Flattening Statecharts without Explosions

Andrzej Wasowski
Department of Innovation
IT University of Copenhagen
2300 Copenhagen S, Denmark
wasowski@itu.dk

ABSTRACT

We present a polynomial upper bound for flattening of UML statecharts. An efficient flattening technique is derived and implemented in SCOPE—a code generator targeting constrained embedded systems. Programs generated with this new technique are both faster and smaller than those produced by non-flattening code generators. Our approach scales well for big models and exhibits good properties with respect to memory usage, automatic analysis of worst-case reaction time and automatic validation of memory safety.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State Diagrams*

General Terms

Algorithms, Performance, Languages

Keywords

Program synthesis, automatic code generation, statecharts, semantics, embedded systems

1. INTRODUCTION

The language of statecharts [17, 20] is a popular modeling formalism for design and development of reactive systems. Statecharts owe this popularity to their intuitive visual syntax and formalized semantics. The language is centered around the finite state machine paradigm extended with concurrency and hierarchy.

Flattening, understood as removing hierarchy and retaining concurrency, is a central transformation for statecharts. Its applications range from constructing operational semantics for hierarchical models [18] through model-checking [12], automatic testing [4, 6, 21] up to software and hardware synthesis [5]. Flat statecharts can be implemented using a simpler runtime system and less mutable data structures, which results in reduced consumption of RAM. Both properties are crucial in development of resource-constrained embedded systems. Understanding computational properties of flattening emerges as one of the vital steps in exploration of properties of statecharts. This paper approaches the problem from two angles. First, a formal statement of complexity is given. Second, a practical optimized implementation and experiments are evaluated.

There is a bit of controversy on the exact meaning of the term *flattening*. Note that our definition is substantially different from the expansion to a product machine. A product machine is easy to implement in a code generator, but the expansion process is trivially exponential, ruling out practical applications. The flat form we consider is a predecessor language of statecharts: a set of traditional Mealy machines operating concurrently. Our notions of hierarchy, concurrency and flattening are similar to those of [2, 3, 25, 12] and substantially different from [9, 22, 24].

Our complexity proof proceeds in a standard way presenting an algorithm producing a flattened statechart that is only polynomially bigger than the given hierarchical one. The result is relatively surprising as it contradicts a widely held but informal belief that a polynomial solution does not exist.

The complexity–theoretic proof is easily adapted to a more efficient flattening algorithm that replaces expensive signal communication with cheap sequential rule processing. This algorithm is implemented and evaluated using SCOPE [23]—an efficient and compact code generator for embedded systems. Experiments show that the quality of resulting code exceeds the quality of programs synthesized by an advanced non-flattening code generator.

The presentation is divided into 4 parts. Section 2 formalizes the problem: the source and target statechart languages, an implementation relation and a specification of flattening. Section 3 rigorously describes the flattening algorithm and assets its correctness. Section 4 focuses on the practical implementation: efficiency-oriented mutations and optimizations, as well as experimental evaluation. Section 5 examines the meaning of the result against the existing research and compares to other code generation approaches.

2. FLATTENING PROBLEM

Let us start with a description of hierarchical statecharts (source language), flat statecharts (target language) and the desired properties of the flattening translation from one to the other. The statechart of Fig. 1 will be used as a running example.

LCTES'04, June 11–13, 2004, Washington, DC, USA.

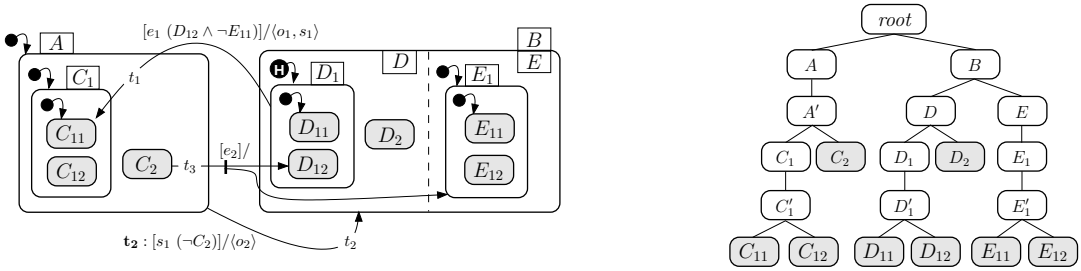

Figure 1: Hierarchical statechart and the hierarchy tree defined by the $\searrow$ relation. Entry and exit actions on states are suppressed. So are all uninteresting transitions. It is assumed that all states are reachable. A node with primed label denotes a dummy or-state (a single thread in a trivially concurrent and-state).

2.1 Hierarchical Statecharts

Consider a subset of UML statecharts. Let *State* be a finite set of states, divided into two disjoint classes $State_{\mathsf{and}}$ and $State_{\mathsf{or}}$ of and-states and or-states respectively. A *root* state and the substate relation $\searrow$ form a hierarchy of states such that:

[or_root]	$root \in State_{\mathsf{or}}$
[and_leaves]	$\forall s_1 \in State_{\mathsf{or}}.\ \exists s_2 \in State_{\mathsf{and}}.\ s_1 \searrow s_2$
[alternation]	$\forall s_1, s_2 \in State.\ s_2 \searrow s_1 \Rightarrow s_1 \in State_{\mathsf{and}} \wedge$ $s_2 \in State_{\mathsf{or}} \vee (s_1 \in State_{\mathsf{or}} \wedge s_2 \in State_{\mathsf{and}})$
[rooted]	$\forall s \in State.root \searrow^* s$
[acyclic]	$\forall s_1, s_2 \in State.\ \neg(s_2 \searrow^+ s_1 \wedge s_1 \searrow^+ s_2)$
[one_parent]	$\forall s_1, s_2, s_3 \in State.$ $s_2 \searrow s_1 \wedge s_3 \searrow s_1 \Rightarrow s_2 = s_3$

The relation imposes a directed tree structure on states, rooted in *root*. The *root* node is an or-state and all leaves are and-states. State types alternate between and and or on all paths from *root* to leaves. If $s_1 \searrow s_2$ then we say that s_2 is a child of s_1 with $s_1 = parent(s_2)$ and $s_2 \in children(s_1)$. Members of the path from s to *root* are denoted by $ancest^*(s)$. Below we present some values for the example of Fig. 1:

$$State_{\mathsf{and}} = \{A, B, C_1, C_2, D_1, D_2, E_1, C_{11}, C_{12}, D_{11}, D_{12}, \dots\}$$
$$State_{\mathsf{or}} = \{root, A', D, E, C'_1, D'_1, E'_1\}$$
$$(\searrow) = \{(root, A), (root, B), (A, A'), (B, D), (B, E), (A', C_1), (A', C_2), (D, D_1), (D, D_2), (E, E_1), (C_1, C'_1), (D_1, D'_1), (E_1, E'_1), (C'_1, C_{11}), \dots\}$$

Each or-state s has a distinguished child called an *initial state*. This state, written $ini(s)$ and marked ●, is entered by default by any transition that targets s without specifying further targets. Some or-states are called *history states*. Whenever a history state is entered, the child that was active most recently is entered instead of the usual $ini(s)$. D is a history state. D_1, marked Ⓗ, is its initial history value.

$$ini : State_{\mathsf{or}} \to State_{\mathsf{and}} = [root \mapsto A, A' \mapsto C_1, C'_1 \mapsto C_{11}, D \mapsto D_1, D'_1 \mapsto D_{11}, E \mapsto E_1, E'_1 \mapsto E_{11}]$$

