Compile-time Scope Resolution for Statecharts Transitions *

Andrzej Wąsowski and Peter Sestoft

IT University of Copenhagen
{wasowski,sestoft}@it-c.dk

Abstract. Despite the success of statecharts, surprisingly little research effort has been devoted to improving code synthesis techniques for them. The work presented below is an outcome of growing interest in the area. We discuss one possible improvement for code generators retaining explicit information about the model hierarchy. The problem of dynamic scope for multitarget transitions is described and an algorithm for compile-time detection and resolution is presented. Finally we examine the possibilities of employing this technology in various compilation optimization for classical UML state diagrams which do not allow multitarget transitions.

1 Introduction

UML languages are receiving increasing interest in the embedded systems community. Especially statecharts[8, 14] enjoy a constantly growing number of publicly available implementations supporting design, verification and code synthesis. Increased reliability of the final product encourages use of high level description techniques. Unfortunately at the same time runtime overheads in synthesized code discourage applications with hard time requirements and space constraints. Relatively inefficient code generation techniques are probably the main reason why the vast part of embedded devices are still programmed with low level tools and languages (assembly languages, C, mutations of C++, etc).

Available code synthesizers targeting embedded systems focus on fully automatic synthesis of efficient and compact code [15, 10, 5, 11]. This contrasts with traditional approach of CASE tools for standard PC software, which emphasize readability and ability to manually modify the final result. Automatic nonmodifiable code generation gives much better results in terms of efficiency and size.

IAR visualSTATE [11] is a CASE development system including a statechart implementation following the unified notation of UML. IAR visualSTATE is specifically devoted to development of embedded software. Its state diagrams are compiled to a set of compact runtime structures representing the model, which are then interpreted by a simple runtime driver (mostly model independent). From both safety and efficiency perspective, it is desirable that the logics of the

* Revised 3/10/2002

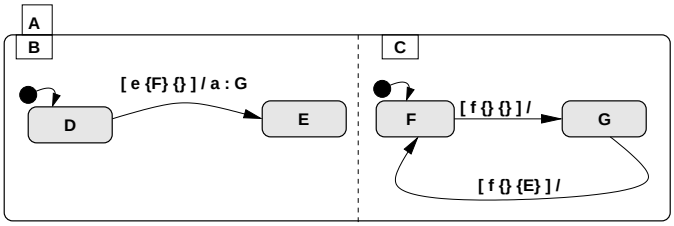

Fig. 1. Statechart with single-level transitions. A is an and-state, which is a parallel composition of or-states B and C. State B is in turn a sequential composition of basic and-states D and E.

interpreter is as simple as possible, speeding up execution process and decreasing trusted code base.

In this paper we analyze a specific problem of code synthesis for visualSTATE statecharts: *dynamic scope resolution for multitarget transitions*. The problem affects those code generation schemes which preserve high-level information about the model hierarchy at runtime. A straightforward solution computes the scope of transition at runtime. This approach however would slow the execution of all transitions down (not only multitarget transitions).

We propose an algorithm shifting the task of detection and scope resolution from runtime to compile time, simplifying the runtime engine and improving its efficiency. The algorithm has been implemented and evaluated in our experimental code generator.

2 Multitarget Transitions

A precise operational semantics for the visualSTATE language has been given in [16]. A brief description should suffice for the purpose of current paper. Statecharts of visualSTATE follow the semantics of UML state diagrams in general. See figures 1 and 2 for basic examples.

States are organized in a hierarchy by means of parallel composition of or-states and sequential compositions of and-states. States belonging to concurrent or-states are called orthogonal. The decomposition tree shown on figure 2 is a convenient way to explicitly represent the hierarchy. The root of the tree is a top level and-state. Types of internal nodes alternate between and-states and or-states on each path from root to leaves. A childless leaf (basic state) is always an and-state. Formally two states are orthogonal if their nearest common ancestor is an and-state.

Each state may have some actions assigned, which are fired whenever the state is entered (entry action) or is left (exit action).

