Controlling Data Flow Applications in Gedae: Is a Finite State Machine the Answer?

Mr. Ian Alston & Dr. Bob Madahar
BAE SYSTEMS Advanced Technology Centre
West Hanningfield Road,
Gt. Baddow, Chelmsford CM2 8HN, U.K.
Tel: +44 1245 242195
Fax: +44 1245 242124
ian.alston@baesystems.com & bob.madahar@baesystems.com

ABSTRACT

Control is an integral part of many applications, in particular in embedded processing systems. The complexity of these systems and applications, and therefore the level of control, has risen with advanced high performance systems capable of sustaining ~1Tflops practical now compared to ~1Gflops in the 1980's. These systems are multi-functional and multi-mode by design and require high performant control solutions to deliver the performance capabilities. This adds an extra dimension of difficulty to the designer, particularly when errors in the control logic can cause reduced performance or even failure of the application. The finite state machine (FSM) offers a convenient method of specifying and simulating control functionality of a (sub-)system. Combining the control capability of FSMs with the model based design flow available within Gedae allows the control logic of complex applications to be implemented. Two implementations of FSMs will be shown, one based on emulating the behaviour of conventional state transition diagrams and an alternative based on a state transition table lookup method. Indeed it will be shown that an FSM implementation greatly simplifies the Gedae flow graph compared to using the basic discrete control primitives and/or dynamic data flow.

1. INTRODUCTION

The defence industry is aiming to field state-of-the-art products in less time and with lower costs based predominantly on commercial off the shelf (COTS) components. In order to achieve this goal, model based design tools employing automatic code generation are being used to seamlessly move from a functional representation to implementation onto real-time COTS test beds. Gedae [5] offers such a model based design tool for general data flow applications, such as signal and image processing systems.

It is well known [8] that Moore's law states that the capability of these COTS components will double within 18 months. This is demonstrated by the fact that over the past three decades the functionality of embedded sensor based applications has moved from single mode applications to extended multi-mode/multi-function applications with complex dynamic control of the switching between these modes/functions. Thus a designer needs to be able to design, simulate and finally implement a variety of application control algorithms as well as the application itself.

As an example, Figure 1 shows a typical radar system and the interaction of the Radar Control functions with other elements of the system. In particular the control functions play a major role in configuring the multi-mode behaviour through the interaction with the antenna, beamformer, waveform generation and the processing performed. The interest in this paper is signal processing and Gedae.

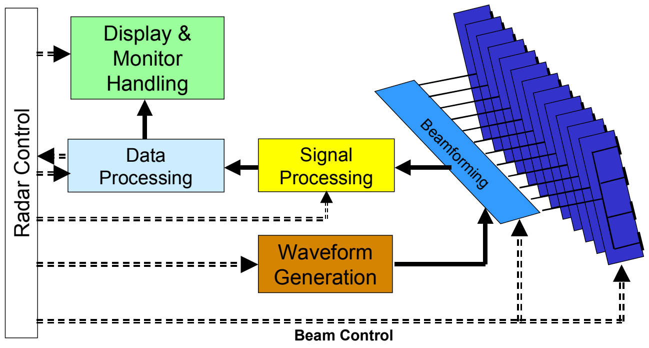

Figure 1: Typical Radar System

Various languages and tools have been developed to model control processing. Of particular note is the Finite State Machine [7] (FSM) paradigm that offers a means to describe and analyse sequential logic and control functions. There has already been extensive research [6] into the combination of FSMs with various models of computation including data flow. It would therefore seem logical to investigate how FSMs could be implemented within the Gedae data flow language thus allowing complex applications, including their control logic, to be implemented in a single tool environment.

The paper will provide a brief overview of control processing languages and analyse the current capabilities of Gedae for control processing. The paper will demonstrate that whilst Gedae is not optimised for the definition and implementation of complex control structures, FSMs can be implemented within Gedae and enables the designer to include the necessary control parts for their application. The paper will therefore provide an overview of FSMs and describe two methods of implementing FSMs within Gedae. The first is based on a state transition diagram model whilst the second is based on a state transition table look-up method. Whilst the implementations discussed are based on the use of the discrete "trigger" primitives it will also be shown that similar techniques can be applied to conventional data flow primitives using both static and dynamic data flow properties. A demonstration of the practical use of FSMs within Gedae to model the control logic of a highly multi-mode radar will also be reported.

2. CONTROL PROCESSING LANGUAGES

The complex multi-mode/multi-function systems described above are essentially a mixture of transformation and reactive systems. A transformational system is characterised be being data driven, where input data is transformed into output data. A purely reactive system is one which is event driven continuously having to react to its environment or some event stimulus. In practice systems will be a combination of both (c.f. machine vision). Therefore while the data-flow portions of a Gedae design belong in general to the general class of transformational systems, the complex, highly dynamic control elements belong to the general class of reactive systems. Therefore in order to understand the most appropriate way in which to implement these control elements an understanding of the languages and techniques for the modelling of reactive systems is important.

