On Efficient Program Synthesis from Statecharts

Andrzej Wasowski
Department of Innovation
IT University of Copenhagen
2400 Copenhagen NV, Denmark
wasowski@it-c.dk

ABSTRACT

Program synthesis from hierarchical state diagrams has for long been discussed in various communities. My aim is to provide an efficient, lightweight code generation scheme suitable for resource constrained microcontrollers.

I describe an initial implementation of SCOPE—a hierarchical code generator for a variant of the statechart language. I shall discuss several techniques implemented in the tool, namely imposing and exploiting a regular hierarchy structure, labeling schemes for fast ancestor queries, improvements in exiting states, compile-time scope resolution for transitions, and various details of compact runtime representation.

The resulting algorithm avoids the exponential code growth exhibited by tools based on flattening of hierarchical state machine. At the same time it demonstrates that it is possible to maintain reasonable speed and size results even for small models, while preserving the hierarchy at runtime. SCOPE currently produces code that is comparable with that of industrial tools (IAR visualSTATE) for small models and clearly wins for bigger ones.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

Program synthesis, automatic code generation, statecharts, embedded systems

1. INTRODUCTION

Statecharts[8], or hierarchical state diagrams, have proved to be a successful behavioral modeling formalism. The intuition behind their visual syntax increases productivity, while formal semantics supports development of tools for program synthesis and validation. Inclusion into UML standard has brought statecharts into the very center of software engineering technology. One of the emerging research problems is development of efficient automatic program synthesis techniques, as existing solutions are mostly focused on code readability and round trip engineering. The efficiency aspects are usually left behind. This is acceptable for production of workstation software, but still remains an issue for smaller target platforms such as resource constrained embedded systems, which quickly become ubiquitous due to tremendous success of mobile devices.

One approach to efficient program synthesis from statecharts uses flattening—a process of translation into a set of parallel Mealy machines. This results in simplification of runtime interpreter and allows its very efficient implementation. Unfortunately abandoning the hierarchy may cause exponential growth of the model[4], which leads to exponential growth of the program size. Thus the flattening technique seems to be useful only for smaller models. The code generation scheme presented here avoids the code explosion problem by preserving the hierarchy information at execution time.

Although important, linear code size is not the only demand. The complexity term should have a small constant factor, so that the gain is obtained already for small models, not only asymptotically. The time cost of single reaction should be minimized, which seems to be a difficulty, as maintenance of hierarchy introduces slow operations at runtime. Countermeasures should be taken to reduce number of ancestor queries, improve efficiency of single query and propose an internal runtime representation with small time overheads.

SCOPE[13] is a code generator, demonstrating several techniques for decreasing the practical disadvantages of hierarchical approach. This paper describes some of its more important implementation details.

2. OVERVIEW

We shall use a statechart of figure 2a to introduce some basic concepts. Each statechart consists of states organized in a hierarchy. Two states are in *parent-children* relation if one directly contains another. The sets of *descendants* and *ancestors* are defined by transitive closures of respectively children and parent relations. For example C is a child of A and H belongs to descendants of A. The outermost state A is called a *root* state. The innermost states (like D and H) are *basic* states. B, C and G are called or-states as only

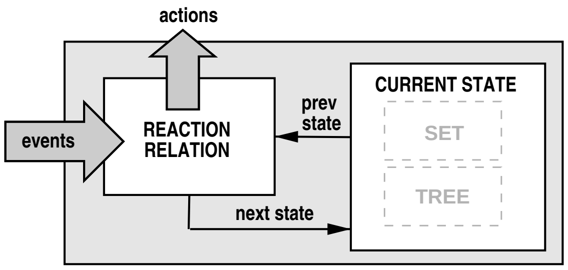

Figure 1: Structure of synthesized program.

one of their children can be active at a time, whereas A is an and-state, because both of its children (B and C) shall be active simultaneously.

Each transition has a single explicit source state and a single explicit target. Label of the transition contains the name of activating event, set of positive conditions (states demanded to be active, when transition fires), negative conditions (states demanded to be inactive) and an action. The action may contain host language function calls, expressions, triggering signals (local events) and additional targets to be activated.

