Tony Clark and Balbir Barn

Goal Driven Architecture Development using LEAP

Methods for goal driven system engineering exist and propose a number of categories of goals including behavioural, formal, informal and non-functional. This article goes further than existing goal driven approaches by linking goals directly to the semantics of an architectural modelling language called LEAP with an operational semantics. The behavioural goals are expressed using a Linear Temporal Logic and the non-functional goals are expressed as functions over meta-properties of the model. The meta-properties are supported using an encoding represented using Java reflection. The article describes the LEAP approach using a simple case study written in the LEAP language supported by the LEAP toolset.

1 Introduction

The architectures of modern IT systems are distributed and heterogeneous and therefore lend themselves to design using component-based approaches. A component based approach, as opposed to large scale ERP implementations leads to multiple possible configurations of system components raising questions such as what is the best component configuration and how to develop component-based designs. Any development that changes an architecture should start with a requirements analysis phase, yet most modelling approaches focus on *what* the system should do. In Architecture Design Languages (ADLs) or System Design Languages such as UML, functional behaviour is expressed using invariants and pre and post-conditions. In Enterprise Architecture (EA) these are represented by as-is and to-be architectures most clearly characterised by approaches such as TOGAF (Spencer et al. 2004). Other approaches such as Archimate (Lankhorst et al. 2010) also utilise informational, behavioural and structural models organised as different architectural layers such as business, application and technical infrastructure to express these architectures. Despite the exhaustive modelling performed to develop an architecture model for an organisation's new requirements, scant effort is applied to understand and codify the knowledge that represents the rationale of the *why* behind these architectural modelling decisions.

Requirements in the form of functional system specifications are supported by a number of technologies including UML and formal languages such as B and Z, and various logics. UML can be categorised as structured but imprecise and the formal languages as being precise but generally unstructured or difficult to map to implementation features.

In both cases, these technologies do not support motivational aspects of system development that are often expressed in terms of non-functional properties such as *cost*, *reliability* and *usability*. Motivation or Intention of business requirements, if supported satisfactorily, can provide a means for analysing the relationships between business requirements or needs and IT infrastructure and thus address one of the perennial issues in Information Systems/Information Technology (IS/IT) research, that of business-IT alignment. This relationship between business and IS/IT performance has received much attention from as far back as 1977 (McLean and Soden 1977) and a more recent review of the key issues being identified in Chan and Reich (2007).

Motivation or intention aspects of requirements engineering has resulted in a new branch requirements modelling based around goal oriented requirements engineering (GORE) techniques

(Mylopoulos et al. 1999) such as i* (Yu 1997; Yu and Mylopoulos 1994) and KAOS (Dardenne et al. 1993; Letier and Van Lamsweerde 2004; Van Lamsweerde 2008). GORE based techniques present a variety of options for analysis such as providing a more formal basis of how goals realise other goals, conflict between goals and the positive and negative contributions goals make to other goals. Further, the relationship between the proposed solution and actual need is more clearly delineated.

Requirements Engineering methods such as KAOS and i* aim to address the structured aspect of requirements in terms of goal modelling. Goals capture the motivation behind system design and goal modelling languages provide a mechanism for structuring the goals and linking them to system elements that are responsible for achieving the goals.

In order to be effective goal-modelling must address the following:

precision In the early stages a requirements engineer is likely to have a broad understanding of the required system. Therefore, goals should support informal discursive requirements. However, as requirements are refined, their precision should increase to the point where they can be, in principle at least, processed mechanically.

semantics Whether a goal is informal or formal, it must be possible, in principle, to provide its meaning. In general a goal is a predicate over some features of a system. Behavioural goals are predicates over system executions and therefore, it is important to be able to articulate such executions to the required level of precision. Non-functional goals are typically more difficult to express, however our proposal is that non-functional goals are predicates over *meta-properties* of a system (whether static or dynamic). If, in principle, a non-functional goal cannot be expressed precisely in these terms then it is not measurable and is of limited use in system development.

structure The use of requirements engineering techniques are justified in terms of system size and complexity. Therefore, requirements must have structure that allows them to be decomposed and analysed independently. Decomposition should support the analysis of alternatives. Ultimately, behavioural goals should decompose into system component responsibilities and therefore the goal model structure should support links to design elements that realise the specified behaviour.

LEAP is a technology for constructing and animating architectural models (Clark and Barn 2011, 2012; Clark et al. 2011). It is based on a small collection of features including: components and connectors, messages, operations, operation specifications, information models, events, state machines, and rules. The design of LEAP has been motivated by a desire to provide a simple collection of orthogonal executable modelling features that can be used as a basis for system design from enterprise-wide architectures through to individual IT components. Our approach is to use components as containers of information and behaviour, and to use messages between connected components as a basis for computation. Components can be used to represent both physical and logical features of a system, and the data stored in components and passed in messages between components, may include components themselves. Our claim is that by making components higher-order features of the LEAP language offers a highly expressive basis for system modelling at all levels without the need for a diverse collection of different elements.

LEAP is based on existing languages and approaches including the port-and-connector models of UML, class and object models of UML, state machines, the Object Constraint Language (OCL), functional programming languages particularly higher-order functions, list comprehensions and pattern matching, event driven programming, and KAOS. LEAP uses a functional language in two ways: to implement component behaviour and to abstract over system models. The LEAP

tool supports the leap language and provides textual and graphical editors for constructing and viewing LEAP models. The definitions in this article are LEAP source code and the diagrams (with the exception of Fig. 8) are generated by the LEAP tool.

Existing goal-modelling languages address precision, semantics and structure as described above. However the degree to which this is achieved is limited because the languages are either imprecise (such as BMM) or general purpose (such as KAOS and i*). In particular KAOS provides a formal language for behavioural goals based on temporal logic, however this does not map on to any specific executable system and therefore remains very general. In addition, no existing goal modelling notation addresses the issue of non-functional requirements in a precise way.

LEAP makes a contribution to architectural modelling in the following ways:

- LEAP brings together an integrated collection of orthogonal features that we propose as a basis for the design of component based architecture. Our claim is that these features are appropriate for high-level architectures such as those found in EA and also appropriate for smaller scale system architecture. This article provides an example of a system architecture but see Clark and Barn 2011, 2012; Clark et al. 2011 for other examples.
- LEAP extends component-based modelling with intentional features in the form of goals, this together with the executable features of LEAP makes it unique as a component-based design language. This article provides examples of the use of the intentional features.
- LEAP uses a formal logic to express behavioural goals over component executions, whilst other systems provide such mechanisms, LEAP is unique in that it integrates the logic with a traditional component-based modelling language. This article provides many examples of LEAP behavioural goals and defines the formal language.
- Our proposal is that non-functional goals can be formalised as meta-predicates over extra-calculational system properties. This allows so-called *soft* goals to be precisely defined in LEAP compared to other approaches that require non-functional goals to be expressed in natural language. This article provides a number of examples of non-functional goals in LEAP and describes the technical machinery that allows the non-functional goals to be checked.

This article describes the LEAP approach to goal modelling. We introduce the approach using a case study and then define the languages used. Finally we compare LEAP with other goal modelling approaches.

2 Case Study

Ruritanian General Practitioners (GPs) are required to provide an automated consultation booking system. Patients register with a medical practice in order to use it. When they register they may indicate a particular GP that they wish to see during consultations. The Ruritanian medical system is entirely on-demand: patients walk in to the practice and request a consultation. If the patient is registered with a particular GP then they will see that GP when they are free, otherwise the patient sees any GP. Being a Ruitanian GP is tough since they must always be available in the surgery. A record must be kept of all consultations and the medicines that are dispensed.

Apart from the functional requirements given above, Ruritania defines some non-functional requirements in terms of *fairness*, *cost*, *efficiency*, and *risk*. It requires that its medical practices are *fair* in the sense that no GP is overworked, all patients are seen within a sensible time, and no consultation takes too long. In addition, the *total cost of ownership* for a medical practice should be below a specified amount. The costs will include the development cost of the software, the costs of running the software, and the costs of the GPs. The *risk* of sensitive medical knowledge being leaked is reduced if the system is distributed

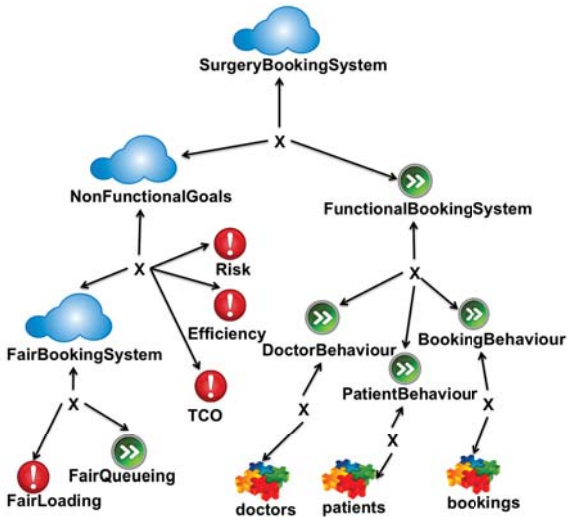

Figure 1: The Surgery Goal Tree

over more than one site. Finally, a system implements the requirement functionality as *efficiently* as possible.

