Statechart Verification with iState

Dai Tri Man Le, Emil Sekerinski, Scott West

McMaster University, Hamilton, Ontario

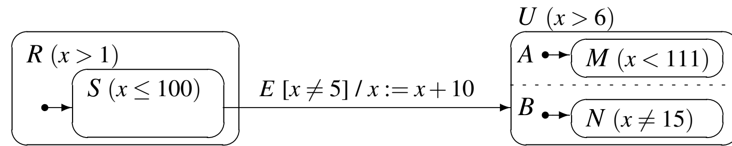
1 Introduction

The statechart formalism, proposed by Harel [6] as an extension of conventional finite state machines, is a visual language for specifying reactive systems. It addresses the state explosion problem of state transition diagrams when modeling systems with parallel threads of control by introducing the concepts of *hierarchy*, *concurrency*, and *communication*.

The *iState* tool translates statecharts into various programming languages, currently the Abstract Machine Notation (AMN) of the B method [1], Pascal, and Java. The translation is based on a definition of statecharts in terms of an extension of Dijkstra's guarded commands [10, 11]. This work demonstrates a novel statechart verification approach using *state invariants* that has been added to *iState*.

2 Invariants

Statecharts allow executable specifications to be derived from user requirements. We propose to supplement a statechart specification by *invariants*. These are attached to states and specify what has to hold in a state configuration. Invariants are also derived from the requirements. They are not meant for execution, but they allow the statechart specification to be cross-checked. By themselves, statecharts do not lead to opportunities for consistency checks beyond well-formedness; invariants address this limitation and give a way of documenting the "purpose" of states. Formally, invariants are predicates over global variables, like x in the example below, and states (state tests):



The definition of statecharts in [10, 11] translates states into variables and events into (nondeterministic) operations, in which use of the *independent (parallel) composition* of statements is made; the parallel composition operator is essential for translating events with transitions in concurrent states. Using AMN, the states of the previous statechart are translated to variables $root \in \{R, U\}$, $r \in \{S\}$, $a \in \{M\}$ and $b \in \{N\}$ and the event E is translated to:

$$\begin{aligned} E \triangleq\ & \textbf{if}\ \ root = R \wedge r = S \wedge x \neq 5\ \ \textbf{then} \\ & \quad x := x + 10 \parallel root := U \parallel a := M \parallel b := N \\ & \textbf{end} \end{aligned}$$

Let *si*: *State* → *Condition* be a function that assigns to each state the invariant specified by the designer, or *true* if none is specified, together with a test for being in that state. For example:

$$si(S) = (r = S \wedge x \leq 100)$$
$$si(U) = (root = U \wedge x > 6)$$

By the hierarchical structure of statechart, being in a state also means being in all of its ancestor states, in exactly one of its child states if the state is an XOR state, and in all of its child states if the state is an AND state. Hence, we have to compose state invariants together to create the *accumulated invariant* $ai(s)$ of state s. For example:

$$ai(S) = (root = R \wedge x > 1) \wedge (r = S \wedge x \leq 100)$$
$$ai(U) = (root = U \wedge x > 6) \wedge ((a = M \wedge x < 111) \wedge (b = N \wedge x \neq 15))$$

Formally, let *Basic*, *XOR*, *AND* be disjoint subsets of the set *State*. The accumulated invariant *ai*: *State* → *Condition* is defined with the help of the *child invariant ci*: *State* → *Condition* as follows:

$$ci(s) \triangleq \begin{cases} si(s) \wedge \dot{\bigvee} ci[children[\{s\}]] & \text{if } s \in XOR \\ si(s) \wedge \bigwedge ci[children[\{s\}]] & \text{if } s \in AND \\ si(s) & \text{if } s \in Basic \end{cases}$$
$$ai(s) \triangleq \bigwedge si[parent^{+}[\{s\}]] \wedge ci(s)$$

Here, $children[\{s\}]$ denotes the set of all child states of a state s and $parent^{+}[\{s\}]$ denotes the set of all ancestor states of s, where *parent* is the inverse of the *child* relation, $parent = child^{-1}$ [10, 11]. The operator $\dot{\vee}$ stands for *xor*. The definition reflects the meaning of XOR and AND states.

