Event Driven Architecture Modelling and Simulation

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

***Abstract*—Enterprise Architecture (EA) Modelling aims to analyze an organization in terms of its components, IT systems and business processes. Current modelling approaches are based on Service Oriented Architecture (SOA) whereby components publish interfaces of operations that are used via message passing. It has been argued that SOA leads to tight coupling between components and does not handle complex component interactions, with resulting maintenance difficulties. Event Driven Architecture (EDA) is an alternative strategy, based on listening for events, that is designed to address SOA shortcomings. However, there are no EA modelling technologies based on EDA. This paper reviews EA, SOA and EDA, identifies EDA characteristic features and proposes modelling and simulation technologies that are introduced through a simple case study.**

I. Introduction

Enterprise Architecture (EA) describes a collection of approaches that support the design and analysis of an IT infrastructure and how it relates to the goals, directives, processes and organization of a business. There are various approaches to organizing an architecture, but most involve the identification of logical or physical business units, or *components*, that manage their own data and resources, implement a collection of business processes, and communicate with other components using a variety of message passing styles.

Different styles of message passing lead to different types of architecture. A *Service Oriented Architecture* (SOA) involves the publication of logically coherent groups of business functionality as *interfaces*, that can be used by components using synchronous or asynchronous messaging. An alternative style, argued as reducing coupling between components and thereby increasing the scope for component reuse, is Event Driven Architecture (EDA) whereby components are event generators and consumers. EDA is arguably more realistic in a sophisticated, dynamic, modern business environment, and can be viewed as a specialization of SOA where communication between components is performed with respect to a single generic event interface.

An important difference between SOA and EDA is that the latter generally provides scope for *Complex Event Processing* (CEP) where the business processes within a component are triggered by multiple, possibly temporally related, events. In SOA there is no notion of relating the invocation of a single business process to a condition holding between the data passed to a collection of calls on one of the component's interfaces.

Given that EDA and CEP are claimed to be more flexible than SOA, how should we design and verify an EA using these concepts? Some specialized languages for CEP exist, however they are specifically designed to efficiently process relatively simple event streams, and do not integrate into a component architecture required by EA. The most widely used language for modelling systems is UML, however UML does not provide any specific support for this type of EA.

The contribution of this paper is to identify the key characteristic features of EDA and to use them as the basis for both an EDA modelling notation and a simulation language where no such technology currently exists. The modelling notation is defined as a UML class diagram stereotype and an extension of OCL that supports event processing. The simulation notation is defined as an action language for the modelling notation. The simulation language has been implemented as an interpreter in Java.

The paper is structured as follows. Section II reviews complex event processing and EA modelling languages. Section III performs a domain analysis on EDA and describes an EDA modelling language. Section IV describes a case study as an EDA model. Section V describes an EDA simulation language and provides an example by implementing the case study. The simulation language has been implemented in Java and is described in section VI.

II. Event Driven EA

A. Service Oriented Architecture

Service Oriented Architecture (SOA) organizes a system in terms of components that communicate via operations or *services*. Components publish services that they implement as business processes. Interaction amongst components is achieved through *orchestration* at a local level or *choreography* at a global level.

Its proponents argue that SOA provides loose coupling, location transparency and protocol independence [1] when compared to more traditional implementation techniques. The organization of systems into coherent interfaces has been argued [2] as having disadvantages in terms of: extensions; accommodating new business functions; associating single business processes with complex multi-component interactions. These can be addressed in terms of CEP as described in the next section.

B. Complex Event Processing

Complex Event Processing (CEP) [3] can be used to process events that are generated from implementation-level systems by aggregation and transformation in order to discover the business level, actionable information behind all these data. It has evolved into the paradigm of choice for the development of monitoring and reactive applications [4].

CEP can be viewed as a specialization of SOA where components are decoupled from multiple interfaces and where each component implements a single generic event interface. Components both raise and handle events in terms of this interface and therefore it is more flexible in terms of extension and maintenance. In addition, CEP implements events in terms of business rules compared to SOA that implements operations using business processes. Typically, a business rule can depend on multiple, possibly temporally related, events, whereas a business process is invoked on receipt of a single operation request. Therefore, SOA can implement CEP by enforcing a single operation interface across an architecture and by providing special machinery to aggregate multiple operation calls.