The set of active states is called a *configuration*. Transitions specify how configuration is changed in reaction to discrete events. Black marks attached to states (*initial markers*) indicate members of the initial configuration of active states. Finally both states and transitions have *actions* assigned. These are executed when states are entered and exited or transitions fired.

The program build of statechart model has two fundamental components (figure 1): the current state configuration and the reaction relation. The configuration describes the set of active states. In hardware synthesis techniques it is usually realized using a feedback register. The reaction relation (sum of all transitions) defines how the configuration advances in response to external stimuli. It corresponds to combinational block of hardware implementations (see [6] for an example).

In a flattened representation the configuration component is implemented as a vector of activity indicators for all states. Operations performed on the configuration are limited to testing and setting values of those boolean flags. The reaction relation component is a list of rules containing guards, actions and target states. Both the configuration and the relation representation are rather simple. In the hierarchical case the configuration component contains both the hierarchy tree and the set of active states (gray on figure 1). It can be seen as a query answering engine, making decisions based on the content of the set and shape of the hierarchy tree. The operations performed are activity tests and exiting/entering a state.

SCOPE's hierarchy tree is represented in integer arrays. States are divided into two disjoint sets: or-states and and-states. This solution brings a multitude of small savings, starting from shortening integers used for identifying states ending up on elimination of runtime checks for state types. States are labeled in an order that allows fast ancestorship checks based on label comparisons. Moreover, whole parts of the hierarchy can be exited by performing set difference of current state set and the interval characterizing the subtree to be exited. Last but not least, an extension of labeling scheme is proposed, which exploits monotonicity of label sequences, to detect the end of children list and thus eliminate some markers from representation.

SCOPE supports a dialect of statecharts implemented by the industrial tool IAR visualSTATE[9]. Similarly to STATEMATE it allows transitions with targets concurrent to the source of the transition. Such transitions introduce a problem of dynamic scopes: one cannot decide at compile time what is the actual scope of the state change (what part of hierarchy should be affected). An algorithm for imposing static scopes has been designed and implemented. It ensures that all transitions have scopes detectable at compile time, thus removing this severe disadvantage of hierarchical representations.

The SCOPE code generator performs reasonably on small and simple models. Results are especially good for bigger models, when it clearly wins compared with the industrial implementation based on flattening. More work remains to be done, especially in the area of implementation of the reaction relation, which should improve the results further.

The structure of the paper is as follows. Section 3 describes technical details of SCOPE's implementation. Section 4 evaluates both theoretically and empirically solutions described in section 3. Section 5 comments on some examples of similar work. Finally section 6 concludes and indicates future work directions.

The semantics of statecharts language itself is not presented here. Should details be needed please refer to the original paper by Harel[8] and to our formal definition[16].

3. TECHNICAL DESCRIPTION

The runtime representation has been designed with modest space requirements and fast access in mind. The basic observation was that despite the multitude of attributes for states and transitions, developers hardly ever use all of them. Thus commonly used elements (initial markers, source states, targets) should be implemented cheaply, whereas it is acceptable to use more space and access time for exotic ones (multiple targets, complicated conditions, exit/entry actions, history).

For instance an initial marker is an example of commonly used element: it must be present once for each or-state. Initial markers are entirely eliminated from SCOPE's runtime representation in terms of space usage. Instead children lists are reordered, so initial states become lists' heads.

An average transition most likely will only have a simple condition (a source state and a discrete event), an action and a single target state. These fields, stored in the static part of transition record, are quickly accessible using fixed offsets. Multiple targets, a complex guard, and a transition scope are kept in the variable section of the record. Slower and more expensive field type indicators are used in this part, which is acceptable for rarely used elements.