Reactive systems have at their heart some form of state machine i.e. an event causes the internal state of the reactive system to change. Therefore many of the design languages appropriate for designing reactive systems are also based on state machines. The following sub-sections first give an overview of finite state machines followed by a brief description of a number of general languages for designing reactive systems which use FSMs in different ways.

2.1 FSM

The finite state machine is a technique that allows the simple and accurate design of sequential logic and control functions. They have a wide application domain and can be applied to the design of computer programs, control systems for machinery, electronic equipment and digital applications, or telecommunications protocols. The basic premise is that a system can only have a limited (finite) number of states which represent the internal "memory" of the system by implicitly storing information about what has happened before. Transitions (which represent the "response" of the system to its environment) between states depend upon the current state of the machine and the current input.

The FSM *M* can be represented mathematically as a tuple of the form:

$$M ::= < I, O, S, S_0, T >$$

where	I	is a set of input events,
	O	is a set of output events,
	S	is a finite set of states,
	$S_0 \in S$	is the initial state,
and	T	is a set of transitions.

In a single reaction of the FSM, a subset of the input events present in the current state causes a transition to the destination state while also creating any output events associated with the transition. Therefore a transition has associated with it the following:

- $S_S \in S$ is the source state,
- $t \in T$ is a named transition defining a "guard/action" pair where:
  - a guard is a logical combination of a subset of the occurring input events I,
  - action lists a subset of the output events O, including the empty set, that will be generated for this transition,
- $S_d \in S$ is the destination state.

In most cases the strict mathematical definition presented above is replaced by a state transition diagram. A typical example is shown in Figure 2. Each node represents a state and each arc represents a transition. The arc without a source state points to the initial state i.e. S_0. The arcs are labelled with the "guard/action" pair defined for the transition linking the source and destination states and the output events to be generated respectively. In cases where the action is empty it is often absent from the transition label as shown in Figure 2, guard e_2.

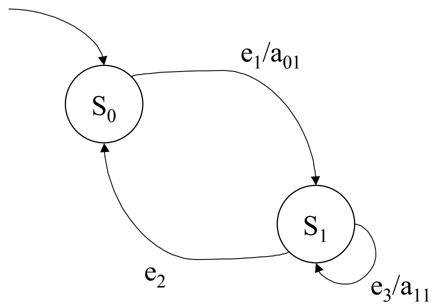

Figure 2 A simple FSM defined as a state transition diagram

2.2 Statecharts

The state transition diagram is one of the most popular graphical means of describing FSMs. However, such a simple "flat" manner of specifying the complexity of reactive systems having ever increasing number of states leads to unstructured and illegible state transition diagrams. For such diagrams to be useful a modular, hierarchical and well structured approach must be adopted. In order to cater for such an approach Harel [7] developed a visual formalism for describing states and transitions known as Statecharts. Statecharts extend the simple graphical representation of FSMs to include three important attributes namely hierarchy, concurrency and communications as described below.

Statecharts overcome the growing number of states to describe even simple systems by the introduction of depth and concurrency. Depth allows a state transition diagram to be viewed at different levels of abstraction. Systems specified in a hierarchical manner are usually easier to understand as each level of abstraction focuses on the details that are important at that level of the hierarchy. The notion of depth can also be used to cluster groups of states in order to reduce the transition arrows.

In a conventional state machine model states are an or-state (i.e. they are either in one state or another) but never in two states concurrently. Concurrency in Statecharts uses the notion of the and-state. The and-state allows multiple sub-states of a higher level state to be active at the same time. The resulting sub-states are said to be orthogonal. In addition, communication between the sub-states occurs in a defined manner:

- all orthogonal sub-states of a state accept events sent to that state,
- one sub-state may create an event as a result of a transition that is consumed by one of its orthogonal sub-states.

An example of the use of the and-state is shown below in Figure 3 using the notation in [7].

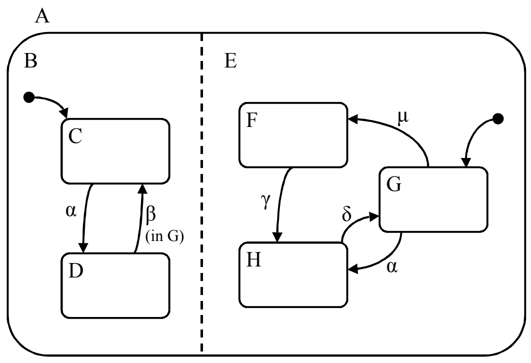

Figure 3: AND-states in Statecharts

The diagram shows a state A consisting of two orthogonal sub-states B and E with the property that being in A requires being in some combination of C or D with F, G or H. Entering A from the outside into the default states (i.e. those identified by the transition from a small black circle) would result in the combined state (C,G). If a subsequent α event occurs, a simultaneous transition to the new combined state (D,H) occurs.

