ConStaBL - A Fresh Look at Software Engineering with State Machines

Karthika Venkatesan[1,2] and Sujit Kumar Chakrabarti[1]

[1] International Institute of Information Technology Bangalore, INDIA
{karthika.venkatesan,sujitkc}@iiitb.ac.in
[2] Centre for Development of Advanced Computing Bangalore, INDIA

Abstract. Statechart is a visual modelling language for systems. Our variant of the statechart has local variables, which interact significantly with the remainder of the language semantics. Our semantics do not allow transition conflicts in simulations and are stricter than most other available semantics of statecharts in that sense. In this paper, we extend our earlier work on modular statecharts and present updated operational semantics with concurrency. It allows arbitrary interleaving of concurrently executing action code, which allows more precise modelling of systems and upstream analysis of the same. We also establish the criteria based on our semantics for defining conflicting transitions and valid simulations. We present the operational semantics in the form of the simulation algorithm. Our executable semantics can be used to simulate statechart models and verify their correctness. We present a preliminary setup to carry out fuzz testing of statechart models, an idea that does not have precedent in the literature to the best of our knowledge. We have used our simulator in conjunction with a well-known fuzzer to do fuzz testing of statechart models of non-trivial sizes and have found issues in them that would have been hard to find through inspection.

1 Introduction

Statecharts have been a popular modelling notation for several decades now. Many implementations are in use: Stateflow [23], Yakindu [18], Boost [30], Sismic [8], Raphsody [10], QM [27], Specgen [29], and Uppaal [20], both in the commercial and free domains, to name a few. Several surveys have been carried out on statecharts to date, indicating their effectiveness and usefulness in modelling complex systems [1, 3, 5]. Extensive research has been conducted in the areas of semantics, formal verification, and testing for a range of statechart variants. Even though research on statechart notation has seemingly reduced in pace over the last decade or so, we believe there are still various issues that need to be addressed.

1. The inherent complexity of statechart models needs efforts in the direction of simplification and modularisation of notation to make statechart modelling a scalable practice.

2. The semantics of the widely available statechart variants have important shortcomings [21]. These need investigation and proposals for mitigation.
3. Due to their higher level of abstraction and complex structure, statecharts need sophisticated verification, testing, and simulation capabilities to be integrated as a part of the modelling environment.

In this paper, we present ConStaBL (**Con**current **Sta**te **B**ased **L**anguage) a statechart variant that includes local variables. Local variables increase modularity but have a bearing on the semantics of the rest of the statechart language. In all reported work on statechart semantics, the modelling semantics is treated separately w.r.t. the action language. However, we believe that, to fully treat the concurrency semantics of statecharts, action language cannot be treated as an entirely external feature. Hence, we take a fresh look at the operational semantics of ConStaBL by integrating action language as a part of modelling semantics.

We present ConStaBL, a novel statechart language that serves as both a high-level modelling and a rapid prototyping language. As a high-level modelling language, it enables the construction of abstract representations of the system's states, transitions, and events, taking inspiration from design integrated development tools like Simulink [24], LabView [25], and many other statechart modelling tools [1]. This abstraction enables a clear visualisation of the system's functionality and behaviour to capture complex system dynamics and specify desired system behaviours in a concise and intuitive manner. In addition, our statechart language is an effective rapid prototyping tool, as it allows system designers to run simulations to validate and evaluate the system's behaviour, test various scenarios, and identify potential issues or enhancements during the earliest stages of development. However, there are notable distinctions in our approach to handling the action language, as we have discussed in Section 5.

An overview of our work is presented in Fig. 1. ConStaBL statecharts have integrated action language and variable scoping. The parsing phase verifies syntax problems and produces an Abstract Syntax Tree (AST). The semantic checker detects static errors in the statechart model, such as typing, and scope errors. The type-checked statechart can be executed through the simulation engine. The execution begins by initialising the statechart to its default state and subsequently consuming an event from the event queue. During execution, processing an event begins with the identification of transitions that are *triggered* by an *event* for which the *guard* evaluates them to be *true*. Afterwards, the execution moves from the source state to the destination state, passing through several intermediate configurations. This is done by running the appropriate action blocks for the states and transitions that have been identified. The process involves generating code for the identified transitions and executing this code in an interleaved manner. Environment (σ) and configuration ($\mathbb{C}$) are updated during execution. The event and input variables are prompted manually during simulation. The execution stops and reports when a *nondeterminism* is encountered (*nondeterminism* is presented in detail in section 3.2); otherwise, execution proceeds as an event occurs. We have used fuzzer to generate events and inputs

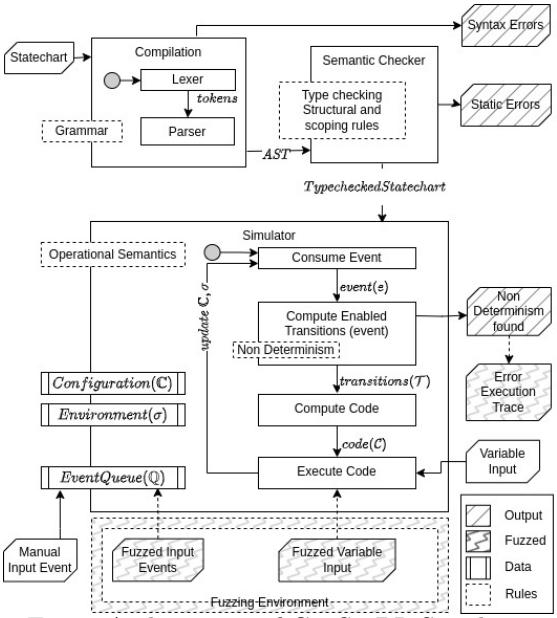


Fig. 1: Architecture of ConStaBL Simulator

for the variables to test the simulation with random inputs and generate the execution traces when nondeterminism occurs.

Through this work, we make the following contributions:

1. Concurrent StaBL (ConStabl): an extension of StaBL, a state-based specification language with local variables, to include concurrency.
2. Operational semantics of ConStaBL.
3. Simulator for ConStaBL.
4. Fuzz testing of statecharts, presented as an application of the simulator.

The overview of the ConStaBL language is presented in section 2 with an example. The Structural Semantics is presented in section 2.2. We present the design details of a simulator and operational semantics in section 3. Application and usage of fuzz testing is detailed in section 4.

2 The Language

In this section, we present the upgrades to the abstract syntax and structural semantics of StaBL [6] with the constructs and terminologies that are necessary to discuss the concurrency semantics of *ConStaBL*.

2.1 Abstract Syntax

A statechart model consists of three components: a set of states (S), a set of transitions (T), and a set of events (E).

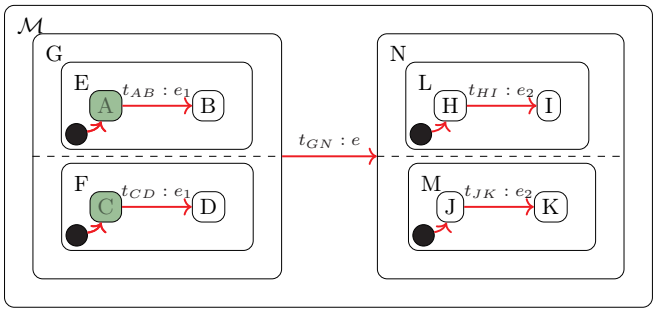


Fig. 2: A concurrent statechart model in a configuration $\mathbb{C} = \{A, C\}$.

A ***state*** is a *tuple* of $(p, S, I, \mathcal{V}_l, \mathcal{V}_p, \mathcal{V}_s, a_N, a_X, \tau)$. Here, $\mathcal{V}_l, \mathcal{V}_p, \mathcal{V}_s$ are the variable sets declared within the state with storage classes: *parameter*, *local*, and *static*, respectively. $a_\mathcal{N}, a_\mathcal{X}$ are the entry and exit actions. S is the set of all substates of the state. I is the set of default or initial substates of the state and I is a subset of S ($I \subseteq S$). $\tau \in \{statechart, atomic, composite, shell\}$ is the type of the state.

A ***transition*** is a *tuple* of (p, s, d, e, g, a). A transition is annotated as $e[g]/a$ on the arrow that connects the *source*(s) and *destination*(d) states. e is the event, g, the guard is a boolean value. g has to be *true* for a transition to be enabled/taken, and a is the action code.

In these *tuples*, p is the parent state of the state/ transition and a is an arbitrary piece of code in any imperative programming language (sample of such language is shown in section A.1).

A statechart model has only one state of the *statechart* type, which is itself. The states contained within *shell* execute concurrently. A *shell* state bears resemblance to an *AND* state [23] [14], but it distinguishes itself in terms of how transitions are defined as described in Section 2.2. The substates of the *shell* state are of type *composite*, aka. *regions*, and all of them will be active during execution, so I will contain all of its substates. For a composite state, I is one of its substates, and for an atomic state, I is *empty*. Operational semantics are detailed in Section 3.

Example 1. In Fig. 2 the ConStaBL model $\mathcal{M}$ is of type *statechart*. The state G – a *shell* state – is represented as $G : \{p = \mathcal{M}, S = \{E, F\}, I = \{E, F\}, V_l = \{x_1\}, V_p = \{p_1\}, V_s = \{n_1\}, \mathcal{N} : \{x_1 := 0; p_1 := 0; n_1 := 0\}, \mathcal{X} : \{x_1 := 1; p_1 := 1; n_1 := 1\}, shell\}$. I corresponds to $\{E, F\}$ (all substates of G). t_{AB} is a transition from source state A to destination state B, denoted as $t_{AB} : e_1[x_1 = 1]/p_2 := 1$. Here, t_{AB} is the name of the transition, e_1 is the event, $x_1 = 1$ is the guard, and $p_2 := 1$ is the transition action. □

2.2 Structural semantics

Containment (denoted by $\prec$) is function between two states s_1, s_2, denoted by $s_1 \prec_i s_2$, where i is the level of containment. When $i = 1$, s_2 is the *parent*

of s_1 and s_1 is *child/substate* of s_2. When $i = *$, s_2 is the *ancestor* of s_1 and s_1 is *descendent* of s_2 at an arbitrary level. Containment is antisymmetric (i.e., $\forall s_1, s_2 \in S, (s_1 \prec_* s_2) \wedge (s_2 \prec_* s_1) \implies (s_1 = s_2)$), not reflexive (i.e., $\forall s \in S, (s \not\prec_* s)$) and transitive (i.e., $\forall s_1, s_2, s_3 \in S, (s_1 \prec_* s_2) \wedge (s_2 \prec_* s_3) \implies (s_1 \prec_* s_3)$).

Only a few type substates are valid. There are 4 state types: *statechart*, *shell*, *composite* and *atomic*. The root state of a model must be of type *statechart*, and it can have substates of any type but itself. A *composite* state must have substates of type *composite*, *shell*, or *atomic*. A shell state must have substates of type *composite* only. An *atomic* state cannot have substates.