Let *Event* denote a finite set of environment events, *Signal* a finite set of model internal events (signals) and *Output* a finite set of model generated outputs. In the example:

$$Event = \{e_1, e_2\} \quad Signal = \{s_1\} \quad Output = \{o_1, o_2\}$$

An *action* is a sequence of signals and outputs. We assign an entry and exit action to each and-state. Those are generated whenever s is entered (exited). Exit outputs are generated in a bottom-up manner: first basic active states are exited, then their parents, and so on. Entry actions are generated in a top-down manner from outer to inner states.

$$en, ex\ :\ State_{\mathsf{and}} \to (Output \cup Signal)^*\ .$$

A *guard* is a logical formula over and-states. The set of all guards *Guard* is generated by the grammar:

$$g ::= true \mid s \mid g \wedge g \mid \neg g\ ,\ \text{where } s \in State_{\mathsf{and}}. \qquad (1)$$

A *transition* is a tuple $(s, e, g, os, ts) \in Trans$, where

$$Trans = State_{\mathsf{and}} \times (Event \cup Signal) \times Guard \times \times (Output \cup Signal)^* \times \mathcal{P}(State_{\mathsf{and}})\ .$$

We shall usually write $s \xrightarrow{[e:g]/os} ts$, where s denotes the source state, e is a triggering event, g is a guard, os is the action generated when the transition fires and ts is a set of target states. The model of Fig. 1 has the following transitions:

$$Trans = \{t_1, t_2, t_3, \dots\}\ ,\ \text{where}$$
$$t_1 = D_1 \xrightarrow{[e_1 : D_1 \wedge D_{12} \wedge \neg E_{11}]/\langle o_1, s_1\rangle} C_{11}\ ,$$
$$t_2 = A \xrightarrow{[s_1 : A \wedge \neg C_2]/\langle o_2\rangle} B\ ,$$
$$t_3 = C_2 \xrightarrow{[e_2 : C_2]/\langle\rangle} \{D_{12}, E_1\}\ .$$

The source state condition s is included in the guard condition g. It is also indicated separately to guarantee a unique visual representation of transitions and to support the notion of *scope* for transitions. A scope t is defined as the nearest common or-ancestor of the source and target states ($\{s\} \cup ts$). Informally the scope represents the extent to which a transition may modify the state configuration. The scopes of transitions on Fig. 1 are: $scope(t_1) = scope(t_2) = scope(t_3) = root$.

Using the above definitions we define a statechart $\mathcal{S}$ as a tuple:

$$\mathcal{S} = (State_{\mathsf{and}}, State_{\mathsf{or}}, \searrow, ini, Event, Signal, Output, ex, en, Trans)\ .$$

The set of active states σ is called an active *configuration*. In every run of the statechart the *root* state is entered upon initialization and remains active throughout execution. The following invariant holds for all states before and after processing an event: if an and-state is active, then all its

children are active, if an or-state is active, then exactly one of its children is active as well:

$$\begin{array}{ll} \text{[root\_active]} & \mathit{root} \in \sigma \\ \text{[ancestors\_active]} & \forall s \in \sigma.\ \mathit{ancest}(s) \subseteq \sigma \\ \text{[and\_active]} & \forall s \in (\mathit{State}_{\mathsf{and}} \cap \sigma).\mathit{children}(s) \subseteq \sigma \\ \text{[or\_active]} & \forall s_1 \in (\mathit{State}_{\mathsf{or}} \cap \sigma).\exists! s_2 \in \sigma.\ s_1 \searrow s_2\ . \end{array}$$

Active configuration is used to establish a satisfaction relation for guards. References to states are considered true if the given state is active, i.e. $\sigma \models s$ iff $s \in \sigma$. The remaining part of the satisfaction relation is defined in a usual way.

Apart from the configuration, recently active children for all history or-states needs to be maintained in the execution state. The set of history values η contains exactly one and-state child, for each history or-state.

The initial configuration σ_0 of the statechart is computed by closure of the *ini* function and the above invariants. The initial history set η_0 is computed by selecting the range of *ini* function on the domain of history states. In the example:

$$\sigma_0 = \{\mathit{root}, A, A', C_1, C_1', C_{11}\} \qquad \eta_0 = \{D_1\}\ .$$

A signal queue $q \in \mathit{Signal}^*$ is maintained for local events. This is a FIFO-like structure, where signals generated by actions are appended at the end and signals yet to be processed are popped from the beginning.

A transition is *enabled* if its triggering event e is present and its guard is satisfied in the current configuration. An enabled transition fires: it exits its scope generating exit actions; generates its own action; and enters its targets generating entry actions.

A statechart is executed in synchronous steps. The finest step is called a *microstep*. A microstep in presence of a given single event or a signal e advances the execution state from $\langle \sigma_0, \eta_0, q_0 \rangle$ to the new value $\langle \sigma_1, \eta_1, q_1 \rangle$ generating a sequence of outputs os, which is the concatenation of actions with the local signals filtered out: $\langle \sigma_0, \eta_0, q_0 \rangle \xrightarrow[\text{micro}]{e\ os} \langle \sigma_1, \eta_1, q_1 \rangle$. The generated local signals are appended to the end of the new queue.

Statecharts are *input-enabled*: a microstep can be taken for any event or signal in any execution state. In a nondeterministic statechart several choices may be possible for a given event.

A complete reaction to an environment event $e \in \mathit{Event}$ is called a *macrostep*: $\langle \sigma_0, \eta_0 \rangle \xrightarrow[\text{macro}]{e\ os} \langle \sigma_1, \eta_1 \rangle$. A macrostep, executed in a configuration σ_0 places an external event e in an empty signal queue and iterates microsteps popping queue events and signals one-by-one until the queue is emptied. The final configuration and history reached by the iteration of microsteps is the final configuration of the macrostep. The sequence of outputs generated is the concatenation of sequences generated by microsteps. In Fig. 1, the presence of event e_1 may give rise to a macrostep consisting of a single microstep. The e_2 event may trigger a macrostep consisting of two microsteps. First e_2 is processed, which generates s_1, and then s_1 is processed.

The *model size* is the sum of size of states plus the size of transitions plus the maximal length of the signal queue in any execution. The sizes of states and transitions are defined as numbers of primitive elements they contain.

2.2 Flat Statecharts

A flat statechart comprises a set of state groups and an initial configuration with sets of events, signals, outputs and transitions. We shall denote all elements of flat statecharts with primed names, to distinguish them from elements of the hierarchical statecharts.

$$\mathcal{S} = (\{M_1', \ldots, M_k'\}, \sigma_0', \mathit{Event}', \mathit{Signal}', \mathit{Output}', \mathit{Trans}')$$

State groups $M_{(i)}'$ are a partitioning of the state set and $\mathit{State}' = \bigcup_{i=1}^{k} M_i'$. Guards are defined over elements of State' as in (1). Events, signals and outputs are categorized in sets as previously. States do not have any actions assigned (only transitions have). There are no history states. State machines are never reinitialized, so history would not make much sense.

A configuration $\sigma' \subseteq \mathit{State}'$ of the flat statechart is a set containing exactly one state for each state group:

$$\text{[unambigous]} \quad \forall i = 1 \ldots k.\ \exists s \in M_i.\ M_i \cap \sigma' = \{s\}$$

The initial configuration σ_0' is an explicit part of the model. Signal queues are used as before and the microstep and macrostep relations are defined analogously as for hierarchical statecharts in terms of transition firing.

The type of transitions is the same as for hierarchical statecharts, but the transition firing is somewhat different. There is no notion of scope and exiting. Transition fires by generating its output action and adjusting the active configuration. If $\{t_1', \ldots, t_l'\}$ is a possibly empty set of targets of a transition t' (each of them a member of $M_1', \ldots, M_l'$ respectively) then the new state configuration σ_1' is computed by substituting the old members of state groups for the targets:

$$\sigma_1' = \sigma_0' \setminus (\bigcup_{i=1}^{l} M_i') \cup \{t_1', \ldots, t_l'\}\ . \tag{2}$$

The size measure of the flat statechart should be understood similarly as before, reflecting the sum of transition and state sizes plus the maximal queue length.

