Cogent: Code generation from statecharts for real-time reactive code

Theodore S. Norvell
Department of Electrical and Computer Engineering
Memorial University of Newfoundland
theo@mun.ca

***Abstract*—David Harel's statecharts, which have been standardized as UML's StateMachine diagrams, provide an excellent way to describe the high-level behaviour of reactive, real-time, concurrent systems.**

However, manual translation of diagrams to code in an ordinary programming language can be error prone and lead to code that is hard to read. Furthermore, once diagrams are manually translated to code, keeping the diagrams and code consistent as the software evolves is a doomed cause.

This paper presents a diagrams-as-code approach that starts with a modular, textual description of statecharts using the widely used PlantUML notation. From that description, the existing PlantUML tool generates a set of diagrams and a new tool, Cogent, generates code in the C programming language. A further benefit is that we get non-preemptive concurrency and deterministic behaviour.

CONTENTS

I. INTRODUCTION

We present a tool for translating statecharts to space efficient code for embedded systems.

This paper presents a quick introduction to statecharts and the problems with using statecharts, describes the tool built, and concludes with some reflections on the project.

II. STATECHARTS

Statecharts were invented by David Harel in the 1980s [1]. In the 1990s they were incorporated into the UML standard under the name "StateMachine Diagrams" [2].

Statecharts extend ordinary finite state machines in two ways:

1) Hierarchy. States can contain substates. This allows common transitions to be shared.
2) Concurrency. A state can contain multiple regions, each of which contains a set of states. All regions on a state operate concurrently.

The combination of hierarchy and concurrency allows very complex systems to be modelled far more concisely that with ordinary state machines.

Each *statechart* has one *region* called its root region. Each region contains a set of *states*, one of which is designated its *initial state.* Each state can contain a set of zero or more regions. Thus the regions and states with the relation of containment form a bipartite graph which is a rooted tree. This tree is finite and contains all the states and regions of the statechart.

A configuration of a statechart is a set of states and regions with the following properties:

1) The root region is in the configuration.
2) If a region is is in the configuration, exactly one of its children is in the configuration.
3) If a state is in the configuration, all its children are in the configuration.
4) If a state or region is in the configuration, all its ancestors are in the configuration.

Since the regions of a configuration can be inferred from the states, I won't list regions in configurations.

Figure 1 shows an example statechart. The root region contains states A and B. State A contains one region with two states. State B contains two regions, each with two states. The remaining six states have no regions. There are

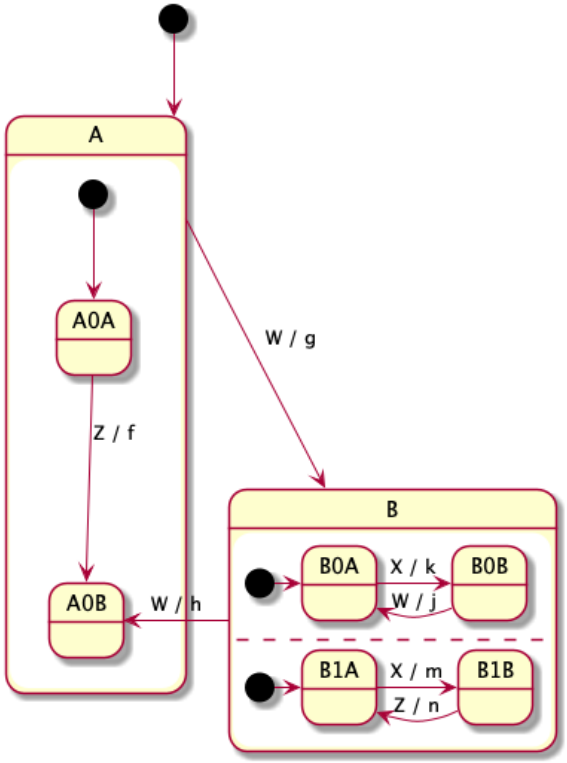

Fig. 1. A Statechart

six configurations. The initial configuration is $\{A, A0A\}$, with the root region and the region in A being implicitly included. The black disks are not states, but serve to indicate the initial states.

When an event occurs, the statechart transitions from one configuration to another. Edges in the statechart travel from one state (its *source*) to another (its *target*). When an edge is followed, the source state and all its ancestors up to the least common ancestral region are removed from the configuration; the target state is added along with all its ancestors up to the least common ancestral region. In order that the new configuration follows the rules above, if a state is removed from a configuration, so are all its regions and if a state is added to a configuration, so are all its children.