Many of the recent tools developed to model and implement FSMs use a variation of the Statechart formalism. Statemate MAGNUM [9] from I-Logix is a particular example.

2.3 Specification and Description Language

As an alternative representation to Statecharts, the telecommunications industry adopted the Specification and Description Language (SDL) [3] which was developed and standardised by the International Telecommunications Union - Telecommunications Standardization Sector (ITU-T) (formally the CCITT). SDL, though aimed at the telecommunications industry, can also be used for other event-driven or reactive systems.

The theoretical model of an SDL system consists of a set of extended FSMs running in parallel, each FSM being independent of each other and communicating via discrete signals. As the language is designed for the specification of a complete system there are 3 basic levels of abstraction within the SDL in order to specify structure, communications and behaviour of a system:

- System

  The system description consists of the top level of detail - an abstract machine communicating with its environment. It contains everything that is to be specified including: block descriptions (see below) and channel descriptions for connecting blocks to each other and to he environment.

- Block

  A block is a sub-item within the system that can be treated as being a self contained object. It is hierarchical in nature and is composed of lower level blocks and process definitions (see below). In addition the block description contains specifications of the connections within the block (to processes and blocks) and to the environment of the block (i.e. external).

- Process

  The dynamic behaviour of the system is described by a process description. A process is a communicating FSM, with possibly many instances of the same process, being driven by and producing signals conveyed between the blocks and processes by means of the channels.

The most common method of describing the process description is using a process diagram. Basic constructs have been defined for the specification of a process which include:

- State control - identifying idle, current and next states and terminating conditions. Also includes the ability to create sub-states via "procedures".
- Signal control - acting on received and producing new signals.
- Task/data control - provides a means to manipulate its own local variables.
- Decision control - controlling the flow through the process diagram based on the values of local variables.
- Process control - setting timers and producing new instantiations of processes.

The corresponding process diagram for the state transition diagram of Figure 2 is shown in Figure 4.

A simple and effective way of implementing a process diagram is achieved by defining the processing to be performed for each event/state pair as a separate "function" which must include internal state modification, the generation of output events and if appropriate a change to a new state. Pointers to these "functions" are placed in a matrix lookup table, the state transition table (STT), which references the processing to be performed when an event occurs in any particular state. Transition through the FSM is simply a means of a dispatch handler identifying the processing in the STT for each event/state pair and calling the associated "function". While the state transition diagram and the SDL process diagram convey similar meanings it could be argued that the latter is more expressive in that there is the ability to show the manipulation of internal state variables, set internal timers, perform internal decisions within a process definition and dynamic process creation.

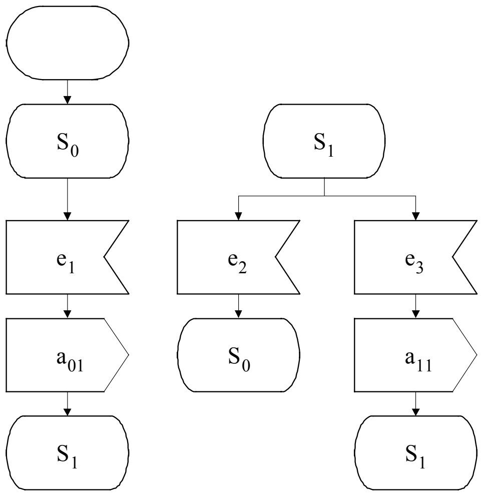

Figure 4: Description of the FSM of Figure 2 by an SDL process diagram

2.4 Synchronous Languages

There is a class of languages for modelling reactive systems, termed synchronous languages, in which it is assumed that the system reacts in zero time, i.e. the computation of the reaction takes no time and thus the output actions are synchronous with the input actions. A reactive system can thus be viewed as a set of subsystems that evolve simultaneously and communicate with each other to achieve the required behaviour. The Statechart formalism described in Section 2.2 also falls into this broad category of languages. Another language is Esterel [1][2] which provides a high level language to describe parallelism for deterministic systems.

Esterel was designed in order to obtain a better understanding of the semantics of parallelism. It is a textual language enabling the compact specification of complex systems. The underlying model behind Esterel can be summarised by:

- Reactivity

  The reactive model forms the basis of all Esterel models i.e. communicating systems continuously interacting with their environment. Within Esterel, the life of a reactive system is divided into instants when it reacts to the events and the Esterel language reactive statements are ones which are defined by reference to instants.

- Atomicity of reactions

  Esterel assumes the synchrony hypothesis which indicates that reactions are instantaneous so that activations and production of outputs are synchronous. Another way of looking at this is to say that reactions are atomic. Any reaction does not interfere with other reactions.

- Instantaneous broadcast

  The Esterel language contains a specific parallelism operator enabling the user to directly program parallel entities. In order to communicate with these parallel entities provides its own unique mechanism called broadcast. That mechanism relies on signals for both internal and external communications. Broadcast is limited to instants i.e. the creation of a signal lasts for the current instant and this signal can be seen by all receptors during this instant. As the generation of a signal does not terminate a particular instant, multiple signals can be created and received in the same instant. This leads to the characteristic of "instantaneous decisions" which are specific to the Esterel language.

