And-Or Dependence Graphs for Slicing Statecharts

Chris Fox and Arthorn Luangsodsai
Department of Computer Science
University of Essex
foxcj@essex.ac.uk, aluang@essex.ac.uk

January 2006

Abstract

The construction of an And-Or dependence graphs is illustrated, and its use in slicing statecharts is described. The additional structure allows for more precise slices to be constructed in the event of additional information, such as may be provided by static analysis and model checking, and with constraints on the global state and external events.

1 Introduction

We take a *slice* to be some syntactic projection of a syntactic object expressed in a language that has a well-defined operational interpretation. Usually we are interested in selecting a projection based upon some *slicing criteria*. The slice should in some sense preserve the operational behaviour of the original object's with respect to the slicing criteria.[1]

Here we consider slicing of statecharts. Slicing statecharts can help with specification comprehension, static analysis (Heimdahl and Whalen, 1997), and model checking (Wang et al., 2002). It may also help with specification based testing.[2]

As originally introduced, slicing was original targeted at programs (Weiser, 1979). Following our generic definition, a program slice is then an order-preserving projection of statements from the original program. The slicing criteria is usually a pair consisting of a variable and program point of interest (or a collection of such pairs). The slice should preserve the behaviour of the original program with respect to those variables at

[1] Sometimes additional work is required for the slice to be well-formed, in which case the unadulterated slice can be taken to indicate some of the syntactic constituents of an appropriate projection of the original object.

[2] R. Hierons, PC.

the relevant program points.[3] There are variants of program slicing (for example, forward slicing, dynamic slicing, conditioned slicing), which we will not consider here.

In the case of a statechart, a slice will be a well-formed structure preserving projection of the original. The slicing criteria can be stated in terms of collections of states, transitions, actions, and variable names (assuming that there is a global or local state with variables that are updated by actions and tested by guards). Actions and variables can be preprocessed to give an appropriate collection of states and transitions (Wang et al., 2002).

This paper proposes the use of dependence graphs (Kuck et al., 1981) for slicing statecharts. Such graphs are already used for efficient slicing of programs (Ottenstein and Ottenstein, 1984; Horwitz et al., 1990, for example). In itself, this may not be particularly novel (Heimdahl et al., 1998). However, here we propose to use slightly richer to dependence graphs, that record And-Or dependencies. This promises to give rise to smaller slices when supplemented with additional analysis; if it can be shown that an apparent dependence can never be realised (e.g. by model checking, interference analysis, or conditions on external events and global variables), then all the dependencies with which it is conjoined can also be eliminated when constructing the slice.

2 Statecharts

Statecharts were originally conceived by Harel et al. (1987). 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 what might be described as 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 (Harel et al., 1987).

The notion of a statechart has been adopted and extended by the Open Management Group (OMG)[4] as one of the specification formalisms of the Unified Modelling Language (UML)[5], and also by W3C in the form of SCXML.[6]

[3] There is room for debate about whether or not the slice should preserve all side-effect behaviour, such as input and output activity. If input behaviour is not preserved, then the relevant behaviour may only be preserved if we also "slice" the input stream in an appropriate fashion. In the case of output, were we considering the behaviour of the program within some pipeline, then we might have to reflect on how to manage the impact of the slice through any down-stream programs.

[4] `http://www.omg.org`.

[5] `http://www.uml.org`.

[6] SCXML (`http://www.w3.org/TR/scxml/`) was originally targeted at specifying voice

Statecharts are usually presented in a graphical form, although there are standard "textual" formats, including SCXML, and XML Metadata Interchange (XMI)[7] for UML.

Typically, a state is represented by a rectangle and a transition between states is shown by an labelled arc. The label specifies a trigger event, a guard condition and an action, although one or more of these elements may be absent. Following the example in Figure 1, if event t occurs (either externally, or as a result of the activity of some concurrent process) and condition g is true then the state of the system will be changed from state x to state y after completing action a.

In general, guards may be propositions about variables in the global state, perhaps involving Boolean valued functions and method calls, and actions may be operations on the global state that update the values of global variables, or make method calls. However, in order to simplify the presentation, hereinafter we take the actions, guards and triggers to be atomic.[8]

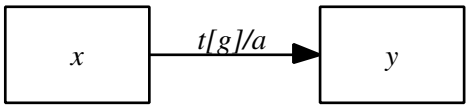

Figure 1: Basic Transition

We can organise states into a composite state as shown in Figure 2. Only one of the states in the composite state is active. The figure also shows a default state or an initial state represented by a small arrow pointing to state x. That means when entering this composite state, it will enter to state x.