3 LEAP: Goal Directed Models

3.1 Goals

Figure 1 shows the LEAP goal decomposition tree for the GP case study. All goals denote predicates over aspects of a system architecture, for example *SurgeryBookingSystem* are the conditions under which any architecture represents an acceptable system, whereas *Doctors* are the conditions for the correct behaviour of the part of the system that manages information about GPs. Goal types differ in terms of the type of language used to express the condition and in terms of the things that the goal can denote. LEAP supports goals of the following types:

informal An informal goal is expressed using natural language. It is intended to scope out an area of the system that is subsequently refined. A goal decomposition tree usually has an informal goal at its root. Informal goals are expressed in LEAP diagrams as clouds.

behavioural A behavioural goal uses *linear temporal logic* to precisely define the behaviour of some aspect of the system. A LEAP model consists of a collection of components each of which contains a database of terms. LEAP execution occurs in terms of messages that cause changes in component databases; therefore LEAP executions are sequences of states and messages. A behavioural goal is a condition that applies to LEAP executions. Behavioural goals are shown as nodes labelled ».

Behavioural goals are not necessarily limited to the scope of a single component. As a goal-decomposition tree is developed from root to leaves, the scope of behavioural goals nearer the root are likely to be scoped over sub-systems that comprise multiple interactive components. Behavioural goals near the leaves of the tree tend to relate to single components and may even be limited to single operations.

Invariant An invariant goal is something that must be true at all times during system execution. A behavioural goal that specifies a condition that must hold in all states of an execution is an invariant. However, the behavioural goals described above are specified using a language that is limited to component messages and states. Other types of invariant relate to meta-properties of a system such as cost, reliability, etc. Therefore, LEAP invariants can be expressed using the following meta-features: **`state`** which is used to reference the current system state in terms of its messages and database terms; **`calc`** that is used to reference the sequence of states in a system execution; **`reify`** that is used to map between model elements and database terms; **`intern`** that is used to map between database terms and model elements. Invariants are expressed on LEAP diagrams as `!`.

component Goals can be linked to specific LEAP components. The goal may be used to specify that the component has particular behaviour or meta-properties. Behavioural goals

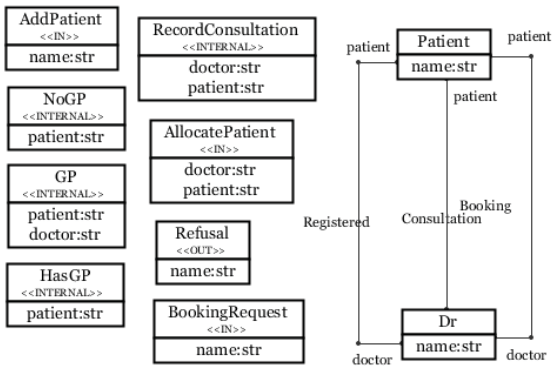

Figure 2: Goal Data

can be used to specify the behaviour of particular component operations and as such will map directly on to the specification contained in the component.

Goal decomposition is shown as nodes and links that connect the goal types described above. Decomposition may be in terms of conjunction (X) or disjunction (+). Decomposition is a mechanism for separation of concerns and for refinement.

The model shown in Fig. 1 has a root goal `SurgeryBookingSystem` that is decomposed into `NonFunctionalGoals` and `FunctionalBookingSystem`. The details of these goals are described in the following sections.

3.2 Behavioural Goals

The behavioural goals are defined with respect to the data types shown in Fig. 2. In LEAP, both the state of a component and the messages that are processed by a component are represented as *terms* whose types are defined by classes. We use UML-style stereotypes to designate the difference between messages and passive data. Figure 2 uses the tags «`IN`» and `OUT` to represent messages that communicate with the system environment. The tag «`INTERNAL`» is used to represent a message that is both produced and consumed by the system. The reason for using data for both active and passive information is that goal-based requirements need not commit to specific distinctions at such an early stage of development.

The `FunctionalBookingSystem` goal specifies a range of system behaviour. Behavioural goals use *Linear Temporal Logic* (LTL) to express constraints, for example:

```
always {
  forall BookingRequest(p) {
   Patient(p) implies
     before(10) {
       Consultation(_,p)
     }
  }
}
```

requires that after a booking request is recorded, if the patient is registered with the practice then a consultation must be recorded before 10 minutes have passed. The following goal requires that a request is politely and immediately denied for customers not registered with the practice:

```
always {
  forall BookingRequest(p) {
    not(Patient(p) implies next { Refusal(p) })
  }
}
```

Finally, the following requires that a patient is eventually seen:

```
always {
 forall BookingRequest(p),Dr(d) {
  Patient(p) and Registered(Dr(d),Patient(p))
   implies
    eventually {
      Consultation(Dr(d),Patient(p))
    }
  }
}
```

3.3 Non-Functional Goals

Functional goals can be expressed in terms of system behaviour that is represented in terms of *calculations* (sequences of run-time states). This may be expressed as pre and post-conditions (one step in the calculation), invariants (every step in the calculation), or as LTL expressions (sub-sequences of the calculation). Our proposal is that non-functional goals are those that require extra-calculational information, i.e., data that relates to any aspect of the system execution, but may not be directly necessary to express the

execution rules. Examples include the cost of resources that are used during execution, whether or not an architecture satisfies given structural guidelines, and the rates of component failure that lead to system exceptions.

Therefore, non-functional goals are expressed in terms of meta-properties of the system. The properties should be made sufficiently precise so that, at least in principle, they can be mechanically checked. Meta-properties may be static or dynamic. Static meta-properties include structural properties of the system models, for example checking how many connections a component has or placing a requirement on the overall number of components. Checking for architectural patterns is a meta-property of a system. In addition, it is important to allow developers to extend the basic meta-types of a system to support their own static meta-information that can be checked in constraints. A typical example of this is the extension of standard UML classes to introduce a new RDBMS table meta-type.

Static meta-properties can be extended to dynamic meta-properties in a straightforward way providing the system has a well defined dynamic semantics. Typically this will involve defining a static structure for the system and then extending it to sequences, trees or graphs of system states. Once these system execution structures are defined it is possible to define measurable dynamic non-functional properties in terms of the meta-properties of the individual states.

LEAP supports meta-access in the following ways. The contents of the current system state is available via the variable **`state`** and the sequence of states in a system calculation is denoted by **`calc`**. Any LEAP value can be transformed into a LEAP term that has a uniform structure and which can be processed using pattern matching, using the meta-operator `reify`. The inverse of `reify` is called `intern`.

In the following example goals, we will make use of some operators that allow a LEAP component to be processed as a LEAP term. The operator `walkComp` is defined below as a standard LEAP operator (all the code in this article is written in the LEAP programming language). The arguments of `walkComp` are `map` that transforms all components in a tree, `cons` that combines a mapped component with its mapped children; `sib` that combines mapped components with their siblings; `base` that is the result of mapping the empty sequence of component siblings.

```
walkComp(map,cons,sib,base) {
 fun (comp) {
  let f = fun(children) {
   case children {
    c:cs →
     sib(walkComp(map,cons,sib,base,c),f(cs));
    [] → base
   }
  } in cons(map(comp),f(comp.children))
 }
}
```

The operator `getComponents` is constructed by supplying `walkComp` with the identity mapping `id` and operators that build lists. The operator `getMess` maps a component to the messages it processes:

```
getComponents = walkComp(id,cons,app,[])
comp2Messages = fun(c) [ m |
  p ← c.ports,
  m ← p.messages
]
getMess = walkComp(comp2Messages,app,app,[])
```

Risk: The Ruritanian Government has identified that public sector systems are at risk if they entirely hosted in one place. Therefore, the goal `Risk` requires that any compliant system must be distributed over at least 2 hosts. The goal needs to reflect on the structure of the system and requires that each component has meta- information defining where the component is hosted:

```
let system = reify(self);
    C = getComponents(system);
    hosts = set([ intern(c).host | c ← C ])
in #hosts > 1
```

Efficiency: The efficiency of a system can be defined in relation to the amount of inter-component communication that is performed. The Ruritanian Government requires that any IT solution to the surgery requirements are *efficient* where

this is defined in terms of a measure of the number of possible messages within the system (recall that *risk* requires at least 2 different components).

