bCMS in LEAP

Tony Clark
Balbir S. Barn
School of Engineering and Information Sciences
Middlesex University
London NW4 4BT, UK
{b.barn,t.n.clark}@mdx.ac.uk

August 2, 2012

Abstract

This paper contains an overview of the features of the LEAP component-based executable modelling language applied to the bCMS Crisis Management System Case Study. The paper aims to show the key features of the LEAP technology in terms of various aspects of the case study.

1 Introduction

LEAP is an executable modelling language and associated tool that aims to provide a basis for component-based development. LEAP is based on a collection of simple concepts that are intended to span the development life-cycle from requirements through analysis, animation and eventually deployment. LEAP is currently in development [3, 4, 2, 1] and its application to case studies such as bCMS[1] and the associated comparison with other modelling approaches is an important part of the development process.

Our aim with LEAP is to support both enterprise models and system models using the same collection of features. Our view is that such systems can be represented as hierarchically organized collaborating components expressed using a suitable collection of modelling languages. Therefore, a specification language is provided that expresses conditions over static and dynamic features of component models. Components manage their own internal state as a form of *knowledge base* and a deductive theory language is provided together with a functional language for expressing behaviour. Declarative component behaviour can be achieved using a state-machine language together with a language for event driven rules.

This document describes describes some key features of LEAP as they are applied to the bCMS system. The key features are as follows: section 2 briefly de-

[1] http://cserg0.site.uottawa.ca/cma2012/CaseStudy.pdf

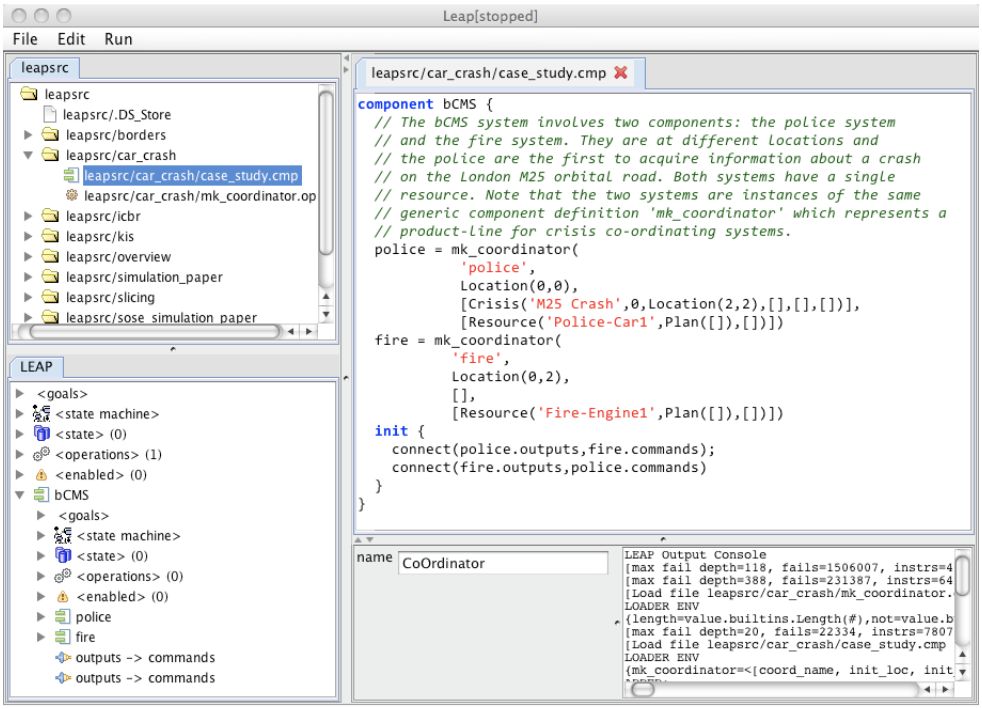

Figure 1: The LEAP Tool

scribes the LEAP tool and how it supports application development; section 3 describes how architecture models are constructed; section 4 describes the bCMS information models; section 5 describes data invariants; section 6 describes how components contain theories that can be used to deduce facts about the current state of a component; section 7 describes how system requirements are captured as goal models; section 8 describes the operational semantics of LEAP components using operations; section 9 describes how the behaviour of a component can be encoded as a state machine; section 10 describes how the current state of a component can be displayed as an object model; section 11 describe how Java classes can be included in LEAP models shows how this feature can be used to implement a simple GUI for bCMS.

2 Tool and Model Development

LEAP is both a textual language and an associated tool. The tool provides a collection of views on the models under construction and on the underlying file system. Currently, the functionality of the graphical model editors is limited and therefore they are used to view and interact with models that have been constructed using text editors. LEAP models constructed using the tool can be saved as text files or in an XML format that retains graphical information and tool state that can be

reloaded at a later date.

Figure 1 shows a snapshot of the LEAP tool after loading in the bCMS definition that is contained in the text files `case_study.cmp` and `mk_coordinator.op`. The former contains the definition of a component model (hence the `.cmp` and the latter defines a function (hence `.op`) that is used as a product-line definition for co-ordinatoing components.

The top-left panel is a view on the file-system that can be used to create, edit and delete LEAP source files. The top-right panel shows a LEAP text editor that contains the contents of the `mk_coordinator.op` file. The bottom-right panel contains messages printed from the system. The middle-bottom panel is a property editor that is used to view and change properties of selected model elements. The bottom-left panel shows all the model elements in the system as a tree; it shows the `case_study` component with two sub-components called `police` and `fire`.

In order to run the bCMS model, the user opens up the `police` component and navigates to an operation named `display` that can be invoked using the mouse. As will be described in the rest of this document, from that point on the bCMS system is driven from a simple GUI embedded as a panel within the LEAP tool.

All the diagrams in the rest of this document are exported from the LEAP tool.

3 Architecture Models

LEAP models consist of hierarchically organized collections of components. Components bind names to values and therefore act as name-spaces. There is a root component called `leap` and therefore, all named components can be reached by navigation from `leap`. However, this does not mean that the structure of LEAP models is static since components can be both anonymous and can be used as normal data values, *i.e.* they are *first-class* data values.

Components behave like black-boxes: they consume messages from their input ports and produce messages on their output ports. Messages consist of a name and a sequence of values. Component ports are named and can be *connected* so that the output port of one component is connected to the input port of another component. When a message is sent to an output port it is broadcast to all connected input ports. An input port can also be connected to an input port of a child component so that messages sent to a component are immediately delegated to its children.

