Dynamic Slicing Research of UML Statechart Specifications

Chunyu Miao
College of XingZhi, Zhejiang Normal University Jinhua, China
E-mail: netmcy @zjnu.cn

***Abstract*—This paper extends the well-known technique of dynamic slicing to Statechart specifications of reactive systems. Statechart language extends state machines along hierarchy, concurrency and communication – resulting in a compact visual notation that allows engineers to structure and modularize system descriptions. Dynamic slicing is well known in the domain of sequential transformational programs and has been found to be useful in understanding, analysis and verification. The classical definition of dynamic slicing is unsuitable for Statechart specifications. In this paper, we firstly formally define a formal semantics model -- observable semantics, which is very suitable for dynamic slicing, because that it only describes outside observable behavior and conceals unobservable behavior of Statechart specifications, and it fully captures the run-time dependence relation among the state transitions in the Statechart specification. Then we propose a new notion of dynamic slicing that, in our opinion, is more natural for Statechart specifications. We formally define notions of dynamic slicing criterion, dynamic slice and minimal dynamic slice, and we also explain how to produce valid dynamic slicing criterion and propose a simple and practical approximation algorithm for minimal dynamic slice generation using observable semantics as an intermediate representation.**

***Index Terms*— Statechart Specification; Slicing; Slice Criterion; Specification analysis; Specification Verification; State Explosion Problem; Formal Semantics; Reactive Systems**

I. INTRODUCTION

Statechart language was introduced by David Harel in 1987 [1] as a visual formalism for complex reactive systems: event-driven systems which continuously react to external stimuli. Examples of such systems include communication networks, operating systems, and embedded controllers for telephony, automobiles, trains and avionics systems. The primary motivation behind Harel Statechart was to overcome the limitations inherent in conventional state machines in describing complex systems – proliferation of states with system size and lack of means of structuring the descriptions. Harel Statechart extends state machines along three orthogonal dimensions – hierarchy, concurrency and communication – resulting in a compact visual notation that allows engineers to structure and modularize system descriptions.

Normally the application domain requires very high quality reactive systems, so an important phase in the development of reactive systems is the understanding, verification and analysis of their specification before the implementation. However, the size of the obtained system specification greatly increases with the system becoming larger and more complex and leads to state explosion problem, what makes industrial system specification difficult to be understood, verified and analyzed.

To decompose a large system specification manually is very costly, laborious and error-prone. Then automation of decomposition seems a logical solution. Automation may help in making the decomposition process faster, in making it less susceptible to human error by automating routine or error-prone tasks, and in making it more reproducible by making it less dependent on human interpretation.

Slicing is a functional decomposition technique that has been successfully used in verification and analysis of sequential software: to analyze a large program, slice the program into functionally independent units that can be analyzed with relative ease [2]. Various slightly different notions of sequential program slices have since been proposed, as well as a number of methods to compute slices, such as backward slicing, forward slicing, static slicing, dynamic slicing and so on. The main reason for this diversity is the fact that different applications.

Although we can find the original idea of slicing in the context of sequential programs [3,4], the theory of slicing is independent of the ways and languages of programming. It can be used for concurrent programs, distributed programs and also for system specifications written in abstract languages such as process algebra, Statechart and Z[5,6].This paper describes an attempt to extend the technique of slicing to Statechart specifications, i.e., to automatically decompose a large Statechart specification into functionally independent subspecifications that can be understood, verified and analyzed with relative ease.

Because that Statechart language is an abstract, non-sequential, visual specification language with structuring mechanism of hierarchy and concurrency, we should keep following differences between Statechart specification and sequential programming languages in mind when we study the notion and method of its slicing.

1. The formal semantics of sequential programs are very simple compared to the formal semantics of

doi:10.4304/jcp.6.4.792-798

Statechart specifications, so we need to devote large parts of our energies to formal semantics definition of Statechart specifications.

2. The predominant notation of sequential programs are variables, whereas, states, transitions and interaction are more predominant in Statechart specifications.

