

StateLogics - State Background

An innovation in software engineering

StateLogics - generates high quality software code directly from state chart specifications and has been implemented as a set of macros and generated header files within C+. It is of general applicability and of *particular* interest to the software engineering community engaged in building embedded and real-time systems.

GLO Intl Ltd

June 2004

Direct and optimal finite state machines

The invention is in the realm of software, in particular the implementation and execution of a specification.

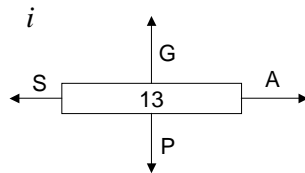
It improves on the current state of the art in that it:

- Reduces both the size of data structures and the number of processor cycles required to execute a finite state machine. (Usually there is a trade-off between these parameters)
- Relies on very simple mechanisms. (Current approaches rely on complex code generation and compilation technologies.)
- Lets software engineers work with state charts instead of code for the most error prone aspects of their system.

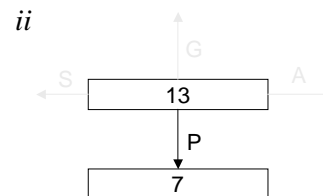
Finite State Machines

A finite state machine specifies the behaviour of a system in terms of the limited (finite) number of states the system can occupy and the allowable transitions between these states. Transitions are triggered by inputs or other events, of which there are a finite number of types.

A common notation for designing finite state machines is the state transition diagram:



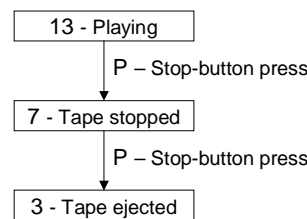
In state 13, the system can deal only with inputs of types S, G, A or P.



An input of type P causes a transition from state 13 to state 7.

iii

The same input (P) causes different transitions depending on the current state of the system.



Part *i* of this diagram shows that, in a given state, the system's behaviour depends on which one of potentially many different inputs occurs. Part *iii* shows that, for one given input (P), the system responds differently depending on which one of potentially many states the system is currently in. Such a many-to-many mapping can be expressed as a matrix or table. In this case, a table with states as its column headings; inputs and events labelling the rows; and transitions in the cells – where the combination of state and input is a valid one.

Both state transitions diagrams and state tables, however, quickly become unwieldy when they attempt to specify a system composed of sub-systems (which in turn may have sub-systems, and so on), where each sub-system traverses a distinct set of states of its own.

Two things now happen that complicate matters:

- Depending in the state of the parent system, some transitions in the sub-system may or may not be allowed. For example, a car has a gearbox as a sub-system. Transitioning from fifth gear to reverse is allowed, but not if the car is travelling at any appreciable forward speed. So for each input, the finite state machine implementation must be cognisant of the composite state of the system – this results in a multiplication of possible states. The state transition diagram gets uglier and the matrix gets fatter.
- Sub-systems may step through their states more or less independently: the gearbox does not care much about the air-conditioning. But it is imaginable

that both respond (in their own ways) to the same single input. Say the press of the Economy button engages a cog in the gearbox to set it to overdrive and switches the air-conditioning from cooling to humidity control only. So there are two valid *current* states and two transitions. Both state transition diagram and matrix need to multiply out the combination of each state of one sub-system with each state of the parallel sub-system – again getting uglier and fatter respectively.

Implementing a finite state machine

When an input arrives, the software has to determine the type of the input and the current state of the system. It can then look up the requisite transition and resulting state.

A state table or matrix is one of two common ways in which finite state machines are implemented. Such a table quickly gets quite large. Dealing with the possibility of more than one valid transition (in independent sub-systems) makes it larger. Optimising it (sparse matrix compression) reduces the size, but adds machine cycles to the lookup. Because of the size of the table and the absence of clear structuring devices (to reflect the hierarchy of sub-systems) a state-table implementation is hard for engineers to debug.

The other common implementation is using branching statements (if then else; case of; switch) to test the input type and state in all valid combinations. A moderately complex finite state machine will give rise to a deeply nested hierarchy of branching statements. Compilation technology can do much to reduce the size of the data structure required, and optimise machine cycles used in the tests, but on average it still takes half as many tests as there are states to locate the correct branch. Given the multiplication of states, that quickly amounts to a large number of tests, resulting in a big and slow program. Nested branching statements are notoriously hard to write and debug – the bigger and deeper they are, the harder and more error prone they get.