The precise dynamic behavior of the model is specified using transitions describing state changes. Each transition comprises two parts: a condition side and an executable side. The condition side declares a single triggering event and

a guard. The guard is a boolean condition generated from the following grammar over terms on activity and inactivity of states in current configuration:

$$g \quad ::= \quad true \mid g \wedge s \mid g \wedge \neg s \ , \tag{1}$$

where s stands for any state name.

The executable side of a transition specifies a number of actions to be executed when the transition is fired, and a set of target states which should become active after the transition is taken. We use textual syntax of rules for convenient representation of visual transition objects:

$$t: \ [e \ pos \ neg] \ / \ a \ : s_1 ... s_k \ , \tag{2}$$

where t is an optional name of the transition, e is a triggering event, *pos* is a set of states which have to be active, *neg* is a set of states demanded to be inactive in order for the transition to be enabled, a is an action to execute, and s_i are target states. Note that the source and target states of transition arrow are not represented explicitly in this notation. If visual transition has the shape

$$s_s \longrightarrow s_t \ ,$$

then the source state s_s is included in the set of positive guards *pos* and the target state s_t, indicated by arrow head, is included in general set of targets $\{s_1, ..., s_k\}$ of formula (2).

Standard UML transitions may be forked to several targets located in orthogonal components. The intended meaning is that control forks from one state (perhaps itself concurrent) to several concurrent states. In a well formed statechart targets must be mutually orthogonal and they may not be orthogonal with any of the source states of transition. It is therefore not possible to affect already active components concurrent to the transition itself in the same step when transition is fired. In other words, the transition arrow can never cross dotted region boundaries. Changes in concurrent components may be achieved by triggering a local event e which will cause a transition to fire in demanded concurrent region.

State diagrams of visualSTATE relax standard UML conditions imposed on targets. It is allowed to enter states in components orthogonal to source state. Figure 1, region B, provides an example.

The syntax is slightly modified in visualSTATE; a transition is not visually forked anymore. Instead one target is selected as the *explicit* target, pointed at by the arrow head of transition. Following textual rule notation – see formula (2) – remaining targets are added to the executable part of transition, avoiding peculiarity of arrows crossing boundaries between concurrent regions. Thus the transitions of figure 1 may be written down as following textual rules:

$$\begin{array}{lllllll} t_1 & : & [\ e & \{D, F\} & \{\} &] \ / \ a & : \ E \ G \\ t_2 & : & [\ f & \{F\} & \{\} &] \ / \ - & : \ G \\ t_3 & : & [\ f & \{G\} & \{E\} &] \ / \ - & : \ F \ . \end{array} \tag{3}$$

Consider transition t_1. Assume that current event is e and states D, F are active in current state configuration σ. Then t_1 is enabled. The semantics of firing t_1 is to execute exit actions of D and F, remove D, F from current configuration, execute action a and entry actions of E, G, finally adding E, G to current configuration.

The interesting point is to note that a transition with several targets actually takes place in *several scopes*. For the first target (E) all active descendants of B have to be exited. The state change to state E takes place within region of B. We call B the *scope of the state change* to state E in configuration σ. The scope for the second target (G) is region C. A slightly more detailed textual notation can be proposed for rules, making scopes explicit:

$$\begin{array}{lllllll} t_1 & : & [\ e & \{D,F\} & \{\} &] & / \ a \ : \ [B]E\ [C]G \\ t_2 & : & [\ f & \{F\} & \{\} &] & / - \ : \ [C]G \\ t_3 & : & [\ f & \{G\} & \{E\} &] & / - \ : \ [C]F \ . \end{array} \tag{4}$$

Transitions with generalized targets as described in this section are called *multitarget transitions* as opposed to the classical join and fork transitions of UML.

The execution semantics for multitarget transitions may seem a bit complicated. It should be stressed though, that in most applications multitarget transitions are used in a very restricted and straightforward way. An average multitarget transitions will only have two targets and is used in a manner very similar to broadcasting of signals. Thus it can intuitively be used by developers.