The component features described above are used in the definition of the bCMS system defined below:

```
1  component bCMS {
2    police = mk_coordinator(
3               'police',
4               Location(0,0),
5               [Crisis('M25 Crash',0,Location(2,2),[],[],[])],
6               [Resource('Police-Car1',Plan([]),[])])
7    fire = mk_coordinator(
8             'fire',
9             Location(0,2),
10            [],
```

```
11              [Resource('Fire-Engine1',Plan([]),[])])
12    init {
13      connect(police.outputs,fire.commands);
14      connect(fire.outputs,police.commands)
15    }
16  }
```

Each component can define the following: bindings; sub-components; goals; operation specifications; ports; operations; state; initialization code; rules; a theory; a state machine. Most of these features will be used in the definition of bCMS. The component `bCMS` contains two bindings: `police` and `fire`. The function `mk_coordinator` is defined in the file `mk_coordinator.op` and is below, it returns a new component based on the supplied arguments that initialize the name, location, crises and resources of the coordinating component:

```
1  mk_coordinator(coord_name,init_loc,init_crises,init_resources) {
2    component {
3      ...
4    }
5  }
```

All of the textual definitions in the rest of this report are part of the definition of the component returned by `mk_coordinator` and therefore should be understood to be part of the body represented by `...` above.

The component `bCMS` also contains an **init** block that is used to connect the two components together: output messages from the `police` component are sent to the `fire` component and vice versa.

The component architecture for bCMSis shown in figure 2. It shows the two components together with their input ports (white boxes) and their output ports (black boxes). The ports have names and the connections between them are labelled with the interfaces that the ports support. The two connections support the same interface type, for example is `fire` sends a message `contact(name,loc)` to the output ports named `outputs` then the message is sent to the input named `commands` of the `police` component. The ticks in the top-left hand corner of each component should be ignored for the purposes of this case study, they denote whether or not all of the data invariants for the component are satisfied.

The messages defined in interfaces are typed. LEAP supports basic types **str**, **int** and **bool** along with types `[t]` that denotes a list of all elements of type `t`. The type **void** is used to denote an asynchronous message. LEAP supports both synchronous and asynchronous message sending. This case study only uses asynchronous messages.

4 Information Models

In LEAP each component maintains a private state. The state is a list of terms of the form `Name(v,...,v)` where `Name` is any name starting with an upper-case letter and each `v` is a LEAP value that is defined as follows: a LEAP value `v` is one of: a string `'example'`; an integer; a boolean; a list of values `[v,...,v]`; a

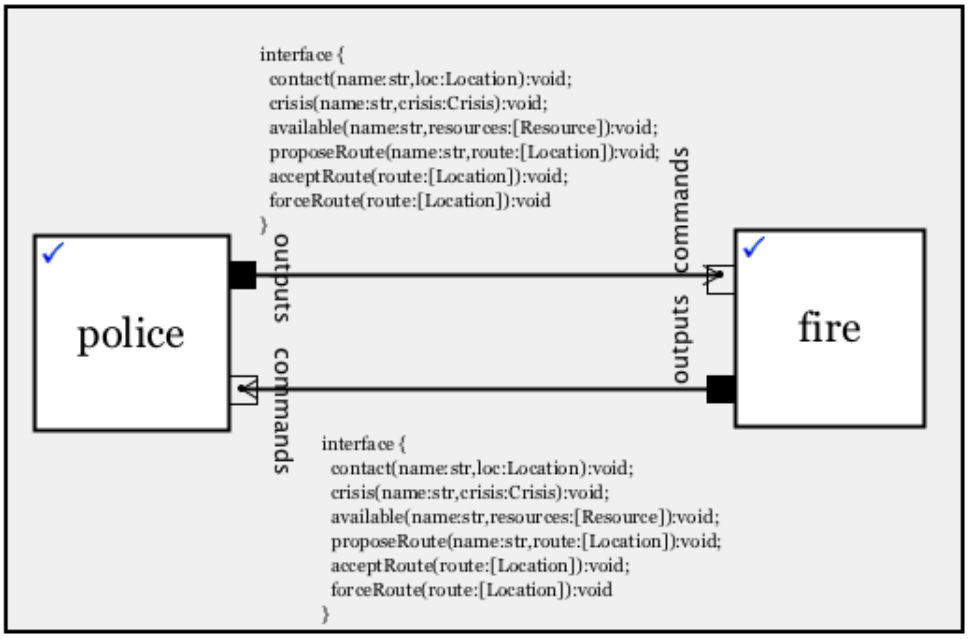

Figure 2: The bCMS Architecture

cons-pair `h:t` where head `h` and tail `t` are values; a term `Name(v,...v)`; a function `fun(x1,...,xn) e` where `xi` are argument names and `e` is a LEAP expression; a component. LEAP models are values too, but we do not need to treat them as such for the purposes of bCMS.

A component state is defined as a class-model. The model for bCMS is shown in figure 3 and is defined in file `mk_coordinator.op` as follows:

```
1  model {
2    class CoOrdinator {
3      name:str;
4      location:Location;
5      colleagues:[CoOrdinator];
6      crises:[Crisis];
7      resources:[Resource]
8    }
9    class Crisis {
10     description:str;
11     time:int;
12     location:Location;
13     victims:[Victim];
14     witnesses:[Witness];
15     allocated:[Resource]
16   }
17   class Time { time:int}
18   class Location { x:int; y:int }
19   class Person { name:str }
20   class Victim extends Person { medical:str; location:Location  }
21   class Witness extends Person { statement:str }
```