The target language of SCOPE is C[10]. Despite the advances in optimization technology, C compilers face hard problems caused by the type system and highly imperative semantics of the language. For instance, an automatic code generator is rather likely to produce redundant identical pieces of code, including complete function bodies. The C compiler must maintain all identical pieces to guarantee correctness of pointer comparisons (if function pointers are used). To avoid the problem identical pieces of code should not be generated to begin with. A dynamic table

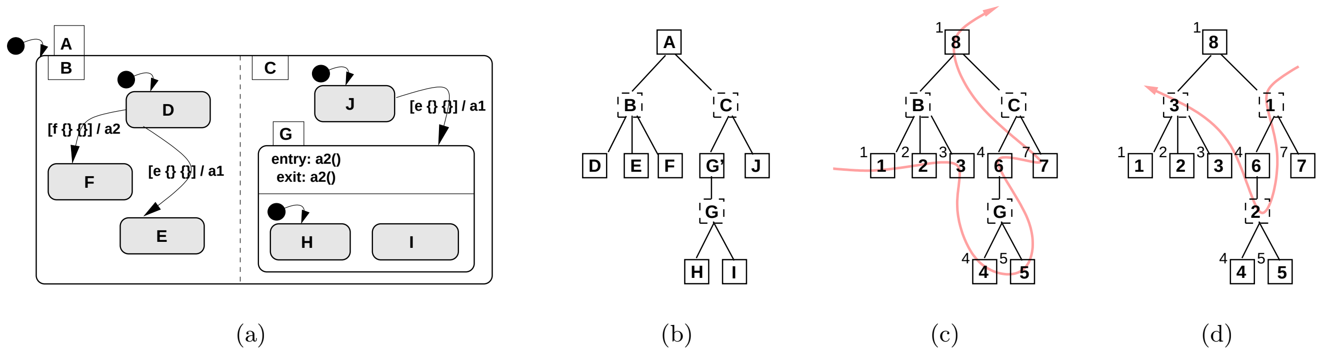

Figure 2: (a) A simple statechart (b) hierarchy tree (c) and-states labeled in DFS postorder, children visited from right to left. Superscripts indicate left edges of descendants interval. (d) Dual ordering of or-states.

of C code snippets is implemented, which only saves fragments not seen before. Program fragments are saved in this table, which is then used to build the actual C program. This uniqueness detection uses a trivial syntactic criterion (identity), sufficient for automatically generated code and reasonable for user written code (as we speak of short actions and expressions without local variables).

3.1 Hierarchy Tree

The hierarchy tree (see figure 2b) is the essential data structure of runtime representation. The language requires that children of and-states are always or-states, but or-states can have both types of children. This asymmetry increases modeling flexibility, but seems to be a burden at runtime. Regular alternation is introduced by inserting dummy single-childed and-states between two consecutive or-states (see G' on figure 2b). This permits to recognize state type by its position in the tree, saving both space (no runtime type information) and time (no dynamic type-checks). Additionally, separate name spaces for and-states and or-states can be used. State identifiers are reused and become shorter (space saving). The two parts of the tree can be saved in separate arrays. A simplified example is given in figure 3. All additional state attributes including parent information and entry/exit actions are omitted. The # marks used in th figure denote end-of-record markers.

In practice state addresses (array indexes) are used as state identifiers. If any of the two arrays becomes longer than integer type sufficient to represent the number of states of given type, state offsets double their size, which would immediately affect the size of identifiers and hence all arrays. To defer this undesirable effect an intermediate dictionary (an array of offsets) is created in such case and states are addressed by an extra indirection at runtime. It can be shown that the space cost of dictionary is always smaller than the saving on the array size.

Ancestorship queries are the most important operation on the hierarchy tree. They are performed whenever a state activity condition is evaluated, i.e. when selecting transitions to fire and when selecting routes of activation for cross-level transitions or transitions targeting non-basic states. IAR visualSTATE statecharts contain even more activity checks than UML state diagrams, since synchronization by states is much more natural for them than synchronization by signals.

The trivial implementation of hierarchical ancestorship check demands traversing the path between two states. Unfortunately the most expensive case, when ancestorship does not hold, seems to be the most common one. To diminish the problem a simple labeling scheme is proposed supporting checks based only on state labels.

Assume that and-states are numbered in depth-first-search order (postorder, with left-to-right visiting of children). Then the identifier assigned to given state s is greater than the identifier assigned to any of its descendants and all of them are greater than the identifier assigned to the leftmost descendant of s (see figure 2c). For each and-state an interval of ancestorship identifiers can be computed. Then the left end-point of the interval needs to be saved for each and-state. The right end-point of the interval is the state identifier, which is always known when reaching the state. As such it does not need to be saved separately.