There are various proposals for how complex events can be used efficiently to process streams of data such as those generated in applications including hotel booking systems, banking on-line credit systems, business activity monitoring (BAM), real-time stock analysis, and real-time security analysis. Most proposals aim to address efficiency issues related to the scale and frequency of the information that is generated [5]. The current state of the art is described in [6] where the key features of an event driven architecture (EDA) are outlined as including an architecture diagram showing the processes of the system and their interconnections, a behaviour specification including the rules used to process events and to control data, and the specification of inter-process communications.

As described in [7] events can be extracted from services, databases, RFID and activities. The events are processed by rules that detect relationships between event properties and the times at which the events occur. Each rule matches against multiple events that occur from a variety of sources and, when all required events have been matched, the rule performs a business action. In [7] the authors describe the implementation of a complex event processing architecture that involves attaching an extractor to event sources and compiling event processing rules into complex event recognition tables. The language does not address modularity issues and how the complex event architecture maps onto modern approaches to EA. Wu et al [8] describe a language called SASE for processing complex events from RFID devices. The language is based expressing patterns of events over events in time-windows and the authors describe various optimizations that can be performed. The language is general purpose but does not implement negation or offer features for modularity.

The approach described in [9] is based on logic programming for complex event processing and in a way is the opposite to our forward-driven approach. The authors use Prolog-style backtracking to find solutions to goals.

C. Enterprise Architecture

Enterprise Architecture (EA) aims to capture the essentials of a business, its IT and its evolution, and to support analysis of this information: '[it is] a coherent whole of principles, methods, and models that are used in the design and realization of an enterprise's organizational structure, business processes, information systems and infrastructure.' [10].

A key objective of EA is being able to provide a holistic understanding of all aspects of a business, connecting the business drivers and the surrounding business environment, through the business processes, organizational units, roles and responsibilities, to the underlying IT systems that the business relies on. In addition to presenting a coherent explanation of the *what*, *why* and *how* of a business, EA aims to support specific types of business analysis including [11], [12], [13], [14], [15]: *alignment* between business functions and IT systems; *business change* describing the current state of a business (*as-is*) and a desired state of a business (*to-be*); *maintenance* the de-installation and disposal, upgrading, procurement and integration of systems including the prioritization of maintenance needs; *quality* by managing and determining the quality attributes for aspects of the business such as security, performance to ensure a certain level of quality to meet the needs of the business; *acquisition and mergers* describing the alignment of businesses and the effect on both when they merge; *compliance* in terms of a regulatory framework, e.g. Sarbanes-Oxley; *strategic planning* including corporate strategy planning, business process optimization, business continuity planning, IT management.

EA has its origins in Zachman's original EA framework [16] while other leading examples include the Open Group Architecture Framework (TOGAF) [17] and the framework promulgated by the Department of Defense (DoDAF) [18]. In addition to frameworks that describe the nature of models required for EA, modelling languages specifically designed for EA have also emerged. One leading architecture modelling language is ArchiMate [19].

As described in [20] and [21], complex events can be the basis for a style of EA design. Event Driven Architecture (EDA) replaces thick interfaces with events that trigger organizational activities. This creates the flexibility necessary to adapt to changing circumstances and makes it possible to generate new processes by a sequence of events [22]. Whilst a complex event based approach to architectural design must take efficiency concerns into account, the primary concern is how to capture, represent and analyze architectural information as an enterprise design.

EDA and SOA are closely related since events are one way of viewing the communications between system components. The relationship between event driven SOA and EA is described in [23] where a framework is proposed that allows enterprise architects to formulate and analyze research questions including 'how to model and plan EA-evolution to SOA-style in a holistic way' and 'how to model the enterprise on a formal basis so that further research for automation can be done.'

A number of commercial EA analysis and simulation tools are available [24]. Many of these are based around industrial standards such as UML and BPMN. However they are generally very complex and lack a precisely defined semantics. Our work aims to provide a precise, well-defined basis for event driven EA analysis and simulation.

III. Event Driven Modelling

A. Features

Our aim is to provide a modelling and simulation language for EDA that is based on features provided by UML. In order to do this we need to identify characteristic EDA features. This section lists the features and the following section describes how they are to be implemented using UML class model stereotypes and an extension to OCL.