3. The behaviour of a Statechart specification is not to produce desired result at the end of execution but to maintain certain ongoing relationship with its environment. More precisely, the behavior of a Statechart specification is a set of I/O sequences.

From above description we can see that standard notion of slicing defined for sequential programs is not very suitable for Statechart specifications. So we need to define a new notion of slicing that, in our opinion, is more natural for Statechart specifications.

II. STATECHART SPECIFICATION

Over the years, a number of variants of Harel Statechart have been proposed -- MODECHARTS, RSML, OMT, ROOM and UML, to name a few. They retain the Harel Statechart structuring mechanism of hierarchy and group transitions, but differ significantly in semantic aspects, which make them more powerful for specific applications. In this section, we propose our Statechart notation, which keeps the main aspects of most commonly used Statechart notations related to concurrency and state hierarchy, inter-level transitions, transition priority, broadcast communication.We give such a Statechart specification in Figure1.

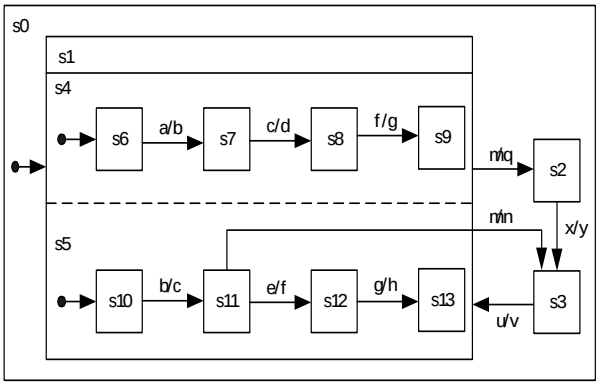


Figure1. A Statechart specification example

A formal syntax definition of a Statechart specification is given as follows.

Definition 1. Syntax: A Statechart specification is a 8-tuple **SC**=(S, LI, LO, s0, T, D, g, h) where S is a finite set including all possible Statechart states; LI is an inside inputs/outputs set; LO is an environmental inputs/outputs set; s0 is the root Statechart state, $s0 \in S$; $T \subseteq S \times L \times L \times S$ ($L=LI \cup LO$) is a finite set including all Statechart transitions; D is a set including all default active substate of all OR states; g: S→2^S is a simple decomposite function; h: S→2^S is a parallel decomposite function; g and h are defined as follows:

$$\forall s \in S,\ g(s) = \begin{cases} \{s\}, & \text{if s is not a OR state} \\ \{\text{all substates of s}\}, & \text{if s is a OR state} \end{cases}$$

$$\forall s \in S,\ h(s) = \begin{cases} \{s\}, & \text{if s is not a AND state} \\ \{\text{all substates of s}\}, & \text{if s is a AND state} \end{cases}$$

III. OBSERVABLE SEMANTIC

For the convenience of discussion of formal semantics of Statechart specifications, we introduce name rewrite technique and conflict resolution mechanism. The rewrite rule is: The rewrite name of a state must include all its ancestors' original name before its original name. for example, the rewrite names of s0 and s7 are s0 and s0s1s4s7.

A. Observable Semantics

One of the most important problems of automatic slicing Statechart specifications is their semantics. In itself, the semantics of a Statechart specification describes its related behavior subset and consists of some semantics states and their transition relation.

Broadly speaking, a semantics state, also called global state, consists of several Statechart states those can be active simultaneously. We denote with all the semantics states of a Statechart specification.For the convenience of slicing algorithm design and slice notion definition, here, we formally define a semantics model -- observable semantics, which is very suitable for slicing, because that it only describes outside observable behavior and conceals unobservable behavior of Statechart specifications, and it fully captures the run-time dependence relation among the state transitions in the Statechart specification.

Definition2. Observable semantics: The observable semantics **OS** for a Statechart specification (S, LI, LO, s0, T, D, g, h) is an input/output labeled transition system (Σ, LO, σ, M-step) where:

- $\Sigma \subseteq \Lambda$ is the state space of the observable semantics. Because **OS** only describes outside observable behavior, not all semantics states are valid in observable semantics, only those can react to environmental inputs consist the state space of observable semantics, which called observable semantics states.
- σ is the initial observable semantics state, $\sigma \in \Sigma$.
- LO is environmental inputs and outputs set.
- M-step $\subseteq \Sigma \times$ LO $\times$ 2^T $\times$ 2^LO $\times \Sigma$ is a set including all possible observable semantics state transitions, also called M-step transitions.