The labeling can be exploited even further to eliminate some record markers from the structure. Recall that and-states and or-states have separate name spaces, which means that they can be labeled in different ways. Note that the interval labeling of and-states does not necessarily demand for state identifiers to be consecutive numbers. Recall also that state identifiers are used as state pointers, which means

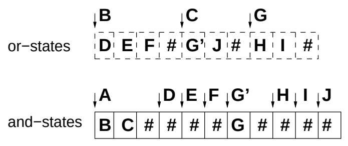

Figure 3: Hierarchy of fig.2b encoded in two arrays

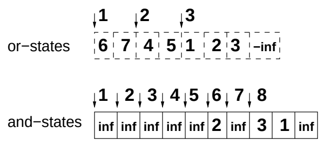

Figure 4: Array encoding of tree in figure 2d

that states are arranged in the same order in the arrays in which they are visited by labeling algorithm.

It can be shown that if or-states are labeled in order dual to the one presented for and-states (DFS, preorder, with right to left visiting of children) both arrays exhibit an interesting property. Children lists in the or-state array form strictly increasing sequences of values, while sequences of children on lists in the and-state array are strictly decreasing (see figure 2d). Moreover the monotonicity is always broken between lists of children of two neighboring states. This information can be used to distinguish record boundaries and remove end-of-state markers. If the state contains more attributes than children list (namely entry/exit actions) they are saved in front of the state record preceded by a marker: negative infinity for array of and-states and positive infinity for array of or-states. A guarding mark should also be appended in the end of the arrays and for basic states (having empty children lists).

Figure 4 shows runtime representation for the tree of figure 2d. Additional fields have been suppressed and consecutive numbers instead of actual offsets were used to increase readability. The saving is especially visible in or-states array, which by definition does not contain any leaves, and for deeper models with many internal nodes.

3.2 State Configuration Encoding

I have experimented with two simple state encoding techniques: set-based encoding and flag-based encoding—a variant of classical 1-hot state assignment adopted for software implementation of statecharts. The set encoding is more compact, while the flag encoding yields faster programs. The set encoding relies heavily on optimized ancestor queries described in section 3.1. Performance of the encodings is compared in section 4.

3.2.1 Set-based Encoding

Instead of the full vector of activity flags for all states, the set-based encoding maintains only a set of active basic states. Activity queries for non-basic states are answered using this set and the hierarchy tree.

Programming in constrained environment discourages use of dynamic memory management. This means that static buffers, rather than dynamic data structures, should be used to represent sets and queues. The set of active states is a simple buffer of elements with empty cells in the end. The lack of gaps between elements is important as we shall see that efficiency of state activity tests depends on the actual number of elements filled in. The configuration implementation does not prevent overflows. To guarantee safety, a bound on configuration size must be found statically by SCOPE.

The exact maximum size of the configuration can be computed using reachability analysis. Unfortunately this is expensive and may be impossible in practice for bigger models. A cheaper estimation is needed.

A simple recursive algorithm is used to give an upper bound of the configuration size. The bound for each basic and-state is assumed to be 1. The bound for each or-state is the maximum of bounds for its children. The bound for each and-state is the sum of children's bounds. Although very simple, this algorithm gives a good improvement over the trivial bound—the number of all basic states. The estimation is exact for purely sequential or entirely flat models.

Interval labeling and set-based encoding can also be used for optimizing some of the state exit operations. Observe that whenever a transition is fired all states within its scope should be exited and then new states should be activated. Standard exit procedure starts from active leaves of the respective subtree and proceeds towards the top executing all exit actions on the way. This can be improved for some states.

An and-state is said to be *exit-pure* if none of its descendants has any exit actions assigned. For such state another exit algorithm may be proposed. Instead of traversing the subtree and executing empty exit actions, one can scan the set of active states and simply delete all states between the end-points of the interval representing the subtree.

3.2.2 Flag-based Encoding

An alternative encoding for statechart configuration would preserve the information about all states (not only the basic states). The idea is to store identifier of active child for each or-state (or a distinct value if the state is inactive itself). This way activity checks become very efficient (and constant time) at the cost of updating the information for more states, whenever a transition fires. Also more writable memory is needed, which often is the most scarce resource.