Edges leaving a state are labelled with triggers, guards and actions. In Fig. 1, there are no guards. Triggers are shown as capital letters, X through Z; each trigger describes a set of events. Lower case letters represent actions; actions describe actions to be executed. Suppose a sequence of events W, X, W, Z, W, X occurs. The sequence of configurations will be:

$$\begin{aligned}
\{A, A0A\} &\xrightarrow{W} \{B, B0A, B1A\} \\
&\xrightarrow{X} \{B, B0B, B1B\} \\
&\xrightarrow{W} \{B, B0A, B1B\} \\
&\xrightarrow{Z} \{B, B0A, B1A\} \\
&\xrightarrow{W} \{A, A0B\} \\
&\xrightarrow{X} \{A, A0B\}
\end{aligned}$$

The first X event shows concurrency. In this case both the transition out of $B0B$ and that out of $B1B$ are followed as they these states are in separate regions of state B. The second W event shows an example of pre-emption. Although there is a W transition out of B, the transition from $B0B$ has priority as $B0B$ is a descendent of B. The final X event shows that events are ignored if no transition out of a state that is in the configuration applies. The sequence of events implied is $g; (k \parallel m); j; n; h$. Conceptually k and m execute concurrently, but one interpretation of this is that they execute sequentially, but in an undefined order. This is the interpretation that we will take and it allows statecharts to express concurrency while running in a single underlying thread.

UML defines a number of other features of statecharts, some of which will be discussed later.

III. The problems with statecharts and their mitigation

Statecharts are understood to be useful in modelling and designing the behaviour of event-driven systems, such as embedded systems.

The usual approach to designing with statecharts is to use statecharts for high-level design, but then to manually translate statecharts to code. This approach has a few problems:

1) Translation to code is a tedious and error prone task.
2) The code will be hard to read compared to the diagrams, meaning that the statecharts should be kept as crucial design documentation.
3) Thus, as the design evolves, the statecharts and the code need to be maintained in parallel and kept consistent. This is time-consuming and error prone.

A separate problem is that if the statecharts are created using a graphical tool such as Visual Paradigm or Draw.io, it can be hard to understand changes to the diagram as the software evolves, as diagram differencing is not straight-forward and rarely, if ever, supported; at the same time it is important to check that changes to the diagrams and changes to the source code are concordant.

Issues related to manual translation can be overcome by using translation software to translate diagrams to source code in a conventional language. Several tools and approaches to translating statecharts have been implemented or proposed.

Issues related to difficulty in visualizing differences can be mitigated by using a text based description of the statecharts —

an example of "diagrams as code", although "diagrams as text" might be a better term. This allows common text differencing software, such as 'git diff' to be used to highlight the differences between versions. It also allows revision control systems such as git to merge independent changes automatically.

Of course using a text-based description alone means losing much of the appeal of statecharts, as a graphical representation allows the brain's visual processing system to aid in comprehension; therefore it goes almost without saying that the textual descriptions will be used to generate diagrams. I think this is only a partial mitigation because looking two diagrams and the differences between the two text files that describe them is cognitively demanding. Visualizing merges of changes requires comparing three or four versions of a file and is even worse. A graphical tool to show how a diagram has changed from one version to another or how two diagrams were merged to make one would be useful.

There are a number of textual languages that can generate diagrams from textual descriptions [3], [4], [5]. Among these, the PlantUML language [3] is particularly interesting; rather than using one language for many types of diagrams, PlantUML is more of a collection of domain specific languages, each intended for describing a particular category of diagrams; PlantUML includes a sublanguage specifically for describing UML statecharts.

Combining code generation with "diagrams as code" we get "code as diagrams as code", i.e. we replace writing source code with drawing diagrams and replace drawing diagrams with describing them with text.

Listing 1. Generated code

```
1  bool_t dispatchEvent_figs ( event_t *event_p,
2          TIME_T now ) {
3      bool_t handled = false ;
4      switch( currentChild_a [ G_INDEX_root] ) {
5      case L_INDEX_A: {
6          bool_t handled_A = false ;
7          code for A's region
8          if ( ! handled_A ) {
9              switch( eventClassOf(event_p) ) {
10             case EVENT(W) : {
11                 status_t status = OK_STATUS ;
12                 handled_A = true ;
13                 exit_A( -1 ) ;
14                 status = g( event_p, status ) ;
15                 enter_B( -1, now ) ; } } }
16         handled_Root = handled_A ; }
17     break ;
18     case L_INDEX_B: {
19         bool handled_B = true ;
20         code for one region
21         code for other region
22         if ( ! handled_B ) {
23             event handling code for state B }
24         handled = handled_B ; } }
25     return handled ;
26 }
```

IV. The Cogent tool

In developing software for the Killick-1 satellite at Memorial University and C-Core [6], we faced a complex system that clearly needed some concurrency. For example, we would need to control the attitude of the satellite concurrently with data collection. Statecharts seemed a good solution for design, but the problems mentioned in the previous section served as a deterrent. After some searching, PlantUML appeared to be a good solution for generating diagrams, but we were not able to identify a good existing solution for automatically generating the sort of code that we needed, i.e., code in the C programming language that used very little RAM. I volunteered to create a code generator and gave it the uninspired name Cogent.