An EDA architecture is based on *components* each of which represents an organizational unit. Components map onto physical IT systems or organizational units. Each component manages local private *data* that maps onto databases (relational, files, etc). As in SOA, a component may offer *operations* that can be invoked by sending *messages*. An operation is specified in terms of *pre* and *post* conditions expressed in terms of the component's data. In addition to modifying local data, an operation may send messages and raise *events*. An event is used by a component to signify a significant change in state that may be of interest to any component that is *listening*. Components use *rules* to process events; a rule matches against local state and one or more events that are received by the component. The body of a rule is an action in terms of local state changes, events and messages. Components are designed in isolation, however global *invariants* place constraints on component state synchronization and lead to implementation requirements in terms of event connectivity between components.

Thus the characteristics are drawn from established and well understood concepts from component based design, distributed architectures.

B. An EDA Modelling Language

Section III-A has outlined the key features that characterize EDA models. This section describes how each of the features are represented as a conservative extension to UML in a similar way to component models described in [25] and [26]. The extension is deliberately minimal in order that existing UML tools can support EDA models in terms of stereotypes and, where OCL cannot be extended, using textual comments. Section IV uses the extended UML to implement a case study and is a key contribution of the paper.

Figure 1 shows a simple meta-model and its extension for EDA modelling. The basic modelling language consists of packages of classes and associations. Each class contains a collection of attributes and operations. An operation is specified using pre and post conditions.

The EDA language extends the basic language with the following features. `CmpDef` is a specialization of `Package` that defines a single component. Each of the other classes contained in a `CmpDef` must be structured types since they cannot have any behaviour. All classes in a `CmpDef` must be associated directly or indirectly to the component. `Component` is a specialization of `Class` that has business rules and input/output events. A component listens for output events raised by other components; when an output event is produced it is received as an input event by the listening component. `CmpOperation` is a component operation specification that can involve an action that, in addition to pre and post-conditions, defines the events that are raised by the operation. `Action` is used in a component operation to specify the events that are raised when the operation completes. An action is either single event `Raise` or is a `Loop` through a collection of elements, where an action is performed for each member of the collection. A `BizRule` has a guard that must be satisfied before the rule can be fired. After the rule is fired, the post-condition of the rule is satisfied and the rule action is performed. A `Pattern` matches against the events that have been received by a component. A pattern may match a single event `EventPattern`, may be conjunctions or disjunctions of patterns, or may be a negated pattern.

The meta-model in figure 1 describes an extension to conventional UML class diagrams. Although, UML has components, they do not correspond to `Component` and `CmpDef` in the meta-model, so we use stereotypes to tag and extend the appropriate UML elements; this allows standard UML modelling tools to be used to support EDA component modelling. Component definitions are represented as UML packages (class diagrams) with exactly one class with the stereotype `<<component>>`. Components may have attributes and operations which are interpreted as standard class features although operations are invoked using messages in the sense of SOA.

Associations on the UML class diagram from the component to classes define the local structured data of the component. These classes may have attributes and associations, but do not have any operations since all execution is defined by component interaction.

In addition, a component may have operations with stereotypes `<<eventin>>` and `<<eventout>>` The signature of the operation defines the name and internal structure of events. An input event declares that the event will be received via some externally monitored component. An output event declares that the component will raise the event via an operation or a business rule.

Operations are specified using standard OCL pre and post conditions. A `CmpOperation` can be defined for operations that includes an action:

```
context C::o(arg*)
  pre exp
  post exp
  action
```

