specgen: A Tool for Modeling Statecharts in CSP

Brandon Shapiro[1] and Chris Casinghino[2]

[1] Brandeis University, Waltham, MA 02453, USA
bts8394@brandeis.edu
[2] Draper Laboratory, Cambridge, MA 02140, USA
ccasinghino@draper.com

Abstract. We present specgen, a tool for translating statecharts to the Communicating Sequential Processes language (CSP), where they may be explored and verified using FDR, the CSP model checker. We build on earlier algorithms for translating statecharts to CSP by supporting additional features, simplifying the generated models, and implementing a practical tool for statecharts built in Enterprise Architect, a commercially available modeling environment. We demonstrate the tool on a standard example.

1 Introduction

Statecharts are a widely-used technique for graphically representing the high-level behavior of complex systems. Since their introduction by Harel [5], support for various versions of statecharts has been implemented in many commercial tools, including Enterprise Architect and Simulink Stateflow. As the use of statecharts has become widespread, so too have techniques for formally verifying their behavior. Classic examples include modeling via translation to SPIN [10] or to SMV [2].

This paper presents `specgen`, a tool for translating statecharts to Communicating Sequential Processes (CSP). This makes it possible to explore and verify the behavior of a statechart using FDR, the CSP model checker [4]. CSP and FDR have been used for modeling and formal verification for decades, in both academia and industry [8,11,9].

Translating statecharts to CSP has two main advantages. First, CSP is a rich, expressive language for writing specifications. We may leverage FDR to check these specifications and to interactively explore the behavior of the translated systems. Second, statecharts are themselves a convenient way to represent specifications for more complex systems already implemented in CSP. For example, the second author has also implemented a tool, called `cspgen`, to translate imperative programs from C source or LLVM IR to CSP [1]. The typical use of `cspgen` involves taking code written by a domain expert and translating it to CSP, then developing specifications to be checked by FDR. As the domain

This work was sponsored by DARPA/AFRL Contract FA8750-12-C-0261. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

expert is typically unfamiliar with CSP, statecharts provide an intuitive, graphical common language for these specifications. Having a tool like specgen to automatically convert the graphical specification to CSP makes this possible.

The specgen tool builds on previous work for modeling statecharts in CSP [12]. We have added support for several additional statechart features and designed a new, simplified algorithm by using new CSP language constructs, as described in Section 3. The tool supports statecharts developed with Enterprise Architect and is the first practical implementation of any such translation. The specgen distribution also includes several examples, described in Section 2, and is available freely under a permissive open-source license [14].

2 The Dining Philosophers: An Example

To illustrate the use of specgen, we consider the classic dining philosophers problem [7]. Our distribution of specgen includes this example, implemented as a statechart in Enterprise Architect, for 2, 3 and 4 philosophers [14]. Figure 1 shows statecharts representing Philosopher 2 and Fork 2 from the four philosopher system. We elide the full system for space—it consists of four philosophers and forks, similar to those shown, as parallel substates of one top-level node.

We begin our explanation with the statechart for Fork 2. Conceptually, it keeps track of which philosopher has permission to use the fork at any time. It begins in the state Free, indicating that the fork is not in use and may be claimed by either philosopher. Transitions to the Phil2Holds2 and Phil3Holds2 states are guarded by the constraints In(WaitingRight2) and In(WaitingLeft3) respectively. This ensures these transitions are not taken until the relevant philosopher is in

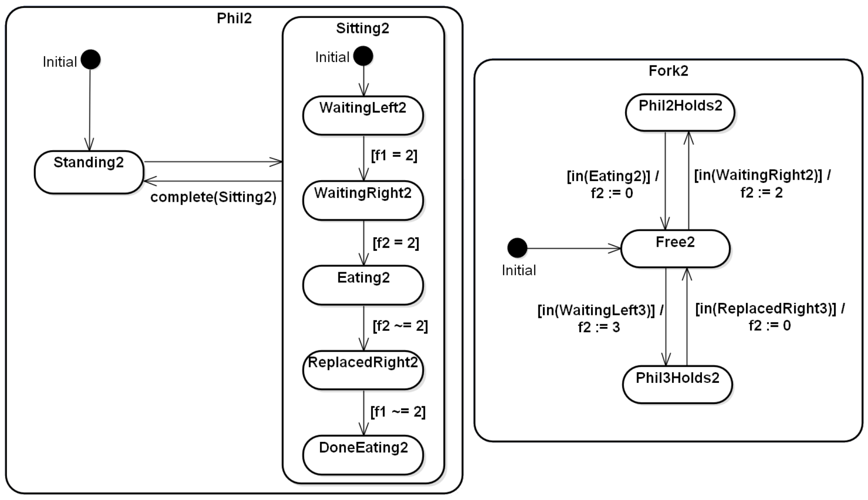

Fig. 1. Statecharts for one philosopher and fork

the state where he is waiting on this fork, so the ownership of the fork is not given to a philosopher until he wants it.