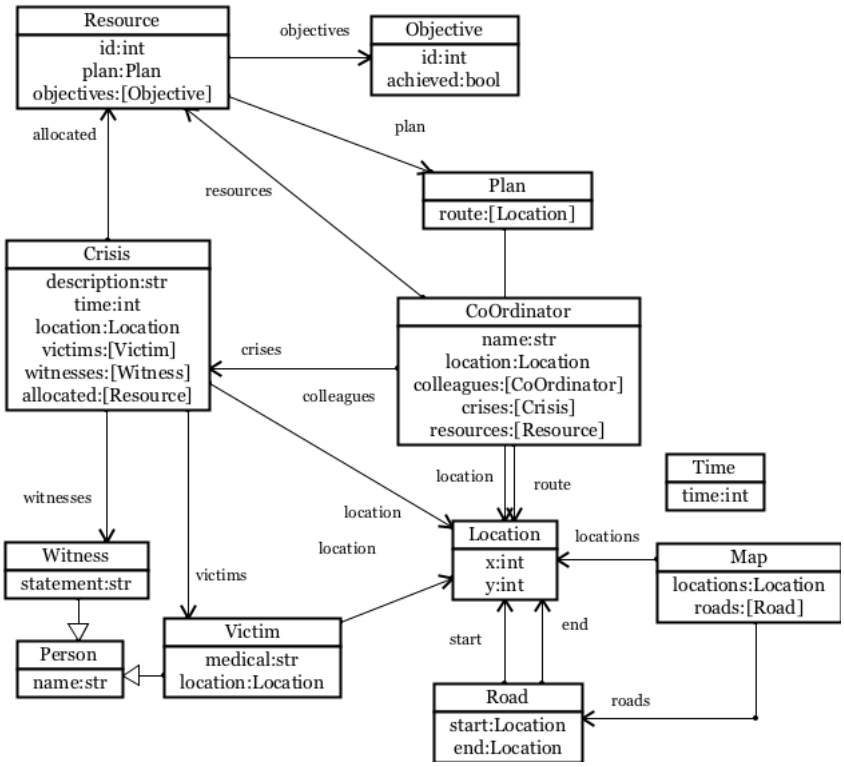

Figure 3: bCMS Information Model

```
22    class Resource { id:int; plan:Plan; objectives:[Objective] }
23    class Objective { id:int; achieved:bool }
24    class Plan { route:[Location] }
25    class Map {locations:Location;  roads:[Road] }
26    class Road { start:Location; end:Location }
27  }
```

A model definition is to be understood as follows. The state of a component is represented as a list of terms. Each term should correspond to an instance of a class definition (or an association which is not used here). For example, consider the class `Time`, it defines a single field named `time` with type **int**, therefore an instance-term is `Time(10)`. Where there is more than one field, such as `Location` the order of the fields is important when constructing a term: `Location(1,2)` has a value of `1` for `x` and `2` for `y`. A possible state for a component with the model given above is: `[Time(1),Location(1,2)]`. By default there can be any number of instance-terms for a given class (invariants can be used to express multiplicity requirements).

The model relates to the bCMS as follows:

CoOrdinator A co-ordinator is used in two ways: to record the details of a component within its state; there should be one such instance-term for each

co-ordinating component. In addition, a co-ordinator will use instances of `CoOrdinator` to record information about its colleagues in order to collaborate on a crisis. Therefore the single instance representing a co-ordinator will contain a list `colleagues` that is builds up over time. In addition, each co-ordinator has a name, a location, a list of crises that it is handling and a list of resources that can be used.

Crisis A crisis has a description, a time at which it occurred, and a location. The crisis contains a list of victims and witnesses. It has a list of allocated resources that should be owned by its managing co-ordinator.

Resource A resource is under the control of a co-ordinator. It has a unique id and a plan for getting from its current location to another location. A resource maintains a list of objectives that describe the tasks allocated by its owning co-ordinator.

Objective An objective is unique and either has been achieved or not. The details of particular objectives are outside the scope of the model.

Plan A plan is just a list of locations.

Map Each co-ordinating component maintains a map of the area in its state. A map is a graph whose nodes are locations and whose edges are roads that link locations.

Location A location is represented as an `(x,y)` co-ordinate.

Road A road simply links locations.

Time A co-ordinating component maintains its local time so that it can determine wether or not time-out events need to be generated.

Person Every person has a unique name.

Witness A witness provides a statement about the crisis.

Victim A victim has a medical history and a location.

Each LEAP component may specify an initial state via a **state** clause in its definition. Recall that some of the information necessary to initialize a co-ordinator component is passed to the `mk_coordinator` operation as argument values:

```
1  state {
2    Time(0)
3    CoOrdinator(coord_name,init_loc,[],init_crises,init_resources)
4    Map([Location(0,0),Location(2,0),Location(2,1),Location(2,2),Location(0,2)],
5     [Road(Location(0,0),Location(2,0)),
6      Road(Location(0,0),Location(0,2)),
7      Road(Location(0,2),Location(2,2)),
8      Road(Location(2,0),Location(2,1)),
9      Road(Location(2,1),Location(2,2))
10   ])
11 }
```

5 Invariants

LEAP allows a component to specify data invariants that are conditions which must hold at all times for the state of the component:

```
invariants {
  ...
}
```

The LEAP expression language includes a boolean expression sub-language that is similar to first-order predicate calculus and OCL. For example the following invariant requires that a component contains only a single instance of the CoOrdinator class at the top-level (other instances may be nested as colleagues):

```
1 oneCoOrdinator {
2   length([name | CoOrdinator(name,_,_,_,_) <- state]) = 1
3 }
```

The invariant defined above is named `oneCoOrdinator`. The body of the invariant (line 2) requires that the number of terms in the state of the component that match the pattern `CoOrdinator(name,...)` is exactly 1, *i.e.* there is only one instance of `CoOrdinator`. Line 2 shows an example of a list-comprehension. In general:

```
[ e | p1 <- e1, ..., pn <- en ]
```

denotes the list constructed by selecting all values from lists `ei` that match patternspi in the order that varies selection from `ej` more frequently than `ei` where `j > i`.

The following invariant requires that the location of the co-ordinator is on the map:

```
1 locatedCoOrdinator {
2    exists CoOrdinator(_,loc,_,_,_) {
3       exists Map(locs,_) {
4           exists loc in locs
5       }
6    }
7 }
```

The invariant above provides an example of the use of `exists` (3 times). An `exists` expression consists of a pattern, an optional list (that defaults to **state**) and a boolean valued expression (that defaults to **true**. Therefore the body of `locatedCoOrdinator` states that there must be a co-ordinator and there must be a map, both in the state of the component, and the location must be in the map.

6 Theories

LEAP supports component *theories* that are used to deduce facts from the current state of the component. A theory is defined as a collection of deduction rules. The *theorems* of a component are the facts it contains as terms in its current state and the facts that can be deduced using the rules.

LEAP theories are defined in a **theory** clause in a component definition. The rules are used in the same way as Prolog rules over a Prolog database, except that

each component in LEAP has its own database and collection of rules. Components may use the theories of other components via message passing.