3 Invariant Verification

For each transition $E[guard]/action$ from state S to T, where $action$ is a statement that may read and write to global variables, may include state tests, and may broadcast other events, a verification condition is generated:

$$\{ai(S) \wedge guard\}\ action\ \{ai(T)\}$$

In the case of broadcasting in *action*, the broadcast is replaced by a call to the corresponding operation. In the case of transitions in concurrent states on the same event E, a combined transition is considered. The generation of the verification conditions then follows the same structure as the generation of the code in [11]; the details are beyond the scope of this paper. In the example, the verification condition for event E is:

$$\{(root = R \wedge x > 1) \wedge (r = S \wedge x \leq 100) \wedge x \neq 5\}$$
$$x := x + 10 \parallel root := U \parallel a := M \parallel b := N$$
$$\{(root = U \wedge x > 6) \wedge ((a = M \wedge x < 111) \wedge (b = N \wedge x \neq 15))\}$$

4 Implementation

The *iState* tool currently uses the Simplify theorem prover [5] to discharge the generated verification conditions because of its support of first order logic and linear arithmetic. Simplify also has arrays built in, though currently *iState* does not use them. We are working on extending *iState* with data types like arrays, rational numbers, and real numbers. Once this is completed, we will make *iState* available for downloading. In future, we also plan to extend the verification theory to timed transitions [9].

5 Discussion

Compared to the statechart verification approaches in [3, 4, 8], we use an *event-centric* semantics of statecharts by looking at events as *operations* rather than *data* as in the original *state-centric* semantics [7]. Instead of writing global temporal specification (say in CTL or LTL) separately, inspired by *nested invariant diagram* [2], invariants (*safety properties*) are attached to states.

By attaching invariants to states and utilizing the guarded command representation of statecharts [10, 11], we arrive at a rather straightforward verification method. The approach generating verification conditions leads to many small "local" verification conditions and avoids some impossible configurations, compared to when specifying invariants on the global level. As many small verification conditions are easier to handle automatically than a few large ones, we believe that the approach can more easily scale up for the verification of large systems.

References

1. J.-R. Abrial. *The B Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.
2. R. Back. Invariant based programming revisited. Technical Report 661, TUCS - Turku Centre for Computer Science, Turku, Finland, 2005.
3. P. Bhaduri and S. Ramesh. Model Checking of Statechart Models: Survey and Research Directions. *ArXiv Computer Science e-prints*, July 2004.
4. E. Clarke and W. Heinle. Modular translation of statecharts to SMV. Technical Report CMU-CS-00-XXX, School of Computer Science, Carnegie Mellon University, August 2000.
5. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
6. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
7. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(5):293–333, 1996.
8. E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in PROMELA/SPIN. WIFT, Los Alamitos, CA, USA, 1998. IEEE Computer Society.
9. S. Samet. Timed transitions in statecharts, from formalization to translation and code. Master's thesis, McMaster University, Computing and Software Department, 2005.
10. E. Sekerinski and R. Zurob. iState: A statechart translator. In M. Gogolla and C. Kobryn, editors, *UML 2001*, LNCS, pages 376–390, Toronto, Canada, 2001. Springer-Verlag.
11. E. Sekerinski and R. Zurob. Translating statecharts to B. In M. J. Butler, L. Petre, and K. Sere, editors, *IFM 2002*, LNCS, pages 128–144, Turku, Finland, 2002. Springer-Verlag.

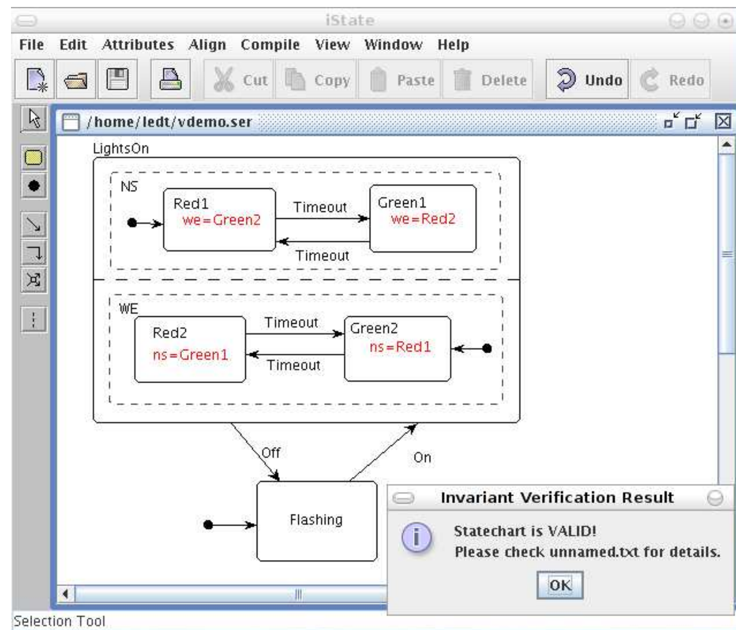
Appendix

The demonstration includes a poster and two prepared examples. The poster explains the translation of statecharts to code in more detail than the current paper, as well as verification of transitions in concurrent states that has been left out from this paper.