Common Ancestors(CA) For a set of states, $S' = \{s_1, s_2..., s_n\}$, $CA(S')$ is the set of states from S that is an *ancestor* of all the states in S'.

$$CA(\{s_1, s_2, ...s_n\}) = \{s \in S | s_1 \prec_* s \wedge s_2 \prec_* s \ \wedge ....s_n \prec_* s\}$$

Closest common ancestors(CCA) (denoted by $\sqcup$) of two states s_1, s_2 (denoted by $s_1 \sqcup s_2$ [3]) is a state s from the **common ancestors** set,which is an ancestor of both s_1 and s_2 such that it is not the ancestor of any other common ancestor (s').

$$s_1 \sqcup s_2 = s | s \in CA(\{s_1, s_2\}) \wedge \nexists s' \in CA(\{s_1, s_2\}) \wedge s' \prec_* s$$

$$\sqcup(\{s_1, s_2, .., s_n\}) = s | s \in CA(s_1, s_2...s_n) \wedge \nexists s' \in CA(s_1, s_2, ..s_n) \wedge s' \prec_* s$$

Further, CCA of set of transitions is the CCA of the source and destination of those transitions (i.e., $CCA(\{t_1, t_2..., t_n\}) = \sqcup(\{t_1.s, t_1.d, t_2.s, t_2.d, ..., t_n.s, t_n.d\})$).

Interlevel transitions. When the parents of the source and destination of a transition are not the same, it is an interlevel transition (i.e., $\exists t \in T | t.s.p \neq t.d.p$).

1. There can be no inter-level transitions between the regions of a shell state (i.e., $\nexists t \in T | \sqcup(\{t.s, t.d\}).\tau = shell$).
2. There can be no transition between a descendent and an ancestor (i.e., $\nexists t \in T, (t.s \prec_* t.d) \vee (t.d \prec_* t.s)$).
3. The state of type statechart cannot have any incoming or outgoing transitions (i.e., $\nexists t \in T, t.s.\tau \vee t.d.\tau = statechart$).

Example 2. In Fig. 2, $\mathcal{M} \prec_1 G$ means $\mathcal{M}$ is the parent of G and $substates(\mathcal{M}) = \{G, N\}$. Also, $\mathcal{M}$ is the ancestor of states like E, L, M and F (it is also the ancestor of all the states contained within these states). Here, $E.\tau = F.\tau = composite$ and $G.\tau = shell$.

$$CA(\{A, D\}) = \{G, \mathcal{M}\} \text{ and } CA(\{A, J\}) = \{\mathcal{M}\}$$
$$CCA(\{t_{AB}\}) = \sqcup(\{A, B\}) = E$$
$$CCA(\{t_{AB}, t_{CD}\}) = \sqcup(\{A, B, C, D\}) = G$$

[3] infix short-form of $\sqcup(\{s_1, s_2\})$

3 Operational semantics

In this section, we explain the operational semantics of ConStaBL. When an *event* is consumed, the following four major stages are executed by the simulator. A simulation processes the *events* from the event queue($\mathbb{Q}$) as shown in the algorithm 1.

1. **Find Enabled Transitions:** In this step, we identify the transitions that are enabled for a specific event. If conflicts arise among these transitions, they are reported. (detailed in section 3.2)
2. **Compute Code:** If no conflicts exist, we proceed to identify the code that must be executed during the simulation step. This is achieved by constructing the corresponding transition state tree. (detailed in section 3.3)
3. **Code Execution:** Subsequently, we execute the code that has been identified in the previous step, and we update the environment(σ) accordingly. (detailed in section 3.4)
4. **Update Configuration:** Following the code execution, the configuration is transitioned from the source configuration to the destination configuration. Additionally, the event responsible for triggering this transition is removed from the event queue. (detailed in section 3.5)

3.1 Terminologies

We now introduce the terminologies and definitions that will be used throughout the work to present the operational semantics. In order to succinctly compute the code from the statechart structure, we use an intermediate *tree* data structure and define the operations on it: *subtree*, *sliced subtree*, and *tree map*.

Tree. TREE : $node \times tree\ set \rightarrow tree$ is function such that $\text{TREE}(n, S)$ creates a tree rooted at node n with all trees in set S as subtrees of n.

Subtree. $\mathbb{T} : node \times tree \rightarrow tree$ is a function such that $\mathbb{T}(n, tr)$ gives the subtree rooted at n in a tree tr.

$$\mathbb{T}(n, tr) = \text{TREE}(n, \{\mathbb{T}(c, tr) \forall c \in childnodes(n)\})$$

Sliced subtree. $\widehat{\mathbb{T}} : node \times node\ set \times tree \rightarrow tree$ is a function such that, for any node n, the sliced subtree $\widehat{tr} = \widehat{\mathbb{T}}(n, C)$ w.r.t. to node set C is a subtree of n such that each path in $\widehat{tr}$ from its root goes to a member of C. Here, C is a set of nodes that contains some leaves of $\mathbb{T}(n, tr)$ possibly along with other nodes not in $\mathbb{T}(n, tr)$.

Treemap. $treemap : (\alpha \rightarrow \beta) \times \alpha\ tree \rightarrow \beta\ tree$ is a function such that $treemap(f, t)$ gives a tree t' that is identical in shape as the t except that the value on each node of t' is $v' = f(v)$ where v is the value on the corresponding node of t.

Example 3. Let us consider the statechart shown in the Fig. 2, the tree representation of the statechart is shown in Fig. 3(a). The nodes represent the states, and the edges represent the containment of a state within another. Here we have also represented the transitions between states as arrows between these nodes

at different levels. AND state is represented as rectangle nodes and all other types of states represented by circle nodes. The subtree of the node G is given in Fig. 3(b). The sliced subtree $\widehat{\mathbb{T}}(G, \{A, C\}, tr)$ gives a tree rooted at G and each path leads to a leaf state in the set $\{A, C\}$ as shown in Fig. 3(c). The tree map function substitutes each node in the tree to the value provided by the mapping function $s \rightarrow s.\mathcal{X}$ as shown in Fig. 3(d).

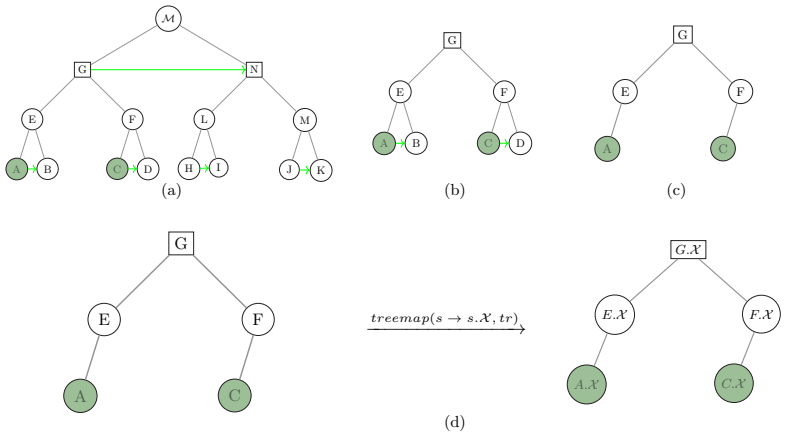


Fig. 3: (a) An equivalent state tree (tr) for the statechart in Fig. 2; (b) Subtree - $\mathbb{T}(G, tr)$ (c) Sliced Tree $\widehat{tr} = \widehat{\mathbb{T}}(G, \{A, C\}, tr)$ (d) $treemap(f, \widehat{tr})$ maps each node in given tree to the valuation of the function f. Here, $f = s \rightarrow s.\mathcal{X}$ outputs exit code $s.\mathcal{X}$ for any given state s.

Based on this, we define *Configuration State Tree*, *Transition State Tree*, *Source Side Tree*, *Destination Side Tree*, *Control Flow Graph Tree*, *Code Tree* later.

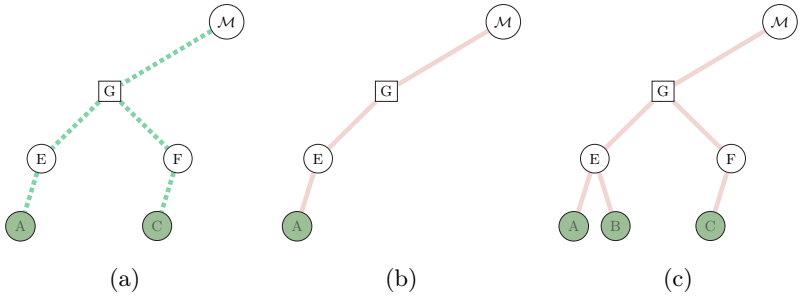
Configuration($\mathbb{C}$) is a set of *atomic* states that are active at a given point. In active hierarchical sequential composition, $\mathbb{C}$ is a singleton set and in active concurrent composition, $\mathbb{C}$ is a set of atomic states. Not all sets of atomic states are ***valid configurations***. A configuration can also be represented as Configuration State Tree (CST) - a tree st, with nodes from the root state to the states specified by the set $\mathbb{C}$. A configuration is valid iff:

1. For each composite state $s \in st$, s must have only one child in st corresponding to one of its substates.
2. For each shell state $s \in st$, s has a child node in st corresponding to each of its regions.
3. Each state in $\mathbb{C}$ is a leaf in st.
4. All ancestors of all states in $\mathbb{C}$ are contained in st.
5. No other state is included as a node in st.

Such a trees that represents a valid configurations is called configuration state tree(CST) as shown in Fig. 4.

$$st = CST(\mathbb{C}) = \widehat{\mathbb{T}}(n, \mathbb{C}, tr)$$

here, n is the root state of the statechart model.



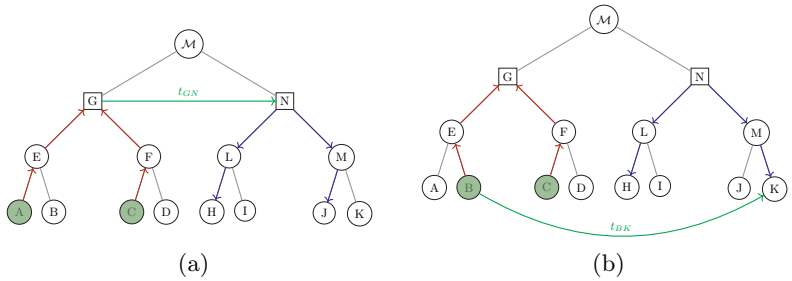
(a) (b) (c)

Fig. 4: Configuration state tree of statechart shown in Fig. 2; (a) A valid configuration state tree for $\mathbb{C} = \{A, C\}$. (b) An invalid configuration tree - shell state G does not have both its regions E and F in the state tree. (c) An invalid configuration tree - composite state E has both its substates in the state tree.

Transition state tree. When a transition is fired in a configuration $\mathbb{C}$, the states corresponding to *source side tree* ($ST_s(t, \mathbb{C})$) should be exited and the states corresponding to *destination side tree* ($ST_d(t)$) are entered. It can be viewed as two trees corresponding to the containment hierarchy of the states that they are a part of.

