges execute the same action, and if no incoming edge of this state has an action other than ϵ, then the action on the outgoing edges, can be linked to the incoming edges. The other five rules are similar. We refer to any application of these rules as **lcn** (label chord normalization) in Proc. 40. Remark that similar rules are used in compiler optimizations to execute actions as early as possible (see for example [12]).

Proposition 42 *Procedure 40 terminates for every simplified statechart, and transforms a simplified statechart into a label-equivalent simplified statechart.*

Procedure 40 consists of two parts: (reflexive) λ-transition elimination and lcn application (Table 3). The proof that λ-transition elimination terminates is trivial: there are only a finite number of states and edges in each region and with every λ-transition removal one edge is eliminated therefore the procedure cannot go on for ever. Reflexive λ-transition removal also terminates for similar reasons. We already proved equivalence of statecharts after (reflexive) λ-transition elimination.

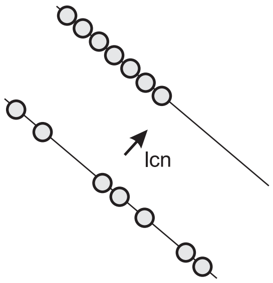

Figure 13: Linking the Labels as a Bead Chord

Remains to show that **lcn**-rules application terminates and results in equivalent simplified statecharts. To prove this, we introduce an *edge weight* function. The arity of an edge is the number of non ϵ positions in the label of that edge. Possible values for the arity of an edge in simplified statecharts are the natural numbers zero to three (0 to 3). We access this arity value for each edge e with the function $ar(e)$. We denote the set of all edges of all s-paths from state σ to σ' with $SP(\sigma, \sigma')$

Definition 43 *For any state σ in any region $\rho(s)$, s in S, of a simplified statechart, the* edge weight *of all outgoing edges of state σ is defined as follows. Given all the shortest s-paths, originating from the start state to state σ ($\in \rho(s)$):*

$$\begin{array}{l} p_1 : s = \sigma_{0,1}, \sigma_{1,1}, \sigma_{2,1}, \ldots, \sigma_{k,1} = \sigma \\ \ldots \\ p_m : s = \sigma_{0,m}, \sigma_{1,m}, \sigma_{2,m}, \ldots, \sigma_{k,m} = \sigma \end{array}$$

calculate following expression for each shortest s-path p_j:

$$ar(p_j) = \Sigma_{e_i \in edge(p_j,M)} ar(e).4^i \qquad (26)$$

and find the s-path p_{min} for which $ar(p_j)$ is the smallest. The edge weight *of all outgoing edges of state σ to any state σ' connected via edge f is defined as*

$$\mid f \mid_l = ar(p_{min}) + ar(f).4^k \qquad (27)$$

The weight of $SP_{(\sigma,\sigma)}$ is 0. The edge weight *of the set $SP_{(\sigma_i,\sigma_j)}$ of s-path edges between σ_i and σ_j is the sum of the* edge weights *of all edges in this set:*

$$\mid SP_{(\sigma_i,\sigma_j)} \mid_l \;=\; \Sigma_{e \in SP_{(\sigma_i,\sigma_j)}} \mid e \mid_l \qquad (28)$$

Remark that by this definition and by the definition of s-paths we have

$$\begin{array}{rcll} \mid SP_{(\sigma_i,\sigma_j)} \cup SP_{(\sigma_j,\sigma_j)} \mid_l & = & \mid SP_{(\sigma_i,\sigma_j)} \mid_l & (29) \\ \mid SP_{(\sigma_i,\sigma_j)} \cup SP_{(\sigma_j,\sigma_k)} \mid_l & \leq & \mid SP_{(\sigma_i,\sigma_k)} \mid_l & (30) \end{array}$$

The proof that **lcn** terminates, is an application of noetherian induction. We first prove following property of edge weights:

Proposition 44 *For each set of s-path edges $SP_{(\sigma_i,\sigma_j)}$ in region $\rho(s)$ holds that*

$$\mid SP_{(\sigma_i,\sigma_j)} \mid_l \;>\; \mid \textbf{lcn}(SP_{(\sigma_i,\sigma_j)}) \mid_l$$

*with **lcn**$(SP_{(\sigma_i,\sigma_j)})$ the edges of the set of s-paths between σ_i and σ_j after maximal rule application of **lcn**.*