The action part is of a constraint is used to specify the events

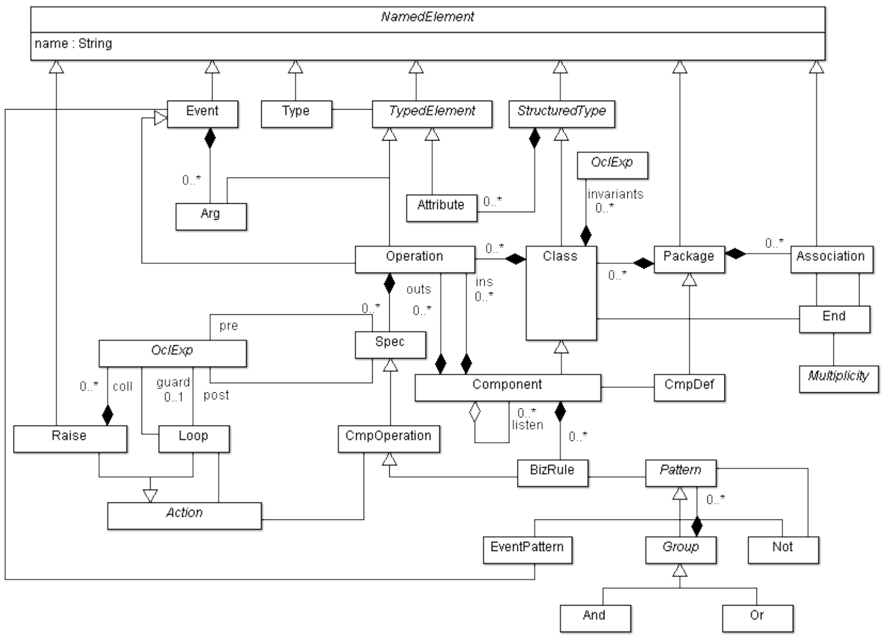


Fig. 1: EDA Meta Model

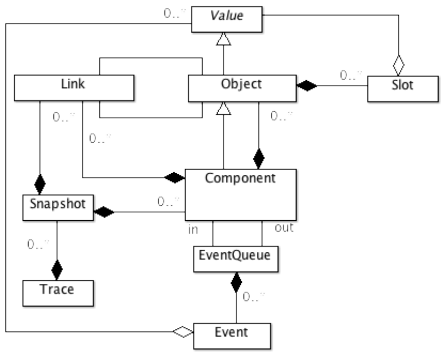


Fig. 2: Semantic Domain

that are raised by an operation (or a business rule). An action may involve a single event or may loop through a collection in order to raise several events:

```
action ::= raise name(exp*) |
           with name:type in exp [ when exp ] do action
```

Business rules monitor events received by a component. A rule may depend on multiple events and the current state of the component. The rule may cause a state change to the component and may raise events. We propose a new form of OCL that supports business rules that match event and data patterns such as those described in [27] as follows:

```
context C
  on pattern*
  pre exp
  post exp
  action
```

The semantics of EDA models is defined in terms of a relationship between the meta-model in figure 1 and the semantic domain model given in figure 2. The semantic domain extends object models (snapshots) to support components. An object model consists of snapshots that contain objects and links. The objects are instances of classes and have slots that are instances of the corresponding class attributes. The links are relationships between objects and are instances of associations between corresponding classes.

A component is a special type of object that has a pair of event queues. Each queue contains a partially ordered sequence of events. The input queue contains events that have been received by a component and the output queue contains events that have been produced by the component. A *trace* is a sequence of snapshots that describe how a system of objects and components change over time.

A semantic relationship holds between the meta-model and the semantic domain that defines the meaning of EDA models. The essential features of the relationship are as follows. Snapshots may only contain objects, components and links that correspond to the classes, component definitions and associations that are defined in the corresponding EDA model. A step in a trace is defined to hold when a rule is fired or when a message is sent. A message is handled by an operation whose behaviour is defined in terms of a pre and post-condition. The pre-condition must hold in the pre-snapshot and the post-condition must hold in the post-snapshot. A rule is ready to fire when its pattern is satisfied by the state of its component in terms of the slots, links and input event queue. The rule's pre-condition must also hold. The post-snapshot in the step is defined by the post-condition of the operation and its action. The action may cause one or more events to be raised and the listening relationships in the model define the co-ordination of input/output event queues. A rule may only fire once using the same component state which implies that events have unique identifiers.

IV. Case Study

A. Overview

Our case study for evaluating the design of the EDA Modelling Language is drawn from a UK higher education (HE) context. In the UK, EA and in particular the use of shared services has become an increasingly important strategic driver. Our approach presents one possible technology for addressing some of the issues currently prevalent in this sector. A recent report into the use of IT in HE [28] examines how successful UK HE has been at exploiting the opportunities offered by ICT. It argues that there is little high-level strategic impetus behind the integration and that the sector is struggling to get systems to talk to each other. The associated JISC report[1] describes the public sector support for SOA in HE with the intention of leading to shared services across the sector. It argues that EA can address problems relating to data silos, information flow, regulatory compliance, strategic integration, institutional agility reduction in duplication and reporting to senior management.

EA can be applied within an organization in order to determine how to comply with externally applied regulations. Architecture models can answer questions about the reuse of existing components, the locality of regulatory information and to identify the need for new information sources. This section describes a case study which is typical of current issues facing the UK HE sector. We describe the case study and then use the EDA modelling and simulation languages to describe an EA for the application.