Statecharts support concurrency. The usual visual presentation is to combine the concurrent parts in one containing box, and separate them by way of dashed lines, as shown in Figure 3. If the concurrent process is active, then one state in each of the regions will be active.

There are many features of statecharts that we do not consider in this paper. Readers can find more details in Harel et al. (1987). For example, features that are not covered in this paper include: the condition and selection circled connectives; delays and timeouts expression; entry/exit activity and activities inside a state; transitions between composite activities; histories.

---

browser related behaviour, but consists of a complete statechart modelling language that is closely related to Harel and UML statecharts.

[7] `http://www.oasis-open.org/cover/xmi.html`

[8] This means that some dependencies are not considered here. It should be straightforward to extend the approach to include operations and expressions involving functions on variables. In general, if Object Constraint Language (OCL) expressions and method calls can appear as labels, as in UML 2.0 statecharts, then there may be additional dependencies that will be hard to take into account without a more complete model of the relevant methods, objects and classes, and the dependencies between them.

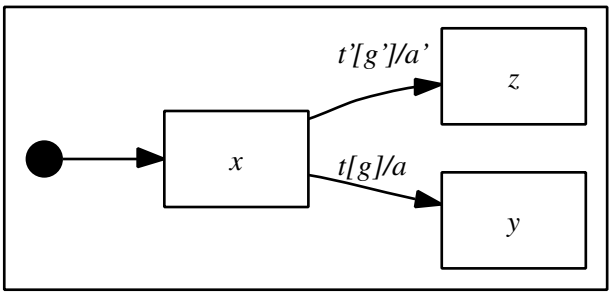

Figure 2: A Composite State

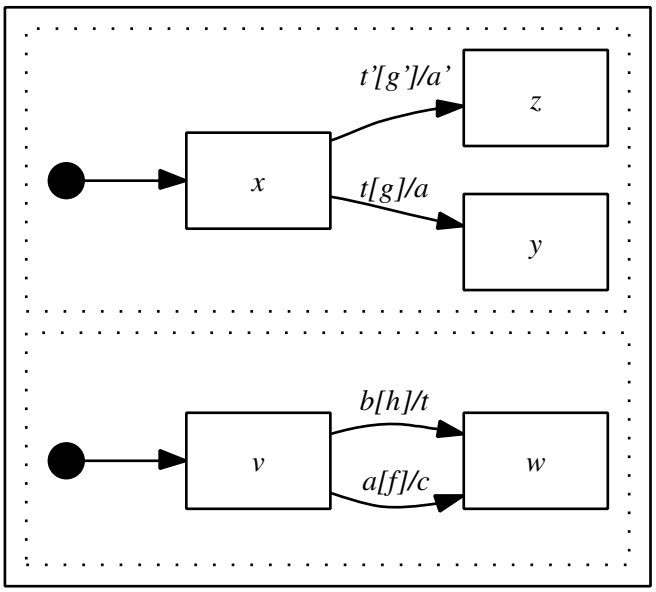

Figure 3: Statechart with Concurrency

We assume the synchronous, microstep semantics for concurrency. In summary, on receiving an external trigger, any transitions that can be taken will be. Any resulting internal actions that are triggers for available transitions will be dealt with in the order that they arise. External triggers will be queued until all internal transitions have completed. Any global state will remain the same for the purposes of evaluation whilst transitions are in progress.[9]

3 Dependencies in Statecharts

In this section we explain how we can model dependencies in statecharts in terms of graph dependencies.

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.

Unlike dependence graphs for programs, dependencies arising from a common transition in the original statechart can be considered as being conjoined; if one of the constituent dependencies can be shown to be infeasible, for example through interference analysis (Ranganath and Hatcliff, 2003, 2004), some form of conditioning (Jiang and Brayton, 2003) or similar techniques, perhaps in conjunctions with constraints on the environment.