- Determinism

  The parallelism introduced by Esterel ensures determinism thus simplifying programming and ensuring that behaviour is reproducible.

Of course for such a language to be useful some form of processing capability must be included i.e. not all real world applications have processing that fits the reaction takes no time assumption. Esterel caters for this with the introduction of asynchronous tasks - a piece of sequential code that is not instantaneous in its execution. Tasks have the following basic properties:

- They can be started and killed during an instant.
- A task is allowed to synchronise only when it terminates its execution.
- Tasks are not allowed to communicate.

One of the major benefits of the Esterel language is that the compilation process, i.e. converting the Esterel language statements into executable statements (C or pseudo output code format), produces sequential code with any parallelism and local communications being compiled away resulting in efficient implementation. Esterel is also based on rigorous mathematical semantics and thus formal verification methods may be applied to Esterel programs [1].

Although Esterel has its own environment for verification, validation, simulation and development it is now finding its way into commercial toolsets such as Esterel Studio [4].

3. FSM IMPLEMENTATION IN GEDAE

From the brief overview provided in previous sections, it is evident that a number of approaches to the utilisation and implementation of FSMs is possible. Gedae being a graphical environment lends itself particularly towards the FSM providing we can define an appropriate representation for them. In particular the graphical representations offered by the state transition diagram (and associated Statecharts) and SDL are particularly suited to implementation within Gedae. These will be discussed in this section as a basis for future comparison of alternative methods such as an Esterel approach.

For the moment we need to consider the basic building blocks required to implement FSMs. Figure 5 shows the key elements of a state and its associated transitions:

- A state S,
- A transition labelled "guard/action" causing transition to a new state with associated actions being performed,
- An entry into the state.

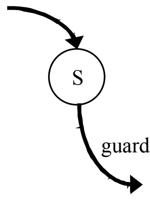

Figure 5: Elements of a single state

If we analyse the behaviour of the above elements in a different (i.e. more dataflow oriented) way we arrive at the following requirements for a basic FSM building block for Gedae to model the elements of a single state:

- The state has no internal memory except to know whether it is in the state or not.
- The guard is a true or false signal indicating that the state will be left.
- During the transition to a new state the action specified must be completed.
- The new state to enter must be specified.
- An entry signal is provided.

This last requirement permits a certain level of consistency checking within the implementation by setting the only internal memory element of the state i.e. in the state or not. An example of where this will be useful is when the guard is true even though we aren't in the state - executing the action is then a mistake.

From a reuse point of view, the programming of the action element into the basic building block would be a disadvantage. Therefore in the proposed implementation below the design uses the concept that the block will produce a signal to inform downstream processing that the action should be performed. This has the advantage that all action programming is then external to the state modelling blocks which because of our assumption of no internal memory is valid.

Therefore a pseudo-code representation of the basic single state building block is:

```
On signal entry:
      If already in this state - do nothing
      Else set internal state to true

On signal guard:
      If not in this state - do nothing
      Else
            set internal state to false
            perform actions (using signal)
            enter new state (using signal)
```

It should be noted that in order to maintain the integrity of the FSM, the two elements in the last Else branch of the "On signal guard" should occur in sequence and complete before continuing with processing of further events.

As we are trying to model reactive systems, the most appropriate domain for implementing this Gedae building block is in the discrete "trigger" domain. An implementation following the pseudo code above is shown in Figure 6 and 7 below.

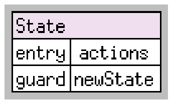

Figure 6: Gedae implementation of basic single state block

```
Name: State
Type: trigger
Comment: ""
Input: {
  trigger int entry;
  trigger int guard;
}
Local: {
  int inState;
}
Output: {
  int actions;
  int newState;
}
Reset: {
  inState = 0;
}
Trigger: {
  int en = dirty(entry);
  int ex = dirty(guard);
  if ( en && ex ) {
    printf("cannot enter and exit at the same time\n");
    OStaticFailed("cannot enter and exit at same time");
  } else if ( en && ! inState ) {
    /* entering this state */
    inState = 1;
  } else if ( ex && inState ) {
    /* exiting this state */
    inState = 0;
    actions = guard;
    push(actions);
    newState = guard;
    push(newState);
  }
}
```

Figure 7: Primitive implementation code for single state

In order to model multi-state FSMs we can connect this simple single state building block to form a state transition diagram. This is demonstrated in Figure 8 and 9.

