Statechart Slicing

Arthorn Luangsodsai and Chris Fox
School of Computer Science and Electronic Engineering
University of Essex
Wivenhoe Park, Colchester, CO4 3SQ, United Kingdom
Email: aluang@essex.ac.uk, foxcj@essex.ac.uk

***Abstract*—The paper discusses how to reduce statecharts model by slicing. We start with the discussion of control dependencies and data dependencies in statecharts. The *and-or* statechart dependence graph is introduced to represent control and data dependecies for statecharts. We show how to slice statecharts by using this dependence graph. Our slicing approach helps systems analysts and systems designers in understanding systems specification, maintaining software systems, and reusing parts of systems models.**

***Index Terms*—Statecharts Slicing, Control Dependencies, Data Dependencies, And-Or Dependence Graph, Backward Slicing, Forward Slicing, Specification-based Slicing, Model-based Slicing.**

I. I

The purpose of slicing statecharts is to reduce statecharts so that systems analysts and systems designers can focus on relevant parts of statecharts. Slicing statecharts helps in software maintenance [1], static analysis [2], and model checking [3].

There has been a significant amount of research in the area of program slicing and code-based slicing, as we can see from program-slicing survey literature [4]–[6]. In contrast, there is relatively little research on specification slicing and model-based slicing. Chang and Richardson [7] presented static and dynamic slicing of Z specification language. Heimdahl et. al. [8] described slicing of Requirements State Machine Language (RSML) to aid specification comprehension. Wang and Qi [3] presented an algorithm for slicing extended hierarchical automata for model checking UML statecharts. Korel et. al. [1] presented an approach to slice extended finite state models (EFSM). Our work is unique in providing the details of computing backward and forward slices using the *and-or* statechart dependence graph.

The *and-or* dependence graph represents control dependency and data dependency for statecharts. The slicing algorithm can implement either backward slicing or forward slicing by traversing the *and-or* dependence graph. The result of backward slicing of statecharts shows the reduced parts of statecharts that is relevant to the slicing criteria. The result of forward slicing of statecharts shows the reduced parts of statecharts that are dependent upon a given slicing criteria.

Section II gives a brief description of statecharts. Section III and Section IV discuss control and data dependencies in statecharts respectively. Section V illustrates how to slice statecharts using the *and-or* dependence graph. Section VI discusses our software for slicing statecharts and its evaluations. Conclusions and future work are discussed towards the end of the paper.

II. S

Statecharts were originally conceived by Harel [9]. He extended state transition diagram with the notions of hierarchy, concurrency and communication. The main purpose of using statecharts is to specify behaviour of complex reactive systems. Reactive systems, unlike transformational systems, have to react to external and internal stimuli. Examples of reactive systems include telephones, automobiles, communication networks, operating systems, missile and avionic systems [9].

Typically, a state is represented by a rectangle and a transition between states is shown by an labelled arc, as shown in Figure 1. The label specifies optionally one or more of a trigger event, a guard condition and an action. We can organise states into a composite state. Only one of the states in the composite state is active. Statecharts support concurrency.

The features of statecharts discussed in the present include transitions, triggers, guards (with expressions involving variables), actions (including assignments) and embedded states. We assume the semantics of statecharts as originally described in [10]–[12].

III. C D S

In this section we explain how we can model control dependencies in statecharts in terms of graph dependencies. Control dependence is usually defined for program statements by using a program dependence graph (PDG). Control dependence captures the notion that a node in the PDG may affect the execution of another node [13]. For example, the statements in the branches of an `if` statement are control dependent on the control predicate of an `if` statement. We extend the control dependence concept to statecharts.

For example, consider statecharts in Figure 1, to change from state x to state y, statecharts must be in state x, and event t must be triggered, and condition g must be true. That means state x, event t, and condition g control whether we take the transition in question and enter state y. We therefore could say that state y is control dependent on state x, event t and condition g. Similarly, before entering state y, action a must be activated. This implies that the occurrence of action a is control dependent on state x, event t and condition g.