There is a significant drawback of multitarget transitions. Broadcasts are definitely more clear notationally: signal is well visualized in both the broadcasting component and in the receiving component. At the same time a multitarget transition will only visualize an additional target on its arrow – in the broadcasting component. There is no notation in the affected component informing about the externally caused state change. For this reason we do not consider multitarget transitions as a good language extension. At the same time we argue that multitarget transitions can be more efficient than broadcasts and because of that they are suitable for use in runtime representations of statecharts.

3 Dynamic Scopes Problem

In general, it is not possible to annotate arbitrary transitions with scopes for each target change, as in the simple example of fig.1. The scope of a state change potentially depends on the current configuration. Consider the example in figure 2. This example contains a single transition that affects both concurrent components.

Let as assume that all statically imposed state configurations are legal for this statechart. [1] Figure 3 shows legal state configurations in which the transition

[1] This is a conservative estimation. Determining the exact set of reachable configurations demands performing a full reachability test. Such estimation has to be made

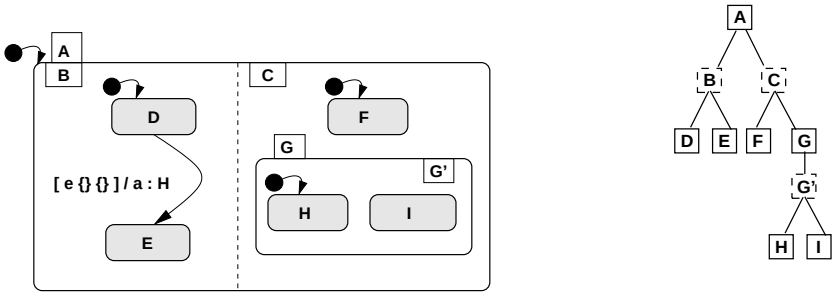

Fig. 2. Statechart exhibiting the problem of dynamic scope. The hierarchical model On the left has the decomposition tree shown on the right. Solid frame nodes represent and-states, dash-line frame nodes represent regions (or-states). H and E are orthogonal because they belong to concurrent regions of B and C. Their nearest common ancestor in decomposition tree is A, which is an and-state.

is enabled. Similarly to the first example the scope of state change to state E will always be the same, regardless of the current configuration. It has been visualized using shaded area surrounding always the same set of descendants $\{B, D, E\}$. This property is guaranteed by the implicit positive condition $\{D\}$ induced by the explicit source of transition. You can observe that the scope will always be independent of current configuration for state change to explicit target of transition (and hence for all transitions of regular UML statecharts, which only have explicit targets).

Let us look at the second target (state H), the implicit target being forced by this transition. If state F is active (figure 3a) then firing process would imply exiting state F and entering appropriately G, G' and H. The scope of the state change would be the region of C. On the other hand if I is active (figure 3b) the scope of state change becomes narrower: state G'. All states below G' must be exited and G must be entered. The same holds for the last case, where state change behaves as a self-loop on H. Thus event states not mentioned in the transition affect the set of entry and exit actions to be executed.

This can be very expressive if one uses the entry/exit actions in resource allocation (or object construction/destruction) fashion of object oriented programming. Lots of transitions can be avoided retaining relatively clear approach. At the same time the logics of runtime interpreter becomes much more complicated. If the scope of state change is dynamic then it has to be computed at runtime. Unfortunately this is an expensive process. As a result a typical interpreter will actually be slow also for majority of normal transitions with static scopes.

Various optimizations can be done to improve efficiency of this computation. We think, however, that the most efficient solution is to precompute scopes at compile time, transforming the model to guarantee that all transitions have

in any statechart compiler lacking extensive state space exploration. The algorithm presented in section 5 can be easily improved using reachability tests.