The number of messages in a system model is a meta-property. It is found by reifying the system and mapping the resulting term to the number of messages it contains. The size of the resulting set is calculated by the following:

```
let system = reify(self)
in fold(add,0,set(getMess(system))) < 20
```

TCO: The total cost of ownership is defined the Ruritanian Government as the development costs, the hosting costs and the staffing costs of any IT system. The non-functional requirement that the TCO should be less than 1000 Ruritanian Rurs is a meta-constraint that is applied to properties of the model. Both the development and hosting costs are meta-properties of the components in the system. The staffing costs are calculated by mapping the state of all system components to the set of GPs that they contain and then multiplying by `GPcost`:

```
let system = reify(self);
   C = getComponents(system);
   devcosts = [intern(c).devcost | c ← C];
   devcost = fold(add,0,devcosts);
   hostcosts =
     [(intern(c)).hostcost | c ← C];
   hostcost = fold(add,0,hostcosts);
   staff =
     set([d | c←C,Term('Dr',[d])←c.state]);
   staffcost = #staff * GPcost * #calc
in devcost + hostcost + staffcost < 1000
```

FairLoading: The Ruritanian Government expects all GPs to put in a fair and equitable amount of effort. Some Ruritanian patients register with a particular GP in a medical practice and expect to see only that GP for any consultation. However most are happy that all GPs are of similar quality and will see the next available GP when they request a consultation. Therefore, fairness is defined to ensure that the difference in the number of GP consultations is never greater then 2 from the average:

```
let doctors = [Dr(n) | Dr(n) ← state];
   consultations(dr) =
    #([1 | Consultation(dr,p) ← state,
           ?([n | Registered(dr,p)←state]=[])])
   M = [ consultations(dr) | dr <- doctors ]
in (max(M) - min(M)) ≤ 4
```

4 Design

This section outlines the LEAP support for operation specification and for animating architecture designs. It provides a specification, architectural diagram, and implementation of the case study. It also describes how LEAP supports animation and visualisation of data in terms of object diagrams. An object diagram is used to show a snapshot of a running system. A simple interface for the case study is constructed using the LEAP built-in components for constructing interactive GUIs.

4.1 Component Architecture

The goal model shown in Fig. 1 links to leaf components named `doctors`, `bookings` and `patients`. The behaviour of these components is captured in the goal model using behavioural goals together with the non-functional requirements for the overall system.

Figure 3 shows the next level decomposition of the system where the components are connected to support message-based communication. Each component is named and has a collection of ports shown as boxes on the outside of the component box. Each port is named and may be designated for input (white boxes) or output (black boxes). For example the `bookings` component has a port that handles `requests` from the `gui` component and produces `patientRequests` to the `patients` component.

Each connection between ports has an interface type that is shown as text positioned close to the connection edge. The interface type defines the messages that may be sent along the connection. For example, the connection between `bookingCommands` in `gui` and `requests` in `bookings` is labelled with the following interface:

```
interface {
  addGP(name:str):void;
  requestConsultation(name:str):void;
  next():void
}
```

The message return types indicate whether the message is *synchronous* or *asynchronous*. Most messages in the case study are asynchronous and have the return type **void**.

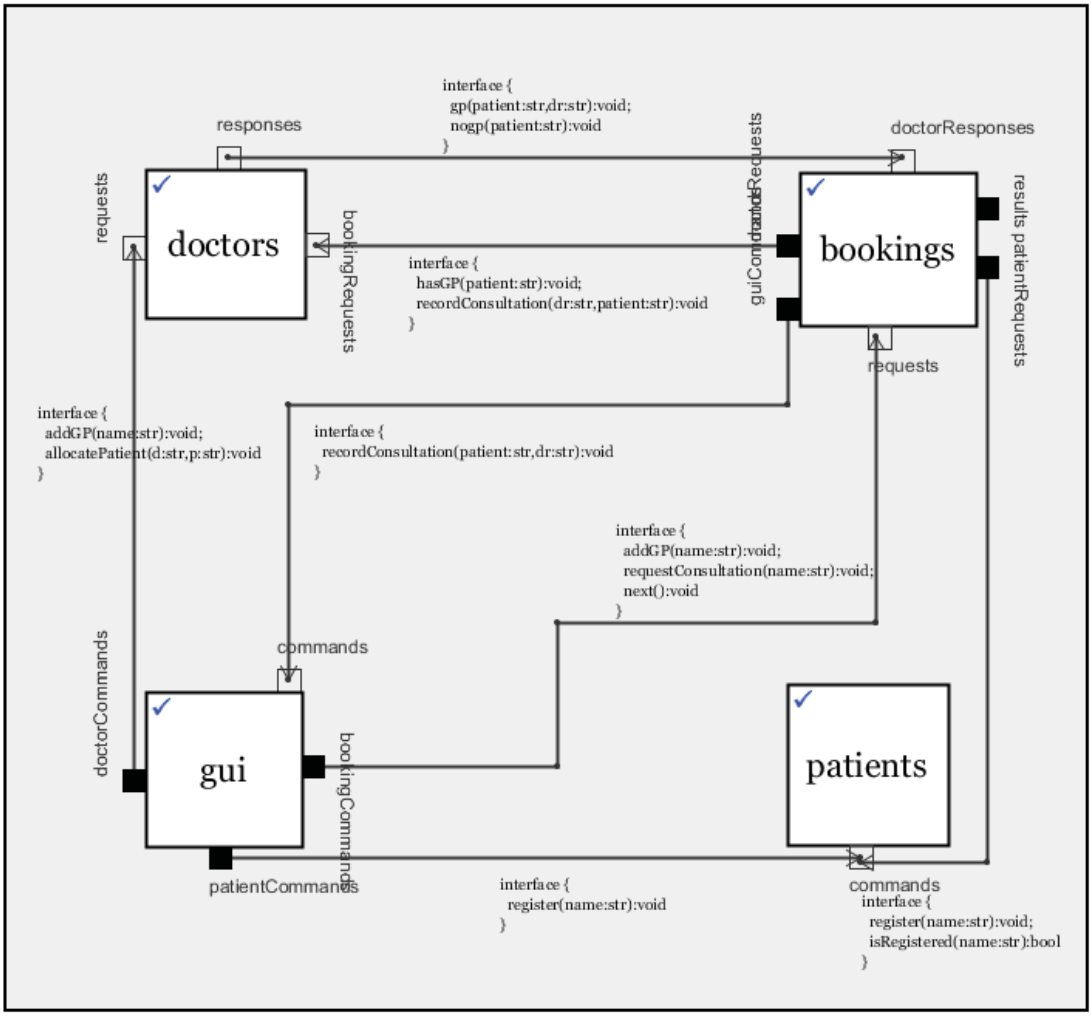

Figure 3: Surgery Component Architecture

4.2 Specifications

A goal decomposition tree should lead to the identification of a collection of components whose individual behaviours are specified by behavioural goals. The goals should identify the information content of each component and also define the messages that the components must support. The designer then has freedom to partition the messages between ports and to specify the behaviour of the components in response to each message.

LEAP provides a *specification clause* in the definition of a component that is used to specify the behaviour the component in response to messages. Individual behaviours are then simulated in LEAP using a variety of mechanisms including state machines, transition rules and operations. This section gives an overview of the specification clause in terms of the case study.

A specification clause contains a collection of message specifications that are defined using three types of sub-clause: pre-condition; post-condition; message-condition. A pre-condition is a predicate that must hold at the time the message is processed in order for the post-condition and the message-condition to hold. Both pre and

post-conditions are expressed in terms of patterns over the state of the component (although both may refer to general boolean expressions `e` via the syntax `?(e)`). The message-condition is a predicate that applies to the output ports of the component and uses pattern matching to determine message membership of the output queue.

A specification clause should follow directly from behavioural goals in the goal model. KAOS uses a similar method to determine patterns in the LTL formulas that can be ascribed to single operation calls. The following shows a fragment of the specification for the `doctors` component and defines the behaviour of the component in response to handling the `addGP` and `hasGP` messages:

```
spec {
 addGP(name:str):void {
  pre not(Dr(name))
  post Dr(name)
 }
 hasGP(patient:str):void {
  pre Registered(Dr(dr),Patient(patient))
  messages responses ← gp(patient,dr)
 }
 hasGP(patient:str):void {
  pre not(Registered(Dr(dr),Patient(patient)))
  messages responses ← nogp(patient)
 }
}
```

When refining the behaviour of a component from that imposed by goals to that provided by a specification clause, it may be necessary or useful to also refine the data model. LEAP goal models are associated with components. The model in Fig. 1 is associated with a component called `surgery` that contains the four components shown in Fig. 3. The data model for `surgery` is shown in Fig. 2 and is therefore used throughout the goal model.