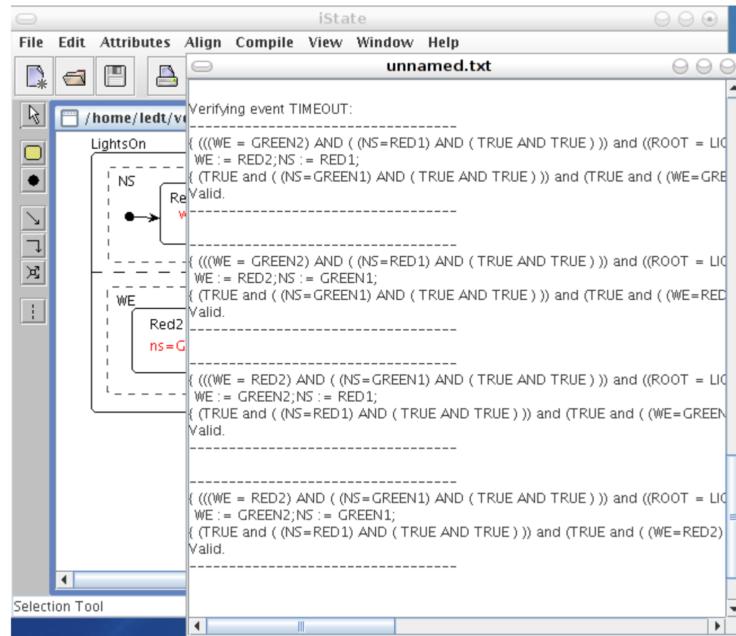
One example is a simple traffic light. The example is easy to grasp and allows the generation of code and of the verification conditions to be illustrated. The second example is the control of luxury sedan car seat. The seat has controls which modify the position of the headrest, seat-base, seat-back, and height controls with front-back up-down adjustments. Additionally, the controls include memory functions which store the settings of the seat to be restored at a later time. The controller has to observe that certain movements cannot occur at the same time and that movements cannot extend past limits. Also, there are calibration procedures that must take place after power is supplied, or there is a break in power. The example is not trivial. The demonstration will show that some of the requirements are easily overlooked in a statechart implementation and that verifying invariants can help detect these violations.

The demonstration will also show how iState works overall, how iState checks the structural validity of statecharts, and how statecharts can be animated. We plan to make iState available for downloading by August.

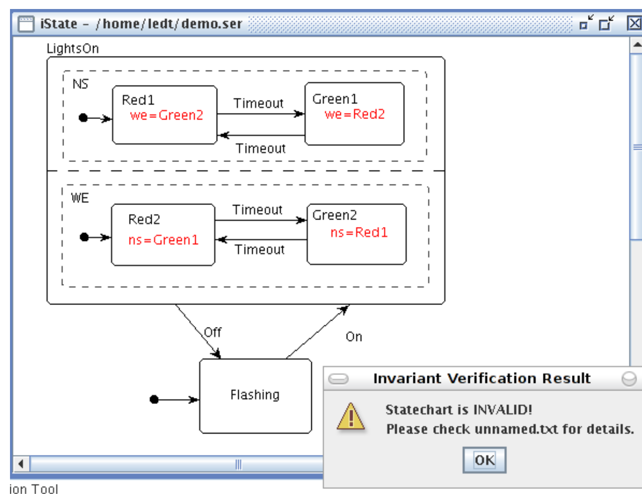
Some screenshots of a traffic light example are given below. We first draw a correct statechart. After using the verification functionality of the *iState* tool, a message box says the statechart is *valid* with respect to invariants attached:



A text area displays the detailed verification results:



We next draw an incorrect statechart by specify the wrong initial state in the WE state, i.e. we begin with a bad state configuration where red lights simultaneously appear in both WE and NS states. A message box says the statechart is *invalid* with respect to invariants attached:



The *iState* tool provides a text area (not shown here) displaying the detailed verification results as in the previous case.