Each rule can be thought of as defining a term based on the existence of other terms which are either deduced using rules or are present in the component database, for example `Append` defines terms of the form `Append(l1,l2,l3)` where `l3` is the list containing elements from `l1` followed by the elements from `l2`:

```
1 Append([],l,l) { }
2 Append([x|l1],l2,[x|l3]) {
3   Append(l1,l2,l3)
4 }
```

The example above defines two rules. The first rule defines a fact `Append([],l,l)` for any list `l`. The second rule (lines 2-4) uses the notaion `[h|t]` that denotes a list with a head `h` and tail `t`. The second rule defines a relationship between two non-empty lists and a third list `l2` in terms of an existing relationship defined by `Append(l1,l2,;l3)`.

LEAP provide the **prove** expression that is used to deduce facts within a component, for example:

```
prove Append([1,2,3],[4,5,6],$l) {
  print(l)
}
```

prints out `[1,2,3,4,5,6]`. Note the use of `$` as a prefix for unbound variables in the term. Another example is `Member` that defines list membership:

```
Member(e,[]) { fail }
Member(e,[e|rest]) {}
Member(e,[x|rest]) {
  Member(e,rest)
}
```

The following rules deduce facts about roads in a map:

```
Road(loc1,loc2,[]) { fail }
Road(loc1,loc2,[Road(loc1,loc2)|roads]) { }
Road(loc1,loc2,[ignore|roads]) {
  Road(loc1,loc2,roads)
}
```

Finally the following rules deduce routes between two locations:

```
Route(loc,loc,route,visited){  }
Route(loc1,loc2,[loc1,loc2],visited) {
  Map(locs,roads);
  Road(loc1,loc2,roads)
}
Route(loc1,loc2,route,visited) {
  Member(loc3,locs);
  not(Member(loc3,visited));
  Route(loc1,loc3,route1,[loc3|visited]);
  Route(loc3,loc2,[loc3|route2],[loc3|visited]);
  Append(route1,route2,route)
}
```

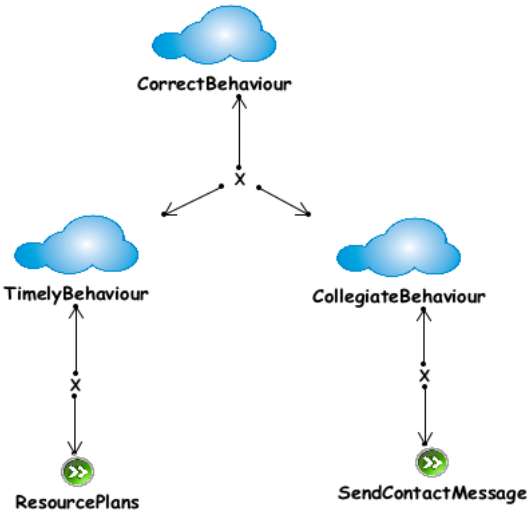

Figure 4: bCMS Goal Model

7 Goals and Requirements

LEAP supports requirements modelling using goal trees based on KAOS. Like KAOS, LEAP goal fall into a number of categories including informal goals and behavioural goals. An informal goal is used to provide a description of a functional or non-functional requirement with the intention that it will be refined further. A behavioural goal is expressed using a logic that expresses conditions over the behaviour of one or more components.

A LEAP component can be viewed as a state machine whose states are component states and whose transitions are labelled with messages. LEAP behavioural goals are expressed in a logic that denotes predicates over such state machines. It is beyond the scope of this report to provide a detailed description of the LEAP behavioural logic, however we provide some examples in the context of bCMS.

Figure 4 provides part of a goal model for bCMS. The goal `CorrectBehaviour` is informal and is the root of the behaviour tree for the components. We decompose the root into two informal sub-goals: `TimelyBehaviour` that relates to agreeing plans and `CollegiateBehaviour` that relates to contacting colleagues in order to work together. In order to provide an example of the different categories of goals, the two sub-goals are refines into two precisely defined behavioural goals: `ResourcePlans` and `SendContactMessage` (the indirect refinement is not necessary, but it provides some structure for the tree to show the reader the kind of refinement envisoned for larger goal models).

Goals are included in LEAP component models in a **goals** clause. The model source is described in the rest of this section. The root goal and its immediate refinements are defined as follows:

```
goal CorrectBehaviour {
  'The system shall behave correctly.'
}
goal TimelyBehaviour {
  'The system will have a plan in place
   for all resources within a given time limit.'
}
goal CollegiateBehaviour {
  'All controllers must introduce themselves
   to their colleagues before a crisis strikes.'
}
refine CorrectBehaviour <- TimelyBehaviour and CollegiateBehaviour;
```

The behaviour goals are expressed using the LEAP behaviour logic, for example:

```
1  behaviour ResourcePlans {
2    'At all times, if a crisis occurs at time t then
3     plans are in place by time t + 3.'
4    [](CoOrdinator(n,l,cs,[Crisis(d,t,l',v,w,[])],rs) ->
5       <>(CoOrdinator(n,l,cs,[Crisis(d,t,l',v,w,rs')],rs') and
6             forall Resource(i,Plan(p),a) in rs' { p != [] } and
7             Time(t') and t' < t+3
8           )
9     )
10 }
11 refine TimelyBehaviour <- ResourcePlans;
```

The formula in lines 3-7 uses the operator `[]` to mean *at all future times*, `->` to mean logical implication, `<>` to mean *at some future time*. The temporal operators allow us to roam over the future states of a component in order to enforce conditions on its contents.

The following example shows an example of the **U** operator; `p` **U** `q` holds when `p` is true until the point at which `q` is true:

```
1  behaviour SendContactMessage {
2    'The contact message must be sent at or before the crisis happens.'
3    <>([contact(name,loc)] and CoOrdinator(name,loc,colls,crises,res)) U
4    CoOrdinator(name,loc,colls,[crisis],res)
5  }
6  refine CollegiateBehaviour <- SendContactMessage;
```

8 Behaviour

Component behaviour is defined in three ways: named functions within a **operations** clause; a state transition machine defined within a **machine** clause; event detection rules defined within a **rules** clause. The bCMS system will use all three mechanisms for expressing behaviour. In principle, everything can be expressed using just one of these behavioural definition mechanisms, however LEAP provides all three because each mechanism has features that make it appropriate in particular circumstances. This section provides an example of LEAP operations.