Example 4. Let us consider the representation of transition t_{GN} on the state tree. It is a combination of Source Side Tree and destination state tree as shown in Fig. 5. Fig. 5 shows a state containment hierarchy. The green line from state G to state N shows an enabled transition t. The current configuration is $\mathbb{C} = \{A, C\}$ and the corresponding states are shown filled with green colour. State $\mathcal{M} = G \sqcup N$. Therefore, state G is the last state to exit on the source side, and state N is the first state to enter on the destination side. The source side tree $ST_s(t, \mathbb{C})$ and the destination side tree $ST_d(t)$ are highlighted in red and blue edges, respectively. □

$ST_s(t, \mathbb{C})$ is the *Source Side Tree*, that is computed based on the transition to be taken and the current configuration $\mathbb{C}$.



(a) (b)

Fig. 5: (a)Source Side Tree (ST_s) is marked in red arrows for the transition t_{GN} and Destination Side Tree (ST_d) is marked with blue edges. The arrows in ST_s indicate that the states will be exited, and the arrows in ST_d indicate that the states will be entered when $\mathbb{C} = \{A, C\}$; (b)shows ST_s and ST_d for $\mathbb{C} = \{B, C\}$ and transition t_{BK}

$$
\begin{aligned}
&ST_s(t, \mathbb{C}) = \\
&\qquad \textbf{let } l = t.s \sqcup t.d \textbf{ in} \\
&\qquad \textbf{let } s = \\
&\qquad\qquad \textbf{if } t.s \prec_1 l \textbf{ then } t.s \\
&\qquad\qquad \textbf{else such that } s \prec_1 l \wedge t.s \prec_* s \textbf{ in} \\
&\qquad \widehat{\mathbb{T}}(s, \mathbb{C})
\end{aligned}
$$

Now, we define the *Destination Side Tree* (ST_d) as follows: Consider the list of states L from s to $t.d$. Here, the state s is a ancestor of $t.d$ such that $t.d \prec_* s \prec_1 lub$ where, $lub = t.s \sqcup t.d$. Let this list be $\text{L} = [d_1, d_2, ..., d_n]$ where $d_1 = s$ and $d_n = t.d$.

$$
\begin{aligned}
&ST_d(t) = f(d_1) \\
&\qquad where, f(d_i) = \\
&\qquad\qquad \textbf{if } i = n \textbf{ then } ST_i(d_i) \\
&\qquad\qquad \textbf{if } i \neq n \wedge d_i.\tau = composite \textbf{ then } \textsc{tree}(d_i, f(d_{i+1})) \\
&\qquad\qquad \textbf{if } i \neq n \wedge d_i.\tau = shell \textbf{ then } \textsc{tree}(d_i, [f'(c_1), f'(c_2), ..., f'(c_m)]) \\
&\qquad\qquad\quad \textbf{where,} \\
&\qquad\qquad\quad d_i.childnodes = c_1, c_2, ..., c_m \\
&\qquad\qquad\quad f'(c) = \textbf{if } c \neq d_{i+1} \textbf{ then } ST_i(c) \\
&\qquad\qquad\qquad\quad \textbf{else } f(c)
\end{aligned}
$$

Here, **Initial subtree (ST_i)** for any node n is given by the following:

$$
\begin{aligned}
ST_i(n) = &\textsc{tree}(n, \{ST_i(n.I)\}) \textbf{ if } n.\tau \in \{Composite, Statechart\} \\
&\textsc{tree}(n, \{\}) \textbf{ if } n.\tau = Atomic \\
&\textsc{tree}(n, \{ST_i(c),\ \forall c | c \in childnodes(n)\}) \textbf{ if } n.\tau = Shell
\end{aligned}
$$

Example 5. For the transition state tree shown in Fig. 5(b), when $\mathbb{C} = \{B, C\}$ and the transition t_{BK} to be taken, two state trees are to be computed. The source side tree (St_s) - a slice of the configuration state tree. The substeps in computing the destination side tree (St_d) as per our definition, is illustrated in Fig. 6. The input for the computing function is a list L, that contains the nodes $[N, M, K]$ and the function $f(d1)$ is a recursive function that computes the Destination Side Tree. Dotted arrows show the intermediate steps and the curved arrow shows the input with the final output on applying the function $f(d_1)$.

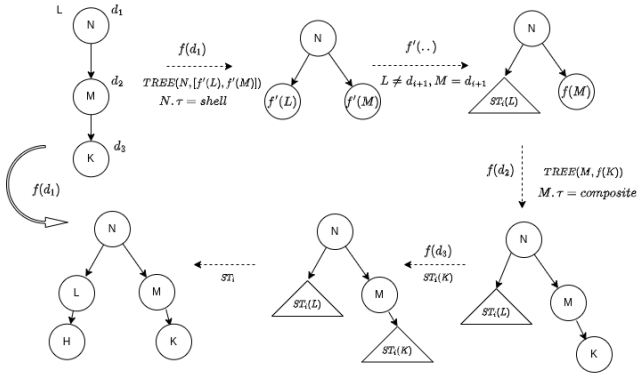


Fig. 6: Computation of destination side tree for transition t_{BK} as shown in Fig. 5(b)

The dependency graph of all the terminologies used in the paper is available in Appendix A.2.

Now we present the detail of the stages introduced in the beginning of this section. The algorithm that combines these stages to simulate the statechart is shown in algorithm 1. The next subsections will focus on explaining the internals of these stages that are integral to the operational semantics.

3.2 Stage 1: Find Enabled Transitions

Enabled Transition. In a given configuration $\mathbb{C}$, when an event e has arrived, a transition t is enabled when all three of the following conditions hold:

1. $t.s \in CST(\mathbb{C})$, (i.e., t's source state is one of the vertices of the configuration state tree of $\mathbb{C}$).
2. $t.e = e$, (i.e., t's trigger event is the same as the one that has arrived).

Algorithm 1: ConStaBL Simulator

Input: SC: Statechart, $\mathbb{Q}$: Event[]
Output: Configuration ($\mathbb{C}$), Environment (σ)

procedure SIMULATE($SC, \mathbb{Q}$)
▷ Initializing σ - environment, $\mathbb{C}$ - configuration
$\mathbb{C}, \sigma \leftarrow initialize(SC)$
while $e \in \mathbb{Q}$ **do**
  ▷ stage 1 : Find Transition
  $\mathcal{T} \leftarrow$ `findEnabledTransitions(e);`
  ▷ Conflict Identification
  `findNonDeterminism(`$\mathcal{T}$`);`
  ▷ stage 2 : Compute Code
  if $\mathcal{T} \neq \emptyset$ **then**
    ▷ Here, $\mathcal{C} \rightarrow$ computeCode, ConcurrentCode=$[c_1|c_2|..|c_n]$
    $\mathcal{C}(\mathcal{T}, \mathbb{C}) = [\mathcal{C}(t_1, \mathbb{C})|\mathcal{C}(t_2, \mathbb{C}), ...|\mathcal{C}(t_n, \mathbb{C})]$
    ▷ stage 3 : Execute Code
    $\sigma \vdash$ `executeCode`$(\mathcal{C}(\mathcal{T}, \mathbb{C})) \rightarrow \sigma'$
    ▷ stage 4 : Update Configuration
    $\mathbb{C}' = \bigcup_{t \in \mathcal{T}} leaves(ST_d(t))$
  $\mathbb{Q} \leftarrow \mathbb{Q} \setminus$ `e`

procedure FINDENABLEDTRANSITIONS(e: Event)
▷ finding transitions that originate from any node of current CST - Configuration State Tree
`return` $\{t | t.s \in CST(\mathbb{C}) \wedge t.e = e \wedge \sigma \vdash t.g \Downarrow true\}$

procedure FINDNONDETERMINISM($\mathcal{T}$: Transition set)
▷ Identifying if ST_s - Source State Tree of two transitions overlap
if $\sigma, \mathbb{C} \vdash \forall t_1, t_2 \in \mathcal{T}, ST_s(t_1, \mathbb{C}) \cap ST_s(t_2, \mathbb{C}) \neq \phi$ **then**
  `raise` $conflict(t_1, t_2)$

procedure COMPUTECODE(t:Transition, $\mathbb{C}$: Configuration)
▷ $CFGTree_s$ and $CFGTree_d$ - CFG Tree derived from ST_s and ST_d resp.
$CFGTree_s(t, \mathbb{C}) = treemap(s \rightarrow$ `cfg`$(s.\mathcal{X}), ST_s(t, \mathbb{C}))$
$CFGTree_d(t) = treemap(s \rightarrow$ `cfg`$(\langle init(s.\mathcal{V}_l), s.\mathcal{N} \rangle), ST_d(t))$
▷ $\mathcal{C}_s(t, \mathbb{C})$ and $\mathcal{C}_d(t, \mathbb{C})$ - Code derived from $CFGTree_s$ and $CFGTree_d$ resp.
$\mathcal{C}_s(t, \mathbb{C}) = \mathcal{C}(CFGTree_s(t, \mathbb{C}))$
$\mathcal{C}_d(t, \mathbb{C}) = rev(CFGTree_d(t))$
▷ `cfg`$(t.a)$ - Code derived from $t.action$, `cfg` - Control Flow Graph, SequenceCode = $\langle c_1, c_2, ..c_n \rangle$
$Code(t, \mathbb{C}) = \langle \mathcal{C}_s(t, \mathbb{C})$, `cfg`$(t.a)$, $\mathcal{C}_d(t, \mathbb{C}) \rangle$
`return` $Code(t, \mathbb{C})$

3. $\sigma \vdash t.g \Downarrow true$, (i.e., in the given value environment σ, t's guard evaluates to *true*). Here, σ is the *environment* that maintains binding of variables to current values is given by σ.

It is possible that, for a given configuration $\mathbb{C}$ and event e, the number of enabled transitions is zero, one, or more than one. Hence, to perform the next step, we compute the set of enabled transitions $\mathcal{T}$. For example, in the example model in Fig. 2, if $\mathbb{C} = \{A, C\}$ and $e = e_1$, then $\mathcal{T} = \{t_{AB}, t_{CD}\}$.

The Source Side Tree helps to identify the transitions that conflicts to identify nondeterminism.

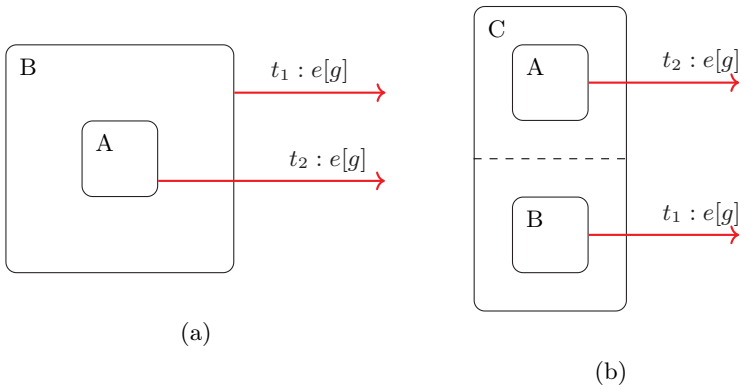


Fig. 7: Conflicting Transitions: (a) if $\mathbb{C}$={A}, then both $t_1.s = A$ and $t_2.s = B$ would cause A.$\mathcal{X}$ and B.$\mathcal{X}$ to execute; (b) Both t_1 and t_2 would cause C.$\mathcal{X}$ to execute.