Any flat statechart can be seen as a hierarchical statechart of a special structure, where all the state groups are really history or-states. By defining a lean language of flat statecharts we can avoid many superfluous empty attributes in the translation scheme. Also the primary difference between hierarchical and flat interpreters is indicated more clearly. The mechanism of transition firing in the flat interpreter performs very simple operations on the configuration—see (2). The hierarchical interpreter demands recursive traversals of state hierarchy, not only consuming more writable memory, but also demanding use of complex data structures at runtime [26].

2.3 Flattening

Definition 1 (Simulation). *A flat statechart $\mathcal{S}'$ in configuration σ_0' simulates a hierarchical statechart $\mathcal{S}$ in configuration and history (σ_0, η_0), written $(\mathcal{S}', \sigma_0') \lesssim (\mathcal{S}, \sigma_0, \eta_0)$, iff*

$$\forall e, os, \sigma_1'.\ \sigma_0' \xrightarrow[macro]{e\ os} \sigma_1' \Rightarrow\ \exists \sigma_1, \eta_1.\ \langle \sigma_0, \eta_0 \rangle \xrightarrow[macro]{e\ os} \langle \sigma_1, \eta_1 \rangle \wedge (\mathcal{S}', \sigma_1') \lesssim (\mathcal{S}, \sigma_1, \eta_1)$$

Definition 2 (Implementation). *A flat statechart $\mathcal{S}'$ implements a hierarchical statechart $\mathcal{S}$, written*

$\mathcal{S}' \lesssim \mathcal{S}$, *iff for initial configurations* σ_0' *and* σ_0 *it holds that* $(\mathcal{S}', \sigma_0') \lesssim (\mathcal{S}, \sigma_0, \eta_0)$.

DEFINITION 3 (FLATTENING). *Let F be an algorithm translating hierarchical statecharts to flat statecharts. F is a flattening algorithm if for any hierarchical statechart $\mathcal{S}$ it yields a flat statechart $\mathcal{S}'$ such that $\mathcal{S}' \lesssim \mathcal{S}$.*

Note that simulation is a strong notion in the input-enabled synchronous setting. In particular it is not the case that the empty flat statechart (with no states and transitions) implements all hierarchical statecharts. An empty statechart can accomodate any sequence of inputs without producing a single output. The hierarchical statechart has to match each of these executions in the same fashion (without any outputs). Clearly this can only be fullfiled by hierarchical statecharts which never produce any outputs.

THEOREM 4. *For any hierarchical statechart $\mathcal{S}$ there exists a flat statechart $\mathcal{S}'$ such that $\mathcal{S}' \lesssim \mathcal{S}$ and the size of $\mathcal{S}'$ is at most polynomial in the size of $\mathcal{S}$.*

In fact this result holds even if the set of internal signals of $\mathcal{S}'$ is restricted to two distinctive values (note that this is a restriction on the target, not the source language, which would be trivial). This can be concluded using binary encoding techniques presented in [27], but shall not be discussed further here.

3. POLYNOMIAL FLATTENING

We shall present the flattening algorithm in a declarative style, as a syntax-driven transformation of statechart elements. Consider a hierarchical statechart: $\mathcal{S} = (State_{\mathsf{and}}, State_{\mathsf{or}}, \searrow, ini, ex, en, Event, Signal, Output, Trans)$. We shall show how to construct a flat statechart $\mathcal{S}' = (M', \sigma_0', Event, Signal', Output, Trans')$ such that $\mathcal{S}' \lesssim \mathcal{S}$.

3.1 States

The state groups M' of $\mathcal{S}'$ are the children sets of or-states in the hierarchical model $\mathcal{S}$. We add an extra state group $\{I\}$ containing a single fresh state I, used to implement administrative internal rules.

$$M' = \{children(s) \mid s \in State_{\mathsf{or}}\} \cup \{\{I\}\}$$

The initial configuration σ_0' is computed by taking the range of the hierarchical function *ini* and adding the extraneous I state, which is the only, hence trivially initial, state in its group. Let us write the state groups and the initial configuration for the example of Fig. 1. Note the states belonging to σ_0' but not to σ_0 (for instance D_1):

$$\begin{aligned} M' &= \{\{A,B\},\{C_1,C_2\},\{D_1,D_2\},\{E_1\},\\ &\quad \{C_{11},C_{12}\},\{D_{11},D_{12}\},\{E_{11},E_{12}\},\{I\}\}\\ \sigma_0' &= \mathbf{rng}\ ini \cup \{I\} = \{A, C_1, C_{11}, D_1, D_{11}, E_1, E_{11}, I\} \end{aligned}$$

The number of groups, the number of states in $\mathcal{S}'$, and the size of σ_0' are linear in $|State|$.

3.2 Guards

Guards are flattened by computing an ancestor closure over states of S. This way the [ancestors_active] invariant of the hierarchical configurations is enforced for the flat statecharts at the transition level. The invariant would not hold for flat configurations directly.

$$flat(g) = \begin{cases} true & \text{if } g = true \\ \neg flat(g_1) & \text{if } g = \neg g_1 \\ flat(g_1) \wedge flat(g_2) & \text{if } g = g_1 \wedge g_2 \\ \bigwedge_{p \in ancest^*(s) \setminus State_{\mathsf{or}}} p & \text{if } g = s,\ s \in State_{\mathsf{and}} \end{cases}$$

This transformation gives raise to at most polynomial growth of transition size.

3.3 Action Transitions

Entry and exit actions are not available in the flat target language. We shall implement them by generating an *action transition* for each of them. For each and-state s, define the following transitions:

$$t_s^{\mathrm{ex}} = I \xrightarrow{[e_s^{\mathrm{ex}}:flat(s)]/ex(s)} I \qquad t_s^{\mathrm{en}} = I \xrightarrow{[e_s^{\mathrm{en}}:flat(parent^2(s))]/en(s)} s,$$

where e_s^{ex} and e_s^{en} denote fresh signals not belonging to $Event \cup Signal \cup Output$. These signals may now be used to trigger entry and exit actions. Note that the exit transition will fire if s is active, while the entry transition only activates s if the nearest and-state ancestor is active, so that invariant [ancestors_active] is preserved. The action transitions for C_1 of Fig. 1 are:

$$I \xrightarrow{[e_{C_1}^{\mathrm{ex}}:C_1 \wedge A]/ex(C_1)} I \text{ and } I \xrightarrow{[e_{C_1}^{\mathrm{en}}:A]/en(C_1)} C_1\ .$$

Similarly an action transition is generated for each hierarchical transition t:

$$t_t = I \xrightarrow{[e_t:true]/os_t} I\ ,$$

where e_t is a fresh signal. This signal may now be used to trigger the action of the original hierarchical transition. An action transition corresponding to transition t_1 in Fig.1 is:

$$I \xrightarrow{[e_{t_1}:true]/\langle o_1, s_1\rangle} I\ .$$

Later on we shall schedule action transitions in the proper sequences to implement traces of original statechart. Note that so far we have added a number of transitions linear in the number of states and transitions in the original model.

3.4 Interface

The input and output alphabets cannot be changed, otherwise the flat statechart would violate the implementation condition trivially. The set of internal signals is extended with the administrative signals mentioned before plus a new one for each history state:

$$\begin{aligned} Signal' = Signal &\cup \{e_s^{\mathrm{en}} | s \in State_{\mathsf{and}}\} \cup \{e_s^{\mathrm{ex}} | s \in State_{\mathsf{and}}\}\\ &\cup \{e_t | t \in Trans\} \cup \{e_s^{\mathrm{h}} | s \text{ is a history state}\}\ . \end{aligned}$$

The size of $Signal'$ is linear in the size of the original model.