A vertex in a dependence graph represents any object that can be depended upon, or which can depend upon other objects. Arcs between vertices indicate a potential dependence, the item represented by the source vertex depending upon that represented by the target vertex. Every item in the statechart which can depend upon another, or which can be depended upon itself, is added as a vertex to the dependence graph. This includes states, events, conditions, and actions.

For every transition of the form given in the statechart in Figure 1, then the dependence arcs in Figure 2 are added to the dependence graph. If any of the event, condition or action are missing, then the relevant arc or node in the dependence graph of Figure 2 is simply omitted.

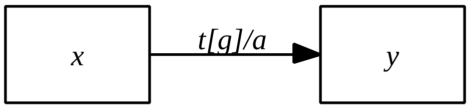

Fig. 1. Basic Transition

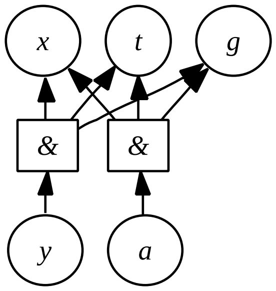

Fig. 2. Dependence graph for the statechart in Figure 1

Note that the dependencies that arise in an individual transition are conjoined; the dependencies are all or nothing. However, the dependencies for distinct transitions are to be considered as disjoined. This is why we call this dependency graph an *and-or dependence graph*.[1] Unlike program dependence graph (PDG) where control dependence is represented with dotted edge, here we use normal solid edge to represent control dependence in *and-or* dependence graphs. We can share common structures in the graph — provided they occur within the same concurrent context — giving the reduced version of Figure 3.

Figure 4 shows an example of branching and merging of transitions in statecharts. Figure 4 also shows the stop

[1]The distinction between *and* and *or* nodes potentially allows for smaller slicers for a form of analysis that is not the subject of the present paper.

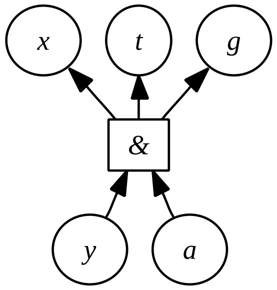

Fig. 3. Reduced dependence graph for the statechart in Figure 1

circle after state $s6$ to represent the stop or exit point for statecharts. However, we will omit the stop circle in later example figures of statecharts in the paper. Figure 5 shows the corresponding dependence graph for Figure 4. We add the *Enter* and *Exit* box in the graph to show that state $s1$ depends on *Enter* box and *Exit* box depends on state $s6$. However, later on in the paper, we will omit the *Enter* and *Exit* box in dependence graphs.