Conflicting transitions. For a given configuration $\mathbb{C}$ and event e, two enabled transitions are said to *conflict* if, fired concurrently, they lead to exiting the same state.

$$\sigma, \mathbb{C} \vdash \forall t_1, t_2 \in \mathcal{T},$$
$$conflict(t_1, t_2) = \textbf{ if } ST_s(t_1, \mathbb{C}) \cap ST_s(t_2, \mathbb{C}) \neq \phi \textbf{ ~~then~~ ~~true~~ ~~else~~ ~~false.~~}$$

In the above, we use the set intersection ($\cap$) to indicate the intersection between the trees blocks in $ST_s(t_1, \mathbb{C})$ and $ST_s(t_2, \mathbb{C})$.

Example 6 (Conflicting transitions). Fig. 7 shows two scenarios in which transitions may conflict. Fig. 7(a) shows a case in a sequential (non-concurrent) scenario where two enabled transitions t_1 and t_2 emerge from two states that are in an ancestor-descendent relationship. This case, which we call *nondeterminism*, makes it impossible to determine which transition to take, unless we use an external conflict resolution rule (e.g. clockwise arrangement [23], outside-in [7] etc). We consider such conflict resolution undesirable, as it may sneak in undesirable behaviour without the knowledge of the modelling engineer. Instead, such cases should be flagged out wherever possible, and the engineer should be given the choice to resolve in a way as desirable to him/her. Fig. 7(b) shows a

case of a concurrent model. Two enabled transitions emerge from the two regions A and B of a shell state C. Both transitions would lead to the execution of C.$\mathcal{X}$, which would be invalid.

3.3 Stage 2: Compute Code

We will now provide the specifics of how to identify the code to be executed. To compute the *transition code*, its corresponding *Source Side Tree* and *Destination Side Tree* are converted to the respective *control flow graph tree*. The transition code is the code composed from the respective CFGTree and transition code blocks. This section gives the details on composition of code from the identified non-conflicting transitions.

Control Flow Graph Tree. The source side CFG tree for an enabled transition t in a configuration $\mathbb{C}$ is the tree of exit code blocks of the corresponding Source Side Tree $ST_s(t, \mathbb{C})$ of t. Similarly, the destination side CFG tree for an enabled transition t in a configuration $\mathbb{C}$ is the tree of exit code blocks of the corresponding Destination Side Tree $ST_d(t)$. Mathematically:

$$CFGTree_s(t, \mathbb{C}) = treemap(s \rightarrow \text{CFG}(s.\mathcal{X}), ST_s(t, \mathbb{C}))$$
$$CFGTree_d(t) = treemap(s \rightarrow \text{CFG}([init(s.V), s.\mathcal{N}]), ST_d(t))$$

Here, the constructor CFG takes a code block (in the form of its abstract syntax tree) and gives its control flow graph (CFG). Control flow graphs are presented in detail in Section 3.4 where it will be needed to discuss code execution. Here, it suffices to proceed with an informal understanding of control flow graphs. While computing the source side CFG tree $CFGTree_s(t, \mathbb{C})$, we include the CFGs of the exit code of each state in the Source Side Tree $ST_s(t, \mathbb{C})$. While computing the destination side CFG tree $CFGTree_d(t)$, we include the CFGs of the concatenation of the variable initialisation code and entry code of each state in the Destination Side Tree $ST_d(t)$.

Code. The code executed during a simulation step is computed from $CFGTree$. The code composition can be either sequential or concurrent and is represented by the following datatype $Code$ as follows:

$$\begin{aligned} Code &= Seq([c_1, c_2, ..., c_n]) \text{ where } c_1, c_2, ..., c_n : Code \\ &= Conc(\{c_1, c_2, ..., c_n\}) \text{ where } c_1, c_2, ..., c_n : Code \\ &= CFGCode(cfg) \text{ where } cfg = \text{CFG}(b) \text{ and } b \text{ a code block} \end{aligned}$$

Here, Seq is an abbreviation of sequential code and $Conc$ is an abbreviation of concurrent code. Henceforth, we abbreviate $Seq([c_1, c_2, ..., c_n])$ as $\langle c_1, c_2, ...c_n \rangle$ and $Conc(\{c_1, c_2, ..., c_n\})$ as $[c_1|c_2|...|c_n]$.

Code reversal function done to a code $c : Code$ is defined as follows:

$rev(c) =$

c **if** c **is** $CFGCode$

$\langle rev(c_n), rev(c_{n-1})..., rev(c_2), rev(c_1)\rangle$ **if** c **is** $\langle c_1, c_2, ..., c_n\rangle$

$[rev(c_1)|rev(c_2)|...|rev(c_{n-1})|rev(c_n)]$ **if** c **is** $[c_1|c_2|...|c_n]$

Code Containment Tree. As is clear, the *code* type is recursive and can represent a hierarchical arrangement of code blocks. We call this hierarchy as code hierarchy and represent the same by a tree called *code containment tree*. Sequence code and concurrent code form the internal nodes of this tree and CFG code form the leaf nodes of this tree. This tree is useful in navigating the code during code execution. Please note that this is different from the code tree mentioned in Section 3.3 which mimics the state hierarchy. As shown in the Fig. 8(a), for any event in a given configuration, the codes of the non-conflicting enabled transitions are concurrently composed. The transition code is further computed with the help of CFGTree introduced in the beginning of this section.

Transition Code. The code to be executed with a transition t gets fired in a configuration $\mathbb{C}$ is given by:

$$\mathcal{C}(t, \mathbb{C}) = \langle \mathcal{C}(CFGTree_s(t, \mathbb{C})) \ , \text{CFG}(t.a) \ , rev(\mathcal{C}(CFGTree_d(t)))\rangle$$

Note the code reversal done to compute code from CFGTree of destination.

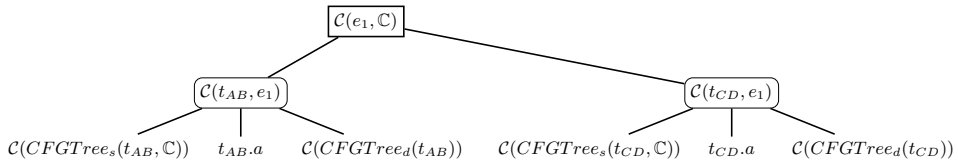
The function $\mathcal{C} : CFGTree \rightarrow Code$ takes a CFG tree ct, and gives its code as follows:

$$\begin{aligned}\mathcal{C}(ct) &= CFGCode(s.cfg) \text{ if } ct = \text{TREE}(s, \{\}) \\ &= \langle CFGCode(s.cfg) \ , \mathcal{C}(c)\rangle \text{ if } ct = \text{TREE}(s, \{c\}) \\ &= CFGCode(s.cfg) \ [\mathcal{C}(ct_1)|\mathcal{C}(ct_2)|...|\mathcal{C}(ct_n)] \\ &\qquad \text{if } ct = \text{TREE}(s, \{c_1, c_2, ..., c_n\})\end{aligned}$$

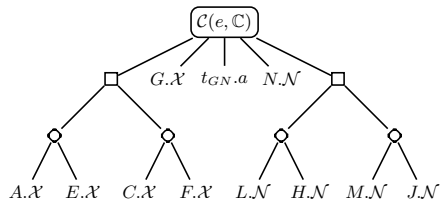
Example 7 (Code containment tree).

The code shown in Example 9 are pictorially illustrated in the form of code containment trees in Fig. 8. We use rectangular nodes with pointed vertices to show a concurrent code, rectangular nodes with rounded corners to show a sequential code and an unbordered node to show CFG codes. □

Finally, the entire code to be executed during a simulation step is the concurrent composition of the codes of the enabled transitions. For the set of enabled transitions $\mathcal{T} = \{t_1, t_2, ..., t_n\}$, code is given by: $[\mathcal{C}(t_1, \mathbb{C})|\mathcal{C}(t_2, \mathbb{C}), ...|\mathcal{C}(t_n, \mathbb{C})]$



(a)



(b)

Fig. 8: Example: Code containment tree of the code in Example 9.

Example 8. In Fig. 4(c), when transition t_{GN} is fired in configuration $\{A, C\}$, the code that gets executed is:

$$\begin{aligned}\mathcal{C}(t_{GN}) = Seq([&Conc(\{Seq([A.\mathcal{X}, E.\mathcal{X}]), Seq([C.\mathcal{X}, F.\mathcal{X}])\}), G.\mathcal{X},\\ &t_{GN}.a,\\ &N.\mathcal{N}, Conc(\{Seq([L.\mathcal{N}, H.\mathcal{N}]), Seq([M.\mathcal{N}, J.\mathcal{N}])\})])\end{aligned}$$

In abbreviated form, the above is written as:
$\mathcal{C}(t_{GN}) = \langle[\langle A.\mathcal{X}, E.\mathcal{X}\rangle|\langle C.\mathcal{X}, F.\mathcal{X}\rangle], G.\mathcal{X}, t_{GN}.a, N.\mathcal{N}, [\langle L.\mathcal{N}, H.\mathcal{N}\rangle|\langle M.\mathcal{N}, J.\mathcal{N}\rangle]\rangle$ □

3.4 Stage 3: Code Execution

After the transition code $\mathcal{C}(\mathcal{T}, \mathbb{C})$ is computed, the simulator will proceed to execute it. In this subsection, we present the details of this process.

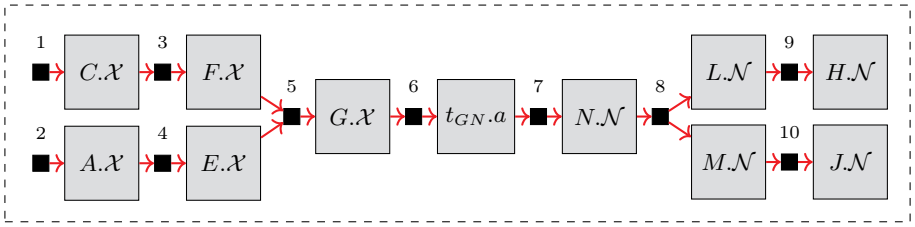
Each step in code execution involves the interpretation of single statements in the code. The code can be concurrent. Unlike many existing implementations of Statecharts [23] [14] [4] [22] [18] [8], we allow arbitrary interleaving between code executed concurrently. We assume individual instructions to execute atomically. Though finer grained interleaving is possible in principle, we consider that not to be necessary feature at the statechart modelling level.

Example 9 (Transition Code). Consider the model shown in Fig. 2. Source configuration $\mathbb{C} = \{A, C\}$. For this example, assume that the code blocks in the model are as follows:

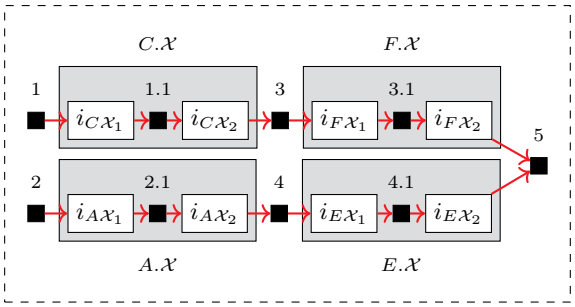
- **Entry codes:** $A.\mathcal{N} = \{i_{A\mathcal{N}_1}; i_{A\mathcal{N}_2};\}$, $B.\mathcal{N} = \{i_{B\mathcal{N}_1}; i_{B\mathcal{N}_2};\}$ etc.
- **Exit codes:** $A.\mathcal{X} = \{i_{A\mathcal{X}_1}; i_{A\mathcal{X}_2};\}$, $B.\mathcal{X} = \{i_{B\mathcal{X}_1}; i_{B\mathcal{X}_2};\}$ etc.
- **Transition action codes:** $t_{AB} = \{i_{AB_1}; i_{AB_2};\}$, $t_{CD} = \{i_{CD_1}; i_{CD_2};\}$ etc.