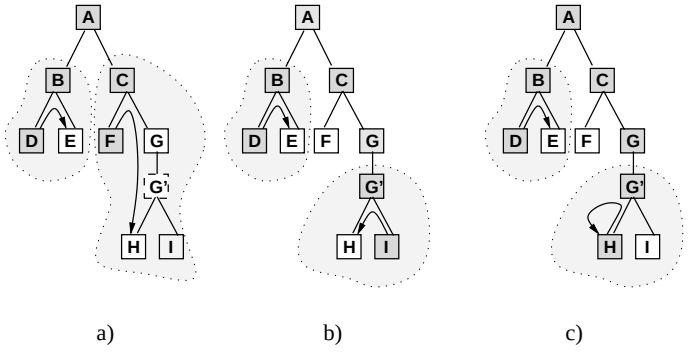

Fig. 3. Legal state configurations for the statechart in figure 2 in which the transition is enabled. Shaded tree nodes with shaded background are active. Nodes with white background are inactive. Shapes in dotted boundaries depict areas affected by firing the transition in this configurations. Arrows represent actual state changes when transition is fired in various configurations.

statically computable scopes. In such case it is possible to remove all scope resolution support from interpreter, making it simpler, more orthogonal and efficient for all of transitions without loss of performance in average, single target case.

Let us give an intuition about the main idea of transition transformation. Instead of having one transition with dynamic scope resolution for the second target (fig.2), we can split it into several mutually exclusive rules for which static scope resolution is possible:

$$\begin{aligned} &[\ e\ \{D, F\}\ \{\}\ ]\ /\ a\ :\ [B]E\ [C]H \\ &[\ e\ \{D, G\}\ \{\}\ ]\ /\ a\ :\ [B]E\ [G']H\ . \end{aligned} \tag{5}$$

We proposed two rules instead of one and ensured that scope can be resolved statically for each of them by extending guards with extra conditions. Following sections discusses how to automate this task.

4 Basic Definitions

Let us start with formalizing some basic notions. Our *state configurations* are defined to be maximal orthogonal sets of basic states [9, 16]. The set is orthogonal if all its elements are mutually orthogonal. Maximality means that adding any basic state to configuration would not preserve set orthogonality.

The visual notation distinguishes one of the positive conditions as a source of the transition arrow. We call this state the *explicit source*. Similarly one of the targets is distinguished as the *explicit target*. Explicit source and target determine the *explicit scope* of transition – the smallest component enclosing

the arrow. Explicit scope plays only a minor role in operational semantics of multitarget transitions. The main role is assigned to *implicit scope*, which directly depends on the current configuration and the target:

Definition 1 (Implicit Scope). *Let s denote the target state and σ the current state configuration. The implicit scope of state change to s is defined recursively:*

$$iscope(\sigma, s) = \begin{cases} parent(s) & \text{if } descend(parent(s)) \cap \sigma \neq \emptyset \\ iscope(\sigma, parent(s)) & \text{otherwise} \end{cases},$$

where $descend(s)$ is the set of direct and indirect descendants of state s, not including s itself.

Implicit scope can be generalized to encompass all targets of transition:

Definition 2 (Generalized Scope). *Generalized scope of transition t in state configuration σ is a set of implicit scopes given by:*

$$iscopes(\sigma, t) = \{iscope(\sigma, s) \mid s \in targets(t)\}$$

Definition 3 (Static Generalized Scope). *Transition t is said to have static generalized scope iff for any pair of state configurations σ_1, σ_2 such that transition t is enabled in both of them, and for all targets $s_1, ..., s_k$ of t, it holds that $iscope(\sigma_1, s_i) = iscope(\sigma_2, s_i)$.*

Clearly if the generalized scope of t is static then it can be computed at compile time.