The construction of a dependence graph for a statechart is sketched here in terms of an algorithm. A method for traversing the initial statechart is assumed. It is also possible to describe the dependence graph for a statechart by way of a declarative definition.

3.1 Initialisation

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, actions, triggers, and guards.

For simplicity of presentation, here we treat actions, guards and triggers as atomic items, as already mentioned. A slightly more elaborate analysis is required to deal with additional dependencies that arise with non-atomic labels, as might be used if actions and guards involve variables in some global state: in this case, variables would appear as vertices in the dependence graph.

Some preprocessing is required to cope with state-internal actions, and transitions involving non-atomic states. Actions and guards that involve method calls may require an even more involved analysis.

[9] This does not exclude various forms of non-deterministic behaviour.

3.2 Adding Dependencies

For every transition of the following form given in Figure 1 in the statechart, then the dependence arcs in Figure 4 are added to the dependence graph. If any of the trigger, guard or action are missing, then the relevant arc in the dependence graph of Figure 4 is simply omitted.

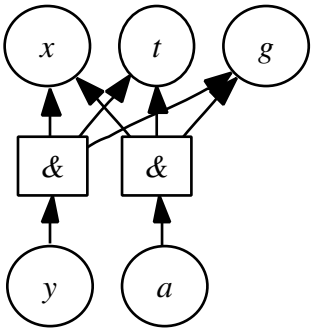

Figure 4: Dependence graph for the statechart in Figure 1

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 (as will be seen in the case of state w of Figure 6, where we consider concurrency).

We can share common structures in the graph, giving the reduced version of Figure 5.

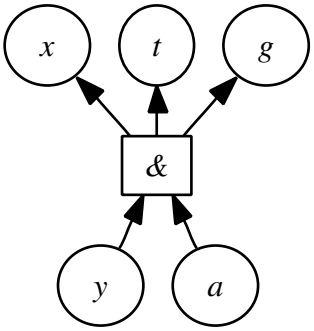

Figure 5: Reduced dependence graph for the statechart in Figure 1

3.3 Concurrency

It turns out that dependencies between concurrent processes will appear in the dependence graph without any additional effort. The dependence graph for the concurrent statechart of Figure 3 is given in Figure 6.

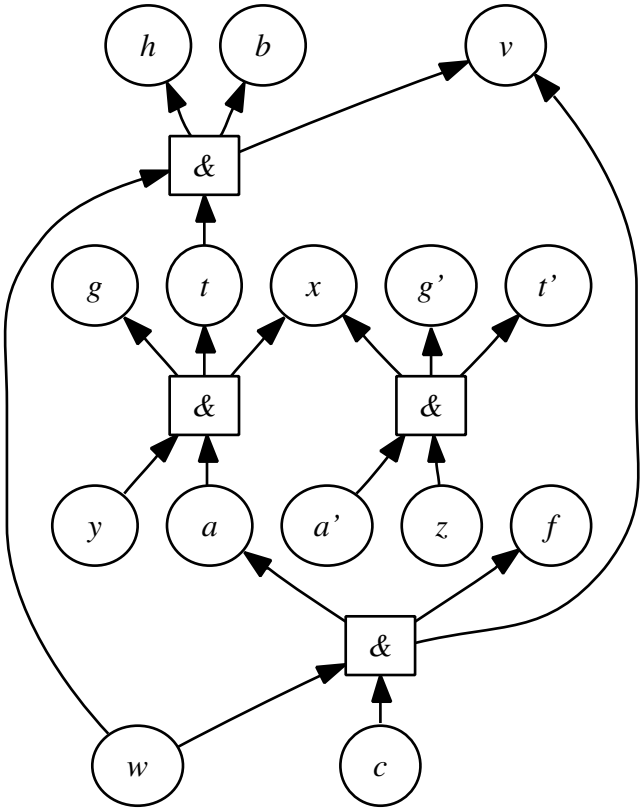

Figure 6: Dependence graph for the statechart given in Figure 3

Of course, without care, all actions may appear to be triggers for transitions with the same label appearing as their trigger, regardless of whether those transitions are syntactically (or operationally) concurrent with action in question. For this reason, it is necessary to augment the vertex labels with an indication of where the relevant action and trigger arises in the statechart, for example by prefixing with labels representing the composite states in which they occur. With appropriate rules for the chosen labelling regime, it is then possible to identify potential/syntactic dependency between concurrent actions and triggers. We have omitted this detail here for presentational reasons.