1. **Case 1 (Concurrent and disjoint).** Suppose, the input event is e_1. The code to be executed in this case is $[\mathcal{C}(t_{AB}, \mathbb{C})|\mathcal{C}(t_{CD}, \mathbb{C})] = [\langle A.\mathcal{X},\ t_{AB}.a,\ B.\mathcal{N}\rangle, \langle C.\mathcal{X},\ t_{CD}.a,\ D.\mathcal{N}\rangle]$. A possible trace generated when the above code executes: $i_{A\mathcal{X}_1}$, $i_{C\mathcal{X}_1}$, $i_{A\mathcal{X}_2}$, $i_{C\mathcal{X}_2}$, i_{AB_1}, i_{CD_1}, i_{AB_2}, i_{CD_2}, $i_{B\mathcal{N}_1}$, $i_{D\mathcal{N}_1}$, $i_{B\mathcal{N}_2}$, $i_{D\mathcal{N}_2}$.
2. **Case 2 (Concurrent, joining and forking).** Suppose, the input event is e. The code to be executed is $\mathcal{C}(t_{GN}, \mathbb{C}) = \langle[\langle A.\mathcal{X}, E.\mathcal{X}\rangle|\langle C.\mathcal{X}, F.\mathcal{X}\rangle], G.\mathcal{X}, t_{GN}.a, N.\mathcal{N}, [\langle L.\mathcal{N}, H.\mathcal{N}\rangle|\langle M.\mathcal{N}, J.\mathcal{N}\rangle]\rangle$. A possible trace generated when the above code executes: $i_{A\mathcal{X}_1}$, $i_{C\mathcal{X}_1}$, $i_{A\mathcal{X}_2}$, $i_{C\mathcal{X}_2}$, $i_{E\mathcal{X}_1}$, $i_{F\mathcal{X}_1}$, $i_{E\mathcal{X}_2}$, $i_{F\mathcal{X}_2}$, $i_{G\mathcal{X}_1}$, $i_{G\mathcal{X}_2}$, i_{GN_1}, i_{GN_2}, $i_{N\mathcal{N}_1}$, $i_{N\mathcal{N}_2}$, $i_{L\mathcal{N}_1}$, $i_{M\mathcal{N}_1}$, $i_{L\mathcal{N}_2}$, $i_{M\mathcal{N}_2}$, $i_{H\mathcal{N}_1}$, $i_{J\mathcal{N}_1}$, $i_{H\mathcal{N}_2}$, $i_{J\mathcal{N}_2}$.

□



(a)



(b)

Fig. 9: Example: code and control points

Control flow graph (CFG). A control flow graph $G(V, E, \mathcal{N}, \mathcal{X})$ is a graph with a set of vertices/nodes V and (control flow) edges E. $V = V_I \cup V_D$. Here,

- $\mathcal{N}$ is a unique node, called the entry node of the CFG g, referred to as $g.\mathcal{N}$.
- $\mathcal{X}$ is unique node $\mathcal{X}$, called the exit node of the CFG g, referred to as $g.\mathcal{X}$. The exit node has no successors.

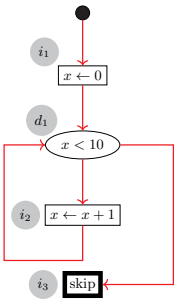
- V_I is the set of instruction nodes, i.e., nodes with an instruction in them with a possible side-effect in the action language. An instruction node $v_I \in V_I$ has zero or one successor (referred to as $v_I.s$).
- V_D is the set of decision nodes. Each decision node $v_D \in V_D$ has a boolean expression in the action language called the condition, referred to as $v_D.c$. A decision node has two successors, namely the *then* successor (referred to as $v_D.t$) and the *else* successor (referred to as $v_D.e$).

Additionally:

- A CFG may have a single node, in which case, $\mathcal{N} = \mathcal{X}$.
- $\mathcal{N} \in V$. $V_{\mathcal{X}} \in V_I$.

Example 10 (Control flow graph).

```
x = 0;
while(x < 10)
  x++;
```



$g.\mathcal{N} = i_1$, $g.\mathcal{X} = i_3$, $g.V_I = \{i_1, i_3\}$, $g.V_d = \{d_1\}$, $i_1.s = d_1$, $d_1.t = i_2$, $d_1.e = i_3$

Fig. 10: Control flow graph

Control Points. To compute the sequence in which the code blocks get executed, we can think of a partial order (or the corresponding directed acyclic graph) of code blocks such that code blocks in the same sequence are in that order. The nodes of the code sequence DAG correspond to the leaf nodes of the code containment tree. The ordering relations between various nodes of this DAG can be directly derived from the structure of the code containment tree.

Example 11 (Code partial order). The model in Fig. 2(a) in configuration $\mathbb{C} = \{A, C\}$ with input event e causes t_{GN} to fire. The code partial order corresponding to this simulation step is shown in Fig. 9(a). As can be seen, the nodes of this DAG correspond to the leaves of the code containment tree shown in Fig. 8(b).

□

As we have pointed out, instructions in concurrently composed pieces of code can interleave. Hence, we work with the idea of *control points* – points in the executing code which are visible to the simulator and are subject to interleaving in case of concurrent composition.

Example 12 (Control Points). A portion of the code partial order shown in Fig. 9(a) is shown in Fig. 9(b). The control points here are: 1, 1.1, 3, 3.1, 2, 2.1, 4, 4.1, 5 and so on. Here, control points like 1, 2, 3, 4, 5 etc. are control points outside individual control flow graphs. However, control points like 1.1, 2.1, 3.1, 4.1 etc. are control points inside individual control flow graphs.

In Fig. 9(b), the two sequences – 1, 1.1, 3, 3.1 and 2, 2.1, 4, 4.1 – are concurrently composed and execute in two concurrent threads. As is clear from the figure, 1.1 is reached strictly after reaching 1; 3 after 1.1 etc. However, Any of 1.1 or 2.1 may be reached earlier depending on which of the two concurrent threads progresses first. At a point when the execution progresses beyond 3.1 and 4.1, the two threads join and give place to a single thread which is at control point 5. □

During code execution, there can be multiple active threads. The simulator keeps track of its progress through each running thread by maintaining a set of control points CP. Every code simulation step will cause the execution of one of the instructions immediately after one of the control points $cp \in CP$. Choice of cp is done randomly from CP. After this, cp gets removed from CP and gets replaced by zero, one or more control points, as the case may be, which succeed cp in the code partial order.

$$
\begin{aligned}
parent(n) = {} & n' \textbf{ if } n' = \text{Seq}([...n...]) \textbf{ or } \text{Conc}(\{...n...\}) \\
& nil \textbf{ otherwise.} \\
first(n) = {} & \textbf{ if } \ n \textbf{ is a } CFGCode \textbf{ then } \{n.\mathcal{N}\} \\
& \textbf{if } n = \text{Seq}([c_1, c_2, ...c_n]) \textbf{ then } first(c_1) \\
& \textbf{if } n = Conc(\{c_1, c_2, ..., c_n\}) \textbf{ then } \bigcup_{c_i \in \{c_1, c_2, ..., c_n\}} first(c_i) \\
nextCFGCodes(c) = {} & \textbf{ if } parent(c) = nil \textbf{ then } \{\} \\
& \textbf{if } parent(c) = Seq([c_1, c_2, ...c_n]) \wedge c = c_i \textbf{ then} \\
& \quad \textbf{if } c_i \neq c_n \textbf{ then } first(c_{i+1}) \\
& \quad \textbf{else } nextCFGCodes(parent(c)) \\
& \textbf{if } parent(c) = Conc(\{c_1, c_2, ..., c_n\}) \textbf{ then} \\
& \quad nextCFGCodes(parent(c)) \\
nextCP(n) = {} & \textbf{ if } (n = n.cfg.\mathcal{X}) \textbf{ then } nextCFGCodes(n) \\
& \textbf{else } succ(n, n.cfg)
\end{aligned}
$$

Fig. 11: Functions - Operational semantics of code simulation

The function $nextCP$ computes the set of control points that replace the currently processed control point. An internal control point cp will be replaced by one of its successors in $cp.cfg$, the CFG it belongs to. However, if cp is an exit node of $cp.cfg$, then the next of control points to replace it in CP are the

entry node of other CFGs. Such CFGs (and their unique entry node) can be identified using the functions $nextCFGCodes$ and $first$. Please note that these are computed through an implicit traversal of code containment tree, discussed in Section 3.3.

When a member $cp' \in nextCP(cp)$ happens to be a join point, (i.e., having multiple predecessor control points), the situation needs to be dealt with separately. This is a point where multiple threads will collapse into one at an appropriate time. This is when all predecessors of cp' have been processed, thus bringing the control points of all these thread to cp'. This concludes their execution. Hence, when the first of these threads reaches cp', we start keeping track of the number of threads waiting to join at cp', and cp' is kept waiting. When, all these threads reach cp', we insert cp' into CP scheduling it for execution in the next code simulation step.

Note that due to the complex hierarchy in which the code blocks in a statechart may be arranged at runtime, it may appear that keeping track of them would be very complex. However, with the above approach, we are able to essentially collapse the control front (the control points of all the active threads) into a flat set.

Example 13 (Code Simulation Trace). Consider the code partial order shown in Fig. 9(b). We track the successive values of the current control point set CP and the join points JP in a typical code simulation.

$$(CP, JP) = (\{\}, \{\}), (\{1, 2\}, \{\}), (\{1, 2.1\}, \{\}), (\{1, 4\}, \{\}), (\{1.1, 4\}, \{\})$$
$$(\{3, 4\}, \{\}), (\{3.1, 4\}, \{\}), (\{4\}, \{5 \mapsto \{4.1\}\}),$$
$$(\{4.1\}, \{5 \mapsto \{4.1\}\}), (\{5\}, \{\}), ...(\{7.1\}, \{\}), (\{8\}, \{\}),$$
$$(\{8.1, 8.2\}, \{\}), ...(\{9.1, 10.1\}, \{\}), (\{9.1\}, \{\}), (\{\}, \{\})$$

Following points in the above trace are noteworthy:

- The set CP and map JP, both start off empty and end up empty over a simulation step.
- The join point 5 is reached when the control front passes control point 3.1. At this point, an entry is added to JP with 5 as key and $\{4.1\}$ as value, as 4.1 is the previous control point to 5 in the currently active threads (only one in this case, as the thread corresponding to 3.1 has already finished execution) which will eventually join at 5.
- As the control front passes 4.1, it gets removed from the values of key 5 in JP. This makes the value set of 5 empty in JP. Therefore, 5 is removed from JP, and added to CP.
- Control point 8 is fork point. 8.1 and 8.2 are the internal control points of $L.\mathcal{N}$ and $M.\mathcal{N}$ respectively. As soon as the control front moves past 8, it is replaced by 8.1 and 8.2 in CP.
- Code simulation for this simulation step concludes when CP becomes empty.