We would like to refine the representation of patient bookings so that LEAP lists are used to implement a queue and therefore impose an ordering on processing the bookings for a GP. The refinement is shown in Fig. 4. It shows an example of LEAP data references that are displayed

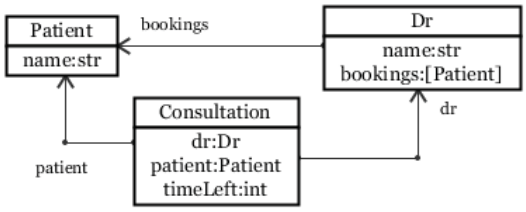

Figure 4: Patient Model

as links between classes with arrows. A reference is shown as a field and an edge, for example `dr:Dr` in `Consultation` and the edge labelled `dr`. This is because the classes are actually term-types and the field order is important, whereas the graphical representation using nodes and edges aids comprehension. A typical consultation term is:

```
Consultation(
 Dr('phibes',[Patient('fred')]),
 Patient('wilma'),3)
```

The specification clause for the `patient` component is shown below:

```
spec {
 addGP(d:str):void {
  pre not(Dr(d,_))
  post Dr(d,[])
 }
 requestConsultation(patient:str):void {
  messages doctorRequests ← hasGP(patient)
 }
 gp(p:str,dr:str):void {
  post Dr(dr,b) ?(exists Patient(p) in b)
 }
 nogp(patient:str):void {
  post Dr(d,b) and exists Patient(patient) in b
 }
 next():void {
  pre Dr(d,Patient(p):b)
      not(Consultation(Dr(d),_))
  post Dr(d,b) Consultation(Dr(d),Patient(p))
 }
}
```

4.3 Implementation

At this point in design, goals have placed behavioural and non-functional requirements on the system, the requirements have been refined into a component architecture including ports and connectors. An initial data model may have been

refined into local data models for each component and an associated specification for each of the messages that the component handles. The final step is to provide an implementation for each message.

LEAP provides three mechanisms for implementing component behaviour. Where a component exists in a number of pre-defined states and where behaviour can be conveniently defined in terms of state transitions, LEAP provides pattern directed state machines that monitor changes in the state of a component and fire transition when guards become satisfied. Our case study does not use state machines.

A less structured form of state-machine behaviour is provided in the form of rules that monitor changes in the state of a component and fire when the rule-condition is satisfied. The `bookings` component uses rules to manage consultations. Finally, a component defines a collection of named operations. Operations can be directly called from within the component and, if the operation name matches a message name, are invoked when a message is processed.

The rest of this section provides examples of the implementation of three of the `surgery` components. The GUI component is the subject of the following section.

patients: The state of a component is modified using **`new`** and **`delete`**. The **exists** operator is used to match patterns over the current state of the component:

```
component patients {
  devcost = 200
  hostcost = 10
  host = 'local hospital'
  operations {
    register(name) {
      new Patient(name)
    }
    isRegistered(name) {
      exists Patient(name)
    }
  }
}
```

doctors: The `doctors` component provides examples of the use of **`find`** that selects an element from the current state of the component that matches a given pattern. the **`replace`** ... **`with`** operator is used to replace a term that matches a pattern. The ← operator sends a message to the named port:

```
component doctors {
  devcost = 500
  hostcost = 10
  host = 'surgery'
  operations {
    addGP(name) {
      new Dr(name)
    }
    allocatePatient(dr,p) {
      new Registered(Dr(dr),Patient(p))
    }
    recordConsultation(dr,p) {
      find Consultations(Dr(dr),ps) {
        replace Consultations(Dr(dr),ps)
        with Consultations(Dr(dr),p:ps)
      } else new Consultations(Dr(dr),[p])
    }
    hasGP(p) {
      find Registered(Dr(d),Patient(p)) {
        responses ← gp(p,d)
      } else responses ← nogp(p)
    }
  }
}
```

bookings: The `bookings` component provides examples of component rules. the rule named `next` has a pattern that matches a `Dr`-term that contains a non-empty queue of waiting patients `p:ps`. The use of **`not`**`(...)` in the rule `next` requires that there is no current consultation for the GP named `d`. The body of `next` creates a new `Consultation`-term, sends a message to record the consultation, and removes the patient from the GP's waiting list.

The rules `consult` and `complete` deal with processing the consultation. The system requires a message `next` to occur in order to start any pending consultations. The rule `consult` then matches any ongoing consultations that have not reached their time limit, and increases the consultation time by 1. The `complete` rule fires when the consultation is over:

```
component bookings {
 devcost = 1000
 hostcost = 100
 host = 'surgery'
 operations {
  addGP(d) {
```

```
  new Dr(d,[])
 }
 requestConsultation(p) {
  if patientRequests.isRegistered(p)
  then doctorRequests ← hasGP(p)
  else results ← refusal(p)
 }
 gp(p,d) {
  find Dr(d,bs) {
    replace Dr(d,bs) with Dr(d,bs+[Patient(p)])
  } else new Dr(d,[Patient(p)])
 }
 nogp(p) {
  find Dr(d,bs) when
       not(exists Dr(d',bs') {#bs>#bs'}) {
   replace Dr(d,bs) with Dr(d,bs+[Patient(p)])
  } else error('cannot allocate gp to ' + p)
 }
 next() {
  new Next()
 }
}
rules {
 next:
   Dr(d,p:ps)
   not(Consultation(Dr(d,_),Patient(_),_)) {
  new Consultation(Dr(d,ps),p,5);
  guiCommands ← recordConsultation(p,d);
  replace Dr(d,p:ps) with Dr(d,ps)
 }
 consult: Next
        Consultation(Dr(d,b),Patient(p),n)
        ?(n>0) {
  delete Next;
  replace Consultation(Dr(d,b),Patient(p),n)
  with Consultation(Dr(d,b),Patient(p),n-1);
  requests ← next()
 }
 complete:
  c=Consultation(Dr(d,b),Patient(p),0) {
   delete c;
   doctorRequests ← recordConsultation(d,p);
   requests ← next()
 }
}
}
```

4.4 State

LEAP is an architecture simulation language. A simulation may be generated interactively, as described in the next section, or programmatically. Component state can be initialised directly by listing a collection of terms, or indirectly by sending components some initial messages to start the simulation. Messages may be synchronous or asynchronous. The ← operator sends a message asynchronously in which case there will often be an issue regarding the relative ordering of groups of asynchronous messages. LEAP provides the **do** construct to place an ordering on message groups: **do** { ms } **then** c sends the messages ms concurrently, but waits for all the messages to be completed before proceeding to command c. Therefore, in the following example, all the GPs are added before the patients are registered and subsequently consultation requests made:

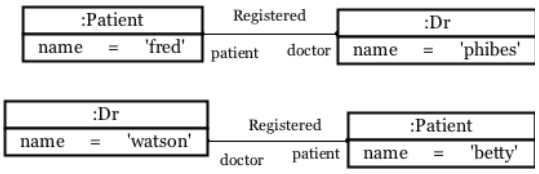

Figure 5: Registered Patients

```
do {
 bookings.requests ← addGP('phibes')
 doctors.requests ← addGP('phibes')
 bookings.requests ← addGP('who')
 doctors.requests ← addGP('who')
 bookings.requests ← addGP('watson')
 doctors.requests ← addGP('watson')
} then do {
 patients.commands ← register('fred')
 patients.commands ← register('wilma')
 patients.commands ← register('barney')
 patients.commands ← register('betty')
 patients.commands ← register('pebbles')
 patients.commands ← register('bam bam')
 doctors.requests ← allocatePatient('phibes','fred')
 doctors.requests ← allocatePatient('watson','betty')
} then do {
 bookings.requests ← requestConsultation('fred')
 bookings.requests ← requestConsultation('wilma')
 bookings.requests ← requestConsultation('betty')
 bookings.requests ← requestConsultation('barney')
 bookings.requests ← requestConsultation('pebbles')
 bookings.requests ← requestConsultation('bam bam')
} then bookings.requests <- next()
```

The state of LEAP components can be constructed or visualised via a tree-browser and using object diagrams. Terms are instances of classes and associations such as those defined in Fig. 2. Terms that are instances of classes are drawn as objects with slots; terms that are instances of associations are drawn as links. The object diagrams are a useful way of vizualising the structure of a component's state.

Figure 5 shows an example object diagram that visualises the state of the doctors component after the allocatePatient messages have

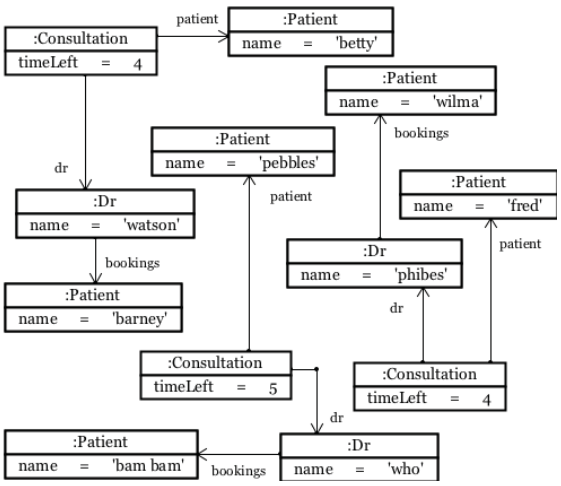

Figure 6: Bookings

been processed. Figure 6 shows an object diagram that visualises the state of the `bookings` component after all the consultation requests have been processed and the simulation has been started via the booking-rules. notice that field references in Fig. 6 are shows as directed links whereas association instances in Fig. 5 are shown as undirected links.