The system also includes four variables, f1, . . . , f4, one for each fork. Intuitively, the value in these variables indicates which philosopher, if any, currently has permission to use a given fork. Thus, the transition from state Free2 to state Phil3Holds2 sets variable f2 to 3. These variables are set by the forks, and used by guards in the philosophers. For example, consider node WaitingLeft2 in Phil2. This node models the state where Philosopher 2 is waiting to pick up his left fork (Fork 1). The guard on this transition prevents it from being taken unless $\mathsf{f1} = 2$, indicating that Philosopher 2 has permission to use Fork 1. Similarly, the transition from Eating2 to ReplacedRight2 is guarded by the requirement that f2 is not 2, indicating that Philosopher 2 no longer has permission to use his right fork. The semantics of statecharts require that all available transitions are taken immediately, ensuring that Fork 2 and Philosopher 2 remain synchronized here.

Finally, we consider the edge from Sitting2 back to Standing2, which is labeled with the *completion event* complete(Sitting2). In statecharts, *events* are named triggers that are often used to represent external events. During execution, a set of enabled events is provided as input, and an edge labeled with an event may only be taken if the event is currently enabled. *Completion events* are special events that are enabled when a node terminates, rather than by input. A node is considered to have terminated when all of its concurrent subnodes have reached states with no out-edges. Here, the event label prevents the philosopher from standing until he is done eating.

It is worth noting that this example is not intended to represent the most efficient or natural implementation of the dining philosophers as a statechart. Rather, we have designed it to highlight several features supported by the tool.

2.1 The Generated Model

When run on an Enterprise Architect statechart like the one described above, `specgen` produces several files containing CSP definitions, including a top-level process `RunSystem` that models the statechart's behavior. The behavior of a CSP process is most easily described by finite "traces" of observable events. In the case of `RunSystem`, the relevant observable events include:

- `transition.N.E`, indicating a transition between nodes. Here `N` is the name of the node that contains the transition, and `E` is the name of the edge itself. Typically, `specgen` will generate node names that match the name given in the statechart if all nodes have unique names, and will otherwise pick a name based on the full path of a node. Edges are given names like `Node1__Node2`, indicating a transition from `Node1` to `Node2`.
- `tock`, indicating the completion of a "step" of the statechart. According to the semantics of statecharts, a step comprises a single transition in every currently-running subchart that can make one.
- `read.x.n` and `write.x.n`, indicating reads or writes of a value `n` in variable `x`.
- `writeerror.x`, indicating that the statechart has a race condition where two parallel subcharts attempted to write to the variable `x` in the same step.

2.2 Finding the Deadlock

The most obvious property to check in the dining philosophers example is deadlock freedom. In our CSP scripts, this property is stated:

```
assert RunSystem \ {| tock |} :[deadlock free]
```

The \ ("hiding") operator here is used to hide the `tock` events of `RunSystem`. A statechate continues to take "steps", represented by these events, even if no subchart can make a transition. Intuitively, to detect the deadlock, we must inform FDR that the mere passage of time does not count as progress.

Asking FDR to check this property results in an assertion failure, as expected. Indeed, because the semantics of statecharts require each parallel process to make a transition in each step if able to, this system will always deadlock. FDR also displays the trace that leads to the deadlock. For the three philosopher system, this trace ends with the events:

```
transition.Sitting2.WaitingLeft2__WaitingRight2,
transition.Sitting3.WaitingLeft3__WaitingRight3,
transition.Sitting1.WaitingLeft1__WaitingRight1
```

We see that the last three events are each philosopher transitioning to his `WaitingRight` node, indicating that each philosopher has picked up his left fork and is waiting on his right fork.

2.3 More Complicated Properties

FDR, more generally, supports checking *refinement* between two CSP processes. This enables the use of CSP as a rich specification language for properties more interesting than deadlock. Our distribution of `specgen` includes many worked examples. For the dining philosophers system in particular, we show how to verify that changing the order in which a philosopher picks up his forks eliminates the deadlock, and include a detailed explanation of how to check the property "after sitting, no philosopher stands without eating". We also show how to check for race conditions in variable writes, and include several other statecharts to demonstrate a variety of properties.

2.4 Performance

The time to find the deadlock in FDR is summarized in the table below, organized by the number of philosophers in the system:

Philosophers	2	3	4
Time	2.0s	6.0s	117s

These times are the averages of 5 runs performed on an Intel Xeon E5-2630 v3. The machine had 32GB of RAM, but all tests consumed less than 6GB.