Operational Semantics – Code Simulation. During the code simulation of a simulation step, the code data structure `C` that gets executed remains constant. In this part of the discussion, we will omit mentioning it in most places.

The first step of code simulation involves populating the control front CP with appropriate set of CFG Nodes in the code. These are nothing but the initial entry nodes of the initial CFGCodes of the code. In Fig. 12, we show this in rule CODE-SIM-INIT. Rule CODE-SIM-INTERNAL presents the operational semantics when the currently processed CFG node n is not an exit node of its CFG. Its successor node in the CFG n' is computed as $nextCFGNode(n, \sigma)$. This is a direct successor if n corresponds to an instruction node. However, if n is a decision node, n' depends on valuation of the condition expression $n.c$ of n. If $n.c$ evaluates to true in the current value environment σ, then n' is n's then-successor; otherwise, it is the else-successor of n.

The rule CODE-SIM-EXIT-NODE handles the case when the current CFG node n chosen randomly from the control front CP (using the function any) is the exit node of its CFG. As discussed above, this step could lead to joining (i.e., two threads collapsing into one) or forking (one thread giving place to multiple threads) of code execution. This could also lead us to the completion of code simulation when CP becomes empty again. In this step, all successors of n which are not join points (shown by set N), directly get added to CP. The other set of successors are join points, shown by set $J = J_i \cup J_c \cup J_o$. J_i is the set of new join points, i.e., for each member of J_i, n is the first of the predecessors to have got processed. In this step, all members of J_i get added to JP. J_c is the set of those join point successors, one of whose predecessors have already been processed in an earlier code simulation step, and hence, they had previously been added to JP. Even after the processing of n, there are other predecessors of the members of J_c which await processing; hence all members of J_c continue to be part of JP. J_o are those join point successors, all of whose predecessors have completed their processing in this step, n being the last of them. Hence, all members of J_o are removed from JC and inserted to the control front CP.

Note that CODE-SIM-EXIT-NODE subsumes the case when the control front CP empties out, which means that the code execution halts.

CODE-SIM-INIT

$$\frac{}{\sigma, \{\}, \{\} \longrightarrow \sigma', first(\mathtt{C}), \{\}}$$

CODE-SIM-INTERNAL

$$\frac{\begin{array}{c} n = any(CP) \\ n' = nextCFGNode(n, \sigma) \qquad \sigma \vdash n.inst \Downarrow \sigma' \qquad CP' = CP \setminus \{n\} \cup \{n'\} \end{array}}{\sigma, CP, \{\} \longrightarrow \sigma', CP', \{\}}$$

CODE-SIM-EXIT-NODE

$$\frac{\begin{array}{c} n = any(CP) \qquad n = n.cfg.\mathcal{N} \qquad \sigma \vdash n.inst \Downarrow \sigma' \\ next(n) = N \cup J_i \cup J_c \cup J_o \qquad N = \{n' | pred(n') = \{n\} \wedge n' \in CP'\} \\ J_i = \{j |\ |pred(j)| > 1, j \notin JP \wedge JP'(j) = pred(j) \setminus \{n\}\} \\ J_c = \{j \mid |pred(j)| > 1 \wedge |JP(j)| > 1 \wedge JP'(j) = pred(j) \setminus \{n\}\} \\ J_o = \{j |\ JP(j) = \{n\}\} \qquad CP' = CP \setminus \{n\} \cup N \cup J_o \qquad JP' = JP \cup J_i \setminus J_o \end{array}}{\sigma, CP, JP \longrightarrow \sigma', CP', JP'}$$

Fig. 12: Operational semantics of code simulation

Example 14 (Operational semantics – Code simulation). In Table 1, we present an example of the trace shown in Example 13 to illustrate the role of sets J_i, J_c and J_o.

CP	JP	n	N	J_i	J_c	J_o	CP'	JP'
{}	{}	-	{}	{}	{}	{}	{}	{}
$\{1, 2\}$	{}	2	$\{2.1\}$	{}	{}	{}	$\{1, 2.1\}$	{}
$\{1, 2.1\}$	{}	2.1	$\{4\}$	{}	{}	{}	$\{1, 4\}$	{}
$\{1, 4\}$	{}	1	$\{1.1\}$	{}	{}	{}	$\{1.1, 4\}$	{}
$\{1.1, 4\}$	{}	1.1	$\{3\}$	{}	{}	{}	$\{3, 4\}$	{}
$\{3, 4\}$	{}	3	$\{3.1\}$	{}	{}	{}	$\{3.1, 4\}$	{}
$\{3.1, 4\}$	{}	3.1	{}	$\{5\}$	{}	{}	$\{4\}$	$\{5 \mapsto \{4.1\}\}$
$\{4\}$	$\{5 \mapsto \{4.1\}\}$	4	$\{4.1\}$	{}	$\{5\}$	{}	$\{4.1\}$	$\{5 \mapsto \{4.1\}\}$
$\{4.1\}$	$\{5 \mapsto \{4.1\}\}$	4.1	{}	{}	{}	$\{5\}$	$\{5\}$	{}
$\{5\}$	...		...	...	...	...	...	...

Table 1: Details of code simulation trace tracking the sets J_i, J_c and J_o as per the operational semantics.

□

3.5 Stage 4: Update Configuration

The new configuration $\mathbb{C}'$ is given by the following function:

$$\mathbb{C}' = \mathbb{C} \textbf{ if } \mathcal{T} = \{\}$$
$$\bigcup_{t \in \mathcal{T}} leaves(ST_d(t)) \textbf{ otherwise.}$$

If there are no enabled transitions, the configuration does not change. The event gets lost. If there are one or more enabled transitions, the new configuration is the union of the set of leaf nodes of the Destination Side Tree of each of these transitions.

We have implemented a simulator according to the operational semantics presented in section 3. Fig. 1 shows an overview of the integral units of the simulator. The *simulator* operates in two modes: *auto* and *interactive*. In *auto* mode the event queue is populated prior to the beginning of execution. In interactive mode, events are taken as input during execution. It works on the AST, and the simulation (procedure SIMULATE) consumes a sequence of events and proceeds with execution as shown in algorithm 1. The simulator maintains the current execution state, configuration, and environment at each execution point.

Our simulator allows users to simulate the models written in *ConStaBL* and is available in our GitHub repository[4].

4 Fuzz Testing of Statecharts

Statecharts are widely used in various domains [12], including safety-critical systems like avionics and automotive control systems. These systems often incorporate subsystems for navigation, obstacle detection, and vehicle communication, resulting in complex interactions and numerous possible execution scenarios. Manual simulation of such systems becomes challenging due to the sheer number of valuations and execution traces. To address this challenge, we have harnessed the advantages of fuzz testing specifically for statecharts. Fuzz testing involves generating a large volume of inputs and observing how the system responds, aiming to uncover software bugs and anomalies. Integrating fuzz testing with statecharts is a novel concept that has not been explored in existing literature.

In this study, we employ fuzzing for the following purposes:

1. Simulating large models with thousands of events, a task that is cumbersome when performed manually. This verifies the capacity and capability of our simulator to handle extensive models and events.
2. Confirming the simulator's functionality with respect to all potential transition combinations in accordance with the structural semantics.
3. Assessing the presence of both desired and undesired properties within the model.

We have integrated Jazzer [17], the java-based fuzzer. The operation of Jazzer is illustrated in Fig. 13. When a *property under test* is detected during simulation, we call $throw\ new\ FuzzerIssue()$ command to halt the fuzzer and record all the inputs generated by the fuzzer up to that point. The inputs are encoded in a serialized format and a reproducer file named $CrashFile.java$ is created. This file calls the *simulate* method using the serialized object as input. By running the crash file, the test can be replicated. Note that, we call $throw\ new\ FuzzerIssue()$ on detection of both desired and undesired properties.

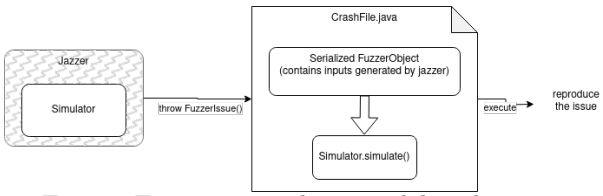


Fig. 13: Fuzzing statechart model with jazzer

[4] Repo link: https://github.com/sujitkc/statechart-verification/tree/skc-simulator-test

4.1 Testing properties of statechart models

When employing fuzz testing on statecharts, we can uncover several *properties* in the statechart at runtime, such as:

1. **Nondeterminism** - Although ConStaBL semantics does not employ priorities for transitions, it is still valuable for designers to be aware of the existence of nondeterminism (as explained in detail in section 3 - *conflicting transitions*). Detecting nondeterminism is a runtime phenomenon, as it depends on the runtime valuation of variables in the *guard* of the enabled transitions.
2. **Configuration reachability** - We intend to check both desired and undesired configurations are reachable. Undesired configurations can occur in concurrent statecharts when multiple regions enter a combination of states that, while individually valid, lead to undesired behaviour. For example, while it is acceptable for two traffic signals to be independently green, if they are at the same junction displaying opposite signals, it can cause accidents. Therefore, it is crucial to examine the orthogonal composition of traffic lights and determine if there are any scenarios where the configuration becomes $\{green, green\}$. When we test for the *reachability* as discussed in next section, a configuration can be desired as well.

We augment the algorithm 1 to throw the fuzzer issue as shown below:

$\text{conflict}(t_1, t_2) \implies$
throw new FuzzerIssue("Nondeterminism detected");

if $\mathbb{C} = \langle c \rangle$ then $\implies$
throw new FuzzerIssue("Configuration detected");

Here $\langle c \rangle$, is a configuration (desired or undesired).

Example 15. Consider the model in the Fig. 14(a) and (b), that are an extension of Fig. 7. It has two transitions $t1$ and $t2$ that can cause nondeterminism as the value of x becomes greater than 5, however, it also contains transition $t4$ which resets the value of $x \leftarrow 1$. Identifying if there is a valuation of x that may cause both $t1$ and $t2$ to be enabled in large models is difficult. Even in this model if the event sequence is $e1, e2, e1, e2, e1, e2, e1, ...$, an alternating pattern between $e1$ and $e2$ or if x is carefully reset at appropriate action blocks, it may never result in nondeterminism. Even when such issues are identified, reproducing the same in large models is difficult.