B. Observable Semantics Generation

According definition 2, the most important thing for observable semantics generation is to generate all M-step transitions. The following algorithm is the M-step transition generation algorithm. In this algorithm we use set C' to include all simultaneously active Statechart states, set I to include all inside outputs, set O to include all environmental outputs, set

Substate(p) to denote all directly or transitively nested substates of Statechart state p (Substate(p) can be easily computed using functions g and h of given Statechart specification), and set U to record all executed Statechart transitions.

Algorithm 1 (M-step-generation)

Input to the algorithm is the Statechart specification **SC**=(S, LI, LO, s0, T, D, g, h), an observable semantics state C containing the current simultaneously active Statechart states and the environmental input i C can react to.

Output of the algorithm is a transitions <C, i, U, O, C'>.

1. Initialize:
   a. I<= Ø, O<= Ø, C'<= C, U<= Ø.
2. Select a maximal set of conflict free enabled transitions:
   a. D={<p,a,b,q> | <p,a,b,q>∈T, p∈C' ∨ Substate (p)∩C'≠Ø, and a=i };
   b. Every t1, t2∈D
      a) If t1 and t2 are in conflict and Priority(t1) > Priority(t2);
      b) Remove t2 from D.
   c. U<= U∪D
3. Execute all Statechart transitions in D in an unspecified order:
   a .For every Statechart transition in D:
      a) If <p,a,b,q> is a same-level transition
         i. Use function g and h to remove p and Substate(p) from C', then C'<= C' ∪ {q};
         ii. If b is inside output, I<=I∪{b}, else O<=O∪{b}.
      b) If <p,a,b,q> is a inter-level transition, the rewrite names of p and q is as shown in Figure2.
         i. Compute the right-most same element, denoted by r, in the rewrite names of p and q (there must be such a r, at least r=s0). pi+1 is the element next to r for p and qj+1 is the element next to r for q;
         ii. Use function g and h remove pi+1 and Substate(pi+1) (including p) from C';
         iii. Put all concurrent states with Statechart states q0…qj+1, q0…qj+2, …, q0…qm-1 into C';
         iv. C'<= C'∪{q};
         v. If b is inside output, I<=I ∪ {b}, else O<=O ∪{b}.
4. Select next maximal set of conflict free enabled transitions:
   a. Do as follows until C' has no composite Statechart state
      a) For every composite Statechart state in C'
         i. If they are a AND state, use set D, function g and h to substitute it by its substates;
         ii. If it is an OR state, use set D, function g and h to substitute it by its default active substate.
   b. D={<p,a,b,q> | <p,a,b,q>∈T, p∈C' ∨ Substate (p)∩C'≠ Ø, and a∈I };
   c. Every t1, t2∈D
      a) If t1 and t2 are in conflict and Priority(t1) > Priority(t2);
      b) Remove t2 from D.
   d. I<= Ø;
   e. If D= Ø, return <C, i, U, O, C'>, else goto 3.

p0=s0, p1, p2, …, pi=r, pi+1, pn-1, pn
q0=s0, q1, q2, …, qj=r, qj+1, qm-1, qm

Figure2 name pattern for p and q

Using above M-step-generation algorithm, we propose an observable semantics generation algorithm. This algorithm uses set B to record all the generated M-step transition and set C to include all simultaneously active Statechart states.

Algorithm 2 (OS-generation)

Input to the algorithm is the Statechart specification **SC**=(S, LI, LO, s0, T, D, g, h).

Output of the algorithm is an observable semantics **OS**=(Σ, LO, σ, M-step).