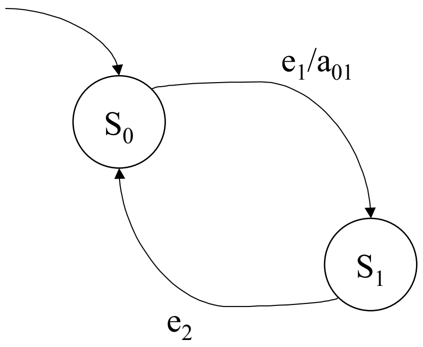

Figure 8: Example state transition diagram

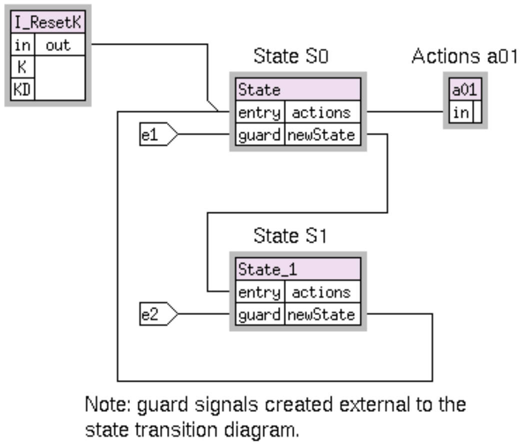

Figure 9: Gedae implementation of Figure 8

As can be seen from the figures, the Gedae implementation using the basic single state building block mirrors the original state transition diagram. Also note that the Gedae implementation contains an I_ResetK primitive to ensure that the correct initial state is entered when the data flow graph is executed.

The above realisation is somewhat limited and just as the work of Harel extended the standard state transition diagram concept to Statecharts, it requires extending for practical use within complex control structures. With minor modifications of the basic state building block and the inclusion of additional wrapper code the following functionality can be easily achieved:

- Depth - as Gedae DFGs are already hierarchical in nature, the concept of depth is automatically included. However the inclusion of an addition "sub" output when the state is entered adds to the functionality and permits improved sub-state modelling. In addition, this output can be used to execute actions when entering a state.

- Multiple entry and exit conditions - this is achieved using the notion of Gedae families converting the multiple entry and exit conditions into single events for use with the basic state machine primitive.

Figures 10, 11 and 12 show the new basic building blocks for FSM modelling in Gedae. Their practical use is given in Section 4. Note that the implementation of such building blocks could be greatly simplified if trigger primitives could employ families of inputs for their trigger inputs.

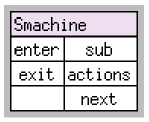

Figure 10: State machine primitive

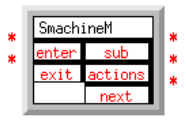

Figure 11: Multi-event state machine primitive

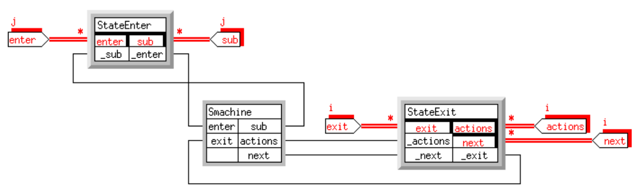

Figure 12: Hierarchical implementation of the multi-event state machine primitive

3.1 Alternative Implementation

The state transition diagram approach to modelling FSMs was originally chosen as it provides an intuitive representation for engineers who are familiar with FSMs. One of the drawbacks of this approach is that Gedae imposes fairly rigid constraints on the way primitives are connected and thus it is possible, if hierarchy isn't used sensibly to end up with a "spiders web" of connecting arcs. Fortunately, it isn't the only representation and an alternative approach is that based on an implementation approach commonly adopted when using SDL process diagrams for modelling FSMs. In Section 2.3 it was noted that a common implementation method is based on a dispatch handler identifying the processing in a STT for each event/state pair. This can be modelled in Gedae quite simply using the 3 basic primitives:

- Event handler - converts each independent event into a unique event number.

- STT lookup - based on the input state and the unique event number, lookup the action to be performed in the STT matrix and emit this action (as a unique action reference number).

- Action Handler - converts the action reference to specific actions. Note that this must also update the state ready for the next invocation of the STT lookup.

Figure 13 shows a particular implementation of these primitives.

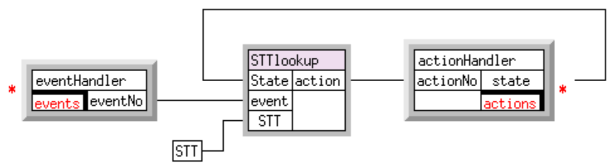

Figure 13: STT implementation of FSMs within Gedae

The STT lookup primitive is the key element of the three blocks but is in itself very simply coded having a trigger method containing lines of code similar to:

```
a = STT[event][State];
if ( a > 0 ) {
        action = a;
        push(action);

}
```

This coding assumes that an entry of zero or below in the STT matrix implies perform no action. The event handler primitive must thus convert independent events, specified through the family of inputs, into a unique event number to act as the trigger of the STTlookup primitive and also to as a reference into the STT matrix. Similarly the action handler converts a unique action number into a "pushed" event notification contained within one element of the family of action outputs. Note that as with the method based on state transition diagrams, the coding of the actions is independent of the event handler, STT lookup and action handler, although they may of course generate new events. The STT matrix corresponding to the state transition diagram of Figure 2 or the SDL process diagram of Figure 4 is then simply:

Event	State	
	S_0	S_1
e_1	1	0
e_2	0	0
e_3	0	2

where action 1 and 2 correspond to actions a_{01} and a_{11} respectively. Note that the action handler updates the state input parameter of the STTlookup primitive for the next execution.