Example 16. Consider the model in Fig. 15 that depicts two concurrent subsystems in an automotive scenario: Emergency Vehicle Avoidance (EVA) and Collision Avoidance (CA). These systems process inputs from sensors and radars and control actuators such as throttle, steering, and brakes. For the purpose of this paper, we have presented a simplified model in Fig. 15 extracted from the automotive dataset by Juarez et al. (2007) [19]. We have identified conflicts in which

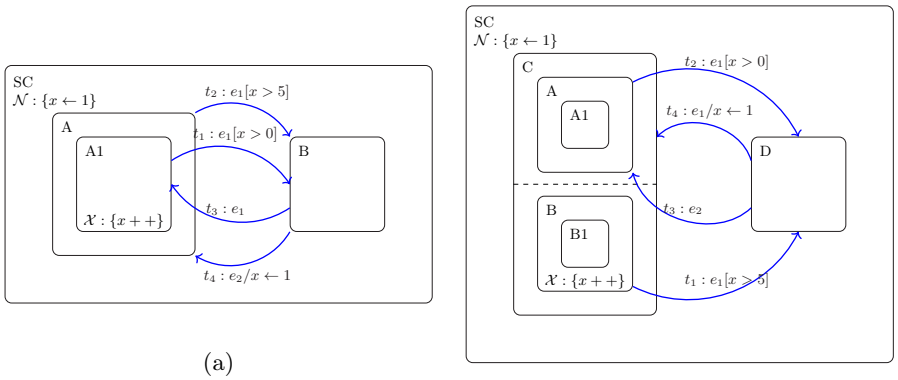


Fig. 14: Conflicting Transitions: (a) if $\mathbb{C}$={A1}, then both $t_1.s = A$ and $t_2.s = B$ would conflict when $x > 5$; (b) if $\mathbb{C}$={A1,B1}, then both t_1 and t_2 would conflict when $x > 5$.

the two subsystems issue conflicting actions, such as simultaneous modifications to the *speed* of the vehicle (known as same-actuator conflicts). Additionally, these subsystems can result in undesired configurations, such as $\{Slow, Mitigate\}$, where EVA instructs the system to slow down, and CA instructs the system to mitigate the risk of a collision by halting the vehicle. However, halting the vehicle may impede the movement of the emergency vehicle, making this an undesired configuration. As *speed* may range from 0 to 100, manual testing of all scenarios is a tedious task. To replicate the actual behavior, we employ fuzzing of inputs, specifically varying the value of *speed*. During simulation, if the current configuration is $\{Slow, Mitigate\}$, the command *throw new* $FuzzerIssue()$ is triggered, to capturing the fuzzed data that caused to reach the configuration.

4.2 Testing the simulator

Fuzzing contributes to thorough testing and verification by methodically traversing a vast number of paths within statecharts, which can be virtually limitless. It can also be employed to assess the *reachability* of all states and transitions, a common consideration in conventional testing. To guarantee the simulator's intended functionality, it is imperative that all transitions, in line with the structural semantics, remain executable, regardless of their depth within the model. The following list enumerates all potential source and destination states for these transitions.

1. Possible source states as shown in Fig. 16(a):
   (a) case 1 - atomic
   (b) case 2 - composite state (active - atomic substate, composite substate, shell substate)
   (c) case 3 - substate of composite state (active - atomic, composite, shell)

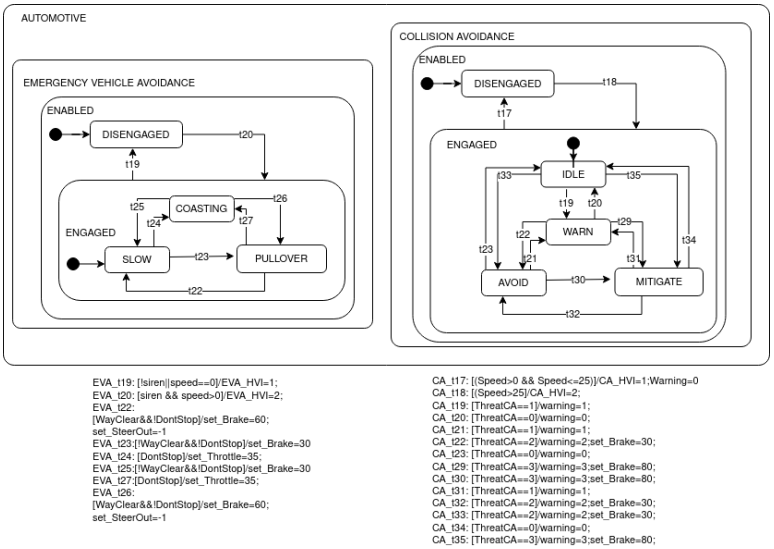


Fig. 15: Collision Avoidance and Emergency Vehicle Avoidance subsystems

(d) case 4 - shell state (active - atomic state, shell state, composite state)
(e) case 5 - region state (active - atomic state, shell state, composite state)
(f) case 6 - substate of region (active - atomic state, shell state, composite state)

2. Possible destination states as shown in Fig. 16(b):
   (a) case 7 - atomic
   (b) case 8 - composite state (default - atomic substate, composite substate, shell substate)
   (c) case 9 - substate of composite state (default - atomic, composite, shell)
   (d) case 10 - shell state (default - atomic state, shell state, composite state)
   (e) case 11 - region state (default - atomic state, shell state, composite state)
   (f) case 12 - substate of region (default - atomic state, shell state, composite state)
3. All possible combinations within a shell state as shown in Fig. 16(c)

In Fig. 16, the hatched states can be replaced with either shell, composite, or atomic states. A model encompassing all possible transitions was created. This was used to randomly test the simulator for the reachability of all feasible configurations by generating event sequences through fuzzing. we utilize the configuration test property (discussed in earlier section) to accomplish this. We conducted the simulation implementation test - without action code (applicable to all statecharts) and with action code (specific to ConStaBL with local variables).

4.3 Running extensive tests

Our experiments were conducted on an automotive dataset [19] that includes seven subsystems: Cruise Control (CC), Collision Avoidance (CA), Parking As-

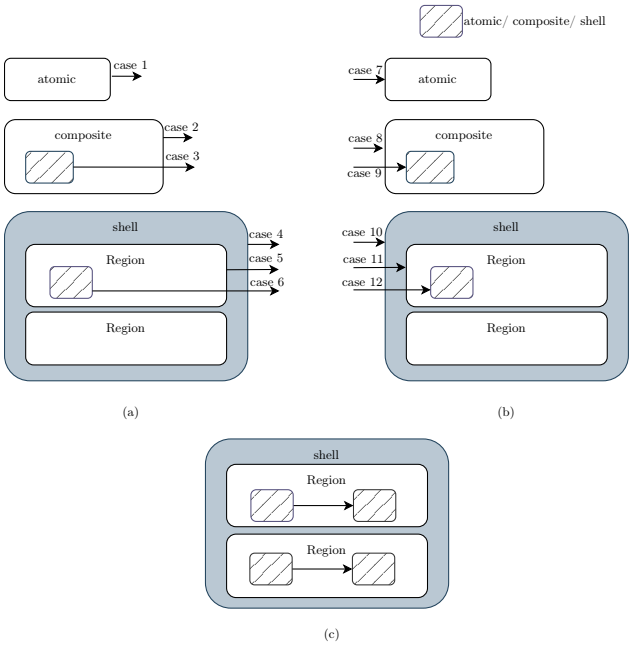


Fig. 16: Testing All Potential Transitions via Simulation

sistant (PA), Lane Guidance (LG), Emergency Vehicle Avoidance (EVA), Parking Space Centering (PSC), and Reversing Assistance (RA). We transformed the dataset from Statemate to ConStaBL semantics, omitting the transition priorities in our model intentionally to test it in a non-prioritised system. Furthermore, we merged the subsystems through orthogonal composition before conducting the experiments. While these models initially did not contain any inherent errors, we ~~deliberately~~ injected errors at various locations in the models to evaluate the effectiveness of our fuzzing implementation in detecting them ~~within a short timeframe~~.

We also injected nondeterminism inducing guards in transitions at various levels of hierarchy and could flag them using our simulator at run-time. The experiments were performed on an Ubuntu machine with 16 GB RAM. Simulations were conducted by generating event sets of sizes 5000, 10000, and 20000 using the automotive model consisting of all seven subsystems, which includes approximately 80 states and 175 transitions. The simulation traces can be found in our GitHub repository[5].

To sum up, we demonstrated the use of fuzzing:

1. To test the simulator itself and evaluate its use as a model exploration tool, we employed the simulator in models of various sizes and generated a large number of events.

[5] https://github.com/sujitkc/statechart-verification/tree/skc-simulator-test/src/dfa/outputs/uwfms

2. To assess the reachability from one random configuration to another, we generated event sequences using fuzzing as part of our tool's coverage testing.
3. To verify properties within the statechart, like nondeterminism and desired/undesired configuration.

This experimentation has provided us with confidence in the effectiveness of fuzz testing in statechart scenarios. It has also opened up possibilities for employing fuzzing to verify different properties, which could be an area for future research.

Finally, the entire experiment stands testimony to the claim that upstream modelling can allow early validation and detection of bugs. In all these experiments, the models were developed using the semantics presented in this paper, proving its practicality. The experiments were run using our simulator, which shows it a useful engine to implement early stage model validation steps.

5 Related Works

David Harel introduced statecharts [11] in 1987 as a visual modelling language for complex reactive systems. While there is no consensus on "one right way" to build semantics for statecharts [15], various approaches have been proposed to accurately capture system behaviour, given their ability to model real-time, event-driven systems. Over time, more than 100 variants [1,3,5,33] of statecharts can be seen in literature, each with different features, semantics, and domains.

As research on the execution semantics of statecharts progressed [15] [13] [32] [31] [18] [9], the topic of concurrency-related semantics became significant. Two major divisions of statechart semantics, namely interleaving and true concurrency [3], were predominant. Interleaving semantics use priorities to ensure determinism and resolve data races. Transition priorities determine the execution preference in cases of nondeterminism, while sequential ordering of substates within an AND composition determines the execution order in the presence of concurrent enabled transitions. This mechanism, though widely used in commercial tools like Yakindu, StateMate, and Stateflow, may limit the analysis of more realistic behaviour in concurrent systems that use multiprocessing architectures. The semantics of UML 2.5.1 mentions that the order in which the enabled transitions in an orthogonal region are executed is left undefined. Whereas, certain variants of statecharts and tools [18] that follow UML semantics, requires the addition of priorities and sequential ordering during design. These design decisions perculates into implementation, as these tools generate sequential code from statecharts (to imperative languages like C, Java, and TypeScript etc.). This may not align with the original intention of designing and analysing concurrent systems. Rhapsody [13], an execution semantics of UML, allows modelling of non-prioritised orthogonal states and acknowledges the undetermined execution order among orthogonal siblings. The arbitrary order of execution is achieved by "locking" each AND state once a transition is triggered within one of its components [16]. Raphsody insists that tools should permit to design without

defining priorities for transitions. It is intractable to statically determine if the guards of two enabled transitions may evaluate to true, so many variants of statecharts insists on defining these priorities. SCXML [2] resolves the execution order in parallel states by its document order. Sismic [8], an execution engine for SCXML (that also support majority of UML features) takes a different approach by processing enabled transitions based on the decreasing order of the depth of their source states. This aligns with the inner-first and source-state semantics, processing transitions from deeper states before those from less nested ones. In case of ties, the lexicographic order of the source state names is considered. It also prefers to flag nondeterminism rather than following a priority based design. As far as we are aware, none of these semantic approaches provide specific details on how to interleave the action code.