Start nodes present a related problem, in that there is no instrinisic distinction between different occurrences of such nodes. However, as there should only be one start node at a given level within a composite or concurrent process, we can once again distinguish between them by way of the position in which the occur in the statechart. As before, we omit this detail for clarity of exposition.

3.4 Other Dependencies

The dependence graph can be extended to include variable dependencies in a natural way, following the data-dependence rules given by Wang et al. (2002), or a refinement of them. Dependencies between method calls would require additional information about the details of the methods' behaviours.

4 Slicing statecharts using the dependence graph

The dependence criteria for program slicing is usually a collection of pairs of program variables and program points.

After we represent statecharts in the form of the dependence graph, and determining the initial transitions and states of interest from the slicing criteria, we can apply slicing algorithm using graph-reachability from the point of interest in the dependence graph. Ottenstein and Ottenstein (1984) proposed slicing algorithm by define slicing as a graph reachability problem over the dependence graph. Our approach simply adapts this method to And-Or dependency graphs.

As an example, if we slice with respect to action a, then from Figure 6 we can see that the part of the original state chart that is relevant involves g, t, x, h, b, v and any mention of a itself. The sliced statechart will the be as given in Figure 7. We also need to keep any relevant start nodes.

4.1 Infeasible dependencies

Simple syntactic slicing of statecharts can be augmented with additional analysis. For example, using model checking and interference analysis

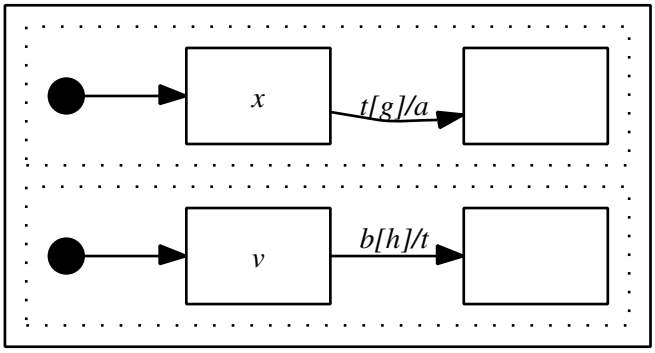

Figure 7: Slice of the statechart given in Figure 3 with respect to a

Ranganath and Hatcliff (2003, 2004) we can determine whether a specific action that arises within a specification can actually act as a trigger for a transition in a concurrent process.

We may also use various forms of conditioning, for example by imposing constraints on external events (perhaps as specified by some transition system, or even another statechart), and on global variables (following the idea of program conditioning Canfora et al. (1998); Fox et al. (2000) and infeasible transitions Jiang and Brayton (2003)) which may constrain potential dependencies in the slice.

We believe that in these cases, the use of And-Or dependency graphs will offer a significant opportunity for refining the slice; in the event that static analysis, or model checking reveals that one dependency in a conjunction of dependencies can never be realised (either in general, or in the context of interest), then all of the other dependencies with which it is conjoined can be ignored.

As an example, assume we slice the statechart of Figure 3 on criteria w, then this would include the entire statechart except for a', t', g', the action label c, and the state labels z, y, as in Figure 8. However, if we know that the guard f can never be satisfied where it occurs, then a close examination of the And-Or dependency tree shows that we can also prune t, x, a, f, and g itself from the slice, giving the slice of Figure 9.

In the event that there is no other way of reaching the nodes in question, this "pruning" of the transition dependencies can be propagated upward through both states and actions, and downward through states. Note that in general we cannot propagate the pruning downward through actions/triggers, as there may be external actions that match the trigger in question. For this reason, if an action is pointed at in relevant part of the dependence graph, then it must remain present in the slice wherever it appear as a trigger.