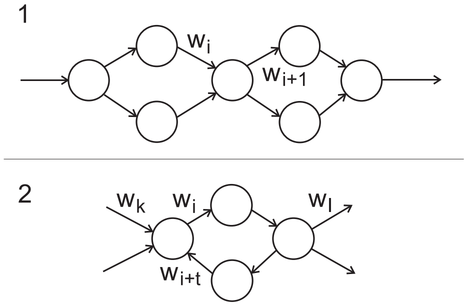

Figure 14: Analysis of Two Types of s-paths

We proceed with the proof of this property. The rationale of the edge weight function, is that any linking of label parts, due to **lcn**, which brings label parts closer to the start state, will result in a lower edge weight for at least one edge, therefore also for a set of edges containing that edge. With w_i, w_{i+1} label arities, $x < w_{i+1}$, we have for the edge weight of some edges in $SP_{(\sigma_i,\sigma_j)}$ and **lcn**$(SP_{(\sigma_i,\sigma_j)})$

$$\begin{array}{l} \ldots + w_i.4^i + w_{i+1}.4^{i+1} + \ldots > \\ \quad \ldots + (w_i + x).4^i + (w_{i+1} - x).4^{i+1} + \ldots \end{array}$$

for any linking which brings label parts closer to the start state. There is one case where label parts can be linked up to edges further away from the start state, namely when a loop participates in some path in $SP_{(\sigma_i,\sigma_j)}$. Every loop has at least one entry and one exit point, otherwise it is not reachable from the start state or would not participate in the paths to σ_j, hence wouldn't be present in $SP_{(\sigma_i,\sigma_j)}$. We therefore can always find states σ_k and σ_l with $k < l$ which are entry and exit points respectively for the loop with incoming and outgoing edge weights w_k and w_l, $x < w_i$, $y < w_l$, $k < i < i+t < l$, such that for an exhaustive application of **lcn** on all paths of $SP_{(\sigma_i,\sigma_j)}$, we have

$$\begin{array}{l} \ldots + w_k.4^k + \ldots + w_i.4^i + \\ \quad \ldots + w_{i+t}.4^{i+t} + \ldots + w_l.4^l + \ldots > \\ \ldots + (w_k + y).4^k + \ldots + (w_i - x).4^i + \\ \quad \ldots + (w_{i+t} + x).4^{i+t} + \ldots + (w_l - y).4^l + \ldots \end{array}$$

Therefore in any case we have

$$\mid SP_{(\sigma_i,\sigma_j)} \mid_l \;>\; \mid \textbf{lcn}(SP_{(\sigma_i,\sigma_j)}) \mid_l$$

and this proves the property. Fig. 14 visually illustrates both cases of this proof.

By Prop. 44, every sequence of **lcn** applications, with $s \in S$ and $t \in T$, of the form

$$\begin{array}{l} \mid SP_{(s,t)} \mid_l \;>\; \mid \textbf{lcn}(SP_{(s,t)}) \mid_l \;> \\ \qquad \ldots > \; \mid \textbf{lcn}^i(SP_{(s,t)}) \mid_l \;> \ldots \end{array}$$

is noetherian, and therefore can never be infinitely long. It follows easily that an exhaustive application of **lcn** must terminate and results in an irreducible normal form.

Proposition 45 *Any (exhaustive) application of* **lcn** *to a simplified sc results in a label-equivalent simplified statechart.*

The proof of this property is trivial: match the four constraints of label-equivalence to the six different rules of Table 3. We leave it to the reader.

With Prop. 41, the termination of (reflexive) λ-transition removal, Prop. 45, the termination of **lcn** application, and the independence of λ-transition removal and **lcn** application, we prove Prop. 42.

Another form of normalization for simplified statecharts, is the reduction of the number of states by removing the unnecessary ones. For this we will use an adapted form of regular automata state reduction. The number of states and the length of paths in a simplified sc clearly determine the complexity of calculating execution traces. By reducing the number of states and by shortening paths we can make this calculation more efficient. Unlike UML statecharts, simplified statecharts impose severe restrictions on the labeling of their edges. In [3] we reduced the action language to its least redundant form given the expressive power of statecharts themselves. We removed control flow instructions from the action language because they are implicitly present in statecharts, we removed event-like structures since they are also implicitly present and so on. Also, we abstracted assignment and other operators to their most basic memory influence, the use of which has become apparent in determining all executions and the possible values of all memory locations. The net result was an action language without redundancy matching regular automata. We can now easily apply regular automata theory to simplified statecharts. In doing so we introduce the final part of the normalization procedure for simplified state machines where we will remove all indistinguishable states.

Definition 46 *A state is* indistinguishable *if it is not distinguishable.*

Definition 47 *A state is* inaccessible *if there exists no s-path containing any s of S that also contains that state.*

It may be clear from the definition that the *indistinguishable* relationship is reflexive, symmetric and transitive, and therefore an equivalence relation. We split up this relationship by region and denote it with $=_s$ with s of S, for each region $\rho(s)$. We continue with the specification of the second normalization procedure.