1. Initialize:
   a. C<={s0}, □<=Ø, M-step<=Ø, B<=Ø, Empty(stack).
2. Obtain initial observable global state:
   a. Do as follows until C has no composite Statechart state
      a) For every Statechart state in C
         i. If they are a AND state, use set D, function g and h to substitute it by its substates;
         ii. If it is an OR state, use set D, function g and h to substitute it by its default active substate.
   b. σ<=C; Σ<=Σ∪{C}.
3. Find all possible M-step transitions:
   a. EO={i | <p,i,*,q> ∈ T, p ∈ C ∨ Substate (p)∩C≠Ø, and i∈LO };
   b. For every i in EO
      a) If <C,i>!∈B, then push <C,i>.
4. Generate M-step transitions and state space:
   a. If stack is empty, return (□, LO, σ, M-step).
   b. Else if stack is not empty
      a) Pop <C,i>, B<= B∪{<C,i>};
      b) Use (S, LI, LO, s0, T, D, g, h), C and i to call M-step-generation algorithm, obtain <C, i, U, O, C'>;
      c) M-step= M-step∪{<C, i, U, O, C'>}; □<=□ ∪{C'};
      d) C<=C', goto 3.

IV. DYNAMIC SLICING OF STATECHART SPECIFICATION

A. Dynamic Slicing Criterion

The choice of the slicing criterion was motivated by considering situations where taking a Statechart specification slice would be useful. Statechart

specification slicing is typically used for complex Statechart specifications understanding, analysis and verification.

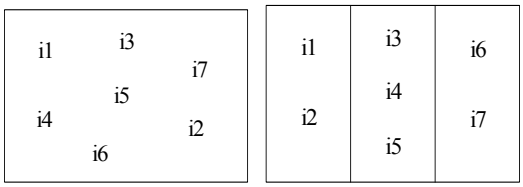
Normally, a complex reactive system needs to continuously react to all kinds of environmental inputs in its whole operational life. These environmental inputs are from different kinds of system users (users represent someone or something that needs to exchange information with the system; but they are not part of the system). For every possible environmental input sequence the system generates specific corresponding output sequence. Coupling all these input sequences and corresponding system reactive behavior together results in a very large complex Statechart specification and make Statechart specifications understanding, analysis and verification difficultly if not impossible

In order to large complex Statechart specifications be understood, verified and analyzed with relative ease, we try to slice them into several functionally independent subspecifications. If we do not make assumptions regarding the inputs to the system and just adopt static information such as some state, some input or output as slicing criterion, we will obtain static slices. Normally, the highly conservative nature of static slicing leads to highly imprecise and considerably larger slices and the usefulness of a slice in understanding, analysis and verification tends to diminish. Since the main purpose of slicing is to identify the subset of a Statechart specification that are of interest for a given application, conservatively computed large slices are clearly undesirable. Recognizing the need for accurate slicing, in this paper we propose the idea of dynamic slicing: a slice computed for a particular fixed input. The availability of run-time information, finite environmental input sequences, makes dynamic slices smaller than static slices.

Definition 3. Dynamic Slicing Criterion: A dynamic slicing criterion for a Statechart specification **SC**=(S, LI, LO, s0, T, D, g, h) is a finite environmental input sequence α, $\alpha \in I^*$, $I \subseteq LO$ is the set of environmental inputs that can be fed to **SC**.

Usually, the requirement analyzers have inexplicitly or explicitly described all possible environmental inputs that can be fed to **SC** in the requirement specification. We need to extract from the requirement specification these environmental inputs. If the requirement specification is written formally, we can sketch out some generation algorithm to generate these environmental inputs automatically, and if the requirement specification is written informally, the slicers have to derive these environmental inputs manually. For example, we have a reactive system, which has seven possible environmental inputs {i1, i2, i3, i4, i5, i6, i7} as shown in Figure3(a). Normally, we can find that all obtained environmental inputs are related, and there are two types of temporal relationships between every two environmental inputs: they are either concurrent or sequential. Sequential environmental inputs are called predecessors and successors, and the predecessors happen before the successors. Concurrent environmental inputs can happen simultaneously. For example, the temporal relationships among {i1, i2, i3, i4, i5, i6, i7} is shown in Figure3(b). The environmental inputs in {i1, i2}, {i3, i4, i5}, and {i6, i7} are concurrent; every environmental input in {i3, i4} is the predecessor of every environmental input in {i6, i7} and the successor of every environmental input in {i1, i2}.