4.5 GUI

LEAP provides a language for constructing simple graphical user interfaces in order to interact with a simulation. The language uses a term representation for a display defined as follows:

```
d ::=
  Table([[d,...],...]) // tables of elements.
| Text(s)              // text.
| Input(s,s)          // a text input field.
| Button(s,f)         // a button with a label.
```

where `s` is a string and `f` is a closure. Each input field is named. When a button is pressed, it is supplied with a table that contains all input names with their associated values.

Figure 7 shows the LEAP tool running the case study. The panel labelled `LEAP` shows a browser view on all components loaded into the tool. The panel labelled `leapsrc` shows a view onto the file system containing leap source code, the file `surgery.cmp` has been selected and its source code is shown in the middle panel on the right. The upper right panel shows the case study user interface. The rest of this section describes how the user interface is defined as a LEAP component.

The surgery `gui` component makes use of a general LEAP feature whereby a user-defined Java class can be dynamically loaded into the system and participate as a LEAP component. The LEAP operator `jcomponent` loads a Java class. The class must implement a predefined LEAP interface that allows it to participate in message passing. The `GUI` class provides an input port `in` that can be send a `display` message. The outline of `gui` is as follows:

```
component gui {
  display = jcomponent('frames.GUI')
  show() {
    display.in ←
      display(Table([[manage()],...]))
  }
  // definitions of ports and operations...
}
```

The definition of the `manage` operation shows how user input is handled for adding a new GP:

```
manage() {
  // Set up a table for managing the surgery
  Table([[Text('Manage')],
         [Table([manageDoctors(),
             managePatients(),
             manageBookings()])]])
}

manageDoctors() {
 // Return a table row.
 // Include dummy text for padding...
 [Text('Name'),
  Input('name',''),
  Text(''),
  Text(''),
  // A button function has a single argument
  // a table containing all input fields...
  Button('New GP',fun(e) addGP(e.name))]
}

addGP(name) {
  // The GUI keeps track of a GP's state...
  new GP(name,Free);
  // Send messages to the other components...
  doctorCommands ← addGP(name);
  bookingCommands ← addGP(name);
  // Update the display...
  show()
}
```

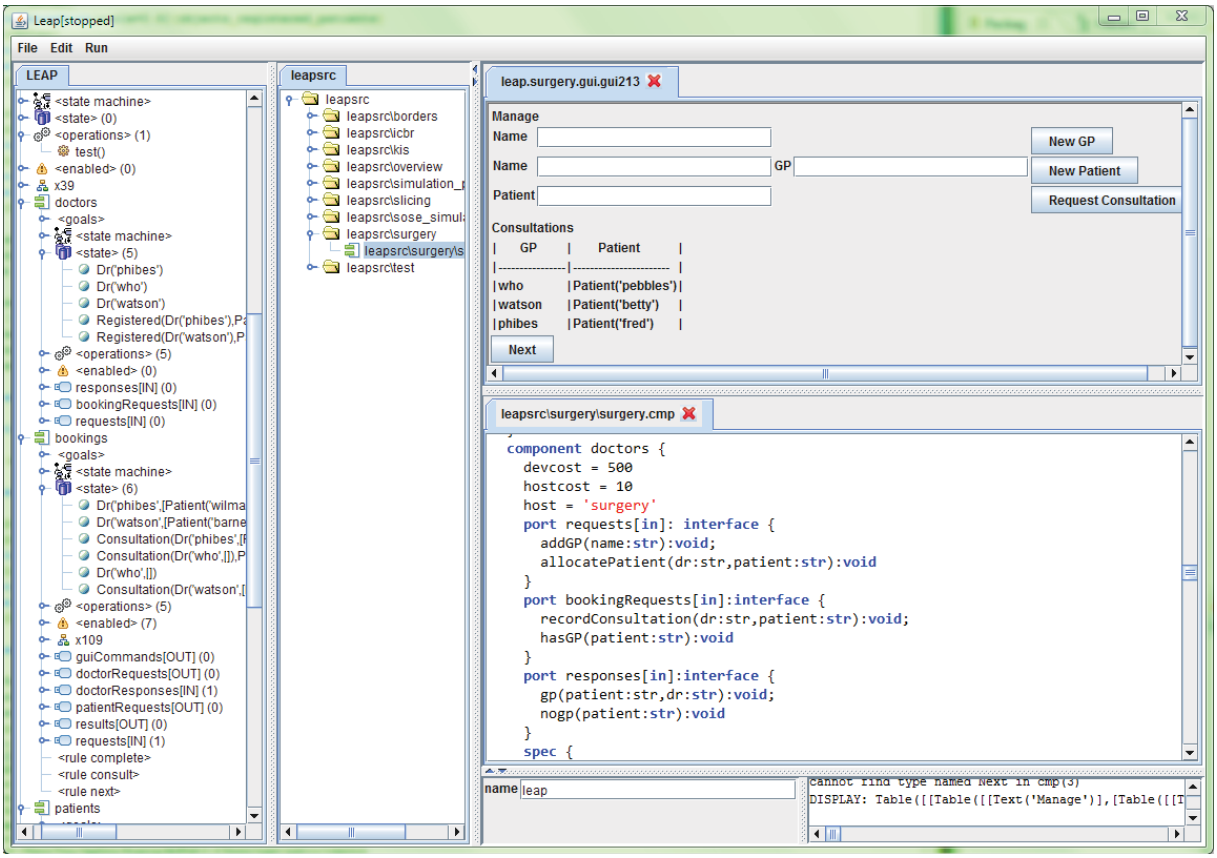

Figure 7: LEAP Showing Case Study Simulation

5 Semantics for Goal Models

We have shown how goal modelling can lead to an architecture model using LEAP. The goals are identified as informal, behavioural and non-functional. Our approach is based on other goal-driven approaches, most notably KAOS. However, whilst KAOS proposes LTL as a language for expressing behavioural goals, it does not provide a link to a language with operational semantics and it does not define how non-functional goals should be precisely expressed. our proposition is that non-functional goals are predicates over meta-properties of system models.

This section describes how LEAP goal models can be given a precise semantics in terms of behavioural and non-functional goals. We define the LEAP value domain and outline the LEAP operational cycle in section 5.1. Section 5.2 describes how *reification* is performed whereby LEAP values at the Java-level are translated automatically into LEAP data at the user-level in order to support meta-constraints. Finally, behavioural goals are written in a LTL that is defined in section 5.3 in terms of the value domain.

5.1 Values

Figure 8 shows the value model for LEAP. Developers can use the LEAP tool to produce diagrams of type: goal; model; component; state; state machine, or can develop their models using the LEAP textual language and then transform the models into diagrams for visualisation.

The execution semantics for LEAP continually removes messages on the queue of component input ports. The messages are matched against

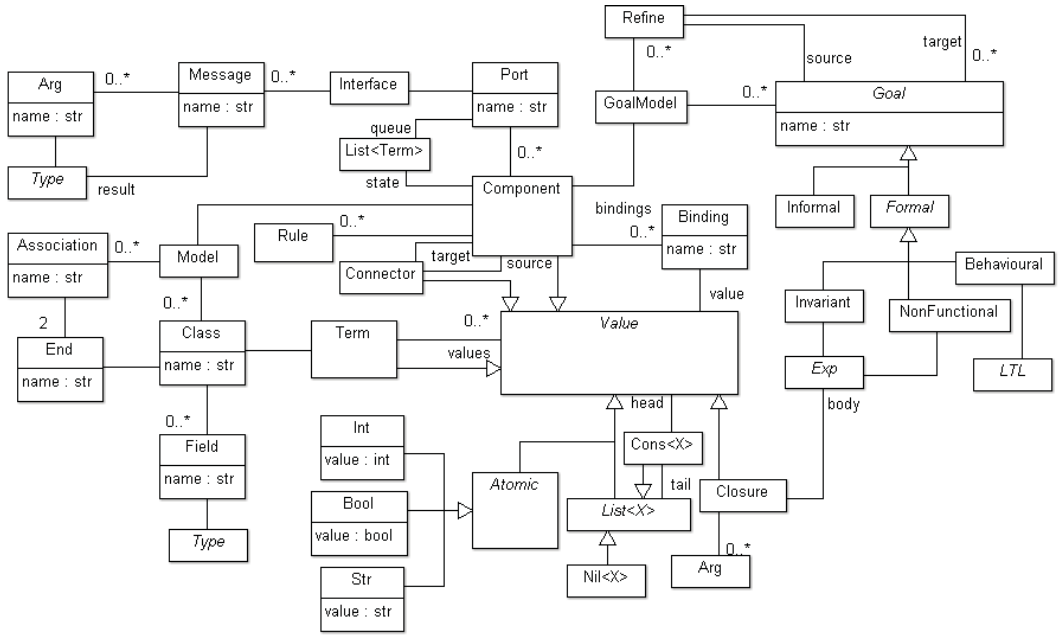

Figure 8: LEAP Values

closures bound by the component and handled by calling the closure with the supplied arguments in the message. The body of the closure is performed and may send messages to the output ports of the component. Messages sent to an output port are transferred to the message queue of any connected input ports. Rules continually monitor the state in a component and when the rule condition matches, the rule body fires.