5 Static Scope Resolution

The generalized scope is independent of current configuration if guard conditions are strong enough to guarantee this. In particular it suffices that guards include a *branch excluding expression* over each target to ensure that the scope is static:

Definition 4 (Branch Exclusion). *An expression $(s_1 \wedge \neg s_2)$ is called a branch excluding expression (or simply branch exclusion) iff s_2 is a child state of s_1.*

Note that it is not strictly necessary for a transition to contain branch exclusion in its guard in order to have static generalized scope. It is however necessary that branch exclusion can be implied from transition guards and model structure. This is exactly the case in our examples. None of the transitions explicitly contains a branch exclusion in its guard, but it can be trivially inferred. The strength of branch exclusion follows from the following observation and from the exclusion theorem.

Observation 5 (Guard Propagation). *If state s is active in given state configuration then all its ancestors are active. If state s is inactive then all its descendants are inactive.*

Theorem 6 (Exclusion). *Let ϕ be a formula representing all statically legal state configurations of given statechart model. Assume that state s is among the targets of transition t. The scope of state change to s is static iff one of the following conditions is satisfied:*

1. *There exist ancestors s_1, s_2 of s (possibly $s_2 = s$), such that s_1 is a parent of s_2 and $\phi \wedge guard(t) \Rightarrow (s_1 \ \wedge \ \neg s_2)$ or*
2. *$\phi \ \wedge \ guard(t) \Rightarrow s$,*

Moreover if the first condition is satisfied, then s_1 is the corresponding static scope. If the second condition is satisfied, the static scope is the parent of s.

Proof. (sketch for nontrivial case 1). Let us take any two legal state configurations σ_1, σ_2 such that transition t is enabled in both. From assumption we know the same branch exclusion is implied regardless of the choice of configuration (and we can choose any configurations satisfying ϕ). From guard propagation it follows that s_1 is the active state closest to s in both configurations. By definition of implicit scope it is the scope of state change to s for both configurations. The proof in reverse direction proceeds in a similar manner. □

The exclusion theorem helps to detect if given transition is static. It does not solve the general problem, what to do if the transition is dynamic. The way to proceed is to multiply each transition, extending its guard with suitable branch exclusions. Static resolution becomes possible and trivial for each transition decorated in this way.

Let us start with constructing a formula representing the set of legal state configurations for our state diagram. The formula will be defined over a set of boolean variables – one for each state. Assignments to these variables identify state configurations of the statechart. If a variable is assigned true then the state is meant to be active in respective configuration. It is inactive otherwise. For notational simplicity we will not distinguish between states and boolean variables representing them, similarly as in syntax for guard conditions above.

To begin with, the formula ϕ will concern all state configurations not violating the static structure of the model. To build it we conjoin following simple terms corresponding to decomposition steps:

1. if s is an and-state and $s_1, ..., s_k$ are its children then $(s \Rightarrow s_1 \wedge ... \wedge s_k) \wedge (\neg s \Rightarrow \neg s_1 \wedge ... \wedge \neg s_k)$ is a simple term.
2. if s is an or-state and $s_1, ..., s_k$ are its children then $(s \Rightarrow s_1 \ \text{XOR} \ ... \ \text{XOR} \ s_k) \wedge (\neg s \Rightarrow \neg s_1 \wedge ... \wedge \neg s_k)$ is a simple term.
3. $(root)$ is a simple term

Thus the formula for statechart of figure 2 would be:

$$^{(fig2)}\phi \ = \ A \ \wedge \ (A \Rightarrow B \wedge C) \ \wedge \ (B \Rightarrow D \ \text{XOR} \ E) \ \wedge \ (C \Rightarrow F \ \text{XOR} \ G) \ \wedge \\ \wedge \ (G \Leftrightarrow G') \ \wedge \ (G' \Rightarrow H \ \text{XOR} \ I) \ \wedge \ (\neg G' \Rightarrow \neg H \wedge \neg I) \ . \quad (6)$$

We proceed by restricting the formula to describe only configurations enabling transition t. If $pos(t) = \{s_1, ..., s_k\}$ and $neg(t) = \{s_{k+1}, ..., s_l\}$ then the restricted formula is defined to be:

$$\phi'(t) = \phi \ \wedge \ (\bigwedge_{i=1..k} s_i) \ \wedge \ (\bigwedge_{i=k+1..l} \neg s_i) \tag{7}$$

