

StateLogics - Introduction

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

Real-time systems

The 21st century will see the realisation of ubiquitous computing: everything will get smart, everything will be running software. Unfortunately the software engineers building these systems have experienced very little improvement in productivity. Tool development has been largely directed towards the programmers of business and desktop systems, where Microsoft tools dominate. These tools address powerful processors with vast amounts of disk and memory. In real-time and embedded systems, processor cycles and memory are at a premium. Software engineers still have to handcraft complex code, in order to make it small and fast enough.

GLO's core technology, StateLogics, addresses these critical challenges:

- It is small and fast
- It manages complexity by directly implementing the structure of Harel State Charts used to analyse the required behaviour of the system.

The first characteristic means we can improve on existing approaches across the full spectrum of applications: from embedded systems running on tiny processors with minimal memory such as SmartCards, to massive server-based systems that manage the state of tens of thousand of live objects across multiple countries such as advance call services in telecommunications networks.

The second characteristic benefits both system engineers and the business. Existing approaches create data and control structures that bear no obvious relation to the structure of the problem they set out to solve. These structures are time consuming to write and hard to read; error prone and difficult to debug; challenging to maintain and nearly impossible to upgrade. This is true for code currently generated from Harel State Charts and for 100%-handcrafted code. In contrast, the GLO algorithm allows a direct mapping from the structure of the problem to the structure of the software. Thus the software is quick to write and easily read; reliable and simple to debug; readily maintainable and as effortless to upgrade or enhance, as it was to create.

Small And Fast currently means handcrafted, special-purpose code. This is true even where code is automatically generated from State Charts. Generated code serves as an initial skeleton and is just the start of a long drawn out development process. At the end of this, there is no recognizable mapping between the code and original state chart. In contrast, StateLogics generates 100% of the delivered state management code. It is smaller and faster than the best handcrafted code, yet requires no intervention from design to delivery of the final program.

Exit the finite state machine

Finite state machines are a well-understood way of specifying the required behaviour of a system. State-transition diagrams are commonly used in the analysis and design of such machines. But they suffer from two flaws:

- Even modest complexity of behaviour gives rise to a sprawling state transition diagram. All the detail is laid out before you, without any device for structuring it. They lack an organising principle
- Parallel states give rise to 'state-explosion'. Parallel states can arise, for example, where components each have their own more or less independent lifecycle. The 'state explosion' occurs because the overall state of the machine is the Cartesian product of the states of its components. This very quickly gets very out of hand.

No wonder then that the state table or nested condition statement in your program code also gets out of hand.

Enter Harel state charts

Harel solved the twin problems of finite state machines and added useful extra mechanisms.

- His state charts structure the ramblings of the finite state machine by introducing levelling: a state can have subsidiary states, and so on, recursively. This simple enhancement delivers a very powerful and navigable description of even the most complex behaviours. You can hide or reveal just the level of detail you need, where and when you need it.
- He makes the notion of parallel states into an explicit construct. The Cartesian product is implied and can be derived, but does not need to be shown.

The combination of hierarchy and parallelism allows for complex behaviour to be described clearly and compactly. Harel also allows for the execution of entry and exit code on states, in addition to code that is executed on transitioning between states. This adds a huge amount of power, especially in combination with the hierarchy

This hierarchy makes a transition into a very rich transaction: if you transition from one part of the hierarchy to a different branch, you must execute not only the transition code, but also the exit code of your original state, plus that of all its parent states, up to the parent you have in common with the new branch. Similarly you must execute all entry code from that common ancestor down to your new state, plus the entry code of any 'initial' sub-states on (possibly parallel) branches below your new state

Harel's history mechanism allows for a past state to be remembered and restored.

Harel state charts are now a world-leading standard for the specification of real-time (or, more generally, reactive) systems. They are part of UML and, as a result, many software engineers will be familiar with them.

Implementing state charts

Existing tools that generate code from Harel state charts fall short on three counts:

- Their implementation is either memory hungry or processor intensive, or both
- They do not fully implement all allowable Harel constructs, so you get a code skeleton at best and have to do significant and error-prone work in C or C++
- There is no traceability between the statechart and the implemented code.

Embedded and real-time systems sense inputs and deliver the correct response. The correct response depends not only on the input, but also on the current state of the system. The software has to manage all combinations of input and state, determining for each the correct response and the resulting state. GLO's algorithm radically simplifies and optimises the way software engineers can achieve this.

Our algorithm allows program code to be structured much more clearly than is presently the norm. It enables a one-for-one mapping from the specification of the required behaviours to the structure of the code. This direct mapping from specification to implementation improves quality, reduces time-to-market and accelerates product evolution. All three are key concerns in this market.

StateLogics produces an implementation that is small and fast – engineering concerns that have been all but forgotten in the US-dominated business and desktop software market, but

that are crucial in embedded and real-time systems. It implements 100% of the state management of a reactive system with an efficiency that cannot be matched even by the best special-purpose program code.

The proposition appeals both to software engineers and to business people. Engineers love the elegance and efficiency of the implementation and the ease with which they will be able to use it; the business will see the competitive advantage resulting from time and cost reduction, and improved quality. Both will recognise that there is no existing approach that can yield the same benefits.

The OO advantage

We have discovered a close parallel between the way events cause transitions in Statecharts and the way messages cause method execution in Object Orientation (OO). Briefly, in OO, a message will cause different behaviour (method execution), depending on the class of the object receiving the message. In Statecharts, an event causes different behaviour (state transition) depending upon the state of the object. Class, in OO is a hierarchic concept. In Statecharts, state is a hierarchic concept - with the added richness of parallel states. We found that the Statechart formalism can be seen as a superset of OO, enabling us to map OO fully into this formalism.

The result is staggering: OO method dispatching and Statechart event transitioning are unified in a single paradigm, implemented by the original algorithm! Both are equally transparent to the programmer. The useful OO features of polymorphism and inheritance apply equally transparently to states allows for the inheritance and specialisation of state-based behaviour. For the first time ever, this extends the advantages of OO development to the development of state-based systems. Traditionally, state-based behaviour is a concern that needs to be programmed long-hand for each new OO sub-class. The state of the art in OO is to address this problem with patterns, but patterns are no more than a template facilitating long-hand coding. Long-hand coding inevitably leads to code bloat and increased error rates - as well as being hard to understand and maintain.

Your advantage

The 21st century will see explosive growth in intelligent devices: *everything will get smart*. StateLogics represents a radical breakthrough in creating the software that drives this growth. This market is diverse and extremely large. IDC estimated the embedded systems market as \$1 trillion at factory gate prices in 2000. The market is expanding rapidly as more intelligence is built into a staggering array of devices; into the nascent generation of domestic robots; into the very fabric of our buildings.

Manufacturers uniformly recognise that the key success factor is time to market and the key to ongoing profitability is rapid product evolution so as to reach all high value niches. Current software process frustrates these critical requirements. Our innovation reduces time-to-market and enables rapid, continual improvement.

Reactive systems are a huge, varied and relentlessly growing market and one where Europe has retained world-beating competence. Systems in this market are large and complex. Harel State Charts are the only realistic way to manage complexity on this scale and have been widely adopted as part of the worldwide UML standard. Software quality is also a concern in this market: these systems are business critical or even safety critical. By retaining the Harel structures in the resulting code, the GLO algorithm enhances traceability and hence quality – not just when the software is first written, but throughout its lifecycle.

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 |
|--|--|