In the view of slicers, only those environmental input sequences that satisfy sequential relationship are valid dynamic slicing criterions, such as <i1i3>, <i1i2i6i7> and <i3i3i7> for the example reactive system.



(a) (b)

Figure3 (a) the possible environmental inputs of example reactive system; (b) the temporal relationships among these environmental inputs

B. Dynamic Slice

To define a dynamic slice formally, we define an output function as follows:

Definition 4. Output function: Given a Statechart specification **SC**=(S, LI, LO, s0, T, D, g, h) with observable semantics **OS**=(Σ, LO, σ, M-step), $\alpha \in I^*$, α= i1i2…in, Output(**SC**, α) =On, if there are <σ, i1, O1, p1>, <p1, i2, O2, p2>, …<pn-1, in, On, pn>∈M-step;= Ø, otherwise

Definition 5. Dynamic Slice: Given a Statechart specification **SC**=(S, LI, LO, s0, T, D, g, h) with respect to the dynamic slicing criterion α, the corresponding dynamic slice is also a Statechart specification **DS=(**S', LI', LO', s0, T', D', g', h') such that:

a) **DS** is obtained by removing zero or more transitions (and involved states) from **SC**.
b) **DS** is canonical and has similar hierarchy relationship information and default active state information to SC.
c) For every prefix of α, β, Output(**SC**, β) = Output(**DS**, β), i.e., outside observers can not distinguish **SC** and **DS** by the response of them to the α.

It is obvious that for a Statechart specification and a corresponding dynamic slicing criterion α, there are many subspecifications can be looked as qualified dynamic slices. For example, the original Statechart specification is always a qualified dynamic slice, and the obtained subspecifications by removing some obviously useless Statechart transitions are also qualified dynamic slices.

Since the main purpose of slicing is to identify the subset of a Statechart specification that are of interest for a given application and the usefulness of a slice in understanding, analysis and verification tends to

diminish with the increase of the size of a slice, conservatively computed large slices are clearly undesirable. Recognizing the need for accurate slicing, we need more accurate dynamic slice definition. So we define the notion of minimal dynamic slice.

Definition 6. Minimal Dynamic Slice: Given a Statechart specification **SC**=(S, LI, LO, s0, T, D, g, h) with respect to the dynamic slicing criterion α, the corresponding minimal dynamic slice is also a Statechart specification **MinDS=(**S', LI', LO', s0, T', D', g', h') such that:

1. **MinDS** is also a dynamic slice.
2. Removing any Statechart transition from **MinDS** can not result a correct dynamic slice.

C. *Dynamic Slice Generation*

The process of dynamic slice generation has following two steps:

1. Generate observable semantics **OS**=(Σ, LO, σ, M-step) from the given Statechart specification **SC**=(S, LI, LO, s0, T, D, g, h) according to Definition 2.
2. Generate dynamic slice from the generated **OS** and the given dynamic slice criterion α using DS-generation algorithm.

The basic ideas underlying this algorithm are as follows:

a) Construct set R including all related Statechart transition using **OS** and α.
b) Generate the state space and input output set of the dynamic slice using R.
c) Produce a complete and consistent dynamic slice using the original Statechart specification definition.

In this algorithm we use set R to include all related Statechart transitions, set U to include all related Statechart states, set V to include all related inputs and outputs and α(k) to denote the kth element of α.

Algorithm 3 (DS-generation)

Input to the algorithm is a Statechart specification **SC**=(S, LI, LO, s0, T, D, g, h), the generated observable semantics **OS**=(Σ, LO, σ, M-step) and a dynamic slicing criterion α.

Output of the algorithm is a dynamic slice **DS**=(S', LI', LO', s0, T', D', g', h').