Procedure 48

```
For each $s \in S$,
  for each state $\sigma$ of region $\rho(s)$
    remove $\sigma$ if it is inaccessible and
    remove all $\delta(\sigma, l) = \sigma'$ and $\delta(\sigma'', l) = \sigma$ from $\delta$.
  construct all equivalence classes $C_{si}$ for $=_s$.
For each $s \in S$,
  add $s$ to $\Sigma_=$,
  for each equivalence class $C_{si}$ in region $\rho(s)$
    add a new state $\sigma_{si}$ to $\Sigma_=$,
    for each $\sigma$ in $C_{si}$ with $\delta(\sigma, e, m, a) = \sigma' \in C_{sj}$
      add $\delta_=(\sigma_{si}, e, m, a) = \sigma_{sj}$ to $\delta_=$,
    for each $\sigma$ in $C_{si}$ with $\delta'(\sigma, e, m, a) = \sigma' \in C_{s'k}$
      add $\delta'_=(\sigma_{si}, e, m, a) = \{\sigma_{s'k}\}$ to $\delta'_=$,
  if $s \in \delta'(\sigma, e, m, a)$, $\sigma \in C_{si}$
  then add $\delta'_=(\sigma_{si}, e, m, a) = \{s\}$ to $\delta'_=$.
For each $t \in T$,
  add $t$ to $\Sigma_=$,
  for each $\sigma$ in some $C_{si}$ for which $\delta(\sigma, e, m, a) = t$
    add $\delta_=(\sigma_{si}, e, m, a) = t$ to $\delta_=$
  for each $\sigma$ in $\delta'(t, e, m, a)$, $\sigma \in C_{si}$
    add $\delta'_=(t, e, m, a) = \{\sigma_{si}\}$ to $\delta'_=$.
Construct the simplified statechart
    $\langle \Sigma_=, s_0, \delta_=, \delta'_=, S, T \rangle$
```

Termination of this procedure is trivial, so we proceed with the proof of the label-equivalence.

Remark that an isomorphism relation is an equivalence relation: each set is isomorphic to itself, and if set A is isomorphic to set B, then B is also isomorphic to A. If A is isomorphic to B and B to C, then A is isomorphic to C.

We proceed by induction on the gradual construction of all the equivalence classes C_{si}, from termination (t in T) to start (s in S). Let us call the original simplified sc, sc_0. As base case we prove that the construction of the equivalence classes for states with outgoing edges to any termination state of some region (t in some $\rho(s)$) of sc_0 results in a label-equivalent sc.

In the first iteration (base case) we construct the equivalence classes of indistinguishable states with outgoing edges to t, and only of these states, by Proc. 48. Each equivalence class of indistinguishable states gets exactly one outgoing edge to t. By the definition of indistinguishable states each path in sc_0 is label-equivalent to exactly one path in the base case of the normalized sc (let's call it sc_1) and therefore there exists an isomorphism between the accepted executions of sc_0 and sc_1.

Now suppose that this property also holds for the statecharts of the next iterations in the construction ($sc_2, \ldots, sc_n$) and in all regions of those statecharts.

Notice that all these statecharts are label equivalent to sc_0, because the isomorphic relation is an equivalence relation, and therefore transitive. From the equivalence of sc_n, we prove the equivalence of sc_{n+1} to sc_0, with an analysis by cases. There are four components in the construction of paths: any connection in some path is the result of either δ or δ' transition function application, originates from a termination state (t of T) or is a δ'-connection to a start state (s of S).

If an equivalence class C_{si} of sc_{n+1} is connected to some equivalence class C_{sj}, which was already present in sc_n, because there is a δ application from a state σ in C_{sj} to a state σ' in C_{si}, then by the definition of indistinguishable states, the definition of Proc. 48 and our inductive hypothesis applied on sc_n, all paths resulting from this δ application will be present in both the original sc_0 and sc_{n+1}. For if this would not be the case, then there exist states σ and σ' in the equivalence class C_{sj} with different outgoing edges to equivalence classes C_{si} and C_{sk} of sc_n, in which case we get a contradiction with either the inductive hypothesis or the definition of indistinguishable states. By the definition of Proc. 48 we can't create new paths, because then we would have to have added δ applications, not in sc_0.

If the edge between C_{sj} and $C_{s'i}$, is a result of δ' application, we can follow a similar reasoning as for the case of δ. We can always "postpone" the proof of this case up until the relevant equivalence class $C_{s'i}$ is constructed and the inductive hypothesis becomes available, because the region hierarchy is a finite tree (no looping proof).

In the case of termination or start states the δ' applications are equal to the ones of sc_0 by the construction of Proc. 48 therefore sc_0 is equivalent to sc_{n+1}.