3.5 Entry Schedule

Firing a hierarchical transition has three phases: exit the scope, execute actions, and enter the scope. In a flat statechart we can realize the entering and exiting phases by generating signal *schedules*. Schedules are sequences of administrative signals, which when interpreted by microstep iteration realize the semantics of the original hierarchical transition.

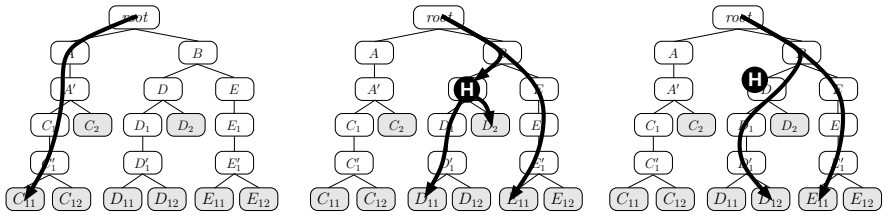

Figure 2: Entry schedules for t_1(left), t_2(middle) and t_3(right). The path of t_1 is static, t_2 relies on dynamic history choice, t_3 circumvents history with explicit guiding targets.

An *entry schedule* has two parts: a statically computable part and a dynamic, history-dependent, part. The static part can be determined at compile time by computing a closure of the *ini* function guided by the set of explicit targets. The computation of the schedule continues until a history state is reached (see Fig. 2). Two mutually recursive functions $entr^{ts}_{\mathsf{or}}$ and $entr^{ts}_{\mathsf{and}}$ realize this hierarchy traversal guided by the set of goal states *ts* stopping at history states and basic states. They generate signals firing the entry transitions of respective states and a history transition if needed. The history state can only be bypassed if the targets below it are explicitly specified (see the example of t_3 on Fig. 2). Otherwise the function needs to follow all possible history versions, relying on the [ancestors_active] invariant encoded in entry transitions to stop entering wrong paths.

$$entr^{ts}_{\mathsf{or}} : State_{\mathsf{or}} \to Signal^* =$$

$$= \lambda s. \begin{cases} \langle e^{en}_s, e^{h}_s \rangle \hat{} entr^{ts}_{\mathsf{and}}(s_1) \hat{} \ldots \hat{} entr^{ts}_{\mathsf{and}}(s_p) \\ \qquad \text{if } s \text{ is a history state and } \forall t \in ts.\ s \not\searrow^* t \\ \langle e^{en}_{def\ ^{ts}(s)} \rangle \hat{} entr^{ts}_{\mathsf{and}}(def\ ^{ts}(s)) \\ \qquad \text{otherwise} \end{cases}$$

where $s_1, \ldots, s_p = children(s)$ in some order.

$$entr^{ts}_{\mathsf{and}} : State_{\mathsf{and}} \to Signal^* = \lambda s. entr^{ts}_{\mathsf{or}}(s_1) \hat{} \ldots \hat{} entr^{ts}_{\mathsf{or}}(s_p),$$

where $s_1, \ldots, s_p = children(s)$ in some order.

$$def\ ^{ts} : State_{\mathsf{or}} \to State_{\mathsf{and}} = \lambda s. \begin{cases} p \text{ if } s \searrow p \wedge \exists t \in ts. p \searrow^* t \\ ini(s) \text{ otherwise} \end{cases}$$

The $\hat{}$ operator denotes concatenation of sequences. The statecharts semantics does not specify in which order concurrent components should be entered by $entr^{ts}_{\mathsf{and}}$. Any order consistent with $\searrow$, including interleaved entry traces of concurrent components, is legal. The $entr^{ts}_{\mathsf{and}}$ function given above corresponds to a single deterministic choice of such sequence. The omission of other choices is permitted because our implementation relation does not require all possible orderings to be preserved.

Function $def\ ^{ts}$ is a helper, which determines a default child for a given or-state. It checks whether further targets are specified below the given or-state, and follows the indicated path if available. A compile-time algorithm cannot continue deeper beyond history states as the actual entering schedule depends on the runtime properties of these states. An administrative history signal e^h_s is triggered instead and a history transition is added for each child p of history state s:

$$I \xrightarrow{[e^{\mathrm{h}}_s : flat(p)]/\langle e^{\mathrm{en}}_p \rangle} I$$

These history transitions are similar to entry transitions, except that they have stronger firing conditions. The guard $flat(p)$ guarantees that only one of these transitions will fire whenever e^{h}_s is triggered—the one entering the most recently active child—which corresponds to a runtime choice of entry schedule. An entry schedule of a transition is computed starting with its scope and the set of targets. The schedule of t_2 is $\langle e^{\mathrm{en}}_B, e^{\mathrm{h}}_D, e^{\mathrm{en}}_{D_{11}}, e^{\mathrm{en}}_{E_1}, e^{\mathrm{en}}_{E_{11}} \rangle$. The schedule of t_3 is $\langle e^{\mathrm{en}}_B, e^{\mathrm{en}}_{D_1}, e^{\mathrm{en}}_{D_{12}}, e^{\mathrm{en}}_{E_1}, e^{\mathrm{en}}_{E_{11}} \rangle$ (see Fig. 2).

3.6 Exit Schedule

Exit schedules are much easier to compute than entry schedules. For a given scope s we define $extr(s)$ to be the sequence of exit signals for and-state descendants of s produced in postorder traversal. For instance $extr(A) = \langle e^{\mathrm{ex}}_{C_{11}}, e^{\mathrm{ex}}_{C_{12}}, e^{\mathrm{ex}}_{C_1}, e^{\mathrm{ex}}_{C_2}, e^{\mathrm{ex}}_A \rangle$. An entry or exit schedule cannot be longer than $|State_{\mathsf{and}}|$.

3.7 Transition Schedule

Each hierarchical transition $t = s \xrightarrow{[e:g]/os} ts$ is translated to a flat transition t' which schedules the relevant action transitions realizing the semantics. The condition part of the transition remains unchanged, except for the flattened guard:

$$t' = s \xrightarrow{[e:flat(g)]/extr(scope(t)) \hat{} \langle e_t \rangle \hat{} entr_{\mathsf{or}}{}^{ts}(scope(t))} I$$

Consider the result obtained for the transition t_1 of our example:

$$t'_1 = D_1 \xrightarrow{[e_1:\gamma]/extr(root') \hat{} \langle e_{t_1} \rangle \hat{} entr_{\mathsf{or}}\{C_{11}\}(root')} I \ , \qquad (3)$$

where $\gamma = (D_1 \wedge B) \wedge ((D_{12} \wedge D_1 \wedge B \wedge) \wedge \neg(E_{11} \wedge E_1 \wedge B))$.

Trans′ is the set of all action transitions, history transitions and scheduling transitions. There are at most $3|State_{\mathsf{and}}| + |Trans|$ new transitions in the flat model and each of them is at most $2|State_{\mathsf{and}}|$ times long. The size change is within the polynomial bounds in the size of hierarchical model, or more precisely in $O(|State|^2 + |State| \cdot |Trans|)$. The size of the flat statechart is bounded by the square of the size of the original hierarchical statechart.

4. CODE GENERATION

Although polynomial, the algorithm of the previous section remains rather inefficient. Our goal is to eliminate excess transitions, simplify guards, and reduce the number of administrative signals. The resulting flat rulesets are extremely simple to represent and interpret using a minimal runtime system. SCOPE's interpreter uses tiny amounts of writable memory: an order of 10 integers plus the size of the current configuration vector and a very shallow call

stack. This is a significant advantage over the previous hierarchical version.