5.2 Reification

Non-functional goals are predicates over meta-data. LEAP provides access to meta-data using two operators: `reify` and `intern` that are inverses of each other. The `reify` operator maps any LEAP data value to a LEAP term and the `intern` operator maps a suitably encoded term into a LEAP value. A simple example is:

```
reify(10) → Int('values',10)
intern(Int('values',10)) → 10
```

In general, reification of a LEAP value produces a term whose type name is the name of the underlying Java class. The first data element in the term is the name of the Java package containing the class followed by the values of the Java fields in a predefined order. The implementation of `reify` and `intern` relies on underlying Java reflection as described in the rest of this section.

In order for a Java class to participate in the reification process it must provide a Java annotation of type `Descriptor`. A descriptor names the fields that will be included as term-data and names the Java methods to be used as accessors and updaters for the fields. Crucially, the descriptor places an order on the fields:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Descriptor {
  String[] accessors();
  String[] updaters();
  String[] fields();
}
```

All LEAP value classes provide descriptors. The following shows part of the `Association` class and its descriptor annotation:

```
@Descriptor(fields={"name","ends"},
            accessors={"getName","getEnds"},
```

```
            updaters={"setName","setEnds"})
class Association {
  String name;
  End[] ends;
  public Association() {}
  public String getName() { return name; }
  public void setName(String n) { name = n; }
  ...
}
```

Mapping between different data representations relies on three meta-operations: instantiation; `getContents`; `setContents`.

Instantiation is directly supported by Java through the `newInstance` method of a class and is used by LEAP in a standard way by providing a 0-arity constructor for each value-class. Accessing the contents of a Java object is achieved through the accessors listed in the descriptor:

```
Object[] getContents() {
  Class<?> c = getClass();
  String[] names = getAccessors(c);
  Object[] contents = new Object[names.length];
  for (int i = 0; i < names.length; i++) {
    String name = names[i];
    Method accessor = c.getMethod(name);
    contents[i] = accessor.invoke(this);
  }
  return contents;
}
```

The `getAccessors` method above transitively retrieves the descriptor annotations of the receiver's class and its super-classes and returns the names of the accessors. Updating a Java object is achieved through `setContents`:

```
void setContents(Object[] os) {
  Class<?> c = getClass();
  String[] U = getUpdaters(c);
  for (int i = 0; i < U.length; i++) {
    String u = updaterNames[i];
    Method um = c.getMethod(a);
    Object o = coerceValue(os[i], argType(um));
    um.invoke(this, o);
  }
}
```

The method `coerceValue` is used to ensure that LEAP values are mapped to Java values, for example LEAP lists are mapped to Java arrays or vectors depending on the type of the field.

Having defined the meta-access machinery we can now define the reification operations. the `reify` method maps a Java object `o` to a LEAP value. Note that the following code has been simplified by omitting the machinery that deals with cyclic data. The `reify` method has three categories of mapping: objects whose class has a descriptor; atomic values; collections. As shown below, the atomic cases are dealt with by directly translating to instances of `Int`, `Bool` and `Str`.

```
Value reify(Object o) {
  Class<?> c = o.getClass();
  else if(c.getAnnotation(Descriptor.class)!=null)
    return reifyDescriptor(o);
  else if(o instanceof Integer)
    return new Int((Integer)o);
  ... // more atomic cases...
  else if(c.isArray())
    return reifyArray((Object[])o);
  ... // also deal with vector...
  else error(...)
}
```

An object whose class has a descriptor is mapped using `reifyDescriptor` defined below. The value is encoded as a LEAP term and the fields, as defined by the descriptor, are recursively reified:

```
Value reifyDescriptor(Object o) {
  String type = o.getClass().getName();
  int dot = type.lastIndexOf('.');
  String packageName = type.substring(0,dot);
  String typeName = type.substring(dot+1);
  Object[] contents = value.getContents();
  Value[] subTs=new Value[contents.length+1];
  subTs[0] = new Str(packageName);
  for (int i = 1; i < contents.length; i++)
    subTs[i + 1] = reify(contents[i]);
  return new Term(TERMCLASS, typeName, subTs);
}
```

An array is reified to become a LEAP list as follows (vectors are treated in the same way):

```
List<Value> liftArray(Object[] objects) {
  List<Value> list = new Nil<Value>();
  for (int i = objects.length - 1; i >= 0; i--)
    list = list.cons(reify(objects[i]));
  return list;
}
```

The `intern` method performs the inverse mapping. Like `reify` is has three categories of translation: terms, atoms and lists:

```
Object intern(Value value, Class<?> type) {
  if (value instanceof Term)
    return internTerm((Term) value, type);
  else if (value instanceof Int)
    return ((Int) value).getValue();
  ... // more atomic cases
  else if (value instanceof List<?>)
    return internList((List<Value>)value,type);
  else eror(...)
```

```
p ::= behavioural constraint
  always { p }
| eventually { p }
| next { p }
| before(n) { p }
| forall t { p }
| p implies p
| not(p)
| p and p
| t

t ::= term patterns
  N(t*)
| k
| t:t
| v
```

Figure 9: Behavioural Constraints

Translation of terms is shown below. The name of the Java class to be instantiated is encoded as a class name and package name in the term. These are extracted and a new Java instance is created. The other data elements in the term are extracted and recursively translated before setting the values in the new Java object using `setContents`:

```
Value internTerm(Term term, Class<?> type) {
  String typeName = term.getName();
  Object[] contents = term.getContents();
  Str str = (Str) contents[0];
  String pname = str.getValue()+"."+typeName;
  Class<?> c = getClass().forName(pname);
  Value v = (Value) c.newInstance();
  Object[] vs = new Object[contents.length-1];
  Class<?>[] ts = v.getTypes();
  for (int i = 0; i < types.length; i++)
   vs[i]=intern((Value)contents[i+1],ts[i]);
  v.setContents(vs);
  return v;
}
```

5.3 Behavioural Goals

Behavioural goals are written in a formal language that is based on *Linear Temporal Logic* (LTL). The syntax of LEAP LTL is shown in Fig. 9 where `N` denotes a term name, `k` is an atomic constant, and `v` is a variable. The semantics of a behavioural goal are defined with respect to system executions. A system execution is a sequence of states where a state is a set of terms as defined by Fig. 8. Although the state of a LEAP model involves multiple components, ports, messages and terms, it is possible to simplify this by equating messages (which are just terms anyway) with states and by flattening the structure of the components (renaming consistently where necessary).

A LEAP system state is a set of terms s. A system execution is a sequence of states $[s_1, \ldots, s_n]$. At any given time the system is in a particular state i and has a history $[s_1, \ldots, s_{i-1}]$ and a future $[s_{i+1}, \ldots, s_n]$.

A term pattern t contains variables. Variables are bound to LEAP values in an environment θ. An environment is applied to a term pattern $\theta(t)$ to produce a term by substituting the variables in the pattern for values. A pattern t occurs in a state s when there is a substitution θ such that $\theta(t) \in s$. Note that a pattern may occur more than once in a state if there is more than one substitution.

A LTL formula p holds at a given point i in a system execution $[s_1, \ldots, s_n]$ when the following relationship holds: $[s_1, \ldots, s_n], i \models p$. The relationship is defined in Fig. 10. The definitions are as follows: (1) defines that P always holds when it holds for all future states; (2) defines that p eventually becomes true when there is a future state for which is true; (3) defines when p is true in the next state; (4) states that **before**$(n)\{p\}$ holds when p is true in the past by skipping back n states into the history; (5) requires that p must hold for all possible elements in the current state that matches t; (6) states that if p holds in the current state then q must also hold; (7) defined that if **not**(p) holds then p must not hold; (8) states that for p **and** q to hold then p and q must both holds in the current state; (9) allows the variables in a term pattern to be existentially qualified in the current state.

6 Related Work

This section provides an overview of related work in the provision of modelling languages and frameworks that have attempted to address the early stages of requirements engineering.

(1)	$[s_1, \ldots, s_n], i \models \textbf{always}\ \{\ p\ \}$	*when* $[s_1, \ldots, s_n], j \models p$ holds $\forall j \geq i$
(2)	$[s_1, \ldots, s_n], i \models \textbf{eventually}\ \{\ p\ \}$	*when* $[s_1, \ldots, s_n], j \models p$ holds $\exists j \geq i$
(3)	$[s_1, \ldots, s_n], i \models \textbf{next}\ \{\ p\ \}$	*when* $[s_1, \ldots, s_n], i + 1 \models p$ holds
(4)	$[s_1, \ldots, s_n], i \models \textbf{before}(n)\ \{\ p\ \}$	*when* $[s_1, \ldots, s_n], i - n \models p$ holds
(5)	$[s_1, \ldots, s_n], i \models \textbf{forall}\ t\ \{\ p\ \}$	*when* $\forall \theta.\theta(t) \in s_i \implies [s_1, \ldots, s_n], i \models \theta(p)$
(6)	$[s_1, \ldots, s_n], i \models p\ \textbf{implies}\ q$	*whenever* $[s_1, \ldots, s_n], i \models p$ *then* $[s_1, \ldots, s_n], i \models q$
(7)	$[s_1, \ldots, s_n], i \models \textbf{not}(p)$	*when* $[s_1, \ldots, s_n], i \not\models p$
(8)	$[s_1, \ldots, s_n], i \models p\ \textbf{and}\ q$	*when* $[s_1, \ldots, s_n], i \models p$ *and* $[s_1, \ldots, s_n], i \models q$
(9)	$[s_1, \ldots, s_n], i \models t$	*when* $\exists \theta.\theta(t) \in s_i$

Figure 10: LTL Semantics

Requirements modelling is an intrinsic and important element of the processes by which system architectures are designed, implemented and managed. However, architecture modelling approaches and techniques, such as design-by-contract, have until recently focused on what a system should do and how it can be achieved. Scant attention has been paid to the "why", the motivations in terms of goals, requirements, rationales. In (Wagter et al. 2012) Wagter et al make an interesting distinction between 'blueprint' styles exemplified by the engineering based approaches such as Zachman (Zachman 1999), TOGAF (Spencer et al. 2004) and Archimate (Lankhorst 2009) and argue that such a blueprint style does not suffice as interests such as stakeholders, informal power structures and other hard-to-quantify factors cannot be easily represented in such an engineering style. They propose instead, that a yellow-print style (as noted by De Caluwe and Vermaak 2003) is necessary.

Efforts to standardise on the motivational or intentional aspects of enterprise architecture have been consolidated in the OMG Business Motivational Model (BMM) (Group et al. 2005) which provides a structure for representing concepts for developing, communicating and managing business plans such that they can be used to model those factors that motivate a busines plan, the elements making up the business plan and the relationship between these factors and elements. The BMM provides a focus for *motivation* such that activities delivering a business plan are defined by *why* specific activities are performed. Concepts in the BMM include:

Ends: the aspirations of the enterprise expressed as Vision, Goals and Objectives.

Means: the mechanism by which Ends are realised and expressed as Mission in terms of Strategy and Tactics; and Directives such as business policy and business rules.

Influencers: how ends and means can influence each other in either positive or negative ways.

The BMM provides a meta model (syntax only) expressed as a UML model that shows these concepts, their subtyping and their relationships. The model recognises that the complexity of the relationship between BMM model elements and process modelling is not fully developed and proposes that a related standard the Business Process Modelling Notation (BPMN) (BPMN 2.0. Notation 2009) should be used as an additional technology. This raises questions about model integration, consistency and shared semantics issues.

The foundational work for BMM can be traced back to goal oriented requirements engineering (GORE) techniques (Mylopoulos et al. 1999) such as i* (Yu 1997; Yu and Mylopoulos 1994) and KAOS (Dardenne et al. 1993; Letier and Van Lamsweerde 2004; Van Lamsweerde 2008). GORE bases techniques present a variety of options for analysis such as providing a more formal basis of how goals realise other goals, conflict between goals and the positive and negative contributions goals make to other goals. Further, the relationship between the proposed solution and actual

need is more clearly delineated. So organisations can more easily and in a systemised manner, articulate choices between alternative configurations of architectures or explore new possible configurations (Halleux et al. 2009; Jonkers et al. 2004). GORE techniques also have potential in that they can be readily combined with viewpoints oriented requirements engineering approaches such as (Finkelstein et al. 1991; Sommerville and Sawyer 1997).

The Reference Model for Open Distributed Processing (RM-ODP) is a comprehensive framework for open systems specification. It is an ISO/ITU Standard (ITU, 1996) that defines a framework for architecture specification of large distributed systems using viewpoints on a system and its environment: enterprise, information, computation, engineering and technology. The theoretical basis of the RM-ODP model resides in object oriented principles and service oriented specification and the mapping of the levels to implementation objects (Raymond 1995). RM-ODP addresses the motivation or intention aspects by the "enterprise viewpoint" and the inclusion of a concept of "Objective" in its enterprise language. The concept is then related to key elements of the enterprise viewpoint including "community" and the objects belonging to the community such as policy, role and process definition. As Almeida et al point out: "In a nutshell, communities are defined to achieve certain objectives. These objectives influence the definition of the policies and roles in the community, which affect the behaviour of the enterprise objects to favour the satisfaction of community objectives" (Almeida et al. 2010). In the same paper, RM-ODP concepts are interpreted using the Unified Foundation Ontology (Guizzardi et al. 2008) to provide a basis for communication and consensus particularly to address the social behaviour dimension of how the use and change of enterprise objects can strive towards motivation and intentions behind business goals.

The i* framework provides a set of concepts for modelling and analysis addressing the early requirements phase, namely, that focusing on the "why" of underlying systems requirements (Yu and Mylopoulos 1994). Key concepts in the framework are centred around the notion of the *intentional actor* that possesses intentional properties such as a belief or ability. Actors collaborate and depend upon each other and collectively perform tasks and in the process of doing so, consume resources. Actors will acknowledge and adapt so that opportunities and threats are addressed in line with the intentional beliefs. Actors are thus part of an agent-oriented system. The i* framework has a provision for two types of models. Firstly, a Strategic Dependency model that is effectively an Actor diagram that is used to model agreements between actors to fulfil a goal, perform a task or to use a resource. Types of goal - hard and soft (for which there is no clear criteria to be fulfilled) can be represented. The Strategic Rationale model is a means of expressing stakeholder interests and concerns and their relationship to various configurations of an enterprise architecture. The model is effectively a drill down of the SD model and describes the actors' goals and rationales in order to justify the actors' relationships and their adoption of particular plans. Related and directly derived from i* is the Tropos methodology (Bresciani et al. 2004; Susi et al. 2005) which adopts the same concepts for the early requirements stage. The ARIS methodology (Scheer 2000) is an approach that is widely used in industry for business process modelling and also includes a high level set of goal-related concepts that include such elements as Objective, Participant, Critical Factor and Function as a means of modelling intentions. Objectives are used to represent a notion of a goal and Functions can be seen as operations applied to objectives for the purpose of supporting goals. Relationships between goals are also supported. The Critical Factor concept represents aspects which need to be considered in the meeting of a particular objective. The limitations of ARIS with respect to the richness of the modelling language for goals and the lack of process modelling capabilities within Tropos and it's parent i* has

been identified as a requirement for some form of integration between the two approaches and Cardosa et al propose a semantics based integration between goals and process modelling as one such approach for provisioning that requirements (Cardoso et al. 2010).