LEAP behaviour occurs in response to message processing and to state change. When a message is processed, there may be an operation with the appropriate name and arity, and there may be a state transition labelled with the messgage and whose guard is enabled. In this case the operation is called and then the transition is fired.

Rules and transitions without messages both monitor a component's state. When the state changes, rules and transitions may become enabled. A single enabled rule or transition is fired on each component execution cycle (whether or not there is a message being processed). Of course, the resulting change of state may cause enabled rules and transitions to become disabled.

The following operation is called when a component receives a contact request from another co-ordinator:

```
1  contact(name',loc') {
2    find c=CoOrdinator(name,loc,cs,crises,rs) {
3      case cs {
4        { CoOrdinator(name',_,_,_,_) } U _ -> print('already registered')
5        else {
6          replace c with
7            CoOrdinator(name,loc,CoOrdinator(name',loc',[],[],[]):cs,crises,rs);
8          outputs <- contact(name,loc)
9        }
10     }
11   } else error('no coordinator')
12 }
```

Line 2 is an example of a **find** expression that matches a pattern against the current component state (or a sequence of values if supplied). The **case** expression on lines 3-10 is used to determine whether or not the co-ordinator making contact has already been registered. Line 4 provides an example of a set-pattern that matches over lists but does not require the elements of the list to be matched in order. A pattern `p` **U** `q` matches a list so that `p` and `q` are non-overlapping subsets of the list. In the case of line 4 the set pattern is used to extract an element from the list of collaborators with the supplied name `name'`. The use of the set pattern ensures that the term will be extracted no matter where it occurs in the list. If the pattern does not match then the **replace** expression (lines 6-7) updates the component state, and a message is sent to the output port (line 8) so that contact is made in the opposite direction.

Operations for `crisis` and `available` that are used to swap crisis information and resource availability are very similar to `contact` and do not involve any new LEAP features.

The co-ordinator that is designated as providing the plan uses the following operation to construct and disseminate the information:

```
1  plan() {
2    find c=CoOrdinator(name,mystart,coordinators,[Crisis(d,t,end,v,w,_)],r) in state {
3      prove Route(mystart,end,$myroute,[mystart,end]) in state {
4        let r' = [ Resource(rid,Plan(myroute),[]) | Resource(rid,_,_) <- r ]
5        in replace c with CoOrdinator(name,mystart,coordinators,[Crisis(d,t,end,v,w,r')],r');
6        for CoOrdinator(name',costart,_,_,_) in coordinators {
7          prove Route(map,costart,end,$coroute,[costart,end]) in state {
8            outputs <- proposeRoute(name',coroute)
9          }
10       }
11     } else print('no route from ' + mystart + ' to ' + end + ' in ' + map)
12   } else error('no coordinator')
13 }
```

The operation uses **prove** to deduce a route for both the co-ordinator and their

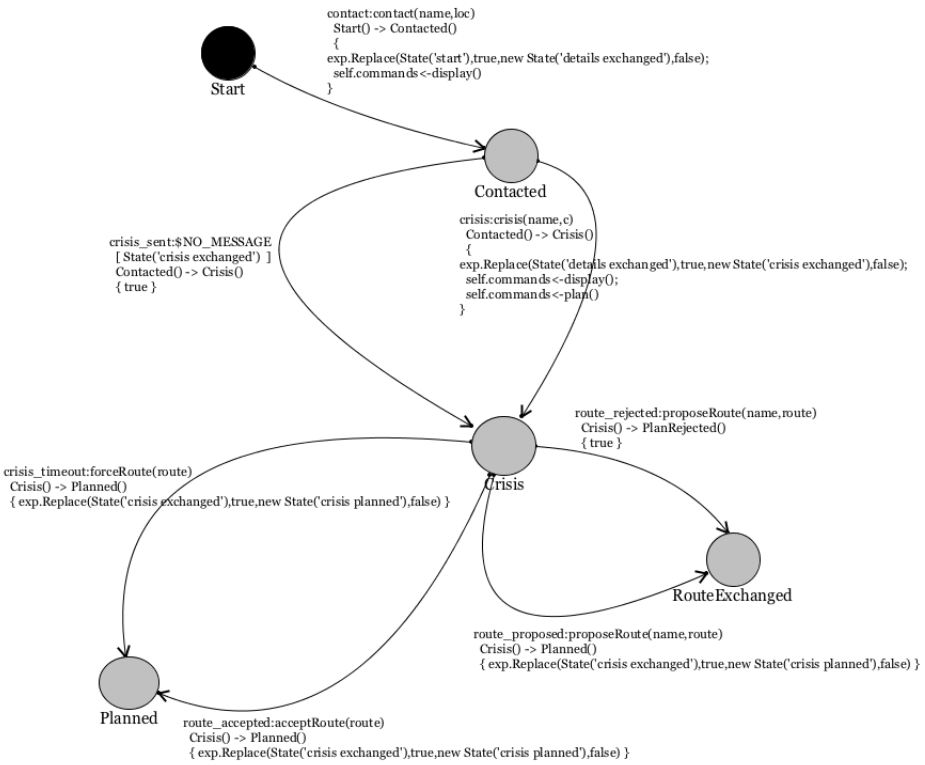

Figure 5: bCMS State Machine

colleagues. The local route `myroute` is added to the component state (line 5), note that a list-comprehension is used in line 4 to update all the local sources with a planned route. For each colleague a new route is planned (line 7) and is sent to any connected co-ordinators vis the output port (line 8).

When a component receives a proposed route, the route may be rejected or accepted. Both of these are handled by message exchange, but are not defined here as they resemble `plan` and do not introduce any new LEAP features. Planning may timeout in which case a route is forced. The operations that handle forced outing are the same as `plan` except routes are not forced, they must be accepted.

9 States and Events

9.1 State Machines

Section 8 shows how LEAP operations can define the behaviour in terms of handling messages (and being directly called internally within a component). The behaviour of a component can also be controlled by a state machine and event driven rules. In principle, any behaviour can be defined using machines and rules, however, it can become very complex. State machines are arguably best used to

control the life-cycle of an object and rules to control exceptional behaviour, leaving operations to handle the more complex aspects of component behaviour.