4.1 Implementation

Let ϕ be a formula overapproximating the set of reachable configurations of hierarchical model. One cheap approximation, used by SCOPE, is the set of all statically legal configurations (configurations satisfying the invariants). Better approximations can be used, including exact reachable state space representations [11]. This would increase the accuracy of the optimizations to follow, at the price of code generation time.

4.1.1 Administrative signals

Massive use of signals is a disadvantage at runtime as it demands writable memory for maintenance of the signal queue. We can get rid of administrative signals by exploiting the fact that code generation is actually solving a simpler than generic flattening. The main difference is that at runtime transitions are processed and fired in some fixed deterministic order. This order may substitute a signal queue in guaranteeing proper schedules of action transitions. For instance the t_1 transition of Fig. 1 can be flattened to:

$$\text{exit:} \quad \begin{cases} D_{11} \xrightarrow{[e_1 : D_{11} \wedge D_1 \wedge (D_{12} \wedge \neg E_{11})]/ex(D_{11})} \emptyset \\ D_{12} \xrightarrow{[e_1 : D_{12} \wedge D_1 \wedge (D_{12} \wedge \neg E_{11})]/ex(D_{12})} \emptyset \\ D_2 \xrightarrow{[e_1 : D_2 \wedge D_1 \wedge (D_{12} \wedge \neg E_{11})]/ex(D_2)} \emptyset \\ D_1 \xrightarrow{[e_1 : D_1 \wedge (D_{12} \wedge \neg E_{11})]/ex(D_1)} \emptyset \\ E_1 \xrightarrow{[e_1 : E_1 \wedge D_1 \wedge (D_{12} \wedge \neg E_{11})]/ex(E_1)} \emptyset \\ \dots \end{cases} \tag{4}$$

$$\text{output:} \quad \left\{ D_1 \xrightarrow{[e_1 : D_1 \wedge (D_{12} \wedge \neg E_{11})]/\langle o_1, s_1 \rangle} \emptyset \right. \tag{5}$$

$$\text{entry:} \quad \left\{ D_1 \xrightarrow{[e_1 : D_1 \wedge (D_{12} \wedge \neg E_{11})]/en(A)\hat{}en(C_1)\hat{}en(C_{11})} \emptyset \right. \tag{6}$$

Remember that the empty set of targets of the flat transition means that the active state configuration remains unchanged (see (2)). Although similar to loops, transitions with no targets are more efficient to execute.

The above sequence has been obtained by instantiating a transition for every signal of the schedules of t_1'. Its top–down interpretation corresponds to firing t_1'. In the generic algorithms of the previous section we would use a signal queue to guarantee this sequencing. Presently no administrative signals are used, but guards are evaluated multiple times and entry/exit transitions cannot be reused.

4.1.2 Guard Analysis

Guard analysis may be used to eliminate superfluous transitions and computations. Note that the *flat* function produces guards with redundant conditions (see γ in (3)). Such guards can be improved, by optimization performed under the assumption that formula ϕ holds. This automatically removes references to constantly true variables, superfluous states (such us E_1, which is equivalent to E). Removal of variables should be accompanied by removal of corresponding states from the flat structure. There is no need to represent them at runtime. For example groups of states containing E_1 and I would be eliminated and guards simplified respectively.

While performing guard analysis, unsatisfiable guards can be found on transitions. Transitions containing them can be discarded. For example the exit transition from state D_2 in (4) will never fire as D_1 and D_2 may never be active at the same time according to ϕ. SCOPE relies on a BDD [10] engine [16] in the implementation of guard analysis.

4.1.3 Merging Transitions

Some transitions in the sequence can be merged saving space and increasing the execution speed. Whenever guards of two subsequent transitions are equivalent (under ϕ), the two rules can be merged into a single one whose action is the concatenation of original actions and whose set of targets is the union of the original sets. A special case, which is particularly easy to detect, is that of the last two entries in every group implementing a single transition. For example transitions (5) and (6) can be combined yielding:

$$D_1 \xrightarrow{[e_1 : D_1 \wedge (D_{12} \wedge \neg E_{11})]/\langle o_1 \rangle\hat{}en(A)\hat{}en(C_1)\hat{}en(C_{11})\hat{}\langle s_1 \rangle} \emptyset$$

Although much more complex, merging is also applied to exit transitions. Exit transitions, or parts of thereof, can be joined if there is some firm knowledge about the source states of given transition (inferred from the guard). In the extreme case, when the knowledge about the source and target states is complete the transition gets translated to a single rule. This is the case with t_3:

$$C_2 \xrightarrow{[e_2 : \gamma]/ex(C_2)\hat{}ex(C_1)\hat{}ex(A)\hat{}en(B)\hat{}en(D_1)\hat{}en(D_{12})\hat{}en(E_1)\hat{}en(E_{11})} \emptyset \ ,$$

where $\gamma = C_2 \wedge C_1 \wedge A$.

4.2 Language Extensions

The source language language can be extended with various features, which do not break the complexity result. Parameterized events (value passing), arithmetic variables, timed transitions, expressions and function calls in guards and outputs are entirely orthogonal to the problem. Deep history, fork and join transitions, do-reactions and internal rules can be described as syntactic sugar, expanded within polynomial bounds. Multiscope transitions and priorities for conflict resolutions can be eliminated using algorithms similar to those of [28, 13, 7].

4.3 Experimental Evaluation

The algorithm has been evaluated using both artificial and real industrial examples. Efficiency has been measured against the older version of SCOPE [26] that used hierarchy preserving code generation, which is known to perform comparably to industrial implementations such as visualSTATE. Executable sizes and execution times are reported for skeleton control programs with dummy action and guard functions, compiled by gcc ver. 3.2 targeting x86 PC (see Table 1). Execution time has been measured for feeding the compiled system with 10^7 random events in repetitive series (run on Linux, 450MHz Pentium II). More experiments has been carried out with GCC for AVR and H8/300, exhibiting similar results (see Table 2). This confirms the expectation that the code generated from statecharts is hard for compilers to optimize and there is a need for advanced analysis and transformations on the model level.

The first model, actions01, is the smallest model which can be built with visualSTATE designer. It contains two states and a single transition. This model exhibits the difference between sizes of runtime libraries. The difference does not increase for bigger examples—both algorithms scale well. However, the flattened code is faster, simpler and

Model	states	trans.	depth	executable size			execution time		
				FL-CG	HI-CG	ratio	FL-CG	HI-CG	ratio
actions01	4	1	3	3 036	3 704	0.82	5.71	6.03	0.95
lift	18	19	3	3 644	4 372	0.83	15.29	21.41	0.71
clockradio	20	27	7	4 652	4 108	0.88	8.84	11.69	0.76
cdplayer	21	16	7	3 560	4 312	0.83	9.38	11.77	0.80
peer	275	192	23	9 252	10 536	0.88	15.19	26.16	0.58
trios01	1121	840	9	20 772	24 108	0.86	335	255	1.31
trios03	1121	840	9	19 972	24 684	0.81	288	259	1.10

Table 1: Speed and size results: hierarchical code generation vs flattening-based code generation. FL-CG denotes code generation for flat models, HI-CG denotes direct code generation for hierarchical models (an older version of SCOPE). Executable sizes in bytes, timings in seconds.