Hardware implementations of similar encodings[6] use $\lceil \log n \rceil$ bits for each or-state, where n is the number of its children. Access to subparts of machine word is relatively inefficient, when it comes to software implementations. A vector of cells with fixed size can be used instead. The cell size should be sufficiently big to store the information for or-state node with highest out-degree.

3.3 Signal Queue

The local events queue is implemented in a ringbuffer. Unfortunately overflow-safety is not guaranteed as no estimation is given for signal queue size currently. The user is obliged to provide a safe value at code generation time. This can be obtained from the visualSTATE model-checker, which however may exhibit usual termination problems of reachability analysis.

3.4 Transitions

Transitions are stored in a simple hash table, with events being hash keys. Each event has a linear list of transitions assigned.

The essential improvement from basic implementation of the semantics is the static scope resolution algorithm. The problem of dynamic scope resolution stems from the presence of transitions that modify part of configuration concurrent to transition's source state. Such transitions usually have several possible scopes and the actual decision on which of them should be used (exited and entered) needs to be postponed until runtime, when the part of configuration in question is known. Compile-time scope resolution formulates and solves a boolean equation for possible scopes for each transition separately.

The boolean equation is formulated over logical variables representing activity of states. If formula ϕ_h describes the set of all statically valid state configurations and ϕ_t the guard condition of transition t, then the algorithm computes satisfiable assignments of the formula $\phi = \exists S.\ \phi_h \wedge \phi_t$, where S is the set of all states, which are not ancestors of target states of t. Each satisfiable assignment of ϕ represents

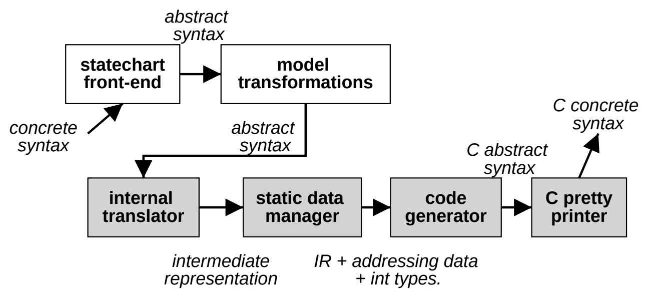

Figure 5: A structure of SCOPE implementation.

a group of state configurations in which t may be enabled. For each target of t consider a path in the hierarchy tree between *root* state and the target. For given satisfying assignment a few consecutive states in the beginning of the path (closest to *root*) are active and the remaining states are inactive—due to hierarchy properties. The last active state in the path (the most remote one from *root*), is the scope of the state shift to the target in all configurations represented by this assignment.

The algorithm multiplies the transition if several scopes are detected. Guards are refined assuring that each of new transitions has a different scope, all scopes are statically computable and transitions are mutually exclusive (no two of them can be active at the same time). To achieve that it suffices to add the scope of transition to the positive conditions and the next state on the path to the target (the first inactive state) to negative conditions. The implementation is based on binary decision diagrams. Precise description has been given in [15]

The transitions, which exhibit the problem, are extremely rare in real life models. This makes the replication feasible. Moreover only the source and target states are multiplied for the transition as the remaining components are shared between the copies. At the same time a significant and expensive computation can be shifted from runtime to compile time.

Another simple, but practically successful, optimization is the introduction of several target types called *modes*. Most importantly we distinguish flat and non-flat targets of transitions. Informally a target is *flat* if an arrow drawn to it from the transition source does not cross statechart levels (it remains within the same or-state). Nonflat targets need to be decorated by scopes as described previously. However, the computation of scope for flat targets is very cheap and can be done by looking up the parent of the source state in the hierarchy tree. Thus scope information does not need to be saved for majority of transitions, bringing yet another space saving.

3.5 Variables

According to the semantics of statecharts updates to the variables should only be visible after the step is completed. The assignment to x in a transition action should not affect any value of x in other expressions evaluated within the same step. This problem is classically solved using double-buffering of variables. The *lvalue* and the *rvalue* of a variable have to be separated.

The technique used is actually identical to maintaining two copies of state configuration: one for the current configuration and one for the next step's configuration. Classical solution maintains two runtime variables for each model variable. One copy (rvalue) is used for reads, the other (lvalue) for write accesses. After the step is completed the lvalues vector is copied over the rvalues vector. If the number of variables is big, the cost of this operation may be significant. Also the size of writable memory employed increases. For this reason developers of highly constraint systems avoid double buffering, following the modeling style which is not prone to such subtleties.