In our concrete example this translates to:

$$^{(fig2)}\phi'(t) =^{(fig2)} \phi \wedge D \tag{8}$$

Note that the set of solutions of $\phi'(t)$ is a subset of solutions of ϕ. Formula $\phi'(t)$ needs to be further restricted to eliminate variables other than ancestors of targets and the targets themselves. These states are the only candidates for exhibiting branch exclusion property of exclusion theorem. The remaining variables are irrelevant and as such should be eliminated to get rid of unnecessary noise in solution set. The restriction can be conveniently done by existential quantification over all unrelated variables:

$$\phi''(t) = (\exists \hat{s}_1, ..., \hat{s}_m).\ \phi'(t) \ , \tag{9}$$

where $\hat{s}_1, ..., \hat{s}_m$ are all the states in the model except for targets of t and their ancestors. This corresponds to

$$^{(fig2)}\phi''(t) = (\exists D, F, I).\ ^{(fig2)}\phi'(t) \tag{10}$$

in our example formula.

Due to guard propagation, the solutions of $\phi''(t)$ have a regular shape: each path down the hierarchy starts with some variables assigned true and switches permanently to false at certain point. See figure 4 for solutions concerning the model from figure 2.

A transition is trivially unreachable if satisfiable solutions for $\phi''(t)$ do not exist. Lack of solutions proves that guard condition is contradictory and transition may be safely discarded (perhaps issuing a warning for user).[2]

If there exists exactly one satisfiable solution to $\phi''(t)$ then, following the exclusion theorem t has static scopes and the scopes can be inferred from solution of formula. One needs to identify the branch exclusion (the switch-point from true to false on ancestors path) and use the exclusion theorem. Note that there is no need to extend guard conditions in this case. Existing guard is sufficient to guarantee desired properties of scopes.

A transition has a dynamic generalized scope if there are more assignments satisfying $\phi''(t)$. For each possible assignment of scopes there is a satisfiable assignment for variables. Each assignment may contain several branch exclusions

[2] Note that much better algorithms for detecting unreachable transitions are known within the model-checking community. See reachable state space analysis in [3, chapter 6] for a basic introduction. Removal of unreachable transitions should rather be considered as model correctness check than an optimization.

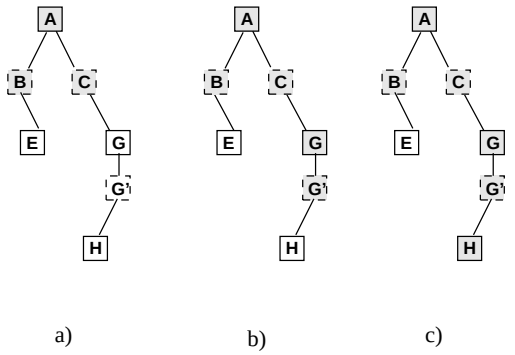

Fig. 4. Solutions of formula ${}^{(fig2)}\phi''(t)$. Shaded nodes represent active states (true assigned to the variable). White background nodes repersent inactive states (false assignment). Three solutions (a,b,c) correspond to scopes depicted in three parts of fig. 3. The same branch exclusion ($B \wedge \neg E$) over state E is present in all 3 solutions. This is because state change to E has static scope. However state H has two diffrent exclusions (and lack of exclusion in case c), which means that the scope is dynamic. It can take several values. In the last case (c) the scope is G' and the state change behaves like a self-loop (second case of exclusion theorem).

but at most one over each target. The branch exclusions may not be contradictory as they come from the same solution of $\phi''(t)$. For each satisfiable assignment we create a new transition t_j extending the guard with branch exclusions found in the assignment. Guards simplification may be used to ensure that no redundant checks are introduced. Minimality of guards for newly created transitions is not guaranteed by the algorithm itself.

Note that this transition cloning should not be expensive. In a typical model there will be very few (if any) transitions with dynamic scope, and only those transitions will be multiplied. At the same time much of the interpreter logic can be removed from runtime code.