1. Construct a set R including all related Statechart transition using **OS** and α:
   a. s<=σ, R<=Ø.
   b. For k=1 to k=|α|, do
      a) If there is a transition <p, i, H, O, q>∈step, where p=s and i=α(k), R <= R∪H
      b) s<=q
2. Generate state set U and input output set V:
   a. U<=Ø, V<=Ø
   b. For every <p, a, b, q>∈R
      a) U<=U∪{p}∪{q}
      b) V<=V∪{a}∪{b}
3. Complement state set U according the underlying hierarchy relationship information
   a. Do as follows until U become stable:
      a) For every state s∈U
      i. If SC.g(s')=s and s'!∈U, then U<=U∪{s'}.
      ii. If SC.h(s')=s and s'!∈U, then U<=U∪{s'}.
4. Produce a complete and consistent dynamic slice DS=(S', LI', LO', s0, T', D', g', h')
   a. DS.T<= R
   b. DS.S<=U
   c. DS.LI <= SC.LI∩V
   d. DS.LO <= SC.LO∩V
   e. DS.D<=SC.D∩DS.S
   f. DS.s0 <= SC.s0
   g. For every s∈DS.S, DS.h(s)=SC.h(s)∩DS.S
   h. For every s∈DS.S, DS.g(s)=SC.g(s)∩DS.S

D. *Algorithm Analysis*

First, as an example, let's consider the Statechart specification shown in Fig.1. The three Statechart specifications shown in Fig.6 is the dynamic slices of the former Statechart specification with respect to dynamic slicing criterions α=x, α=a and α=axmuxu. Clearly, the latter three Statechart specifications have the same behaviour as the original one in Figure1 as far environmental input sequence α is concerned.

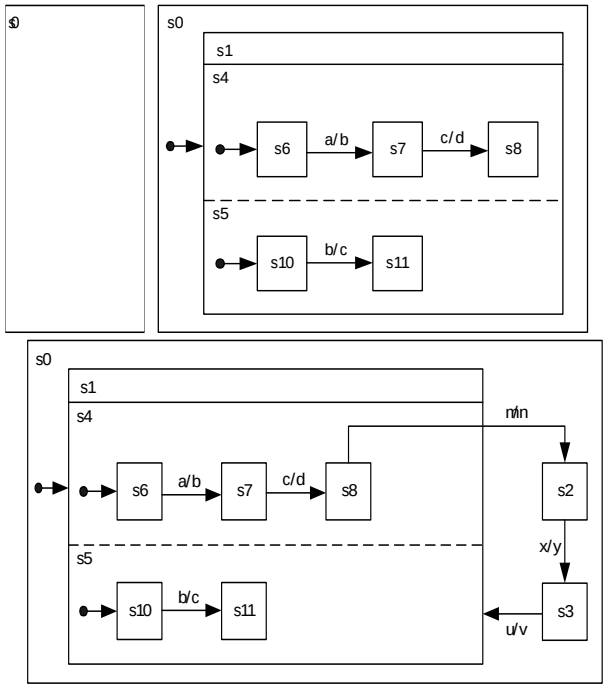


Figure4 Three dynamic slices of Statechart specification in Figure1 w.r.t α=b, α=a and α=axmuxu

Now we have such a question "is the dynamic slice produced by algorithm 3 a minimal dynamic slice?".

Figure5(b) is the dynamic slice generated by algorithm 3 given the Statechart specification shown in Figure(a) and slice criterion α=a. It is obvious that the generated dynamic slice shown in Figure5(a) is not a minimal dynamic slice, because that if we remove Statechart transition <s10, b, -, s11> from it, the remainder part is also a dynamic slice. Actually the

minimal dynamic slice for the Statechart specification shown in Figure5(a) w.r.t slice criterion α=a is the one shown in Figure5(c).