Currently SCOPE does not implement this technique, although the extension would be straightforward. All experiments with visualSTATE (see section 4) have been performed with double buffering switched off to account for this difference.

3.6 Tool Structure

Figure 5 presents the structure of SCOPE implementation. The tool accepts models in IAR visualSTATE file format. After initial stages, model transformations are applied (e.g.removal of dynamic scopes). The actual construction of runtime data structures is done in the *translator*, which saves them in internal bytecode format. The internal format refrains from using actual integer identifiers, concrete types and addresses. Then *static data manager* analyzes the output of translation to decide what are proper lengths for various attributes and whether dictionaries for state names are needed. Actual addresses are assigned at this stage and access macros for various fields generated as needed. Finally the actual code generator uses this information and the intermediate representation to produce the C program. The output should be compiled together with the static runtime library customized by model dependent constants and preprocessor macros.

4. EVALUATION

Let n be the number of all states in the model and t the number of transitions. The hierarchy tree and transition table can easily be implemented in respectively $O(n)$ and $O(t)$ space. The only point where the linearity can be broken is removal of dynamic scopes, which occasionally multiplies transitions. A transition can be multiplied at most $O(d^m)$ times, where d stands for depth of the tree and m is the maximum over number of targets on a single transition. Fortunately this term can be considered constant and is small in real models. The number of possible scopes is usually two or three. Also it is typical to have at most one dynamically scoped target on a transition (statically scoped targets do not cause any multiplication). Finally it is extremely uncommon to actually meet dynamically scoped transitions in real life models.

Linearity of hierarchical approach is a clear improvement over the flattening method. The practical challenge is to lower complexity constants sufficiently, so that use of the technique becomes feasible for relatively small models. This goal is already achieved by SCOPE.

The size of current state vector in the flattening model is proportional in the number of state machines (assuming that the outdegree of each node is small), whereas it is proportional only to the number of leaf state machines in set-based encoding. This is a constant factor improvement observable

Model	states	trans.	depth	**Executable Size [bytes]**						**Execution Time [s]**				
				VS	SC-SE	ratio	SC-FE	ratio	ram	VS	SC-SE	ratio	SC-FE	ratio
actions01	4	1	3	3 596	3 752	1.04	3 704	1.03	0	7.61	6.27	0.82	6.02	0.79
drusinsky89	19	14	7	3 976	4 192	1.05	4 144	1.04	4	9.64	7.67	0.80	7.99	0.83
lift	18	19	3	4 452	4 432	1.00	4 372	0.98	0	15.66	30.33	1.94	21.30	1.36
peer	275	192	23	12 644	10 352	0.82	10 536	0.83	56	20.66	31.81	1.54	26.31	1.27
trios01	1121	840	9	28 164	19 848	0.70	24 108	0.86	271	541	730	1.35	255	0.47
trios03	1121	840	9	60 196	22 048	0.37	24 684	0.41	271	1139	751	0.66	260	0.23

Table 1: Speed and size results: IAR visualSTATE 4.3 vs SCOPE 0.11

in practical applications. The flag-based encoding remains as costly as flattening method.

Similarly, elimination of end-of-state markers brings a constant saving of space in representation of the hierarchy. So does elimination of field indicators for commonly used elements (for instance initial states).

Single state activity test in the flattening approach uses a constant time. With the set-based encoding and traversal of the hierarchy tree the worst case for activity test is $O(dn)$. SCOPE reduces this to theoretical complexity of $O(n)$ using descendants interval labeling. In practice n becomes the number of active basic states (much less than number of basic states). Despite the improvements this is still worse than constant time of flattening approach. One should remember that latter introduces potentially exponentially more tests, which neutralizes the difference. Flag-based encoding enjoys the best of the two—the number of tests is not increased and each test takes the constant time.

Updates on the current state are more expensive in the hierarchical case (at least $O(nd)$ depending on the operation), while they are $O(d)$ for the flat case. There are two reasons why the difference is smaller in practice. First, most of transitions are fired on the level of leaves ($d = 1$), second—(in set-based encoding) the cost is lowered for many of subtrees using pure exits, which perform several updates simultaneously using the same linear time $O(n)$.