Last but not the least it may happen that two or more targets happen to have a common scope in a given transition. In this case the scope should only be exited and entered once, obviously. Thus the transition targets should in general be classified in subsets tagged with common scope. If the scope is dynamic it may happen that actually the grouping will depend on the current configuration. This problem is also nicely solved by the above algorithm. Once the static scope has been inferred for each target, the targets may be grouped into proper categories. If the transition has been multiplied it is trivially guaranteed that each of the new transitions has a static division of targets into common scope groups. The division can only differ between newly introduced transitions.

Finally let us note that such transition multiplication is a semantics preserving transformation of the model. The proof of this claim should use the complete operational semantics of statecharts. The main observation is that new

transitions are active only whenever the old transition was active, their activity is mutually exclusive and whenever the original transition was active in some configuration there is a single new transition which is active in the same configuration. All this follows from construction algorithm relying on solutions to satisfiability problem. Targets and actions of new transitions are identical to the original transition. Consequently if the new transitions are substituted for the old one in the transition set then the model behavior does not change.

6 Implementation

The algorithm has been implemented as part of SCOPE – a statechart code generator being developed at IT University of Copenhagen. In our implementation we used binary decision diagrams [6] for finding satisfiable assignments of formula $\phi''(t)$. BDDs are well known as good platform for solving boolean equations, especially in model-checking community.

One disadvantage of BDDs is the possibility of space explosion while the formula is built. This problem has not been experienced in our evaluation of the algorithm. One of the reasons is that the problem we solve is much simpler than typical dynamic model-checking problems (fewer variables are involved). Additionally we propose several simple implementation improvements to decrease space requirements.

The most important one is to note that since we only consider one transition at a time we never need to build the BDD for the whole model. Each time we only need to build BDD for part of the model which is relevant to the single transition. It is sufficient to consider states that are ancestors of all targets and guard states along with targets and guards themselves. Moreover some states on top of the tree may be cut away as being always true (namely everything outside the component encompassing all parts of the transition in question).

Similarly there is normally no need to represent variables for or-states explicitly in the BDD. A smaller alternative formula may be proposed instead of ϕ to represent legal state configurations, which only concerns and-states.

Regardless of optimizations the final BDD representing $\phi''(t)$ will always be small since it only concerns less than $\rho = |\mathit{targets}(t)| \cdot n$ variables, where n is the model depth (basically only targets and their ancestors are included). This also ensures that enumerating all satisfiable solutions for $\phi''(t)$ is feasible space-wise.

Our implementation, including above improvements, takes currently about 2.5s to compile a model containing about 200 transitions (including the cost of other phases of SCOPE compiler, the static scopes algorithm is called once for each transition in the model).[3]

It shall be stressed that this algorithm integrates well with typical multipass compiler architecture. Being a semantics preserving, source-to-source transformation it can be composed with other model optimizations (like guard minimization, transition compaction, ect) in flexible ways.

[3] Pentium III, 1GHz. The implementation uses fast BDD package Buddy[12], more precisely its Standard ML incarnation – Muddy[7]

7 Related Work and Conclusion

The only hierarchical code generator for statecharts with multiple targets known to authors uses dynamic scope resolution at runtime (unpublished protype work of Behrmann, Iversen and Kristoffersen at Aalborg University, personal communication[2]). The problem solved here is particular to the visualSTATE version of statecharts. The primary reason, for which we consider it important, is that visualSTATE is one of major CASE tools used in commercial deployment of embedded systems and its code synthesis facilities comprise a significant part of its development system.

Classical statecharts avoid dynamic scope problems by disallowing targets in components orthogonal to transition itself. Other versions of statechart languages (Argos[13], SyncCharts[1]) avoid the problem by disallowing multiple targets and even cross-level transitions (flat scoping is trivial).