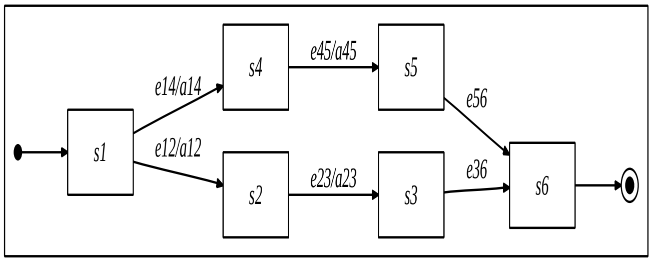

Fig. 4. Statecharts with branching and merging of transitions

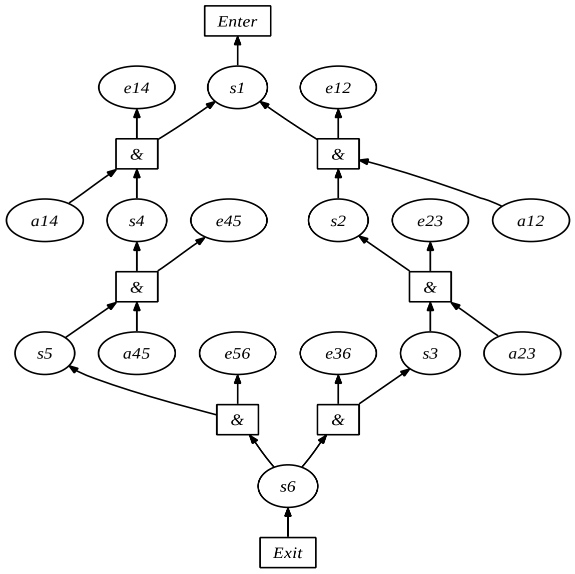

Fig. 5. Dependence graph for the statechart in Figure 4

IV. D D S

In slicing programs, in addition to control dependence, we also have to consider data dependence [4]. Data dependence and control dependence both are represented in PDGs [13]. Assume we have node i and node j in a PDG, and there is an edge from node i to node j. The sets $DEF(i)$ is the set of variables defined at node i, and the sets $REF(j)$ is the set of variables referenced at node j. Node j is data dependent on node i if there exists a variable x such that: (1) $x \in DEF(i)$, (2) $x \in REF(j)$, and (3) there exists a path from i to j on which x is not modified. This definition of data dependence for programs can be adapted to capture data dependence for statecharts.

Data dependencies may occur among variables in conditions and in actions of statecharts. These conditions and actions must exist along the same path in statecharts. For example, Figure 6 shows conditions and actions which are statement expressions[2]. The *and-or* dependence graph can be extended to include data dependencies in a natural way. Figure 7 shows control dependencies in *solid edge* and data dependencies in *dotted edge*. Unlike program dependence graph (PDG) where data dependence is represented with normal solid edge, here we use normal *dotted edge* to represent data dependence in *and-or* dependence graph.

As we see from the figures, action $y = x + 5$ uses variable x which is defined in action $x = 0$. Therefore, there is data dependence edge point to node $x = 0$ from node $y = x + 5$. For this case, the data dependence of a node with action $y = x + 5$ is the data dependence of y which is the left hand side of assignment statement. Also, note that node $y = x + 5$ has control dependent edge point to *and* node as well (as we can see from Figure 7). Both data and control dependent edge from node $y = x + 5$ are treated as being conjoined together.

Condition $[x > y]$ uses variable x and y. Therefore, there are data dependence edges pointing from this node to node $x = y+5$ and to node $y = x+5$, but not to node $x = 0$ because x has been modifed at node $x = y+5$. We may say that data dependencies of a node with a condition are the dependencies that govern its evaluation. Also, note that node $[x > y]$ has two data dependent edges point to node $x = y + 5$ and to node $y = x + 5$. These two data dependent edges from node $[x > y]$ are treated as being conjoined together.

Figure 8 shows an example of branching and merging of transitions in statecharts. Figure 9 shows its corresponding dependence graph. Notice here that action $y = x$ after state $s4$ depends on action $x = 5$ after state $s1$, not action $x = 0$ because action $x = 0$ is in the different transition path from action $y = x$.

[2]Here we use syntax of C programming language for assignment and boolean statements

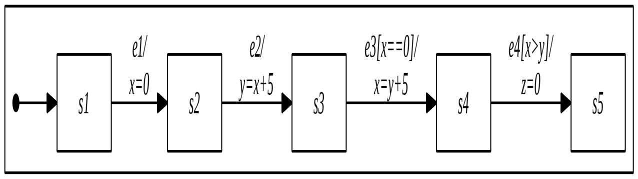

Fig. 6. Statechart with statement expressions

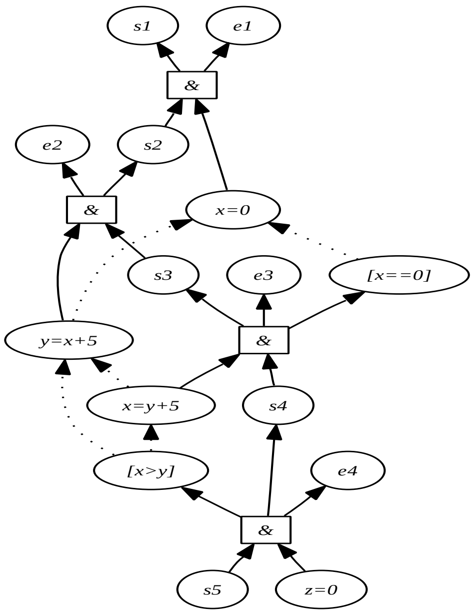

Fig. 7. Dependence graph for the statechart given in Figure 6

V. S S U A -O D G

A. A Slice of Statecharts

A program slice [14] is a reduced part of program statements in which a programmer may have some interest when debugging or maintaining programs. To get the relevant part of programs, programmers must specify some criteria which can be used to identify some relationships or dependencies among these relevant parts of programs. These criteria are called *slicing criteria*. For statecharts, a slice of a statechart is a reduced part of stat-

