Automatic Proofs for Scalecharts

Richard Bosworth

Visual Modelling Group,
University of Brighton, UK
richard.bosworth@bton.ac.uk

Abstract. A scalechart is a set of statecharts, operating in a dense time domain, whose behavior is self-similar at different scales. The simplicity of extracting proofs of behavior from scalecharts is demonstrated, based on the $<$ and *co* relationships. An algorithm is presented which automatically extracts $<$ relationships from a simplified version of scalecharts.

1 Introduction

A *scalechart* [1] is a set of Harel-style statecharts [2] that operate in a dense (ever-divisible) time domain. Structurally each statechart of the set consists of a set of nodes and a set of transitions between those nodes, which form the vertices and arcs respectively of a directed graph. Each statechart has a distinguished start node. A node contains one or more orthogonal components or *orthocomps*, each of which may contain a statechart, forming a self-similar structure in space.

Scalecharts are also self-similar in time. There are no generated events associated with transitions in scalecharts; rather, generated *signals* are associated with nodes. Unlike events, signals have an extended duration in time.

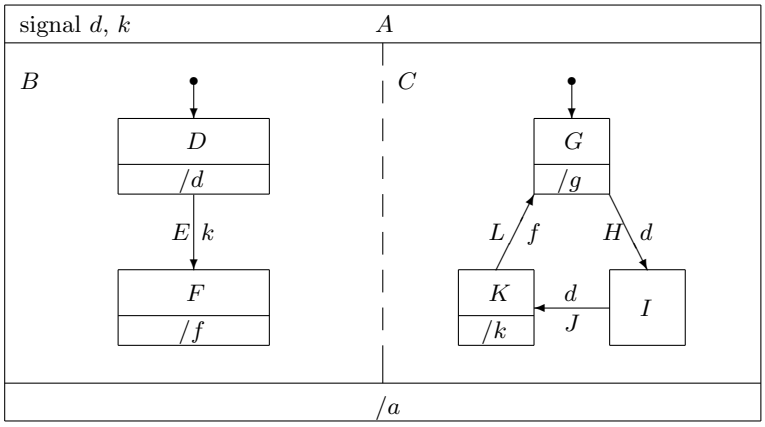

Fig. 1. Signalling inside a compound scalechart node

For example, in Fig. 1, the state D has a signal d associated with it. d remains asserted as long as D is active. D can only become inactive when transition E fires. E can only fire when signal k is asserted.

Now suppose the scalechart of Fig. 1 is instantiated at time t_0. Then according to the semantics in [1], we can guarantee that at some time $t_1 > t_0$ the top-level scalechart will be in state A, orthocomp B will be in state D, and orthocomp C will be in state G. Because G has an enabled out-transition H, it is possible for C to transit to state I and similarly to state K, in which k is asserted, allowing B to transit to state F, which in turn allows C to transit back to state G.

Using the $<$ relation to mean "happened before" (as in [3]) and the *co* relation to mean "happens concurrently with" (as in [4]), and using the convention of expressing occurrence o of signal s as s_o, we can express this situation as

$$\begin{aligned} g_1 < k_1 < g_2 \quad &\wedge \quad d_1 < f_1 \\ g_1 \text{ co } d_1 \quad &\wedge \quad d_1 \text{ co } k_1 \\ k_1 \text{ co } f_1 \quad &\wedge \quad f_1 \text{ co } g_2 \end{aligned}$$

Note that some of the inequalities are independent of the signal-occurrences, for example, within the orthocomp B it is clear that $d_1 < f_1$, and within orthocomp c we can say that $g_1 < k_1 < g_2$, and in general $g_i < k_i < g_{i+1}$ for all $i > 0$.

2 An Extraction Algorithm for Inequalities

We assume that all nodes have a signal expression consisting of one signal. We also ignore input signals completely. This means that a transition is enabled whenever its from-state is active.

The extraction algorithm works as follows: we first build the underlying directed graph from the statechart we are analyzing. We then identify the *cycles*[1] and *parallel trails*[2] of this graph.

Once we have discovered the cycles and parallel paths, it is a simple matter to mark nodes which are internal to parallel paths, nodes which are members of a cycle and nodes which are start/end nodes of a cycle. Once these nodes are identified, it is possible to generate the inequalities.

We can discover all the parallel trails and cycles of a rooted directed graph by growing all the possible trails from the root. We start with the trail consisting of the root vertex only (this is a trivial trail). We then add each outgoing arc from the root and its to-vertex, to form a new trail. We can continue this process with the end-vertex of each new trail, until we cannot find an outgoing arc that we can add, without breaking the rule that a trail does not have duplicate arcs. As the directed graph has a finite number of arcs, this process is bound to terminate.

[1] a cycle is a circuit of the graph which contains no other circuit

[2] a trail is a walk through the graph which contains no repeated arcs, though it may contain repeated nodes

Finding the cycles We grow all the trails we can, starting from the root. When a trail grows to a vertex it has seen before, it contains a cycle. We therefore mark this trail as mature, and stop it growing any more. We continue this way until all our trails have stopped growing (either because they have matured, or because they have run out of arcs). Then we extract the cycle from each mature trail.

Finding the parallel trails We define two trails as being *parallel* if:

- they are both non-trivial
- they have the same start vertex and the same end vertex
- they have no other shared arc or vertex
- they do not contain any circuits at their start or end vertices (unless the start and end vertices are identical). They may contain circuits at other vertices.

We grow all possible trials from the root as before. When we find two distinct trails which share an end vertex, we have two candidate parallel trails. Because we are allowing cycles in trails, we cannot assume that our two candidates are genuinely parallel: both, one or neither of the trails may have a cycle on its end vertex. We discard any pair where only one trail has a cycle on its end vertex.

We now remove the common initial sub-trail of the two trails (this may be null), giving two non-trivial trails with the same start vertex and the same end vertex. Then we reject any pairs which have common internal nodes or transitions. Finally, for trails with distinct start and end vertices, we check that there are no cycles on the start vertex or the end vertex of our pair.

3 Conclusions

We would like to extend the algorithm informally presented here to include transitions with signal labels and guards. Nevertheless the current algorithm shows that it is possible to extract proofs semi-automatically from scalecharts. It is my intention to incorporate this work into a scalechart drawing tool, which would check out designs as they are drawn, and correct errors "on the fly".

Acknowledgements My indebtedness to Harel, Shields and Lamport is clear from the text. I wish to thank Jean Flower, John Howse and Gem Stapleton for crucial help in rescuing this paper from its initial vague and error-laden state.

References

1. Bosworth, R.: Scalecharts - Self-similar Statecharts in a Dense Time Domain. To be published in Software and Systems Modelling. Springer-Verlag, Berlin Heidelberg New York (2004)
2. Harel, D.: Statecharts: a Visual Formalism for Complex Systems. Science of Computer Programming **8** (1987) 231-274
3. Lamport, L.: Time, Clocks and the Ordering of Events in a Distributed System. Comm.A.C.M. **21** (1978) 558-565
4. Shields, M.W.: Semantics of Parallelism. Springer-Verlag, Berlin Heidelberg New York (1997)