Our focus is primarily on analysing concurrency behaviour in ConStaBL statecharts with local variables in a priority-agnostic manner. The presence of local variables in our language makes it important to dwelve into the detailed semantics at the action code level. We provide a detailed perspective on how action code within AND compositions is executed in the interleaved manner at the statement level. While our approach may appear close to tokenized mechanism of Petri nets [26], there is a significant distinction. Each node in the *code graph* that we construct is a *control flow graph* on its own. Also, nesting an *AND* state within another, the composition of action blocks combines sequential and concurrent patterns which needs a special attention. Our methodology formalises this process and enables the early detection and analysis of concurrency-related issues.

Fuzz testing applications and protocols have been a wide area of research. Many fuzzing tools have evolved with focus on security testing. Numerous approaches use fuzzing and model-based testing methods to generate inputs by mutating the models. For instance, Schneider's [28] work creates a classification of fuzzing operators to create invalid message sequences; it provides guidelines on how messages are to be fuzzed, such as by removing messages, adding and modifying them, or changing their behaviour during loops for UML sequence diagram and has claimed to be applicable for message sequence charts as well. Our aim is to test the statechart model itself through fuzzing. While various fuzzing techniques, such as grammar-based fuzzing or mutation-based fuzzing, can be employed, our primary objective is to demonstrate the compatibility of the toolchain we have developed with existing fuzzing engines. For this purpose, we have integrated Jazzer [17]. Nonetheless, it can also be extended for integration with other fuzzing engines and methodologies, which requires further investigation to determine the most appropriate techniques for the specific property or system under test. This constitutes one of our future research areas.

6 Conclusion and Future work

Statecharts are valuable tools for analysing, designing, and implementing complex systems due to their ability to represent different levels of abstraction and

intricate system interactions. In this paper, we introduced ConStaBL, a variant of concurrent statecharts. Our contribution was the development of a semantics and simulator that allow for the interleaved execution of action code, enabling the detection of concurrency-related issues at an early stage. With the emergence of parallel processing systems and distributed execution of entities, there is a need to further enhance our approach to handle parallelism and incorporate analysis methodologies during the design phase.

We presented a novel way to do fuzz testing on statechart models directly. To the best of our knowledge, this idea has not been tried earlier. This illustrates how to detect defects at an early stage of SDLC using testing. It also demonstrates the applicability of the semantics presented in this paper to model realistic systems, and the ability of our simulator as a powerful aid in analysis.

We aim to leverage the fine-grained nature of our simulator to detect a wider range of defects in systems involving interleaved concurrency. Additionally, we will extend the verification capabilities with techniques such as symbolic execution, model checking and other static analysis methods. We intend verify properties specific to Digital Public Infrastructures (DPIs) and automotive component [19] interactions, as they involve localisation and depend highly on the value environment. These extensions will be a focal point of our future studies.

References

1. André, E., Liu, S., Liu, Y., Choppy, C., Sun, J., Dong, J.S.: Formalizing uml state machines for automated verification – a survey. ACM Comput. Surv. (jan 2023). https://doi.org/10.1145/3579821, https://doi.org/10.1145/3579821, just Accepted
2. Barnett, J.: Introduction to scxml. Multimodal Interaction with W3C Standards: Toward Natural User Interfaces to Everything pp. 81–107 (2017)
3. Von der Beeck, M.: A comparison of statecharts variants. In: Formal Techniques in Real-Time and Fault-Tolerant Systems: Third International Symposium Organized Jointly with the Working Group Provably Correct Systems—ProCoS Lübeck, Germany, September 19–23, 1994 Proceedings 3. pp. 128–148. Citeseer (1994)
4. von der Beeck, M.: A structured operational semantics for uml-statecharts. Software and Systems Modeling **1** (2002). https://doi.org/10.1007/s10270-002-0012-8
5. Bhaduri, P., Ramesh, S.: Model checking of statechart models: Survey and research directions. CoRR **cs.SE/0407038** (2004), http://arxiv.org/abs/cs.SE/0407038
6. Chakrabarti, S.K., Venkatesan, K.: Stabl: Statecharts with local variables. In: Proceedings of the 13th Innovations in Software Engineering Conference on Formerly Known as India Software Engineering Conference. ISEC 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385032.3385040, https://doi.org/10.1145/3385032.3385040
7. Croll, P., Duval, P.Y., Jones, R., Kolos, S., Sari, R., Wheeler, S.: Use of statecharts in the modelling of dynamic behaviour in the atlas daq prototype-1. IEEE Transactions on Nuclear Science **45**(4), 1983–1988 (1998). https://doi.org/10.1109/23.710975

8. Decan, A., Mens, T.: Sismic—a python library for statechart execution and testing. SoftwareX **12** (7 2020). https://doi.org/10.1016/J.SOFTX.2020.100590
9. Eshuis, R.: Reconciling statechart semantics. Science of Computer Programming **74** (2009). https://doi.org/10.1016/j.scico.2008.09.001
10. Gery, E., Harel, D., Palachi, E.: Rhapsody: A complete life-cycle model-based development system. In: Butler, M., Petre, L., Sere, K. (eds.) Integrated Formal Methods. pp. 1–10. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
11. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8**(3), 231–274 (1987). https://doi.org/https://doi.org/10.1016/0167-6423(87)90035-9, https://www.sciencedirect.com/science/article/pii/0167642387900359
12. Harel, D.: Statecharts in the making: a personal account. In: Proceedings of the third ACM SIGPLAN conference on History of programming languages. pp. 5–1 (2007)
13. Harel, D., Kugler, H.: The rhapsody semantics of statecharts (or, on the executable core of the uml). In: Integration of Software Specification Techniques for Applications in Engineering, pp. 325–354. Springer (2004)
14. Harel, D., Naamad, A.: The statemate semantics of statecharts. ACM Trans. Softw. Eng. Methodol. **5**, 293–333 (10 1996). https://doi.org/10.1145/235321.235322, http://doi.acm.org/10.1145/235321.235322
15. Harel, D., Naamad, A.: The statemate semantics of statecharts. ACM Transactions on Software Engineering and Methodology (TOSEM) **5**(4), 293–333 (1996)
16. IBM: Semantics of transition selection algorithm in ibm engineering lifecycle management suite: Design rhapsody. https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/design-rhapsody/9.0.1?topic=semantics-transition-selection-algorithm, accessed on: [17-July-2023]
17. Intelligence, C.: Codeintelligencetesting/jazzer: Coverage-guided, in-process fuzzing for the jvm, https://github.com/CodeIntelligenceTesting/jazzer
18. Itemis: Yakindu statecharts, https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide
19. Juarez-Dominguez, A.L., Day, N.A., Fanson, R.T.: A preliminary report on tool support and methodology for feature interaction detection. Tech. rep., Technical Report CS-2007-44, University of Waterloo (2007)
20. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. International journal on software tools for technology transfer **1**, 134–152 (1997)
21. Li, J., Tang, J., Wan, S., Zhou, W., Xu, J.: Performance evaluation from stochastic statecharts representation of flexible reactive systems: A simulation approach. Journal of Systems Engineering and Electronics **25**(1), 150–157 (2014). https://doi.org/10.1109/JSEE.2014.00018
22. paul-j lucas: Github - paul-j-lucas/chsm: Concurrent hierarchical finite state machine language system. (Jul 2018), https://github.com/paul-j-lucas/chsm
23. Mathworks: Semantics of stateflow, http://www.ece.northwestern.edu/local-apps/matlabhelp/toolbox/stateflow/semantic.html
24. mathworks: State Diagram — ch.mathworks.com. https://ch.mathworks.com/discovery/state-diagram.html, [Accessed 11-08-2023]
25. ni: Product Documentation - NI — ni.com. https://www.ni.com/docs/en-US/bundle/labview-statechart-module/page/lvscconcepts/sc_c_top.html, [Accessed 11-08-2023]
26. Peterson, J.L.: Petri nets. ACM Computing Surveys (CSUR) **9**(3), 223–252 (1977)
27. QM: Qm model-based design tool, https://www.state-machine.com/products/qm

28. Schneider, M., Großmann, J., Tcholtchev, N., Schieferdecker, I., Pietschker, A.: Behavioral fuzzing operators for uml sequence diagrams. In: System Analysis and Modeling: Theory and Practice: 7th International Workshop, SAM 2012, Innsbruck, Austria, October 1-2, 2012. Revised Selected Papers 7. pp. 88–104. Springer (2013)
29. Shapiro, B., Casinghino, C.: specgen: A tool for modeling statecharts in csp. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NASA Formal Methods. pp. 282–287. Springer International Publishing, Cham (2017)
30. States, P.: The boost statechart library. Library (2006)
31. Than, X., Miao, H., Liu, L.: Formalizing the semantics of uml statecharts with z. In: The Fourth International Conference onComputer and Information Technology, 2004. CIT '04. pp. 1116–1121 (2004). https://doi.org/10.1109/CIT.2004.1357344
32. Uselton, A.C., Smolka, S.A.: A process algebraic semantics for statecharts via state refinement. In: PROCOMET. vol. 94, pp. 262–281. Citeseer (1994)
33. Van Mierlo, S., Vangheluwe, H.: Statecharts: A Formalism to Model, Simulate and Synthesize Reactive and Autonomous Timed Systems, pp. 155–176. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-43946-0_6, https://doi.org/10.1007/978-3-030-43946-0_6

A Appendix

A.1 A simple imperative language.

```
B::={slist}
slist::= ϵ|s;slist
s::= v:=e|if(cond) then B1 else B2|while(cond) B1
e::= e op e|v|const|op e
//slist: statement list, s: statement, v: variable, e:
    expression
```

Listing 1.1: A minimal representative action language.

A.2 Dependencies of the terminologies

The Fig. 17 shows the dependencies between the concepts and terminologies used throughout this work. The dependencies are transitive.

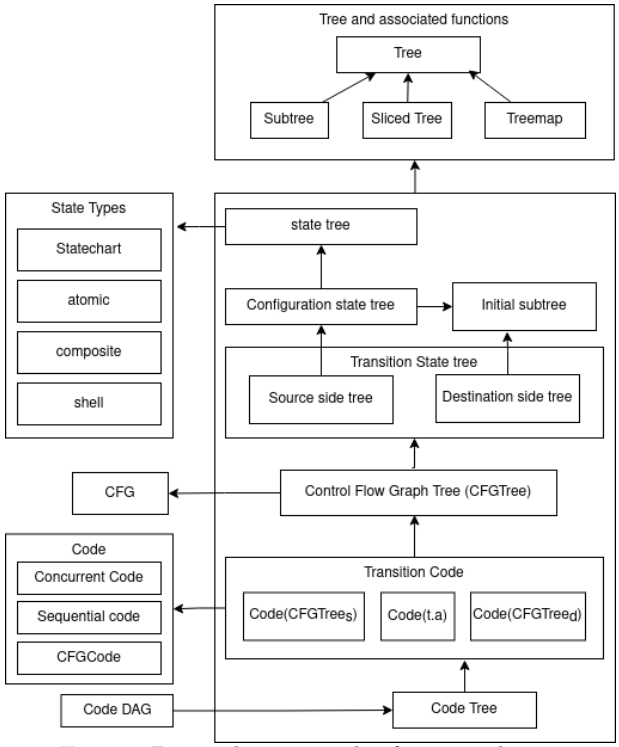


Fig. 17: Dependency graph of terminologies