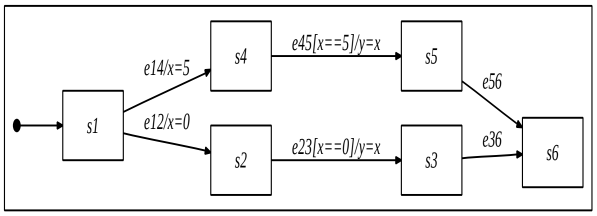

Fig. 8. Statecharts with branching and merging of transitions

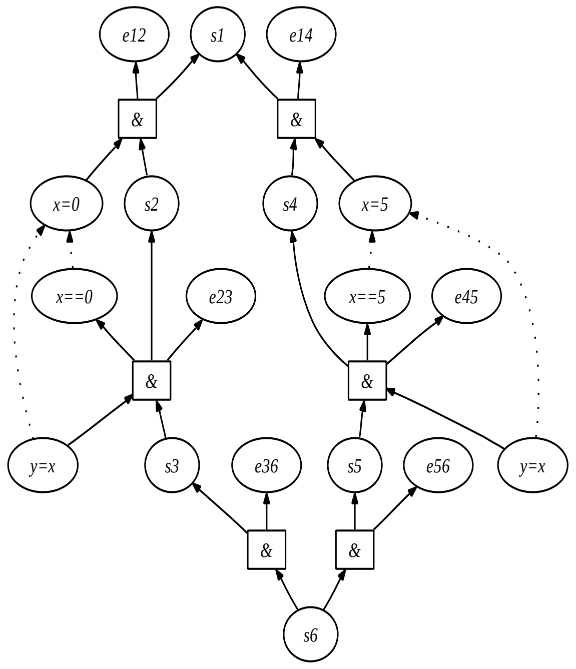

Fig. 9. Dependence graph for the statechart in Figure 8

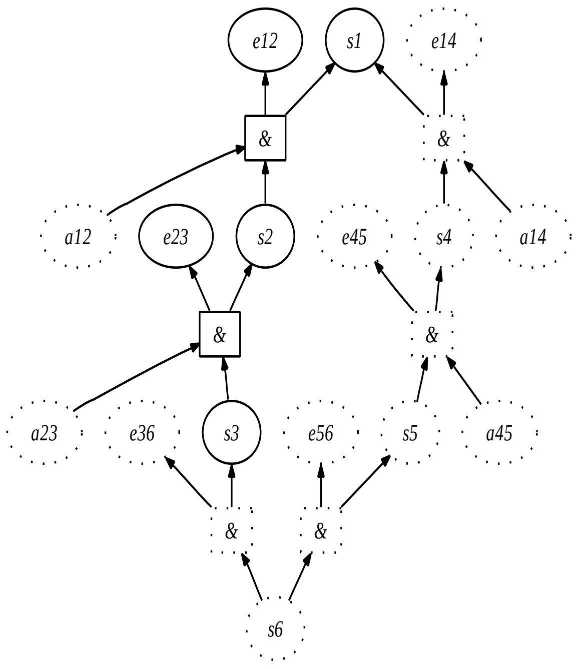

Fig. 10. Sliced dependence graph for the statechart in Figure 4 with backward slicing criteria <state $s3$>

echart in which systems analysts or systems designers may have some interest when designing or maintaining software systems. Similarily, to obtain the relevant parts of a statechart, analysts or designers must define some criteria to be used to isolate relevant relationships and dependencies between relevant parts of statecharts.