Cogent uses a subset of the PlantUML language as its input language; PlantUML allows any text at all as labels on edges, but Cogent parses this text and requires it to follow a particular syntax consisting of three parts, each optional: a trigger, a guard, and a sequence of actions. A trigger is either the name of a class of events or time trigger (described later); a guard is either a boolean expression or the keyword "else". An action is is simply a name. Aside from checking for

The C code generated consists of two procedures representing a state machine. One for initializing and one for processing events. The code that produces events and that decides when to feed an event to the generated state machine is not generated. These procedures depend on an call procedures that must be written conventionally elsewhere. These depended-on procedures are of two sorts: Some represent atomic guards and others represent actions.

For each region, we generate a switch statement to branch to the code of a child state. For each state, we generate:

- Code for each region that is a child of the state.
- A switch statement that jumps to the code for each class of event handled by the event. This entire switch is skipped if the event has been handled by any descendent state. The switch statement branches to the code for the event class. When there are guarded edges, this will be an if-statement; otherwise, it is code that implements the transition. The code for handling a transition exits any states that need to be exited, calls the user-provided procedures representing actions and then enters any states that need to be entered.

Listing 1 shows some of the generated event handling code (slightly cleaned up) for Fig. 1.

The switch from line 4 to line 24 shows how a region is translated. I've glossed over the code for other regions, as it is similar. The cases from lines 5 to 16 and 18 to 24 show how states are translated. The code from line 11 to line 15 shows how a transition is translated.

Each state and region (with siblings) has an enter and exit

routine. The exit routine for a state exits all its regions except for perhaps one; an exit routine for a region optionally exits its current active state. The entry routine for a state enters all its region except for perhaps one; an entry routine for a region optionally enters its initial state. The reason for the optionality is that the required entry or exit may already have been done or soon will be done. For example, a hypothetical transition from $A0A$ to $B0B$ would be coded as.

```
1    exit_A0A(−1, now) ;
2    exit_A( L_INDEX_A01, now ) ;
3    enter_B( L_INDEX_B_region_0 ) ;
4    enter_B_region_0( L_INDEX_B0A ) ;
5    enter_B0B( −1 ) ;
```

The first argument to exit_A indicates that there is no need to exit from its region's current state; this has already happened. (Exiting A's sole region is combined with exiting A.) The argument to enter_B indicates that it's first region should not be entered; all other regions will be entered. The argument to enter_B_region_0 indicates that its initial state should not be entered. One effect of the call to enter_B will be to enter its second region, this call to enter_B_region_1 will have an argument of -1 indicating that its initial state should be entered. Compare this to the call to enter_B on line 15 of Listing 1 which an argument of -1 meaning that all regions should be entered at their initial state.

You can see from Listing 1, that, if B is in the current configuration, its two regions each process the event; in this way events are broadcast and we get a sort of concurrency as multiple regions can process the same event. Transitions are atomic in this model of concurrency. Only if neither region has reacted to the event will B itself get a chance; this implements pre-emption.

In terms of global state, we need a static array with one entry for each region to keep track of the current state, a static boolean array with one entry per state to keep track of whether a state is in the current configuration or not. The demand for stack RAM amounts to having a boolean variable for each state that contains a region to note whether or not the current event has been handled by that state or not. These variables can be locally scoped so that the space requirement is proportional to the depth of nesting.

The Cogent approach tries to give the user as much flexibility as possible; in return some definitions need to be provided. For example, the event_t type is not defined by Cogent's generated code, but in return the user should provide a definition for the 'eventClassOf' and 'EVENT' macros.

A. Other constructs

Here we will look at how Cogent handles most of the many features UML statecharts. More information on these features can be found at [2].

1) Time triggers: A trigger can be of the form "after(n ms)" where n is an integer. In this case the transition is triggered when the state has been in the current configuration for at least n milliseconds. This requires an additional static array to note the last time each state was entered. A special kind of event called a TICK event is used for these triggers. It is up to the code calling the event handler to pass in the current time. Cogent imposes no notion of time on the programmer other than that it can be converted to milliseconds. It's up to the writer of the event loop to ensure that TICK events are sent frequently enough that the state machine meets whatever real-time requirements the system has.

2) Guards and choice pseudostates: Transitions can be guarded with boolean expressions. The atoms of these boolean expressions are translated to calls to boolean functions which must be supplied by the user.