Figure 5 shows the bCMS state machine constructed from the following definition:

```
1  machine(Start) {
2    contact contact(name,loc) Start -> Contacted {
3      replace State('start') with State('details exchanged');
4      commands <- display()
5    }
6    crisis crisis(name,c) Contacted -> Crisis {
7      replace State('details exchanged') with State('crisis exchanged');
8      commands <- display();
9      commands <- plan()
10   }
11   crisis_timeout forceRoute(route) Crisis->Planned {
12     replace State('crisis exchanged') with State('crisis planned')
13   }
14   crisis_sent [ State('crisis exchanged') ] Contacted->Crisis;
15   route_proposed proposeRoute(name,route) Crisis -> Planned {
16     replace State('crisis exchanged') with State('crisis planned')
17   }
18   route_accepted acceptRoute(route) Crisis->Planned {
19     replace State('crisis exchanged') with State('crisis planned')
20   }
21 }
```

The machine is defined in terms of transitions that hold between states. The machine specifies a starting state `Start` and a collection of transitions; each transition has a name, an optional trigger message, an optional guard, a pair of states and an optional action. For example the transition `contact` (lines 2-5) is enabled when the machine is in state `Start` and when the trigger message `contact` is processed. There is no guard, *i.e.* no condition that must be satisfied in terms of the component state. The action updates the state (line 3) and sends a message (line 4).

The bCMS machine uses an extra term `State(s)` in the component's database to keep track of which state is current; this will be used by the GUI in section 11 to determine what information to display to the user.

Transition `contact` is used to detect a new contact and to update the display. Transition `crisis` is used to detect the transfer of crisis information and to initiate route planning. Transition `crisis_timeout` is used to detect when a route is forced on a component. Transition `crisis_sent` is an example of a transition with only a guard. If the state is updated as required by the guard then the transition becomes enabled.

Figure 5 shows the bCMS machine in the initial state. As transitions are fired, the current state (shown as a black circle) is updated on the diagram.

9.2 Rules

State machines offer a structured way to express behaviour. Sometimes, state-based behaviour is less structured, for example when a transition occurs on every state. Rather than clutter a state machine with repeated transitions, it is possible

to factor the transition out as a single rule. Rules are defined in the **rules** clause within a component. For example, the bCMSsystem must enforce an agreed plan before a given timeout. This can be achieved using the following rule:

```
timeout: CoOrdinator(_,_,_,[Crisis(_,t,_,_,_,r)],_)
              Time(t')
              ?(t < (t' - 3))
              ?(noPlan(r)) {
  timeout_plan()
}
```

The rule matches against the current state of the component. There are two state patterns and two conditions. The first state pattern matches against a co-ordinator that has a single crisis. The time of the crisis is `t` and the allocated resources are `r`. The second state pattern matches against the current time `t'`. The first condition requires that 3 time units have passed since the crisis was recorded. The second condition requires that the predicate `noPlan` is satisfied. The predicate is defined as an operator:

```
noPlan(resources) {
  exists Resource(_,Plan([]),_) in resources
}
```

The body of the rule cals the operator `timeout_plan` which is the same as `plan` above except that it forces a route on all of the colleagues.

10 Instance Models

The LEAP tool allows a model to be executed step-by-step and provides views on the state of the components. At any stage the current state can be visualized in terms of the original model. For example, after introducing themselves, after swapping crisis information and then agreeing a plan, the Fire controller is shown in figure 6 and the Police controller is shown in figure 7.

The ability to visualize component states in this way provides a means for checking a model execution and can be used to build up an execution trace. The states can be included as design information to guide system development.

11 Java Components and GUI

The LEAP language supports executable models of component-based systems. LEAP components implement a specific interface that is used by the LEAP execution engine to deliver and receive messages. The internal details of the component is opaque to the execution engine since components simple map input messages to output messages. Therefore, it is possible to load the LEAP execution engine with components implemented in languages other than LEAP. Since LEAP is written in Java, there is a Java interface that can be used to implement any Java class so that it loads on to the engine and behaves as any other LEAP component.

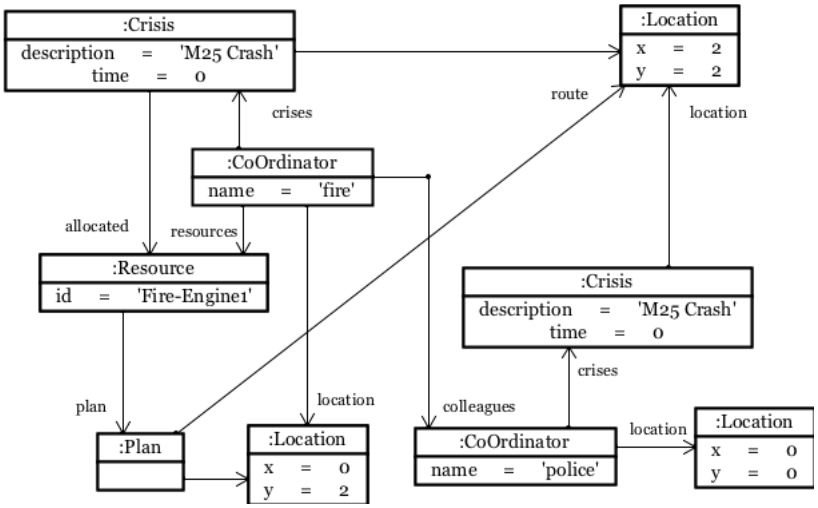

Figure 6: State of Fire CoOrdinator

LEAP provides a built-in function called `jcomponent` to load and instantiate Java classes that implement the LEAP component interface. The single argument to this function is a string that names the Java class. The LEAP tool provides some built-in Java classes in whic way, including a class named `frames.GUI` that implements a simple HTML-style user interface. The definition of the component in `mk_component.op` includes the line:

```
gui = jcomponent('frames.GUI')
```

resulting in a nested component called `gui` that provides a single input port named `in` with a single message `display`. The argument to display is a term `t` defined by the following grammar:

```
t ::= Table([[t...]...)              Tables (like HTML tables).
  |   Text(str)                      Literal text.
  |   Input(str,str)                 A named input field.
  |   Button(str,fun(n) e)           A button with a call-back.
```

The idea when a display term is sent to `gui` it causes the display to be updated. Pressing any buttons causes the associated function to be invoked (functions are proper closures, *i.e.* their bodies are evaluated in the context in which the closure was created). A button-function is supplied with a record that binds all the input field names with their contents. Therefore:

```
Table([
  [Text('name'),Input('name','')],
  [Button('login',fun(r) print('login: ' + r.name))]
])
```