Model	H8 executable size			AVR executable size			AVR runtime vs model size			
	FL-CG	HI-CG	ratio	FL-CG	HI-CG	ratio	FL rtime	FL model	HI rtime	HI model
actions01	1 312	1 906	0.69	952	1 592	0.60	313	25	925	50
lift	1 750	2 356	0.74	1 428	2 076	0.69	459	353	1 047	412
clockradio	2 116	2 640	0.80	1 856	2 412	0.77	377	826	947	800
cdplayer	1 700	2 344	0.73	1 400	2 108	0.66	353	431	1 027	465
peer	6 298	7 226	0.87	6 076	7 004	0.87	530	4 684	1 226	4 888
trios01	17 670	18 326	0.96	17 396	18 212	0.96	409	16 371	1 143	16 454
trios03	16 878	20 064	0.84	16 604	19 980	0.83	409	15 580	1 229	18 134

Table 2: Left: sizes of executables compiled with two microcontroller back-ends of gcc 3.3.2 (h8300-hms-gcc and avr-gcc). Right: sizes of the runtime interpreter and the runtime model representation for gcc-avr compiled code. These numbers are included in the complete executables of the left side. Approximately 600 bytes is left out in each line, used by gcc for internal initialization code.

smaller, despite the fact that less engineering effort had been put in the implementation of the runtime. No size explosions are visible for big models (trios01, trios03). Peer, the biggest real life example, a complicated model of a very advanced coffee machine exhibits a particularly good result, which is perhaps the best recommendation of our approach.

The *lift* example is a flat statechart and as such is not affected by flattening. Still the flattened version operates much faster than hierarchical one as the hierarchical interpretation is relatively expensive. Flat models constitute an important class of models, appreciated by engineers for its simplicity and good applicability to small size tasks. It is thus comforting that a single code generation scheme, the one based on flattening, performs well for both flat and hierarchical models.

The last two models (from *trios* series) are artificial examples. They consist of triples of and-states and or-states interleaved several times. These kind of examples is used in [27], in order to demonstrate the superpolynomial nature of flattening in absence of sequential message passing. As expected, no explosion is observed in present results, where we exploit the sequential nature of code generation. Such highly concurrent models are hardly met in industrial applications, which makes the slow-down reported not essential. Also none of the industrial cases we know suffers from to slow interpretation of the generated control code. Speed seems to be much less an issue than code size and memory consumption.

5. RELATED WORK

Both the upper bound result and the efficiency of the implementation are somewhat surprising as numerous authors presented flattening algorithms suffering from size explosion or informally conjectured about the superpolynomial hardness of this problem [26, 12, 8, 3, 9, 21].

It has been recently established [27] that the flattening problem does explode if the target language of flat statecharts does not include queue-based signal communication. In [27] a family of (α, β)-models is exhibited such that for any family member its flat implementation must necessarily be at least $\Omega(2^{\sqrt{|State|}})$ if we do not allow use of signals. Moreover a formal relation between this bound and the amount of concurrency and sequential computation in the model is established. The lower bound increases together with the increase of concurrency in the statechart. Thus a lower bound, which is arbitrarily close to an exponential function may be shown using (α, β)-models.

The joint conclusion of the present paper and [27] is that equipping the target formalism with signal queues vastly increases expressiveness. Indeed, if the queue is unbounded, one obtains a language of infinite state systems. However in practice the size of the queue is bounded. The algorithm presented here only demands finite bounded queues, with length polynomial in the model size.

Two classic works on succinctness of statecharts are [14, 2]. Drusinsky and Harel [14] discuss the succinctness gain caused by introduction of bounded cooperative concurrency, disregarding the impact of hierarchy on model size. Alur et al.[2] discuss the interplay between nondetermin-

ism, concurrency and hierarchy. Unfortunately among all relations discussed in their paper they omit the one interesting for its application in program synthesis: the relation between hierarchical concurrent statecharts and flat concurrent ones. Also they permit sharing of subhierarchies, corresponding roughly to dropping the [unique_parent] requirement on $\searrow$. This feature certainly affects succinctness but so far has not been incorporated into main stream modeling languages.

Literature contains numerous examples of statechart translation. Many so called hierarchical code generators [26, 1, 29] struggle to represent hierarchy explicitly at runtime. They usually employ complicated execution mechanisms, making the runtime code more error-prone and more expensive to execute, especially in terms of writable memory consumption. The author is not aware of any published material on flattening-based code generator not suffering from the explosion problem.

The full language of statecharts incorporates model variables, which can be used in guards and action expressions. In the implementation these variables need to be *double-buffered* to guarantee stable execution of a microstep, so that results of computations do not affect what transitions are fired in the same microstep. A copy of variable's value at the end of the last step is used in the evaluation of guards, while the actual value may be modified by transition actions. This way transitions can be fired on-the-fly along with the evaluation of guards. Erpenbach [15] exploits the order of transitions to minimize the amount of necessary double buffering between variables. Our flattening algorithm is fully compatible with his approach, because we have only demanded that rules within a group implementing a single hierarchical transition maintain a specific order. We do not impose any restrictions on the relative order among the groups themselves, which suffices for Erpenbach's algorithm.

6. CONCLUSION

I have presented an upper-bound proof for flattening of hierarchical statecharts, one of the most central and widespread transformations applied to statechart models. This result contradicts a common but informal belief that flattening cannot be solved without size explosion. I argued that the algorithm is correct and commented on the applicability of it to typical language variants. Another conclusion following from both present paper and [27] is that signal communication with buffers is a powerful feature of statecharts, which can vastly increase succinctness of models.

In the second part of the paper a detailed description of an optimized implementation has been given. This is the core algorithm of the newest version of the SCOPE code generator. The code generated by the new procedure is fast and compact and exhibits good properties with respect to writable memory consumption. Its simplicity makes it suitable for automatic validation. Unlike hierarchical code generators, our new technique performs well both for flat and non-flat models, which is important in practice.

7. ACKNOWLEDGMENTS

Some concepts of section 4 are inspired by an unpublished work of Gerd Behrmann. I would like to thank Peter Sestoft, for reviewing an earlier version of this material, and to the reviewers for suggestions of important improvements.

8. REFERENCES

[1] J. Ali and J. Tanaka. Converting statecharts into Java code. In *5th International Conference on Integrated Design and Process Technology (IDPT'99)*, Dallas,Texas, June 1999.

[2] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *26th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *LNCS*, pages 169–178, Prague, Czech Republic, July 1999. Springer-Verlag.

[3] G. Behrmann, K. G. Larsen, H. R. Andersen, H. Hulgaard, and J. Lind Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *LNCS*, pages 163–177, Amsterdam, The Netherlands, Mar. 1999. Springer-Verlag.

[4] R. V. Binder. *Testing Object-Oriented Systems. Models, Patterns and Tools.* Addison-Wesley, 2000.

[5] D. Björklund, J. Lilius, and I. Porres. Towards efficient code synthesis from statecharts. In A. Evans, R. France, and A. M. B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods. Countering or Integrating the eXtremists*, Lecture Notes in Informatics P-7, Toronto,Canada, Oct. 2001.

[6] K. Bogdanov. *Automated testing of Harel's statecharts.* PhD thesis, The University of Sheffield, Jan. 2000.

[7] K. Bogdanov and M. Holcombe. Statechart testing method for aircraft control systems. *Software Testing, Verification and Reliability*, 1(11):39–54, 2001.

[8] K. Bogdanov and M. Holcombe. Properties of concurrently taken transitions of Harel statecharts. In *Workshop on Semantic Foundations of Engineering Design Languages (SFEDL)*, Grenoble, France, April 2002.

[9] G. W. Bond, F. Ivancic, N. Klarlund, and R. Trefler. Eclipse feature logic analysis. In *2nd IP-Telephony Workshop*, pages 100–107, New York City, USA, April 2001.

[10] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.

[11] E. M. Clarke. *Model Checking.* The MIT Press, Dec. 1999.

[12] A. David, M. O. Möller, and W. Yi. Formal verification of UML statecharts with real-time extensions. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 2306 of *LNCS*, pages 218–232, Grenoble, France, April 2002. Springer-Verlag.

[13] K. Diethers, U. Goltz, and M. Huhn. Model checking UML statecharts with time. In Jürjens et al. [19], pages 35–51. TUM-I0208.