B. Slicing Criteria

The slicing criteria for program slicing is usually a collection of pairs of program variables and program points [14]. In case of slicing statecharts, slicing criteria can be a state, an event, a condition, or an action. However while states are unique, an event (condition or action) may be duplicated and appear in more than one place. Therefore, we use a state as a specified location to uniquely identify which event (condition or action) we want to be the slicing criteria. For example, if we are interested in which parts of the statecharts affect state $s3$ in Figure 4, we must specify <state $s3$> as the slicing criteria. If we are interested in which parts of the statecharts affect event $e45$, we specify <state $s4$ and event $e45$> as the slicing criteria.

C. Backward Slicing of Statecharts

After we represent statecharts in the form of the *and-or* dependence graph, and decide on an appropriate statecharts' slicing criteria, we can apply a slicing algorithm using graph-reachability from the point of interest in the dependence graph. Ottenstein and Ottenstein [15] proposed a slicing algorithm by define slicing as a graph reachability problem over the dependence graph. Our approach is similar.

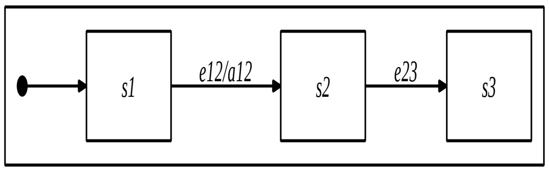

Fig. 11. Sliced statecharts with backward slicing criteria <state $s3$>

For example, consider statecharts in Figure 4 and its dependence graph in Figure 5, if we backward slice this statecharts with respect to slicing criteria <state $s3$>, we start from node $s3$ and follow edges in the *same direction* of arrowheads. We would get the graph as shown in Figure 10, where the solid nodes are the sliced nodes and the dotted nodes are irrelevant nodes. We can see nodes that relevant are node $s3$, $s2$, $e23$, $s1$ and $e12$. The sliced statecharts is shown in Figure 11. This result means that components of statecharts which affect state $s3$ are state $s1$, event $e12$, action $a12$, state $s2$ and event $e23$.

D. Forward Slicing of Statecharts

In program slicing, we can slice backwards or forwards [5]. A backward slice includes program statements which can have effects on the slicing criterion. A forward slice includes program statements which are affected by the slicing criterion. Similarily, a backward slice of a statechart contains the reduced parts of the statechart that have affected the slicing criterion. A forward slice of a statechart contains the reduced parts of the statechart that are dependent upon the slicing criterion.

With backward slicing, we traverse the *and-or* statechart dependence graph by locating the slicing criterion node and then follow the arrowhead of the edges from the criterion node. However, with forward slicing, after locating the slicing criterion node we follow the arrowhead of the edges *backwardly* from the slicing criterion node. Additionally, when we follow backwardly and found the *and* node $\boxed{\&}$, we must include nodes which are attached to this *and* node in a slice as well. This is to ensure that the final statechart is well formed.

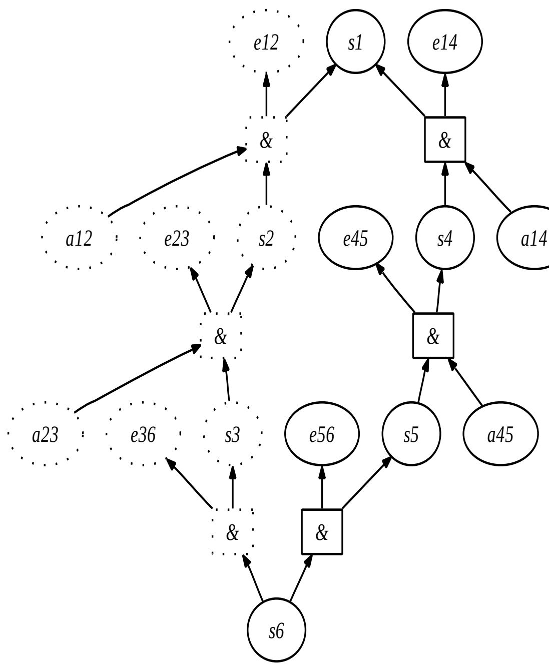

Fig. 12. Sliced dependence graph for the statechart in Figure 4 with forward slicing criteria <state *s*1, event *e*14>