Multitarget transitions do not extend the expressive power of statecharts and thus are not strictly needed in modeling language. We do not argue that multiple targets make development methodology cleaner and easier for users. It is well known that overuse of force target actions leads to bad and unreadable designs. Having said that, we should emphasize that properly implemented force target actions yield much lower execution overhead than issuing and processing local events (typical UML way of passing messages between system components), which is specifically important in systems with hard time requirements.

Although multitarget transitions are visualSTATE specific, authors believe that, once they are efficiently implemented, they may play an important role in runtime representations for other statechart languages. It is a matter of future investigation if UML-like passing of local messages can be, in certain conditions, translated automatically to efficient multitarget transitions. This seems to be trivially possible if instantaneous signals semantics is assumed for statecharts as opposed to the signal queues proposed in UML (see [4, 1] for examples of synchronous languages employing instantaneous broadcasting).

Another common language idiom for classical statecharts, to workaround the lack of multitarget transitions, is manual multiplication of transitions, placing several transitions which will always fire at the same time. Especially with many targets this manual (and often habitual) multiplication may lead to nonoptimal solutions which could be improved by a compiler reducing the problem to single multitarget transition.

The dynamic scope resolution problem defines a class of models having insufficient information for optimization by compiler with only static knowledge about the statechart; namely models demanding multiplication and refinement of some transitions. We believe that in most cases of transitions with dynamic scoping the scope is actually static, because the number of legal configurations we consider is much overestimated. It has been already mentioned that state space reachability analysis may be used to overcome such problems. Alternatively a warning may be issued to the user, who could refine guard conditions to provide compiler with more information to increase efficiency. It is of interests to

identify more algorithms for detecting model deficiencies – in search for enough information to obtain optimizing and efficient code synthesis.

We believe that there are more dynamic properties of statecharts which may be analyzed and precomputed at compile time with only very little memory cost. The one presented here is only the first one to start with. More remain to be investigated. The algorithm presented increases our understanding of multitarget transitions and encourages further investigation.

References

[1] André, C. SyncCharts: A visual representation of reactive behaviors. Tech. Rep. TR 95-52, I3S, Sophia-Antipolis, France, October 1995.

[2] Behrmann, G., Kristoffersen, K., and G.Larsen, K. Code generation for hierarchical systems. In *NWPT'99 – The 11th Nordic Workshop on Programming Theory* (Uppsala, Sweden, Sept. 1999).

[3] Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., and McKenzie, P. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer-Verlag, Berlin-Heidelberg, 2001.

[4] Berry, G. The Esterel v5 language primer. version v5_91, JULY 2000.

[5] Björklund, D., Lilius, J., and Porres, I. Towards efficient code synthesis from statecharts. In *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML'2001* (Toronto,Canada, October 1st, 2001), A. Evans, R. France, and A. M. B. Rumpe, Eds., Lecture Notes in Informatics P-7, GI.

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

[7] Friis Larsen, K., and Lichtenberg, J. MuDDy 2.0 – SML interface to the binary decision diagrams package BuDDy. http://www.it-c.dk/research/muddy.

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

[9] Harel, D., Pnueli, A., J.P.Schmidt, and R.Sherman. On the formal semantics of statecharts. In *Proceedings of 2nd IEEE Symposium on Logic in Computer Science* (New York, 1988), IEEE Press, pp. 396–406.

[10] I-Logix Inc. Rhapsody® in MicroC. http://www.ilogix.com.

[11] IAR Inc. IAR visualSTATE®. http://www.iar.com/Products/VS/.

[12] Lind Nielsen, J. BuDDy – a binary decision diagram package version 2.0. http://www.it-c.dk/research/buddy.

[13] Maraninchi, F. The Argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE Workshop on Visual Languages* (Kobe, Japan, October 1991).

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

[15] Rational Software Corp. Rational Rose® Real Time (RoseRT). http://www.rational.com/products/rosert/.

[16] Wąsowski, A., and Sestoft, P. On the formal semantics of visualSTATE statecharts. Tech. Rep. TR-2002-19, IT University of Copenhagen, Sept. 2002.