There are a couple of immediate advantages of this approach over the state transition diagram based method described earlier. Firstly, additional events or states can be added simply by adding extra entries to the STT matrix. Secondly, it is possible to dynamically change the behaviour of the FSM by changing the value of the STT matrix input parameter. This latter case assumes that the actions used within the STT are already catered for within the action handler.

3.2 Data Flow Implementation

The implementation of the FSM primitives described in this section so far have been based on using the discrete "trigger" domain as these are most suited for modelling reactive systems. However, a current limitation of these types of primitives is that they can only be executed on the Gedae host and can't be embedded onto target platforms. As there may be scenarios when the control logic must be embedded, alternative implementations must be sought.

Luckily, both the implementations described above can also be implemented in the conventional data flow domain if additional assumptions are made. These additional assumptions tend to be concerned with how to handle granularity issues. While there are numerous ways in which a data flow implementation might be constructed, the essential problem to solve is how to mirror the triggered inputs within the data flow domain.

For the state transition diagram approach, the trigger inputs *enter* and *exit* must be converted into non-deterministic inputs. It would then be possible to check the number of available tokens on each input with a number greater than zero indicating that a "trigger" has occurred. One condition that would need an additional assumption is the case when the number of available tokens on one of the inputs is greater than 1 - would this result in multiple outputs or a single output? As the actions performed due to this non-deterministic input could result in a change of state then it can be assumed that multiple available input tokens are converted to a single output token. In some respects this mirrors the code in the discrete trigger primitive which tests whether we are already in or out of the state. As outputs will not be generated on every firing of the primitive the outputs would need to be dynamic or non-deterministic.

For the STT lookup method, it would be possible to have a purely static data flow implementation as a single input token to the STTlookup primitive always results in a single output token. To keep the primitive as simple as possible, the state and STT matrix inputs would remain an input parameter with the event a stream input. The only consideration then is to ensure that any change in state is synchronised to the arriving input events. One possible solution to this is the use of runlength encoded parameters for the *State* input.

In both the above cases, there would have to extensive use dynamic data flow primitives within the event generation/handling and action handling segments of the implementation. This would lead to inefficient implementations. An alternative approach would be to ensure that the embedded scheduler was capable of handling the discrete "trigger" primitives.

4. A PRACTICAL EXAMPLE

The above FSM implementations have been used within the development of the signal processing sub-system of a multi-function radar. The operation of the radar is defined by a hierarchy of search patterns, each of which consists of one or more dwell patterns that in turn call up one or more radar burst definitions. The basic operation of the radar will be governed by a series of default search patterns that will be used until a dynamic data driven series of "tasks" have been generated on-line by the data processing functions of the radar. These dynamic tasks will consist of a number of radar burst definitions. This approach to controlling the radar operation has been adopted for two main reasons:

- To provide the flexibility to operate the radar under a wide range of pulse regimes.
- To allow feed back from the data processing functions to optimise the radar to the current environment and operational needs.

Therefore the control logic of the radar must cater for the following elements:

1. Reading a default search pattern from a repository of such search patterns.
2. Allowing the user to view and edit the default search pattern.
3. Generate radar bursts according to the specification within the search pattern.

4. Update the radar bursts being generated as a result of the data processing.

It is clear that the above logic is well suited to an FSM implementation as the processing consists of a number of states with transitions between the states being controlled by both operator and radar interaction. The first implementation of the FSM for the control logic concentrated on elements 1 to 3. A state transition diagram for the control logic is shown in Figure 14.

The diagrams show the transitions between the states and why the transitions occur. The specific actions to be performed during these transitions are not shown for clarity. Also note that the elements to allow the user to view and edit the search pattern have been specified within the control logic but haven't as yet been implemented within the Gedae implementation.