The UK Borders Agency requires all Higher Education institutions to produce a report that details the number of points of contact between the institution and any student that has been issued with a student visa. This regulation places a requirement on the institution to ensure that the information is gathered at the appropriate points of contact. Furthermore, there is a business imperative for each institution to be able to detect students that may be likely to fail to record the required number of contact points in order to take remedial action and thereby avoid paying penalties with respect to *trusted status* whereby visas are granted via a lightweight process.

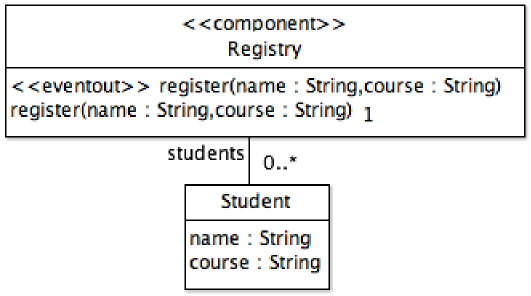
B. Components

The University of Middle England (UME) decides to construct an EA model in order to determine the components, data stores and interactions that are required to comply with the regulations. The model will be exercised through simulation and will be the basis of an as-is and to-be analysis in order to plan how to proceed. The first step is to construct a model of the components that will be required. New components can be designed as part of the model, however reuse of existing components is preferred to keep costs down.

[1]http://www.jisc.ac.uk/media/documents/techwatch/jisc_ea_pilot_study.pdf

UME makes a list of existing systems that can be used to track student interactions: the Library; the Student Office for each University Department; the Registry. A new component, the Monitor, is required in order to aggregate the events raised by the components and to manage a point of contact database.

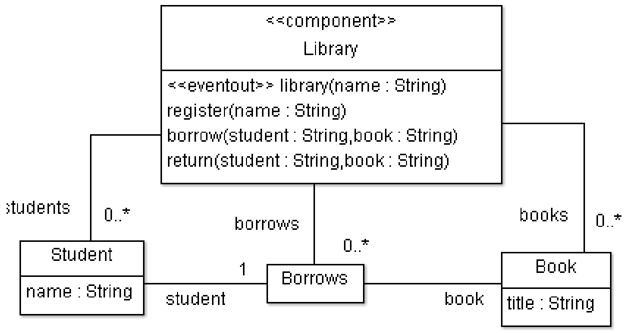
1) The Registry: The registry is responsible for managing the list of all UME students and their course of study. It is the first point of contact for any student where the student registers for a particular course:


<<component>>
Registry
<<eventout>> register(name : String,course : String)
register(name : String,course : String)
1
students
0..*
Student
name : String
course : String

The `Registry` component manages a database of students containing the student's name (assumed to be unique) and the name of the course they are to study. Currently the registry does not inform any other UME system of a new registration. However, student registration is a point of contact and therefore our model requires that an event is raised:

```
context Registry::register(name:String,course:String)
  pre not students->exists(s | s.name = name)
  post students->exists(s | s.name = name)
  raise register(name,course)
```

2) The Library: Once a course has started, students use the library in order to access learning resources. Students must register with the library, after which they can borrow and return books. The current library system manages a database of registered students, books and borrowing records.

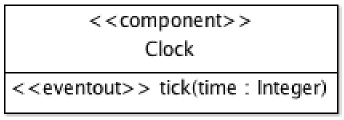


The current library system provides an interface of operations for registration and book borrowing. Each operation counts as a point of contact and therefore should raise events. The event just informs any listener of a library transaction for a particular student:

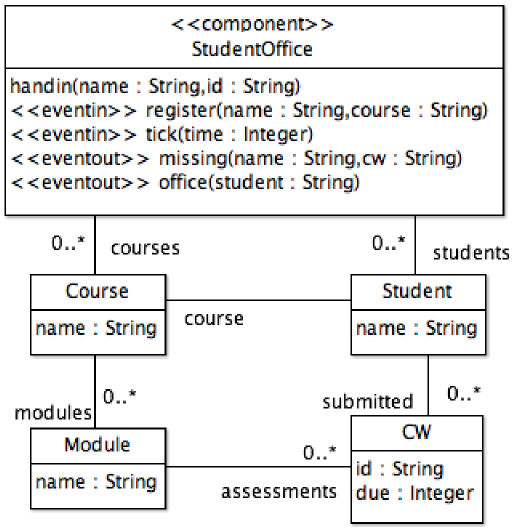
```
context Library::register(name:String)
  pre not students->exists(s | s.name = name)
  post students->exists(s | s.name = name)
  raise library(name)

context Library::borrow(student:String,book:String)
  pre students->exists(s | s.name = student) and
      books->exists(b |
      b.name = book and
      not borrows->exists(b | b.book = b))
post borrows->exists(b |
      b.student.name = student and
      b.book.name = book)
raise library(name)
```