[14] D. Drusinsky and D. Harel. On the power of bounded concurrency I: Finite automata. *Journal of ACM*, 41(3):517–539, May 1994.

[15] E. Erpenbach. *Compilation, Worst-Case Execution Times and Schedulability Analysis of Statecharts Models.* PhD thesis, Department of Mathematics and

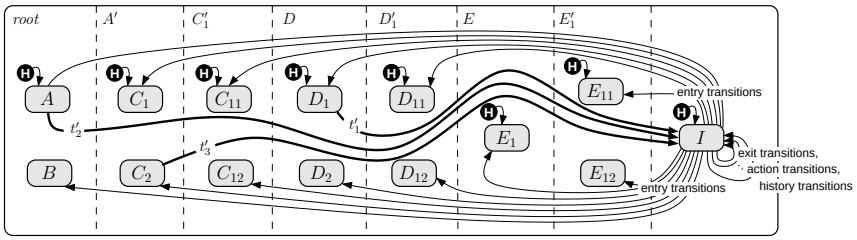

Figure 3: An imprecise but intuitive overview of results of flattening of statechart of Fig. 1.

State groups and initial configuration:

$M = \{\{A, B\}, \{C_1, C_2\}, \{D_1, D_2\}, \{E_1\}, \{C_{11}, C_{12}\}, \{D_{11}, D_{12}\}, \{E_{11}, E_{12}\}, \{I\}\}$

$s^0 = \{A, C_1, C_{11}, D_1, D_{11}, E_1, E_{11}, I\}$

External and internal events:

$Event' = \{e_1, e_2\}$

$Signal' = \{s_1, e^{\text{en}}_A, e^{\text{en}}_B, e^{\text{en}}_{C_1}, e^{\text{en}}_{C_2}, e^{\text{en}}_{D_1}, e^{\text{en}}_{D_2}, e^{\text{en}}_{E_1}, e^{\text{en}}_{C_{11}}, e^{\text{en}}_{C_{12}}, e^{\text{en}}_{D_{11}}, e^{\text{en}}_{D_{12}}, e^{\text{en}}_{E_{11}}, e^{\text{en}}_{E_{12}},$
$e^{\text{ex}}_A, e^{\text{ex}}_B, e^{\text{ex}}_{C_1}, e^{\text{ex}}_{C_2}, e^{\text{ex}}_{D_1}, e^{\text{ex}}_{D_2}, e^{\text{ex}}_{E_1}, e^{\text{ex}}_{C_{11}}, e^{\text{ex}}_{C_{12}}, e^{\text{ex}}_{D_{11}}, e^{\text{ex}}_{D_{12}}, e^{\text{ex}}_{E_{11}}, e^{\text{ex}}_{E_{12}}, e^{\text{h}}_D, e_{t_1}, e_{t_2}, e_{t_3}\}$

Exit transitions:

$I \xrightarrow{[e^{\text{ex}}_A : A]/ex(A)} I,\ I \xrightarrow{[e^{\text{ex}}_B : B]/ex(B)} I,\ I \xrightarrow{[e^{\text{ex}}_{C_1} : C_1 \wedge A]/ex(C_1)} I,\ I \xrightarrow{[e^{\text{ex}}_{C_2} : C_2 \wedge A]/ex(C_2)} I,\ I \xrightarrow{[e^{\text{ex}}_{D_1} : D_1 \wedge B]/ex(D_1)} I,\ I \xrightarrow{[e^{\text{ex}}_{D_2} : D_2 \wedge B]/ex(D_2)} I,$

$I \xrightarrow{[e^{\text{ex}}_{E_1} : E_1 \wedge B]/ex(E_1)} I,\ I \xrightarrow{[e^{\text{ex}}_{C_{11}} : C_{11} \wedge C_1 \wedge A]/ex(C_{11})} I,\ I \xrightarrow{[e^{\text{ex}}_{C_{12}} : C_{12} \wedge C_1 \wedge A]/ex(C_{12})} I,\ I \xrightarrow{[e^{\text{ex}}_{D_{11}} : D_{11} \wedge D_1 \wedge B]/ex(D_{11})} I,$

$I \xrightarrow{[e^{\text{ex}}_{D_{12}} : D_{12} \wedge D_1 \wedge B]/ex(D_{12})} I,\ I \xrightarrow{[e^{\text{ex}}_{E_{11}} : E_{11} \wedge E_1 \wedge B]/ex(E_{11})} I,\ I \xrightarrow{[e^{\text{ex}}_{E_{12}} : E_{12} \wedge E_1 \wedge B]/ex(E_{12})} I$

Entry transitions:

$I \xrightarrow{[e^{\text{en}}_A : true]/en(A)} A,\ I \xrightarrow{[e^{\text{en}}_B : true]/en(B)} B,\ I \xrightarrow{[e^{\text{en}}_{C_1} : A]/en(C_1)} C_1,\ I \xrightarrow{[e^{\text{en}}_{C_2} : A]/en(C_2)} C_2,\ I \xrightarrow{[e^{\text{en}}_{D_1} : B]/en(D_1)} D_1,\ I \xrightarrow{[e^{\text{en}}_{D_2} : B]/en(D_2)} D_2,$

$I \xrightarrow{[e^{\text{en}}_{E_1} : B]/en(E_1)} E_1,\ I \xrightarrow{[e^{\text{en}}_{C_{11}} : C_1 \wedge A]/en(C_{11})} C_{11},\ I \xrightarrow{[e^{\text{en}}_{C_{12}} : C_1 \wedge A]/en(C_{12})} C_{12},\ I \xrightarrow{[e^{\text{en}}_{D_{11}} : D_1 \wedge B]/en(D_{11})} D_{11},$

$I \xrightarrow{[e^{\text{en}}_{D_{12}} : D_1 \wedge B]/en(D_{12})} D_{12},\ I \xrightarrow{[e^{\text{en}}_{E_{11}} : E_1 \wedge B]/en(E_{11})} E_{11},\ I \xrightarrow{[e^{\text{en}}_{E_{12}} : E_1 \wedge B]/en(E_{12})} E_{12}$

Action transitions (for original hierarchical transitions):

$I \xrightarrow{[e_{t_1} : true]/\langle o_1, s_1 \rangle} I,\ I \xrightarrow{[e_{t_2} : true]/\langle o_2 \rangle} I,\ I \xrightarrow{[e_{t_3} : true]/\langle\rangle} I$

History entries:

$I \xrightarrow{[e^{\text{h}}_D : D_1]/\langle e^{\text{en}}_{D_1}, e^{\text{en}}_{D_{11}} \rangle} I,\ I \xrightarrow{[e^{\text{h}}_D : D_2]/\langle e^{\text{en}}_{D_2} \rangle} I$

Schedule transitions (bold on the diagram above):

$t'_1 = D_1 \xrightarrow{[e_1 : (D_1 \wedge B) \wedge ((D_{12} \wedge D_1 \wedge B) \wedge \neg(E_{11} \wedge E_1 \wedge B))]/\langle e^{\text{ex}}_{C_{11}}, e^{\text{ex}}_{C_{12}}, e^{\text{ex}}_{C_1}, e^{\text{ex}}_{C_2}, e^{\text{ex}}_A, e^{\text{ex}}_{D_{11}}, e^{\text{ex}}_{D_{12}}, e^{\text{ex}}_{D_1}, e^{\text{ex}}_{D_2}, e^{\text{ex}}_{E_{11}}, e^{\text{ex}}_{E_{12}}, e^{\text{ex}}_{E_1}, e^{\text{ex}}_B, e_{t_1}, e^{\text{en}}_A, e^{\text{en}}_{C_1}, e^{\text{en}}_{C_{11}} \rangle} I$