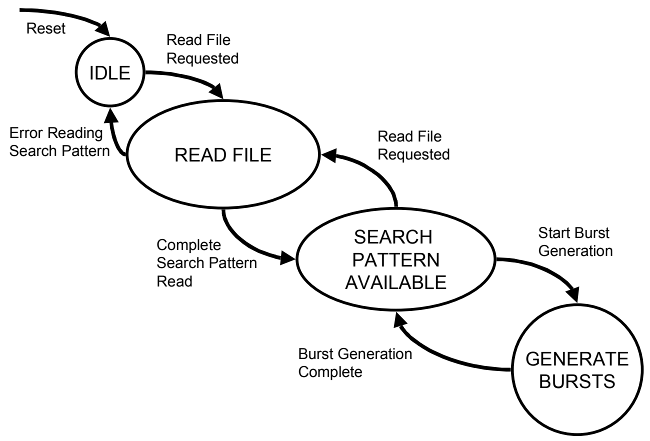

Figure 14(a): Radar control - Top level state transition diagram

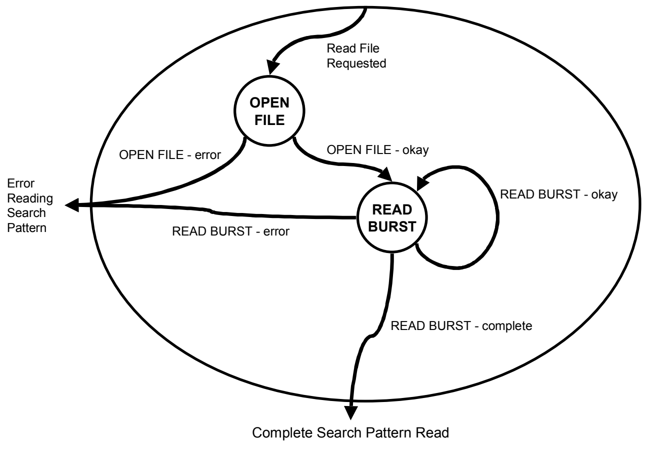

Figure 14(b): Read File hierarchical state

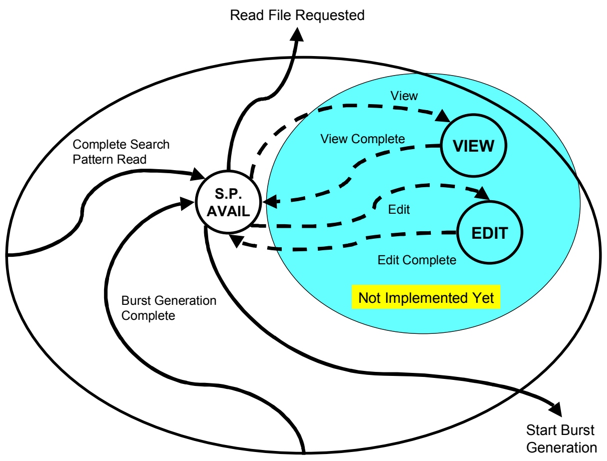

Figure 14(c): Search Pattern Available hierarchical state

The top-level Gedae implementation of these state transition diagrams is shown in Figure 15. As can be seen the model mirrors the state transition diagram of Figure 14(a). For clarity within the model the transition names have been shortened and as discussed in Section 3, the actions to be performed are external to the main FSM structure. As in the state transition diagram the Gedae model contains two hierarchical states, SPread and SPavail.

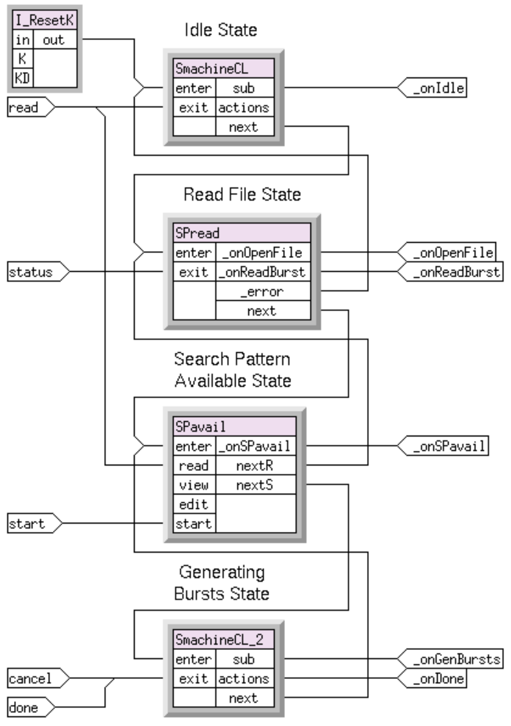

Figure 15: Radar Control - Gedae Implementation

It will be noticed that a SmachineCL primitive has replaced the basic Smachine primitive described in Figure 10. This modified Smachine primitive is shown in Figure 16 and is required to overcome the way the Gedae pushing of parameters operates together with the fact that, in the case of this particular FSM, a “closed loop” or circular action/event structure is present. The act of pushing a parameter using the push() function in a Gedae primitive causes all downstream parameters to be evaluated until no further evaluation can take place. At this point Gedae re-winds to the originating push() function to carry on execution of that primitive code. The push() function is essentially recursive. For most applications this functionality is quite acceptable but in the case of an FSM implementation, this functionality may cause downstream actions to create additional input events prior to any required change in state. In addition, the triggered inputs are not made "clean" (i.e. dirty() returns false) until the execution of the primitive has completed. This results in many FSM errors due to the state being entered and exited during the single firing of the primitive (see sample code in Figure 7).