There is one case where this argument isn't clear to guarantee a non-looping induction hypothesis: the case where there are two (or more) equivalence classes C_{si} and $C_{s'j}$ which are connected in a loop through δ'-applications. In this case, however, we can "postpone" the induction hypothesis again by the same reasoning as the one that was illustrated in Fig. 14.2.

By induction we can conclude that the simplified sc sc_{n+m} (the result of equivalence class connection from the start state) is label-equivalent to the original sc_0. This concludes the proof.

Proposition 49 *Procedure 48 terminates and results in an equivalent simplified sc. The resulting sc has a minimal number of states in all regions.*

Remark that Proc. 40 is completely independent of Proc. 48, and therefore they can be applied in any order to a simplified sc. The net result of our two normalization procedures taken together, is a *normalized simplified sc*. The path length and number of states is minimal, just as we set as the objective of this section.

Proposition 50 *A* normalized simplified sc *is a simplified sc which doesn't change under the application of Proc. 40 or Proc. 48. No simplified sc with less states or shorter paths is label-equivalent to this normalized simplified sc.*

The proof of this statement is left to the reader.

5 Conclusion

To derive state machine normal forms in a mathematically correct manner we highlighted part of our simplified state machine theory in this paper. We introduced s-paths and c-paths in simplified state machines. We defined nrs-paths, nrc-paths and concordant partial orderings $<_s$ and $<_c$ on the set of states of state machine model M. We constructed cc-paths and reasoned on UML state machine semantics to extraxt all legal execution traces from the state machine model description. Restricting the introduction of this theory to the most essential definitions and properties allowed us to concisely represent the complex mathematical system needed to formulate and prove state machine normalization procedures. With behavioral equivalence matched to formal equivalence we are able to make value propagation and executions calculation algorithms more efficient. Normal forms also allow us to reason on many equivalent state machines at once thus making this theory even more interesting.

We conclude this paper with a remark on the comparison of value propagation and verification through model checking. Since we can more or less efficiently calculate the possible values of all variables in all reachable and useful states of the sc, we can check for temporal properties on each of the executions of the simplified sc model M separately by modeling the possible values in a Kripke Model and applying well known model checking techniques to it. Binding our abstraction to model checking algorithms allows the transformation of statecharts into infinite Kripke Models, in which every state knows all possible values of all variables and of the event queue (see for example [13], [14]). Any temporal property formulated in a language like LTL or CTL (see [1],[3],[4],[5]) can be checked on these Kripke Models.

Our theory on memory value propagation and normalization can further enrich model checking techniques by excluding unreachable and useless states. It allows us to first divide state machines into cc-paths or executions, determines reachability by event semantics in each of these executions (questions which are normally verified only at the time of model checking), afterwards determines fairness, liveness and exclusion (with causal loop detection) retaining only a finite number of cc-paths and executions with in every state all possible values of all variables. This technique can also be restricted to memory locations or states of interest to the state machine designer, guiding him throughout the construction process by answering temporal questions albeit reducing calculation time even further.

Making strong abstractions and using the macrostructures of the state machine we tackle one of the biggest challenges to behavioral designs: concurrent, lazy-locking memory behavior. Our theory is far from perfect and tends to struggle with looping constructs but promises guidance to better understanding and design of state machines.

References:

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

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

[3] Laura F. Cacovean, Emil M. Popa, and Cristina I. Brumar. Algebraic algorithm of ctl model checker. *WSEAS Trans. Info. Sci. and App.*, 4(1):1–8, 2007.

[4] Laura F. Cacovean, Emil M. Popa, and Cristina I. Brumar. Implementation of ctl model checker update. In *ICCOMP'07: Proceedings of the 11th WSEAS International Conference on Computers*, pages 432–437, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[6] Benjamin De Leeuw and Albert Hoogewijs. Management and object behavior of statecharts through statechart dna. *WSEAS Trans. Info. Sci. and App.*, 6(5):859–871, 2009.

[7] Benjamin De Leeuw and Albert Hoogewijs. Statechart dna. In *ICAI'09: Proceedings of the 10th WSEAS international conference on Automation & information*, pages 293–299, Stevens Point, Wisconsin, USA, 2009. World Scientific and Engineering Academy and Society (WSEAS).

[8] Benjamin De Leeuw and Albert Hoogewijs. Statecharts disantengled. In *ECC'10: Proceedings of the 4th European Computing Conference*, pages 39–51, Stevens Point, Wisconsin, USA, 2010. World Scientific and Engineering Academy and Society (WSEAS).

[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] 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.

[11] 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.

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

[13] 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.

[14] 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.