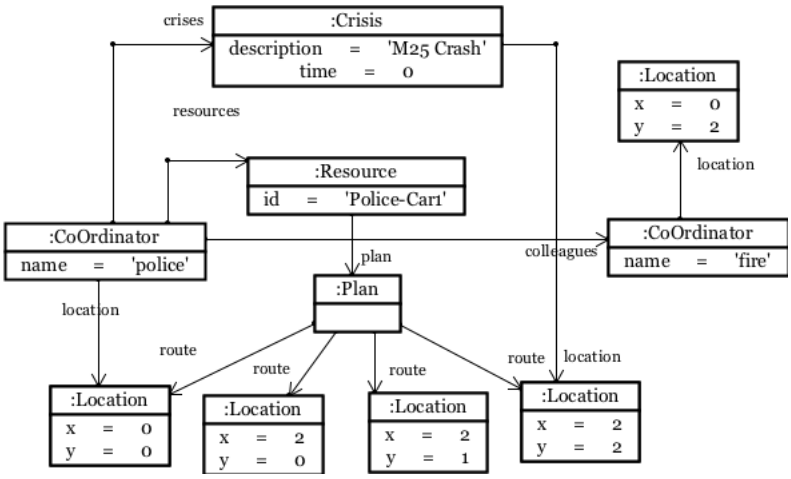

Figure 7: State of Police CoOrdinator

displays a name field and prints out the name that has been entered when the button is pressed.

The bCMS component defines an operation `display` that builds a term and sents it to the `gui` component. The display depends on the current state of the component as defined by its state machine. The code is as follows:

```
display() {
  case getState() {
    State('start') ->
      gui.in <- display(Table([
        timeRow(),
        nameRow(),
        crisisRow(),
        communicateButton(),
        tickButton()]));
    State('details exchanged') ->
      gui.in <- display(Table([
        timeRow(),
        nameRow(),
        crisisRow(),
        tickButton()] + crisisButton()));
    State('crisis exchanged') ->
      gui.in <- display(Table([
        timeRow(),
        nameRow(),
        crisisRow(),
        tickButton()]));
    State('crisis planned') ->
      gui.in <- display(Table([
        timeRow(),
        nameRow(),
```

```
        crisisRow(),
        tickButton()]))
  }
}
```

Each display term relies on row-construction functions that are defined as follows. The time row displays the current time in the co-ordinator:

```
timeRow() { [Text('Time'),Text(getTime())]  }
getTime() { find Time(t) { t } else 0 }
```

The name row displays the name of the co-ordinator:

```
nameRow() { [Text('name'),Text(coord_name)]  }
```

The crisis row displays details about the current crisis or an appropriate message if all is well:

```
crisisRow() { [Text('crisis'),Text(getCrisis())]  }
getCrisis() {
  find CoOrdinator(_,_,_,[Crisis(d,t,Location(x,y),v,w,r)],_) {
     d + ' at ' + t + ' pos (' + x + ',' + y + ')'
   } else 'no crisis'
}
```

The communication button is used by the human operator to inform all connected co-ordinators that they should be registered as a colleague:

```
communicateButton() {
  [Button('Communicate',fun(e) outputs <- contact(coord_name,getLoc()))]
}
getLoc() { find CoOrdinator(_,l,_,_,_) { l } else Location(0,0) }
```

The tick button is used to update the current time and then update the display:

```
tickButton() { [Button('Tick',fun(e) { tick(); display() })] }
tick() { replace Time(t) with Time(t+1) }
```

A crisis button is available only when the crisis has been registered. Therefore it checks to see if the state contains a crisis, if so it returns a list of rows, otherwise it returns the empty list:

```
getCrisisData() {
  find CoOrdinator(_,_,_,[a_crisis],_) {
    a_crisis
  } else error('no crisis in ' + state)
}
crisisButton() {
  case getCrisis() {
    'no crisis' -> []
    else [[Button('Crisis!',fun(e) {
      outputs <- crisis(coord_name,getCrisisData());
      replace State('details exchanged') with State('crisis exchanged')
    })]]
  }
}
```

Figure 8 shows the start of an application where the `gui` components from both controllers have been displayed. The `police` component has the crisis and the `fire` component has yet to be informed. Both components do not know anything about each other. Figure 9 shows the `fire` component at this stage.