Fortunately, this behaviour can be overcome by ensuring that the Smachine primitive completes execution prior to the traversal of the parameter evaluation following the push() function. This is achieved in the modified Smachine primitive using the breakPush and breakPush2 primitives which use an I_GateK primitive and a number of I_Seq2 and I_Pulse primitives to essentially break the "closed loop" nature. These primitives ensure that the Smachine primitive completes execution prior to the sub and actions/next outputs are pushed.

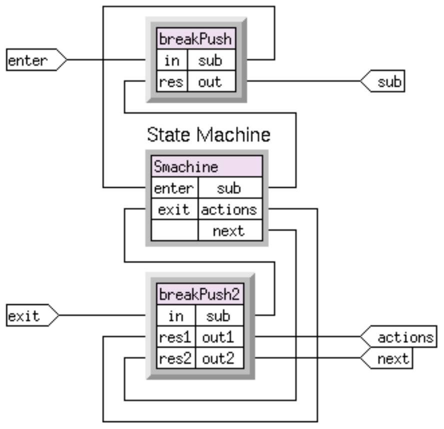

Figure 16: Modified Smachine primitive

The most interesting of the hierarchical states within the top-level Gedae model of Figure 15 is the SPavail state as it contains multiple entry and exit events and also two internal sub-states. The Gedae implementation of this state is shown in Figure 17.

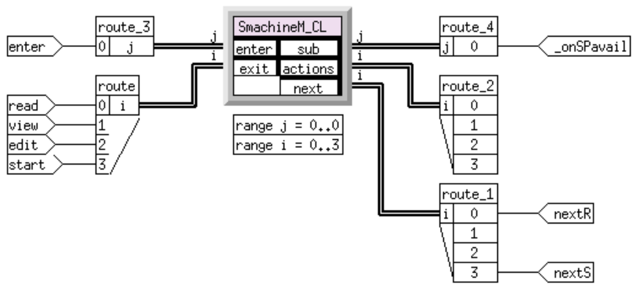

Figure 17: SPavail hierarchical state

This model uses route boxes to convert the input events and output parameters to/from the family input/output members of the SmachineM_CL primitive respectively. This state demonstrates the ease of adding additional transitional events to the FSM. The transition to the two unimplemented internal sub-states is shown by the dangling output members of the route_1 primitive. The inclusion of these sub-states at a later date will be quite straightforward.

5. CONCLUSIONS

It has been shown that FSMs provide the basic elements for the modelling of the complex, highly dynamic control elements found within reactive systems. While the most widely used notation for describing FSMs is the StateChart, the SDL process diagram is commonly used within the telecommunications industry. It has also been shown that these two notations can be effectively implemented within Gedae for the control of data-flow applications.

While Gedae isn't the most appropriate tool for modelling and implementing complex control logic, the FSM approach described in this paper does provide a means of simplifying the control logic. To demonstrate this, a practical realisation of the state transition diagram version has been presented for a multi-function radar. Achieving the same control algorithms using standard discrete logic primitives would be far more complex and much more difficult to modify to include future enhancements.

While the discrete "trigger" primitives are the most appropriate method of implementing FSMs within Gedae, there are some limitations to their use. The two most important limitations are:

- they can only be executed on the host,
- families of inputs/outputs aren't permitted.

It has been shown that the second of these can be overcome, for the discrete "trigger" primitives, by including additional control logic to interface families of inputs/outputs with the basic Smachine primitive. However, at present the first limitation can only be overcome by using standard data-flow primitives and methods for achieving this have been suggested. The removal of these two limitations would greatly enhance the capabilities of Gedae for modelling and implementing FSMs as described in this paper.

6. ACKNOWLEDGEMENTS

Elements of the FSM implementation within Gedae and the practical example of FSM use were developed while the authors were members of an integrated project team (IPT) funded by Alenia Marconi Systems Limited (Chelmsford, UK). The authors would like to acknowledge the support from and contributions of all the IPT team members.

7. REFERENCES

[1] G. Berry, "The Foundations of Esterel", in "Proof, Language and Interaction: Essays in Honour of Robin Milner", MIT Press, 1998

[2] F. Boussinot and R. De Simone, "The Esterel Language", Proc IEEE, Vol 79, No. 9, Sep 1991, pp. 1293-1304.

[3] CCITT Recommendations Z.100 - Z.104, "Functional Specification and Description Language (SDL)", Red Book,Vol.VI.11, 1985.

[4] ESTEREL Studio, ESTEREL TECHNOLOGIES, http://www.simulog.fr/esterelstudio/

[5] Gedae, Blue Horizon Development Software, USA, http://www.gedae.com/.

[6] A. Girault, B. Lee and E.A. Lee, “Hierarchical Finite State Machines with Multiple Concurrency Models”, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 18, no. 6, pp. 742-760, June 1999.

[7] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, vol. 8, pp. 231-274, 1987.

[8] G. E. Moore, "Cramming more components onto integrated circuits", Electronics, Volume 38, Number 8, April 19, 1965.

[9] Statemate MAGNUM, I-Logix, http://www.ilogix.com/.