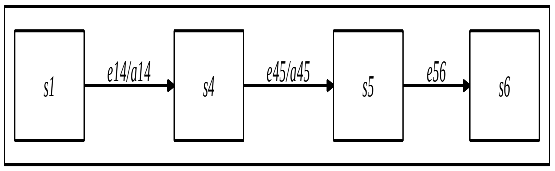

Fig. 13. Sliced statecharts with forward slicing criteria <state *s*1, event *e*14>

To illustrate forward slicing of statecharts, we look again at Figure 4 which shows the example of branching and merging of statecharts, and its *and-or* dependence graph in Figure 5. If we forward slice this statecharts with respect to slicing criteria <state *s*1, event *e*14>, we start from node *e*14 and follow edges backward in the opposition direction of arrowheads. We reach the *and* node after node *e*14 so we must include node *s*1 which is attached to this *and* node in our slice too.

We would obtain the graph as shown in Figure 12 where the solid nodes are the sliced nodes and the dotted nodes are deleted nodes. We can see the relevant nodes are node *e*14, *s*1, *a*14, *s*4, *e*45, *s*5, *a*45, *e*56 and *s*6. The forward sliced statechart is shown in Figure 13. This result means that components of statecharts which are affected by event *e*14 are state *s*1, action *a*14, state *s*4, event *e*45, action *a*45, state *s*5, event *e*56 and state *s*6.

VI. S S S E

A. Statecharts Slicing Software

Implementation of software for statecharts [9] is a complex process. Statechart slicing software has to support diagram drawings and assume a particular semantics of statecharts. For our our implementation of software tools for statecharts, we apply State Chart eXtensible Markup Language (SCXML) to represent statecharts.[3]

The input of the system is SCXML and the output of the system will be sliced SCXML, together with Graphviz's `dot`[4] representation of the sliced statecharts. `Dot` files can be converted into conventional visual representations of sliced statecharts in the form of Postscript or PDF output.

Figure 14 displays the overview of the system. The system gets the input file which is the XML file representing the statecharts. The system then parses this file and creates the dependence graph representing the control dependence and data dependence relationship among states, trigger events, guards, and actions. The user can choose to perform either backward slicing or forward slicing by giving a slicing criteria to the system. The system will display output of the sliced SCXML tags, and then produce a dot output file. Users can produce the graphic file format postscript or jpeg file from this dot file.

B. Evaluations

Traditionally, evaluation of program slicing uses comparing of number of statements in the slice to number of statements in the original program. The survey from Binkley and Harman [16] showed the empirical results on program slicing. The results mostly computed slice sizes in terms of line of code (LOC) and calculated average slice sizes and percentage of reduction. However, for statecharts or hierarchical state machines, there are no standard metrics to evaluate or measure the size and complexity of a state machine [8]. We have chosen

[3]SCXML (http://www.w3.org/TR/scxml/) was originally an XML format targeted at specifying voice browser related behaviour, but includes a complete statechart modelling language that is closely related to Harel and UML statecharts. We assume the semantics of SCXML statecharts as described in the draft of 24th January, 2006, which is essentially a version of the semantics for STATEMATE [10].

[4]http://www.graphviz.org

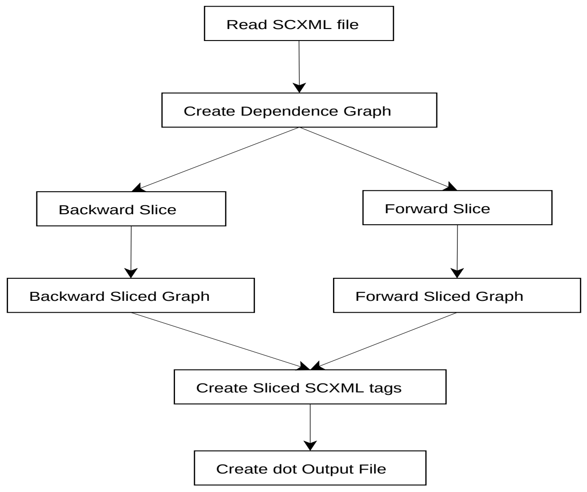

Fig. 14. The System Overview

to measure the number of units such as number of states, number of events, number of guard conditions and number of trigger actions.