$t'_2 = A \xrightarrow{[s_1 : A \wedge \neg(C_2 \wedge A)]/\langle e^{\text{ex}}_{C_{11}}, e^{\text{ex}}_{C_{12}}, e^{\text{ex}}_{C_1}, e^{\text{ex}}_{C_2}, e^{\text{ex}}_A, e^{\text{ex}}_{D_{11}}, e^{\text{ex}}_{D_{12}}, e^{\text{ex}}_{D_1}, e^{\text{ex}}_{D_2}, e^{\text{ex}}_{E_{11}}, e^{\text{ex}}_{E_{12}}, e^{\text{ex}}_{E_1}, e^{\text{ex}}_B, e_{t_2}, e^{\text{en}}_b, e^{\text{h}}_D, e^{\text{en}}_{D_{11}}, e^{\text{en}}_{E_1}, e^{\text{en}}_{E_{11}} \rangle} I$

$t'_3 = C_2 \xrightarrow{[e_2 : C_2 \wedge A]/\langle e^{\text{ex}}_{C_{11}}, e^{\text{ex}}_{C_{12}}, e^{\text{ex}}_{C_1}, e^{\text{ex}}_{C_2}, e^{\text{ex}}_A, e^{\text{ex}}_{D_{11}}, e^{\text{ex}}_{D_{12}}, e^{\text{ex}}_{D_1}, e^{\text{ex}}_{D_2}, e^{\text{ex}}_{E_{11}}, e^{\text{ex}}_{E_{12}}, e^{\text{ex}}_{E_1}, e^{\text{ex}}_B, e_{t_3}, e^{\text{en}}_B, e^{\text{en}}_{D_1}, e^{\text{en}}_{D_{12}}, e^{\text{en}}_{E_1}, e^{\text{en}}_{E_{11}} \rangle} I$

Figure 4: Complete ruleset produced during flattening. Observe the inefficiencies, which can be removed using techniques of section 4, especially redundant checks in guards and need for a long local events buffer.

Computer Science of the University of Paderborn, April 2000.

[16] K. Friis Larsen and J. Lichtenberg. MuDDy 2.0 – SML interface to the binary decision diagrams package BuDDy. `http://www.itu.dk/research/muddy`.

[17] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[18] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[19] J. Jürjens, M. V. Cengarle, E. B. Fernandez, B. Rumpe, and R. Sandner, editors. *pUML Group Workshop on Critical Systems Development with UML (CSDUML)*, Dresden, Germany, Sept. 2002. Technical University of Munich. TUM-I0208.

[20] Object Management Group. OMG Unified Modelling Language specification, 1999. `http://www.omg.org`.

[21] M. Riebisch, I. Philippow, and M. Götze. UML-based statistical test case generation. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World. NetObjectDays, Revised Papers*, volume 2591 of *LNCS*, pages 394–411, Erfurt, Germany, Oct. 2002. Springer-Verlag.

[22] E. E. Roubtsova, J. van Katwijk, R. C. M. de Rooij, and H. Toetenel. Transformation of UML specification to XTG. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Perspectives of System Informatics (PSI), 4th International Andrei Ershov Memorial Conference, Revised Papers*, volume 2244 of *LNCS*, pages 249–256, Novosibirsk, July 2001. Springer-Verlag.

[23] SCOPE: A statechart compiler, 2003. `http://www.mini.pw.edu.pl/~wasowski/scope`.

[24] A. J. H. Simons. The compositional properties of UML statechart diagrams. In C. J. van Rijsbergen, editor, *3rd Electronic Workshop on Rigorous Object-Oriented Methods*. British Computer Society, 2000.

[25] J. Staunstrup, H. R. Andersen, H. Hulgaard, J. Lind-Nielsen, K. G. Larsen, G. Behrmann, K. J. Kristoffersen, A. Skou, H. Leerberg, and N. B. Theilgaard. Practical verification of embedded software. *IEEE Computer*, 5(33):68–75, 2000.

[26] A. Wąsowski. On Efficient Program Synthesis from Statecharts. In *ACM SIGPLAN Languages, Compilers, and Tools for Embedded Systems (LCTES)*, San Diego, USA, June 2003. ACM Press.

[27] A. Wąsowski. On Succinctness of Hierarchical State Diagrams in Absence of Message Passing. In M. Mendler, editor, *Semantic Foundations of Engineering Design Languages (SFEDL)*. A satellite workshop of ETAPS 2004. Barcelona, Spain, March 2004. To be published in ENTCS. Elsevier Science Publishers, 2004.

[28] A. Wąsowski and P. Sestoft. Compile-time scope resolution for statecharts transitions. In Jürjens et al. [19], pages 133–145. TUM-I0208.

[29] A. Zündorf. Rigorous object oriented software development with Fujaba. Unpublished Draft, 2000.

APPENDIX

A. EXAMPLE RESULTS

Fig. 3 presents an overview of the flat statechart produced by applying the algorithm of section 3 to the model of Fig. 1. Notation is imprecise, aiming at intuitive explanation. We use a hierarchical syntax to express flat statecharts. All state machines are marked history, to reflect the fact that the top state is never exited. More precise account of the transformation can be found on Fig. 4.

B. CORRECTNESS SKETCH

DEFINITION 5. *A state configuration and a history set (σ, η) of hierarchical statechart and a configuration σ' of a flat statechart correspond, written $\sigma' \sim_{\mathrm{s}} (\sigma, \eta)$, iff all maintain their respective correctness invariants and:*

1. $\sigma \cap State_{\mathsf{and}} \subseteq \sigma' \subseteq State_{\mathsf{and}} \cup \{I\}$
2. $\forall s \in \sigma'.s \neq I \Rightarrow (s \in \sigma \vee ancest^{+}(s) \cap State_{\mathsf{and}} \not\subseteq \sigma')$
3. $\eta \subseteq \sigma'$

PROPOSITION 6. *The initial configuration σ_0 and history set η_0 of hierarchical statechart and the initial configuration σ'_0 of the flattened statechart correspond: $\sigma'_0 \sim_{\mathrm{s}} (\sigma_0, \eta_0)$.*

PROPOSITION 7. *If $\sigma' \sim_{\mathrm{s}} (\sigma, \eta)$ then*

$$\forall g \in Guard.\ (\sigma \models g \iff \sigma' \models flat(g))$$

PROPOSITION 8. *If t' is a flattened version of t and $\sigma' \sim_{\mathrm{s}} (\sigma, \eta) \wedge q' = q$ then $enabled(t') \iff enabled(t)$.*

DEFINITION 9. *A* ministep *is a sequence of microsteps of a flat statechart iterated until the queue only contains elements of Signal (no administrative signals).*

PROPOSITION 10. *If $\sigma'_0 \sim_{\mathrm{s}} (\sigma_0, \eta_0) \wedge q'_0 = q_0$ and the flattened statechart can perform a ministep $(\sigma'_0, q'_0) \xrightarrow[mini]{os} (\sigma'_1, q'_1)$ then there exist a sequence of hierarchical microsteps such that $(\sigma_0, q_0, \eta_0) \xrightarrow[micro]{os}{}^{*} (\sigma_1, q_1, \eta_1)$ and $\sigma'_1 \sim_{\mathrm{s}} (\sigma_1, \eta_1) \wedge q'_1 = q_1$.*

PROPOSITION 11. *If $\sigma'_0 \sim_{\mathrm{s}} (\sigma_0, \eta_0)$ and the flattened statechart can perform a microstep $\sigma'_0 \xrightarrow[macro]{e\ os} \sigma'_1$ then there exists a hierarchical macrostep $(\sigma_0, \eta_0) \xrightarrow[macro]{e\ os} (\sigma_1, \eta_1)$ and $\sigma'_1 \sim_{\mathrm{s}} (\sigma_1, \eta_1)$.*