Experiments have been carried on both with SCOPE and IAR visualSTATE. Generated programs have been compiled with GCC 3.2 (optimization for size) on x86 PC running Linux. Sizes are bare executables in bytes. Only the control algorithm and the runtime library were linked in. All references to external functions have been substituted with dummies. Running times are given in seconds, measured by triggering 10^7 random events, reinitializing the state machine before each event with probability of 0.01. The measuring was performed on a 450 MHz Pentium II.

Table 1 presents selected results. The *states* column contains the total number of states (both and-states and or-states), *trans* shows the total number of transitions (excluding initial transitions), while *depth* gives the depth of hierarchy tree (counting both and-states and or-states). The minimal depth is 3, which is observed for flat models. *VS* denotes visualSTATE, *SC-SE* denotes SCOPE in state-based encoding mode, *SC-FE* denotes SCOPE using the flag-based encoding. Ratios are computed with respect to the visualSTATE measurements. The *ram* column presents the size of writable memory consumed additionally by the flag-based encoding comparing to set-based encoding.

Similar experiments have been carried on using a nonoptimizing compiler (LCC for Linux) and optimizing embedded systems compilers from IAR Systems. The results were comparable; the only part of the program, that can be optimized by C compiler is the runtime library. IAR compilers for PIC and AVR platforms shown that the flat and hierarchical runtime libraries differ about 5% in size. Static integer tables encoding model data are beyond the scope of ordinary compiler optimizations and thus they remain the same from platform to platform. I report GCC results since this is the most widely accessible reference platform.

Actions01 is a trivial example containing two basic states connected by a single transition. The size differences reflect the sizes of runtime engines. The hierarchical library seems to be only slightly bigger. The version of the library using flag-based encoding is smaller than the one for set-based encoding as the logics involved is much simpler.

Drusinsky89 [5] and *lift* show that the size of code produced by SCOPE is comparable to that of IAR visualSTATE for small models. The difference seems to be acceptable. The latter of the two, *lift*, is a flat statechart (a set of concurrent state machines). It demonstrates the performance strength of flattening approach on flat models, or rather conversely, its relative weakness on hierarchical models for which costly expansions are performed. The interpreter for the flat structure is very efficient. What slows it down is the growth of the structure itself (not observed for flat models).

A typical medium size model with irregular structure is represented by *peer*. The last two models, *trios01* and *trios03*, are highly concurrent and uniformly deep (the whole structure is equally deep). The latter one uses deep history on top level. Such models exhibit the size explosion problem of the flattening approach.

The overall result is that hierarchical generation technique seems to be feasible for small models, and scales extremely well to large ones. Also if the cost incurred on writable memory is acceptable in given application, the flag-based encoding should be used as this brings efficiency gains over the set-based approach.

5. RELATED WORK

There is a multitude of statechart translators available. Most of them take hierarchical code generation approach, without performing more significant optimizations [17, 14, 11]. The tree is encoded in a nested switch statement or in a class hierarchy with virtual methods (the latter is referred to as *state pattern*, see [1] for a generalized version). However much less care is taken to make the implementation efficient. The focus is more on code readability an use of natural constructs of target language than efficiency. The usefulness of such tools for constrained embedded systems is not usually considered.

Erpenbach[7] in his thesis focuses mostly on the worst case reaction time analysis, proposing only a very simple hierarchical representation based on switch statements. He also addresses the double-buffering problem for variables, proposing an optimization which reorders the transitions to decrease the need for double buffering—so that assignments happen after read accesses, if possible.

Björklund, Lilius and Porres[3] devise an intermediate language, that should be compilable efficiently. Nevertheless, the use of flattening in course of translation indicates possible exponential growth of code.

I have chosen the hierarchical approach, inspired mostly by implementation of Behrmann and others [2]. However I considered the idea of guard value memoization (to speed up condition evaluation) as unsuitable for constrained systems, where writable memory is a scarce resource.

