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):

\[
R (x > 1) \quad S (x \leq 100) \quad E [x \neq 5] / x := x + 10 \quad U (x > 6)
\]

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:

\[
E \triangleq \begin{cases} 
\text{if } \ root = R \land r = S \land x \neq 5 \ 	ext{then} \\
\quad x := x + 10 \parallel \ root := U \parallel a := M \parallel b := N \\
\text{end}
\end{cases}
\]
Let \( si: State \to Condition \) 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:

\[
\begin{align*}
    si(S) &= (r = S \land x \leq 100) \\
    si(U) &= (\text{root} = U \land x > 6)
\end{align*}
\]

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:

\[
\begin{align*}
    ai(S) &= (\text{root} = R \land x > 1) \land (r = S \land x \leq 100) \\
    ai(U) &= (\text{root} = U \land x > 6) \land ((a = M \land x < 111) \land (b = N \land x \neq 15))
\end{align*}
\]

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

\[
\begin{align*}
    ci(s) &\triangleq \begin{cases} \\
        si(s) \land \bigvee_{s \in \text{children}}[s] & \text{if } s \in \text{XOR} \\
        si(s) \land \bigvee_{s \in \text{children}}[s] & \text{if } s \in \text{AND} \\
        si(s) & \text{if } s \in \text{Basic}
    \end{cases} \\
    ai(s) &\triangleq \bigwedge_{s \in \text{children}}[s] \land ci(s)
\end{align*}
\]

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

3 Invariant Verification

For each transition \( E[\text{guard}] / \text{action} \) from state \( S \) to \( T \), where \( \text{action} \) 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:

\[
\{ ai(S) \land \text{guard} \} \text{ action } \{ ai(T) \}
\]

In the case of broadcasting in \( \text{action} \), 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:

\[
\begin{align*}
    &\{ (\text{root} = R \land x > 1) \land (r = S \land x \leq 100) \land x \neq 5) \} \\
    &x := x + 10 \parallel \text{root} := U \parallel a := M \parallel b := N \\
    &\{ (\text{root} = U \land x > 6) \land ((a = M \land x < 111) \land (b = N \land x \neq 15)) \}
\end{align*}
\]
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