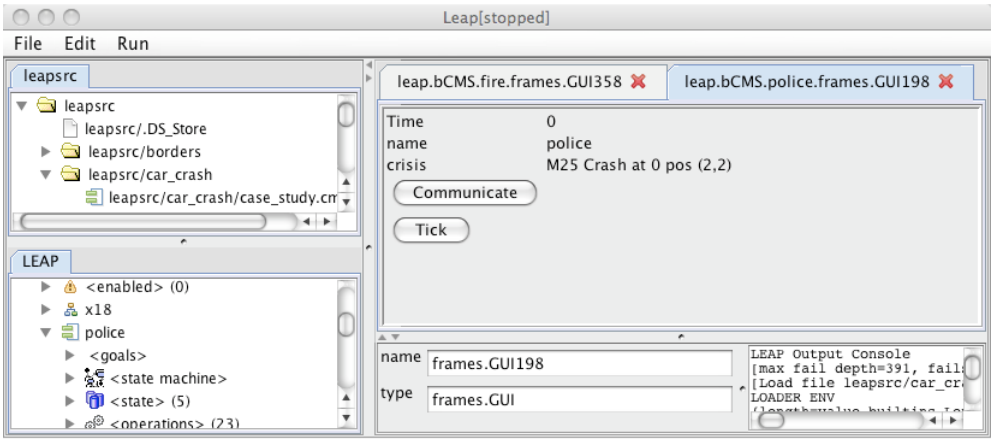

Figure 8: bCMS Police Start

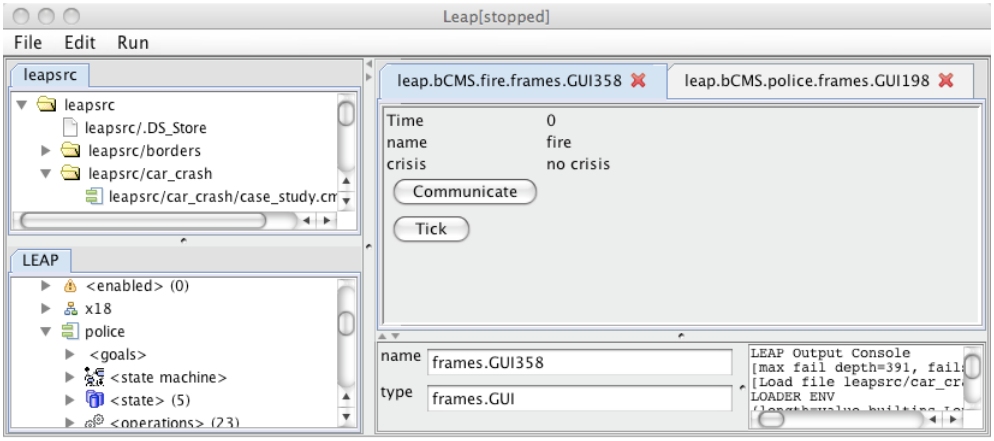

Figure 9: bCMS Fire Start

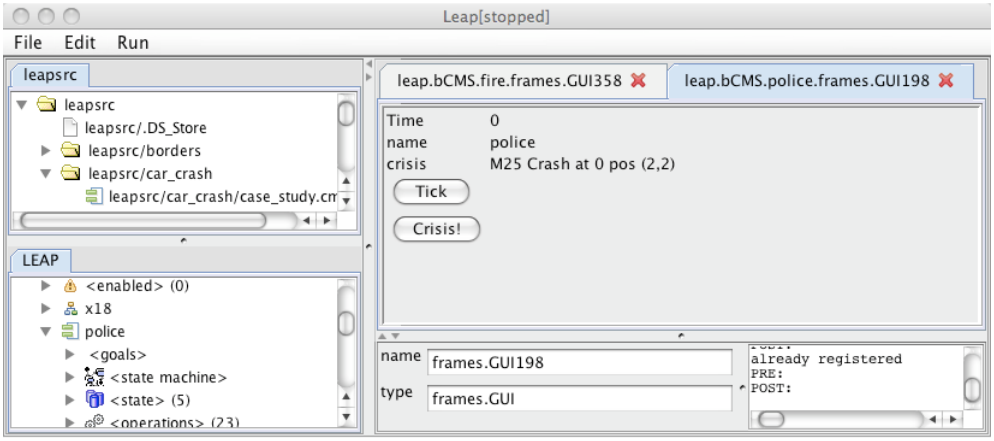

Figure 10: bCMS Police After Introduction

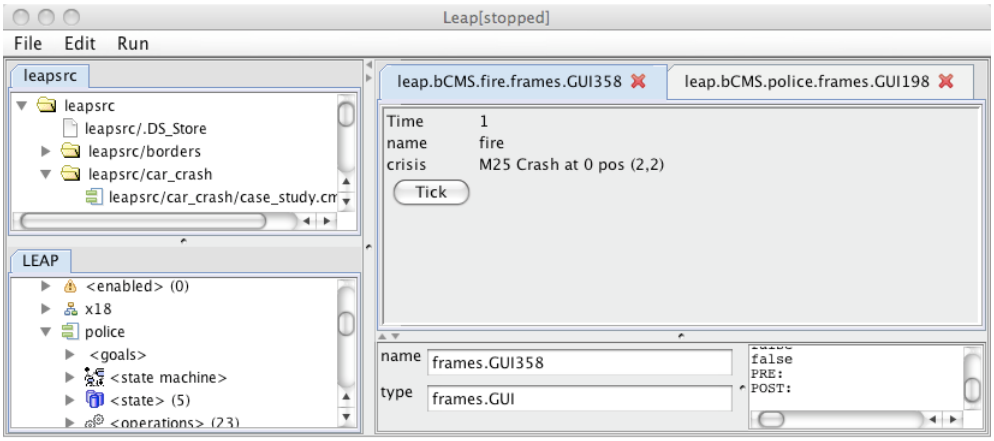

Figure 11: bCMS Fire After Crisis

The `police` component introduces itself to the `fire` component via the `Communicate` button. The result is a new GUI for the `police` component as shown in 10. When the operator presses the `Crisis!` button the crisis is sent from the `police` to the `fire` component and the plans are produced. Figure 11 shows that the `fire` component now has knowledge of the crisis. The internal state of the two components is shown in figures 6 and 7.

12 Conclusion

This document has provided an outline of how the bCMS system can be implemented using the LEAP approach. The key features of the approach are:

components The main modelling element is the component. Components have a simple semantics, the act as black boxes that process input messages and produce output messages.

information The state of a component is defined by a simple database of terms. The use of terms allows a variety of pattern atching styles to be employed. The lack of sharing between terms makes reasoning about component state straightforward. The state of a component can be visualized using object diagrams.

theories Components can contain theories that deduce facts from the current state of the component using rules. Theories can be used to encode knowledge about a domain within each component so that they react to incoming messages in an intelligent way.

higher-order Both components and functions are data values that can be used as arguments, message values and returned from function calls. This feature

makes it easy to define patterns of structure and behaviour. The bCMS model makes use of this to define the `CoOrdinator` component once and then *stamp it out* via function arguments. This is a form of product lines for components.

goals The requirements for a component are expressed as a goal model that consists of a mixture of free-text and formal logic.

constraints Invariants can be expressed over the state of component. The LEAP tool automatically checks the invariant constraints when the state of component changes.

behaviour Component behaviour can be expressed using operations, state machines and event-driven rules. This allows a mixture of approaches including service-oriented architecture and event-driven architecture to be used.

integration The LEAP approach is integrated with Java so that external systems can be wrapped as components and included in models.

References

[1] T. Clark and B.S. Barn. Event driven architecture modelling and simulation. In *Proceedings of the 6th International Symposium on Service Oriented System Engineering*. IEEE, 2011.

[2] T. Clark, B.S. Barn, and S. Oussena. Leap: a precise lightweight framework for enterprise architecture. In *Proceedings of the 4th India Software Engineering Conference*, pages 85–94. ACM, 2011.

[3] Tony Clark and Balbir S. Barn. A common basis for modelling service-oriented and event-driven architectures. In *5th India Software Engineering Conference, IIT Kanpur, ISEC 12*, 2012.

[4] Tony Clark, Balbir S. Barn, and Samia Oussena. A method for enterprise architecture alignment. In *PRET*, pages 48–76, 2012.