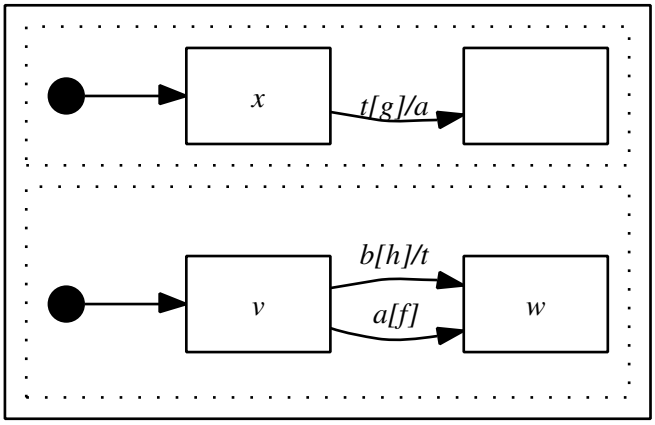

Figure 8: Process of Figure 3 sliced on w

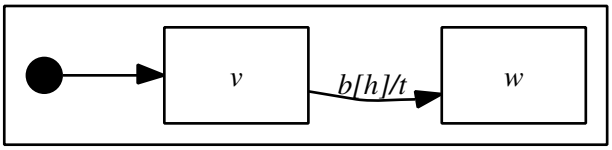

Figure 9: Process of Figure 3 sliced on w when f can never be true at the relevant position

5 Related Work

Wang et al. (2002) present an algorithm for slicing extended hierarchical automata for model checking UML statecharts. The slicing criterion is based on states and transitions. The algorithm can remove hierarchies and concurrent states which are not relevant to the property. They argue that the state space for model checking UML statecharts is reduced efficiently by their slicing algorithm.

Heimdahl et al. (1998) describe slicing of Requirements State Machine Language (RSML) to aid comprehension and static analysis. There is also work that can be considered to amount to a form of conditioned slicing (Canfora et al., 1998; Fox et al., 2000) applied to statecharts (Jiang and Brayton, 2003).

As described in their paper, Wang et al. (2002) produced slices by direct computation of the data and control flow via a collection of dependence rules. Heimdahl and Whalen (1997) adopts an approach based on that of Sloane and Holdsworth (1996). As described, data and control flow slicing are computed separately on the basis of a marked up abstract syntax tree. Although the description of data control slicing does mention a data dependence graph, neither the explicit production of a general dependence graph nor is the possibility of recording conjoined dependencies is mentioned.

There has been work on the use of interference analysis for slicing concurrent programs (Ranganath and Hatcliff, 2003, 2004), but not in the context of statecharts, or And-Or dependence graphs.

6 Conclusions and Future Work

This paper sketches the construction of And-Or dependence graphs for statecharts, and their use in creating slices of statecharts. The paper illustrates how the additional And-Or information can be used to help produce more precise slices when additional information is available, from other forms of static analysis, model checking, perhaps in combination with constraints on the external environment and global state. This is possible because the And-Or information helps to identify additional irrelevant transitions and states.

Some details are omitted to simplify the presentation. These will be fleshed out in future presentations, where a more detailed comparison of the slicing performed by this approach and that of other accounts will be made. Future work includes incorporating static analysis and model checking methods into an implemented system, with the objective of allowing different kinds of constraints on the global environment to be taken into account when slicing. This will correspond to a generalised form of conditioned-slicing for statecharts.

References

Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

Chris Fox, Sebastian Danicic, Mark Harman, and Rob Mark Hierons. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)*, pages 216–226, San Jose, California, USA, October 2000. IEEE Computer Society Press, Los Alamitos, California, USA.

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*, pages 54–64, New York, 1987. IEEE CS Press.

Mats P.E. Heimdahl and Michael W. Whalen. Reduction and slicing of hierarchical state machines. In *Proceedings of the 6th European conference on Foundations of Software Engineering*, pages 450–467. Springer-Verlag, 1997.

Mats P.E. Heimdahl, Jeffrey M. Thompson, and Michael W. Whalen. On the effectiveness of slicing hierarchical state machines: A case study. In *24 th. EUROMICRO Conference*, volume 1. IEEE Computer Society, 1998.

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

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.

D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the 8th annual ACM Symposium on Principles of Programming Languages*, pages 207–218, 26th–28th January, 1981.

Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184, 1984.

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.

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.

Anthony M. Sloane and Jason Holdsworth. Beyond traditional program slicing. In Steven J. Zeil, editor, *Proceedings of the 1996 International Symposium on Software Testing and analysis*, pages 180–186, New York, January 8–10 1996. ACM Press. ISBN 0-89791-787-1.

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.

Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.