Our solution

The hierarchic and parallel nature of all but the most trivial systems means that representations that do not naturally deal with hierarchy quickly become unwieldy. This applies to both state transition diagrams and matrices. Fortunately a well-established notation exists that solves this problem: State Charts, originated by Harel. This notation is incorporated in the popular UML set of software design notations, with which SDL (used widely in the telecommunications industry) is set to converge.

Harel's state charts provide an intuitive and elegant notation for hierarchies of states and for parallel live cycles (either of sub-systems or of co-existing roles of a system or sub-system). Harel also adds a number of augmentations, for example the notion of history, whereby the system state can be restored to an earlier state. A small number of specialist tools including Harel's StateMate) have long provided support for this notation, including code generation.

When it comes to code generation, though, the advantages of the state chart are lost. Effectively it is translated into a finite state machine using one of the two strategies discussed above: a two-dimensional state transition matrix or a deeply nested hierarchy of branching statements.

Our solution compresses the full hierarchy represented in the state chart, including parallel states, into a small, one-dimensional state vector. This is matched with a similar representation of the input, to yield a pointer into a list where the transition (and any attendant actions) is looked up. The list is the largest data structure, but has only precisely as many entries as there are valid transitions.

Compression takes place at compile time (using a bespoke – possibly just-in-time – state chart compiler) leaving to run time merely three or four machine instructions, depending on the optimisation that could be applied by the compiler. One cycle is used to set the target state resulting from the transition. The target state is complex, because of the hierarchy: to transition targets one state, but all its parent states up the hierarchy should also be enabled, and all primary (starting) states down the hierarchy should also be enabled. Because we have a one-dimensional vector representing the entire state space, we can switch on all required states with one machine instruction.¹

We claim that this solution optimises both memory and execution cycles well beyond what can be achieved by the two common approaches discussed earlier, for all but the most trivial finite state machines. This makes our solution suitable for deployment on the widest range of devices, from embedded controllers, to mobile phones, to PDAs, set top boxes, domestic and commercial security and surveillance platforms, PC and server operating systems and applications, database management systems, mainframes and any other device running software or firmware.

The compiler and the execution mechanism are both remarkably simple and should be capable of being formally proven correct. State charts benefit from a number of correctness tests. The combination of state charts with our solution should be extremely robust – exceeding the quality of any approach involving code generation and compilation, where both the code generator and the compiler are complex – and where manual changes to the code destroy any formal rigour the code generator may have imparted.

Furthermore, because this solution directly executes the specification given in the state chart, the engineer can work with and debug the state chart, which is by far the best representation for a human to work with. Even the specialist state chart tools generate code and expect engineers to refine and debug the code, rather than the state chart. We anticipate very large improvements in productivity and quality where our solution is used.

The above can be summarised as three distinct areas of claim:

1. We have the most efficient implementation of a finite state machine (preferably one specified as a state chart) – a technical effect
2. We have the most robust implementation of finite state machines in terms of quality
3. We directly execute the state chart, without intervening artefacts that need concern the engineer – the result is a process improvement yielding higher quality and shorter time-to-market

¹ The count of machine instructions assumes that the state vector is no larger than the word-size used by the processor. This is commonly 32 bits, even for handheld PDAs, but may be as little as 4 bits for some embedded micro controllers. If a state vector of 12 bits had to be dealt with by a 4-bit controller, the number of cycles would be three times as large.

Your next step:

StateLogics - generates high quality software code directly from state chart specifications and has been implemented as a set of macros and generated header files within C+. It is of general applicability and of *particular* interest to the software engineering community engaged in building embedded and real-time systems.

- **Development partner** - work with us to incorporate, adapt and extend our innovation, in particular its integration with established or GLO developed tools. We look to the partner to contribute development resources, in return for significant influence on priorities and earliest access to any results.
- **Investors** – be enthused by amazing upside and patent protected technologies. GLO will consider third party investment from qualified institutions. Our commercial model has low running costs and is highly scalable with limited resources. It is a common and effective model in the software industry and, in addition to license fees; it generates maintenance, consulting and bespoke development revenues.
- **Academics** – get involved with these breakthrough technologies. We require technology validation and research assistance on a variety of exciting software technologies. Our current interest is in establishing research partnerships with academic institutions.

Call us now...

William M Pearson Chief Technical Officer bill@gloltd.com +44 781 425 5901	Michael Tanaka Chief Executive Officer michael@gloltd.com +44 776 777 6141
--	--