Choice pseudostates are nodes where a transition can branch. All the edges out of a choice pseudostate should be guarded and must not have triggers. With choice states, a transition from one state to another may involve a sequence of edges; these are called compound transitions. Cogent supports choice pseudostates.

3) else guards and 'in' guards: Cogent support both of these.

4) Send actions: Cogent does not directly support send actions. Events can be created and queued from within the code of actions. Cogent will translate an action written "!name" to "send_name" and so one can use this naming convention to indicate which actions create new events.

5) Submachines, entry points, and exit points: UML provides a way to break large statecharts into smaller parts called submachines. Support for submachines is essential to handle large applications, and Cogent supports submachines essentially as macros. Each submachine is defined in a separate diagram. Entry points and exit points are pseudo states on the boundary of a state. They allow compound transitions to cross state boundaries without edges crossing state boundaries. As such they are often used with submachines.

6) Entry, exit, internal, and do actions: None of these are currently supported, but support should not be difficult.

7) History, deep history, fork, join, and junction, pseudostates and final states: None of these features are currently supported. History and deep history should not pose a problem, as the current state of a region is remembered even when the region itself is not part of the current configuration. Fork and join are of limited use owing to PlantUML limitations. Junction pseudostates are not supported by PlantUML. Final states require completion events to be generated by the generated code, which is somewhat at odds with our current approach of leaving event creation to non-generated code.

B. Implementation

Cogent is implemented in Scala 3 using a largely functional style. The translator has the following passes:

1) Parser: Parsing is done using components of PlantUML as a library.
2) Middle end: The data structure produced by UML is translated into an abstract syntax tree. During this pass edge labels are parsed.

3) Submachine expansion. Occurrences of submachine states are replaced with their definitions.
4) Checking: A number of checks are made to ensure that the statechart can be further translated.
5) Back end: The abstract syntax tree is traversed to produce C code.

Because of this structure, targetting languages other than C should be straight-forward. Furthermore, a different input language could be used by replacing the first two passes with a different parser.

The code generated is a little verbose, but its size essentially linear with respect to the size of the statechart after submachine expansion. In our project code size is not a limiting constraint.

V. Conclusion

A. Related work

Converting statecharts to other languages is hardly a new idea. Commercial tools have been available since the 1990s. Recent academic projects have targetted object oriented languages and so were not of use to us. A summary of these efforts up to 2012 can be found in [7]. A quick survey did not turn up any tools suitable for our use case.

B. Reflections

I was a bit hesitant to go the route of generating code from statecharts.

The first is that it required students working on the Killick-1 project to learn a new language and a new paradigm; this could also be seen as a positive. The students who have needed to have learned to use statecharts effectively.

The second is that information must be passed between actions and guards must be stored in static storage, something that programmers are normally discouraged from doing; with a multithreaded system such state could be stored in stack variables and passed around via subroutine parameters. By insisting that all static variables are local to a single module, this disadvantage was somewhat mitigated.

The third was doubt that the time spent on implementing Cogent could have been better spent elsewhere. It turned out that the implementation effort didn't take a great deal of time. About 2,800 lines of implementation code and 300 lines of test code have been written, including comments.

On the positive side: It avoids the need for multiple threads, which have their own disadvantages. It makes it clear which actions will be done atomically and which will not. It makes a very clear layer boundary between the statechart code and the C code. In our project, the statechart diagrams are linked into the documentation for the project and so at least these diagrams are always up to date.

Cogent is available for use by others at: https://github.com/theodore-norvell/cogent

C. Acknowledgements

I'd like to thank the Canadian Space Agency who have funded the Killick-1 satellite project in part, Desmond Power of C-Core for leading the project, and the many many students and co-op students who have contributed to the project.

References

[1] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, 1987.
[2] C. Bock, S. Cook, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, and D. Tolbert, *OMG Unified Modeling Language, Version 2.5.1*. Object Management Group, 2017.
[3] "Plantuml," Website, Accessed 2023. [Online]. Available: https://plantuml.com/
[4] "Mermaid," Website, Accessed 2023. [Online]. Available: https://mermaid.js.org/
[5] J. Ellson, E. R. Ganser, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph — static and dynamic graph drawing tools," in *Graph Drawing Software*. Springer, 2004.
[6] A. Quadri and G. Deveaux, "Preliminary design of the Killick-1 earth observation cubesat," in *Proceedings of NECEC*, 2019.
[7] E. Domingúez, B. Pérez, A. L. Rubio, and M. A. Zapata, "A systematic review of code generation proposals from state machine specifications," *Information and Software Technology*, vol. 54, pp. 1045–1066, 2012.