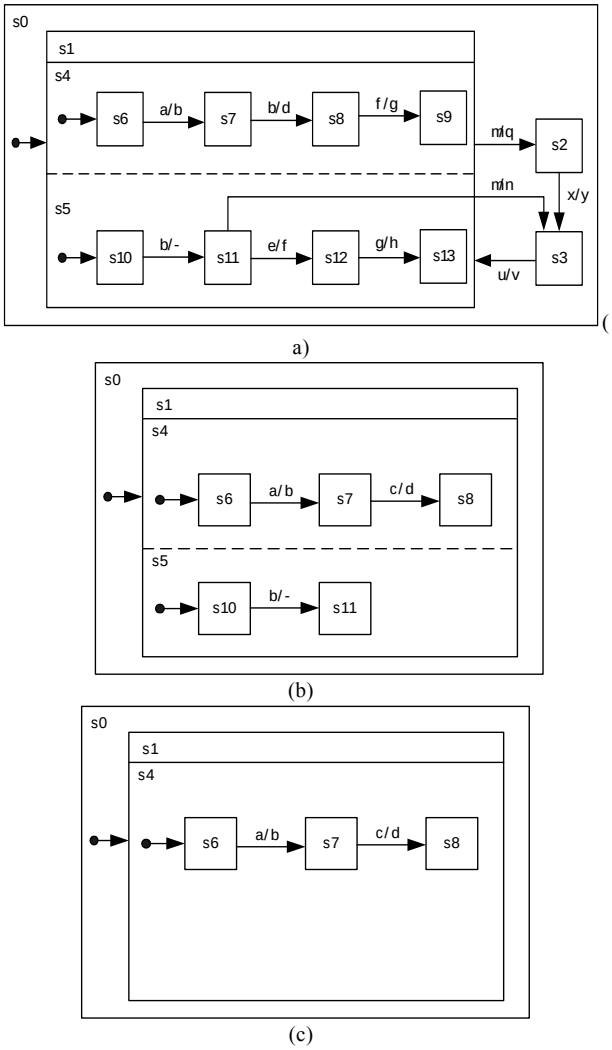


Figure5 (a) An example Statechart specification
(b) The generated dynamic slice with α=a
(c) The minimal dynamic slice with α=a

Because for minimal dynamic slice algorithm 3 is an approximation algorithm, its approximate analysis is unavoidable. By simple observation we can see that the approximation of the algorithm is dependent on the structure of the Statechart specification being sliced.

In the best case, for some compact Statechart specifications, the produced dynamic slices by algorithm 3 are also minimal dynamic slices, e.g., the dynamic slices shown in Figure4.

In the worst case, the generated dynamic slice is a bad approximation. For example, Figure6(b) is the dynamic slice generated by algorithm 3 given the Statechart specification shown in Figure6(a) and slice criterion α=a. There are many unobservable inside Statechart transitions. These Statechart transitions are executed for the possible subsequent environmental inputs u or v and they are no use for the current input a.

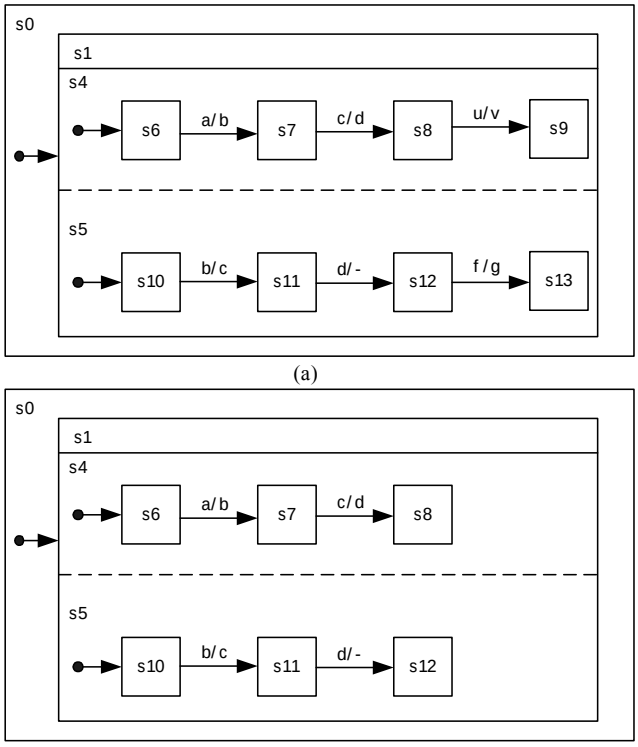


Figure6 (a) An example Statechart specification
(b) the generated dynamic slice with α=a

Although algorithm 3 can not produce minimal dynamic slices always, it has following advantages:

It is very simple. The algorithm is essentially a traversal algorithm that works on the observable semantics corresponding to the Statechart specification being sliced.