We evaluate our software by slicing some statecharts case studies, and compute the percentage of reduction of number of states, events, guards, and actions. The case studies cover a representative range of statechart constructions. The results are given in Table I.

Reduction Percentage Results Across Cases		
Cases	Backward Slice	Forward Slice
Case 1	81.25%	62.68%
Case 2	59.14%	50.78%
Case 3	64.13%	36.38%
Case 4	56.09%	63.15%
Average	65.15%	53.25%

TABLE I
R P R A C

The result shows that for our case studes, the average percentage of reduction is 65.15% for backward slicing. The average percentage for forward slicing is 53.25%. The results suggest that on average statechart slicing does help to eliminate parts of the statechart that are not relevant to the slicing criteria.

VII. C F W

This paper sketches the construction of *and-or* dependence graphs for statecharts, and their use in creating slices of statecharts. We introduce control and data dependencies for statecharts, and illustrate how to do backward and forward slice from these dependencies.

We present results that show slicing reduces the size of a statechart. We hypothesise that this should assist those who need to interpret statecharts.

In future work we will describe how our analysis can be extended to concurrent statecharts. This can be achieved by using a hierarchical name-space convention. Other slicing techniques such as dynamic slicing [17], conditioned slicing [18], [19], and techniques that more fully exploit the *and-or* dependencies will also be investigated. This will include a notion of slicing in which the criteria indicates what should be removed, rather than what should be retained.

R

[1] B. Korel, I. Singh, L. Tahat, and B. Vaysburg, "Slicing of state-based models," in *19th IEEE International Conference on Software Maintenance (ICSM'03)*. Los Alamitos, CA, USA: IEEE Computer Society, 2003, p. 34.

[2] M. P. Heimdahl and M. W. Whalen, "Reduction and slicing of hierarchical state machines," in *Proceedings of the 6th European conference on Foundations of Software Engineering*. Springer-Verlag, 1997, pp. 450–467.

[3] J. Wang, W. Dong, and Z. 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*, ser. Lecture Notes In Computer Science, vol. 2495. Springer-Verlag, 2002, pp. 435–446.

[4] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, September 1995.

[5] M. Harman and R. Hierons, "An overview of program slicing," *Software Focus*, vol. 2, no. 3, 2001.

[6] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[7] J. Chang and D. J. Richardson, "Static and dynamic specification slicing," in *In Proceedings of the Fourth Irvine Software Symposium*, 1994.

[8] M. P. Heimdahl, J. M. Thompson, and M. W. Whalen, "On the effectiveness of slicing hierarchical state machines: A case study," in *24 th. EUROMICRO Conference*, vol. 1. IEEE Computer Society, 1998.

[9] D. Harel, "Statecharts:a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[10] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman, "On the formal semantics of statecharts," in *Proceedings of the second IEEE Symposium on logic in computer science*. New York: IEEE CS Press, 1987, pp. 54–64.

[11] D. Harel and A. Naamad, "The STATEMATE semantics of statecharts," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 4, pp. 293–333, 1996.

[12] E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel, "On formal semantics of statecharts as supported by statemate," July 1997.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, 1987.

[14] M. Weiser, "Program slicing," in *Fifth International Conference on Software Engineering*, San Diego, CA, Mar. 1981, pp. 439–449.

[15] K. Ottenstein and L. Ottenstein, "The program dependence graph in software development environments," *SIGPLAN Notices*, vol. 19, no. 5, pp. 177–184, 1984.

[16] D. Binkley and M. Harman, "A survey of empirical results on program slicing," in *Advances in Computers, 62:105178*, 2004, pp. 105–178.

[17] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 246–256.

[18] G. Canfora, A. Cimitile, and A. De Lucia, "Conditioned program slicing," in *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds. Elsevier Science B. V., 1998, vol. 40, pp. 595–607.

[19] C. Fox, S. Danicic, M. Harman, and R. Hierons, "ConSIT: A conditioned program slicer," in *IEEE International Conference on Software Maintenance (ICSM'00)*. San Jose, California, USA: IEEE Computer Society Press, Los Alamitos, California, USA, Oct. 2000, pp. 216–226.