Predictably, the time to find the deadlock grows exponentially with the number of philosophers. Checking these translated statecharts is slower than checking more natural implementations of the dining philosophers in CSP, because accurately modeling the semantics of statecharts involves substantial coordination overhead and additional features like per-node timers. As statecharts offer the advantage of wider accessibility, we believe this overhead is sometimes justified.

3 Translation Enhancements

As mentioned in the introduction, `specgen` builds on an earlier algorithm for modeling statecharts in CSP, by Roscoe and Wu [12]. In addition to providing a practical implementation, we have improved on that paper's translation by including support for two additional statechart features (the "in" guards and completion events described in Section 2) and exploiting a newer FDR feature to simplify the generated models. The remainder of this section describes this simplification.

The biggest challenge in modeling statecharts in CSP is representing *priority*. In CSP, a process may select freely among its available actions, but in statecharts certain transitions may be favored over others. For example, nodes must be allowed to take an "idle" step if and only if no transitions are available. Also, transitions out of a state may be favored over transitions within that state when both are available, or vice versa—classic Statemate semantics [6] favor outer transitions while UML favors inner ones [3]. (In `specgen` we have followed [12] in modeling Statemate, but it would be straightforward to prefer the alternate order, which is more common today).

Roscoe and Wu's translation models these instances of priority with a subtle renaming and synchronization scheme [13]. Happily, modern versions of FDR include a new feature that `specgen` uses to simplify this: `prioritise`. This function takes as arguments a process `P` and an ordered list `evs` of sets of events. If `P` may perform events from different sets in `evs`, then `prioritise(P,evs)` may perform only events from the first set that contains any of `P`'s events. Combining `prioritise` with *interrupts*, where a CSP process may be preempted by certain events, also allowed for a simplified encoding of "promoted" actions in statecharts. These actions allow an inner node to transition directly to an outer node, terminating its parallel siblings.

4 Conclusion and Future Work

This paper has described `specgen`, a tool for translating statecharts to CSP. We demonstrated the use of the tool on a common example, illustrating how to analyze the behavior of a statechart by model-checking its translation with FDR (Section 2). Many more examples are available with the `specgen` distribution, which is available as open-source software [14]. The translation used by the tool is inspired by earlier work by Roscoe and Wu [12], which has been improved and extended (Section 3).

We are interested in expanding on this work in several directions. First, the generated model can likely be further optimized for model-checking speed in

FDR. In particular, the use of *inductive compression* [13] to reduce the state space created by hidden control events seems particularly promising. Second, it would be interesting to compare our tool directly with other systems for verifying statecharts. Lastly, while the translation is intended to faithfully model one version of statechart semantics, it would be reassuring to formalize and mechanically verify this property with an interactive theorem prover like Coq or Isabelle/HOL.

While `specgen` is intended as a prototype, we have found it to work surprisingly well on a variety of examples. Readers are encouraged to download the implementation and give it a try.

Acknowledgments The authors thank Neil Brock, Thomas Gibson-Robinson, Colin O'Halloran and Cody Roux for their advice on this project, and the anonymous reviewers for their helpful feedback.

References

1. Casinghino, C.: `cspgen`. `https://github.com/draperlaboratory/cspgen` (2016)
2. Chan, W., Anderson, R.J., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J.D.: Model checking large software specifications. IEEE Transactions on Software Engineering 24(7), 498–520 (Jul 1998)
3. Eshuis, R., Wieringa, R.: Requirements-level semantics for uml statecharts. In: Fourth International Conference on Formal Methods for Open Object-based Distributed Systems. pp. 121–140. Kluwer Academic Publishers (2000)
4. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3: A parallel refinement checker for CSP. International Journal on Software Tools for Technology Transfer (2015)
5. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3) (1987)
6. Harel, D., Naamad, A.: The statemate semantics of statecharts. ACM Transactions on Software Engineering and Methodology 5(4), 293–333 (Oct 1996)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Inc., Upper Saddle River, NJ, USA (1985)
8. Lawrence, J.: Practical Application of CSP and FDR to Software Design, pp. 151–174. Springer Berlin Heidelberg (2005)
9. Lowe, G.: Casper: A compiler for the analysis of security protocols. Journal of Computer Security 6(1-2), 53–84 (1998)
10. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing Statecharts in PROMELA/SPIN. In: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques. IEEE Computer Society (1998)
11. Mota, A., Sampaio, A.: Model-checking CSP-Z: Strategy, tool support and industrial application. Science of Computer Programming 40, 2001 (2001)
12. Roscoe, A.W., Wu, Z.: Verifying Statemate Statecharts Using CSP and FDR. In: Proceedings of ICFEM 2006 (2006)
13. Roscoe, A.: Understanding Concurrent Systems. Springer-Verlag New York, Inc., 1st edn. (2010)
14. Shapiro, B., Casinghino, C.: `specgen`. `https://github.com/draperlaboratory/specgen` (2016)