Although the produced dynamic slices may include some redundant Statechart transitions, which have nothing to do with the current environmental inputs and outputs, they cover all the executed Statechart transitions under input α sequence and they are useful for accurate system behavior analysis and verification.

V. CONCLUSIONS AND FURTHER WORK

The understanding, analysis and verification of developed Statechart specifications of the under-developing reactive systems is very important as the application domain requires very high quality systems. However, the size of the obtained system specification greatly increases with the system becoming larger and more complex and leads to state explosion problem, what makes industrial system specification difficult to be understood, verified and analyzed. To automatically decompose a large system specification into functionally independent sub-specifications that can be understood, verified and analyzed with relative ease seems a logical solution.

In this paper, we address the problem of dynamic slicing of a Statechart specification. The major contributions of our work are the following ones:

- The definition of a semantics model -- observable semantics, which is very suitable for slicing, because that it only describes outside observable

behavior and conceals unobservable behavior of Statechart specifications, and it fully captures the run-time dependence relation among the state transitions in the Statechart specification.

- The definition of a new notion of dynamic slicing that, in our opinion, is more natural for Statechart specifications. We formally define notions of dynamic slicing criterion, dynamic slice and minimal dynamic slice, and we also explain how to produce valid dynamic slicing criterion and propose a simple and practical approximation algorithm for minimal dynamic slice generation using observable semantics as an intermediate representation.

Our further work deals with the extension of the subset of Statechart notation we take into consideration, such as the introduction of data values and variables.

REFERENCES

[1] Harel. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 1987, 8(3): 231-274.

[2] M. Weiser. Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method. PhD thesis, The University of Michigan, Ann Arbor, Michigan, 1979.

[3] Venkatesh Prasad Ranganath and John Hatcliff. Pruning interference and ready dependence for slicing concurrent java programs. In Evelyn Duesterwald, editor, CC, volume 2985 of Lecture Notes in Computer Science, pages 39–56. Springer, 2004. ISBN 3-540-21297-3.

[4] Venkatesh Prasad Ranganath and John Hatcliff. Honing the detection of interference and ready dependence for slicing concurrent Java programs. Technical Report SAnToS – TR2003–6, Department of Computing and Information Sciences, Kansas State University, 7th October 2003.

[5] Y. Jiang and R. K. Brayton. Don't cares in logic minimization of extended finite state machines. In Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific, pages 809–815. IEEE, 2003.

[6] W. Damm, B. Josko, A. Votintseva, A. Pnueli, I. Ober, and S. Graf. A formal semantics for a UML kernel language, 2002. IST-2001-33522 OMEGA, Correct Development of Real-Time Embedded Systems, Public document, available at http://www-omega.imag.fr/doc/d1000009_6/ D112_KL.pdf.

[7] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999. Kluwer, 1999.

[8] Ji Wang, Wei Dong, and Zhi-Chang Qi. Slicing hierarchical automata for model checking UML statecharts. In Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, volume 2495 of Lecture Notes In Computer Science, pages 435–446. Springer-Verlag, 2002.

[9] Luangsodsai, A.; Fox, C. Concurrent statechart slicing. Computer Science and Electronic Engineering Conference (CEEC), 2010,1-7 M. Weiser. Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method. PhD thesis, The University of Michigan, Ann Arbor, Michigan, 1979.

[10] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems, 12(1):35-46, January 1990.

[11] Korel and J. Laski. Dynamic slicing of computer programs. Journal of Systems and Software, pages 198-195, 1990.

[12] Author introduction:

[13] Chunyu Miao, was born on 1978 in Jilin Province, China. Software engineering master, graduated from East China Normal University in 2006. And major field of study is computer application and trusted computing.

[14] He is a lecturer of College of XingZhi , Zhejiang Normal University Jinhua,, Zhejiang, China.