Drusinsky[6] and Ramesh[12] discuss the state encoding problem from hardware implementation perspective. Both encodings proposed are more compact than those presented here. A hardware implementation can very efficiently extract single bits and groups of bits from the state register, whereas this seems to be expensive in software. Apart from this difference Drusinsky's encoding is very much alike to the flag-based encoding of SCOPE. Both papers are very brief on explaining the structure of combinational block which implements the reaction relation. The applicability of these results for software synthesis should still be investigated.

My algorithm approximating cardinality of the active basic states set can be seen as a simpler version of Drusinsky's algorithm for finding maximum-cardinality exclusivity set presented in [6].

6. CONCLUSION AND FUTURE WORK

I have presented implementation of SCOPE—a tool for synthesis of programs from statechart specifications. This seems to be the first non flattening translator oriented at efficiency and constrained embedded systems. Only few language elements are not supported yet. Most importantly timer events and do reactions. All of them seem to be straightforward extensions, which will not break the general approach. It should be emphasized that in contrary to other approaches I have not experienced any problems with incorporating history and deep history fields.

A number of solutions have been discussed and their influence on efficiency evaluated both theoretically and empirically. These have been divided into two categories: those keeping the complexity at linear level and those increasing the efficiency of operations. The resulting tool seems to fulfill its initial goal. It performs well on big examples, while retaining feasibility for smaller models. In addition it has been established that the flag-based encoding yields more efficient programs, which however use more writable memory than the set-based encoding.

According to my knowledge this is the first ever attempt to evaluate a code generation technique for statecharts quantitatively, instead of just describing the approach. This contrasts with traditionally well evaluated results for hardware synthesis.

There are still improvements to be made. A shift from a simple hash table of transition lists to a more advanced structure avoiding some of the repetitive condition tests should bring further speed gains. In future I would like to investigate, how more adaptivity could be incorporated into the compilation algorithm, which would allow it to meet hard size constraints set by developers.

7. ACKNOWLEDGMENTS

Henrik Reif Andersen introduced the flattening approach to me. Kaare Kristoffersen disclosed details of implementation only briefly described in [2]. I would also like to thank Peter Sestoft for supervision and Christian Worm Mortensen for patient and fruitful discussions on data structures used in SCOPE. IAR Systems provided me with visualSTATE tool and the compilers for embedded platforms. Last but not least—all three anonymous reviewers made valuable suggestions of improvements.

8. REFERENCES

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

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

[3] 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. Workshop of the pUML-Group.*, Lecture Notes in Informatics P-7, Toronto,Canada, October 1st, 2001. GI.

[4] D. Drusinsky and D. Harel. On the power of cooperative concurrency. In *Proceedings of Concurrency '88*, volume 335 of *Lecture Notes in Computer Science*, pages 74–103, New York, 1985. Springer-Verlag.

[5] D. Drusinsky and D. Harel. Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer Aided Design*, 8(7):798–807, 1989.

[6] D. Drusinsky-Yoresh. A state assignment procedure for single-block implementation of state charts. *IEEE Transactions on Computer-Aided Design*, 10(12):1569–1576, 1991.

[7] E. Erpenbach. *Compilation, Worst-Case Execution Times and Schedulability Analysis of Statecharts Models*. PhD thesis, Department of Mathematics and Computer Science of the University of Paderborn, Apr. 2000.

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

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

[10] International standard. Programming Languages - C. Ref. ISO/IEC 9899:1999(E).

[11] A. Knapp and S. Merz. Model checking and code generation for UML state machines and collaborations. In D. Haneberg, G. Schellhorn, and W. Reif, editors, *Proceedings of 5th Workshop on*

Tools for System Design and Verification, Technical Report 2002-11, pages 59–64. Institut für Informatik, Universität Augsburg, 2002.

[12] S. Ramesh. Efficient translation of statecharts into hardware circuits. In *12th International Conference on VLSI Design*, pages 384–389. IEEE Computer Society Press, Jan. 1999.

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

[14] E. Sekerinski and R. Zurob. iState: A statechart translator. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language, Toronto, Canada, October 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 376–390. Springer-Verlag, 2001.

[15] A. Wąsowski and P. Sestoft. Compile-time scope resolution for statecharts transitions. In *Proceedings of Workshop on Critical Systems Development with UML (CSDUML)*, Dresden, Germany, Sept. 2002. TR of Munich University of Technology.

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

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