One major strand of research in goal modelling is KAOS methodology (Dardenne et al. 1993) and subsequently elaborated further by Letier, Van Lamsweerde and others (Letier and Van Lamsweerde 2004; Van Lamsweerde 2000, 2008; Van Lamsweerde et al. 1998). Consistent with the technologies described previously, KAOS is also an agent oriented methodology for requirements engineering. The key concept is a goal as a "a prescriptive statement of intent that the system should satisfy through cooperation of its agents". Goals are defined at varying levels of abstraction through refinement relationships. Goals are specifically satisfied by a system component and are termed a *requirement*, however there is support for a partial satisfaction via an *expection* relationship. These are not enforceable via automated processing. KAOS, like i* and others supports the notion of conflict modelling by *obstruction* and *resolution* by other goals. Perhaps different from others, KAOS allows the modelling of domain hypotheses represented by domain invariants - properties (and values) that are always hold.

Quartel et al (Quartel et al. 2009) provide a useful overview of some of the technologies described here and also make the observations: that BMM cannot be considered a true requirements modelling language; i* while providing a rich expressive language presents on overhead in learning and using the language; KAOS lacks some of the richness of expressivity but counters that shortcoming in its simplicity. In considering these observations they propose a language called ARMOR which provides a goal/intentional modelling capability to the Archimate language and is thus similar to our proposal outlined in this paper. We argue that ARMOR provides both an abstract and concrete syntax but relies on the limited semantics provided by Archimate. In contrast our integrated offering of goal modelling support within the LEAP language provides intentional modelling with semantics. The language offered here as part of LEAP has a similar expressive power in that most of the pertinent aspects of i* are available. We have also tried to optimise the usability available in KAOS and provide the more advanced facilities of a simulation environment. The semantics offered by the use of LTL further enhances the capability of the language. As Cardosa et al have pointed out, semantics based integration between goals and process modelling (the integration of the why and how) are a necessary step. LEAP and its goal modelling element provides that capability.

7 Conclusion

The motivation or goal behind why a particular requirement manifests itself as system function and the traceability of the relationship that explores the *why* remains an area of relative neglect in the system and enterprise architecture domains. Technologies originating as characterisations of goal oriented requirements engineering such as KAOS and i* could perhaps have had limited impact on architecture modelling because of issues such as: complexity leading to usability, lack of semantics and a traceable rigour from goals through to system functions. In particular, non-functional requirements present specific difficulties. In this paper we have proposed a technology LEAP that attempts to address some of these issues. This contribution exists at several levels. Firstly we have provided a formal model for goal modelling that is supported by a technology that allows goals to be checked during execution. Secondly, the executability of requirements presents an opportunity to provide semantics for goal model executability by the use of LTL. Finally by provisioning a tight meta model based integration between concepts that reside at the early stages of requirements analysis with those that are focussed on the engineering aspects of architecture we provide a route from goal models to architecture models that is supported by a prototype modelling and execution environment.

The LEAP approach is based on the hypothesis that architecture design and analysis should use a small and orthogonal set of concepts based around higher-order components. A component-based language can be used to represent both logical and physical elements of a system. We have demonstrated success with this approach in terms of intentional modelling, goal and IT alignment, architecture simulation, representation of both event-driven and service-oriented approaches to architecture, refinement, and architecture refactoring.

However, there are limitations with the current LEAP technology that we have not yet addressed. It cannot represent low-level architectural designs such as those required by embedded and real-time systems. We acknowledge that the current support in LEAP for specifying behaviour in terms of invariants, pre and post-conditions and LTL expressions is insufficient to express complex component interactions involving time and/or concurrency.

LEAP has been implemented and therefore has an operational semantics given by its implementation, our next steps will include an abstract semantic description of LEAP that integrates with the LTL semantics given in this article that will allow us to study issues such as the complexity of execution and therefore the limitations on the sise of LEAP models. The usability aspects of LEAP tooling, including debugging complex configurations of higher-order components, has yet to be addressed. Part of the strength of LEAP is that it is based on a relatively small number of orthogonal concepts and provides a richly expressive language through higher-order features. However, in order to be usable it is necessary for developers to be able to make distinctions between different categories of elements, for example goals that apply to the system and those that apply to the actors that use the system. A form of domain-specific syntactic sugar may be a suitable way to support these distinctions.

Another area that we feel will be fruitful, is using LEAP as a migration tool from an as-is architecture to a to-be architecture by using the Java interfaces of LEAP to simulate the to-be architecture in terms of the as-is architecture and thereby providing a basis for incrementally replacing the LEAP simulation with new components.

Our proposal for goal modelling has been evaluated with an experiment using a restrictive but representative case study example. We note the limitations of such experiment and as a consequence further research will continue to validate our approach as we attempt to use our tools to evaluate new case studies from both existing literature and from industrial practitioners. We are now engaged in a relationship with leading technology research lab from the commercial sector who will be using LEAP as part of their research activity. We expect this relationship to provide new evaluatory data to support the proposal presented in this article.

References

Almeida J., Cardoso E., Guizzardi G. (2010) On the Goal Domain in the RM-ODP Enterprise Language: An Initial Appraisal based on a Foundational Ontology. In: Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2010 14th IEEE International. IEEE, pp. 382–390

Bresciani P., Perini A., Giorgini P., Giunchiglia F., Mylopoulos J. (2004) Tropos: An agent-oriented software development methodology. In: Autonomous Agents and Multi-Agent Systems 8(3), pp. 203–236

Cardoso E., Santos Jr P., Almeida J., Guizzardi R., Guizzardi G. (2010) Semantic Integration of Goal and Business Process Modeling. In: IFIP International Conference on Research and Practical Issues of Enterprise Information Systems (CONFENIS 2010), Natal-RN, Brazil

Chan Y., Reich B. (2007) IT alignment: what have we learned? In: Journal of Information Technology 22(4), pp. 297–315

Clark T., Barn B. S. (2011) Event driven architecture modelling and simulation. In: Gao J. Z., Lu X., Younas M., Zhu H. (eds.) SOSE. IEEE, pp. 43–54

Clark T., Barn B. S. (2012) A common basis for modelling service-oriented and event-driven architecture. In: Aggarwal S. K., Prabhakar T. V., Varma V., Padmanabhuni S. (eds.) ISEC. ACM, pp. 23–32

Clark T., Barn B. S., Oussena S. (2011) LEAP: a precise lightweight framework for enterprise architecture. In: Bahulkar A., Kesavasamy K., Prabhakar T. V., Shroff G. (eds.) ISEC. ACM, pp. 85–94

Dardenne A., Van Lamsweerde A., Fickas S. (1993) Goal-directed requirements acquisition. In: Science of computer programming 20(1-2), pp. 3–50

De Caluwe L., Vermaak H. (2003) Learning to change: A guide for organization change agents. Sage Publications, Inc

Finkelstein A., Goedicke M., Kramer J., Niskier C. (1991) Viewpoint oriented software development: Methods and viewpoints in requirements engineering. In: Algebraic Methods II: Theory, Tools and Applications, pp. 29–54

Group B. et al. (2005) The business motivation model-business governance in a volatile world. Technical report

Guizzardi G., Falbo R., Guizzardi R. (2008) Grounding software domain ontologies in the unified foundational ontology (ufo): The case of the ode software process ontology. In: 1th Iberoamerican Workshop on Requirements Engineering and Software Environments (IDEAS 2008)

Halleux P., Mathieu L., Andersson B. (2009) A method to support the alignment of business models and goal models. In: Proceedings of BUSITAL 8, p. 121

Jonkers H., Lankhorst M., Van Buuren R., Hoppenbrouwers S., Bonsangue M., Van Der Torre L. (2004) Concepts for modeling enterprise architectures. In: International Journal of Cooperative Information Systems 13(3), pp. 257–287

Lankhorst M. M, H.A. Proper H., Jonkers J. (2010) The Anatomy of the ArchiMate Language. In: International Journal of Information System Modeling and Design 1(1), pp. 397–409

Lankhorst M. (2009) Introduction to Enterprise Architecture. In: Enterprise Architecture at Work. The Enterprise Engineering Series. Springer Berlin Heidelberg http://dx.doi.org/10.1007/978-3-642-01310-2_1

Letier E., Van Lamsweerde A. (2004) Reasoning about partial goal satisfaction for requirements and design engineering. In: ACM SIGSOFT Software Engineering Notes. 6 Vol. 29. ACM, pp. 53–62

McLean E., Soden J. (1977) Strategic planning for MIS. John Wiley & Sons Inc

Mylopoulos J., Chung L., Yu E. (1999) From object-oriented to goal-oriented requirements analysis. In: Communications of the ACM 42(1), pp. 31–37

OMG BPMN 2.0. Notation 2009. In: Object Management Group, p. 496

Quartel D., Engelsman W., Jonkers H., Van Sinderen M. (2009) A goal-oriented requirements modelling language for enterprise architecture. In: Enterprise Distributed Object Computing Conference, 2009. EDOC'09. IEEE International. Ieee, pp. 3–13

Raymond K. (1995) Reference model of open distributed processing (RM-ODP): Introduction. In: IFIP TC6 International Conference on Open Distributed Processing. Brisbane, Australia, Chapman and Hall, pp. 3–14

Scheer A. (2000) ARIS–business process modeling. Springer Verlag

Sommerville I., Sawyer P. (1997) Viewpoints: principles, problems and a practical approach to requirements engineering. In: Annals of Software Engineering 3(1), pp. 101–130

Spencer J. et al. (2004) TOGAF Enterprise Edition Version 8.1

Susi A., Perini A., Mylopoulos J., Giorgini P. (2005) The tropos metamodel and its use. In: INFORMATICA-LJUBLJANA- 29(4), p. 401

Van Lamsweerde A. (2000) Requirements engineering in the year 00: A research perspective.

In: Proceedings of the 22nd international conference on Software engineering. Acm, pp. 5–19

Van Lamsweerde A. (2008) Requirements engineering: from craft to discipline. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM, pp. 238–249

Van Lamsweerde A., Darimont R., Letier E. (1998) Managing conflicts in goal-driven requirements engineering. In: Software Engineering, IEEE Transactions on 24(11), pp. 908–926

Wagter R., Proper H. A. E., Witte D. (2012) A Practice-Based Framework for Enterprise Coherence. In: Practice-Driven Research on Enterprise Transformation. Lecture Notes in Business Information Processing Vol. 120, pp. 77–95

Yu E. (1997) Towards modelling and reasoning support for early-phase requirements engineering. In: Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on. IEEE, pp. 226–235

Yu E., Mylopoulos J. (1994) Understanding ẃhyín software process modelling, analysis, and design. In: Proceedings of the 16th international conference on Software engineering. IEEE Computer Society Press, pp. 159–168

Zachman J. (1999) A framework for information systems architecture. In: IBM systems journal 38(2/3)

Tony Clark, Balbir Barn

Middlesex University
The Burroughs
London
NW4 4BT
United Kingdom
{t.n.clark | b.barn}@mdx.ac.uk