3) Time: The UK Borders Agency requires UME to report on students by a given date which we will measure in the number of elapsed weeks after the start of a course. Therefore, the UME IT architecture must be aware of how many weeks have passed and includes a timing component that *ticks* at the end of each week:



4) The Student Office: Each UME department has a student office that manages student coursework (although since they are all the same we will only include one student office in our UME design). Currently in UME there is no standardization between student offices. However, handing in coursework counts as a point of contact and late coursework is an indicator that things are going wrong for a student. UME decides to standardize student office components:


<<component>>
StudentOffice
handin(name : String,id : String)
<<eventin>> register(name : String,course : String)
<<eventin>> tick(time : Integer)
<<eventout>> missing(name : String,cw : String)
<<eventout>> office(student : String)
0..*
courses
0..*
students
Course
name : String
course
Student
name : String
modules
0..*
submitted
0..*
Module
name : String
0..*
assessments
CW
id : String
due : Integer

The office maintains a database of all courses including modules and courseworks. Each coursework has a due date (measured in weeks).

Registration events can be processed by each student office in order to initialize a student with their department:

```
context StudentOffice
  on register(student:String,course:String)
  post students->exists(s |
    s.name = student and
    s.course.name = course and
    s.submitted->size = 0)
```

Coursework submission counts as a point of contact and therefore the student office is required to raise an event:

```
context StudentOffice::handin(student:String,cw:String)
  post students->exists(s |
    s.name = student and
    s.submitted->includes(
      s.course.modules.assessments->select(a | a.id = cw)))
  raise office(student)
```

The student office processes timing events by checking whether there are any outstanding coursework and generating late coursework events. Such events can be used to remind students that they have a deadline and thereby improve the number of contacts within the limits imposed by the UK BA. The following query operations calculate a set of records of the form `{name=n;cw=i}` where `n` is the name of a student and `i` is a coursework identifier:

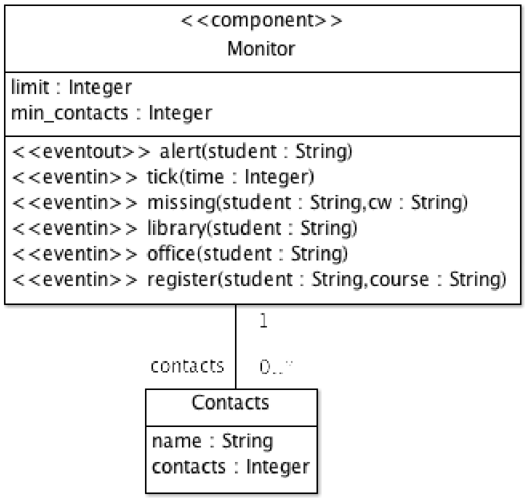
```
context StudentOffice::missed(time:Integer) =
 students->iterate(s R=Set{} |
  let req = s.course.modules.courseworks->select(c |
          c.due < time)
  in R + missed(s.name,req,s.submitted))

context StudentOffice::missed(s,req:Set(CW),done:Set(CW)) =
  (req-done)->collect(cw | {name=s;cw=cw.id})
```

The student office generates a missing event for each missing coursework:

```
context StudentOffice
  on tick(time:Integer)
  with x:{name:String;cw:String} in missed() do
  raise missing(x.name,x.cw)
```

5) The Monitor: Finally, UME requires a new component that monitors student contacts and generates alerts when the limit is reached:


<<component>>
Monitor
limit : Integer
min_contacts : Integer
<<eventout>> alert(student : String)
<<eventin>> tick(time : Integer)
<<eventin>> missing(student : String,cw : String)
<<eventin>> library(student : String)
<<eventin>> office(student : String)
<<eventin>> register(student : String,course : String)
1
contacts
0..*
Contacts
name : String
contacts : Integer

When a student is registered, the monitor must initialize a local record:

```
context Monitor
  on register(name:String,_)
  pre not contacts->exists(c | c.name = name)
  post contacts->exists(c | c.name = name and c.contacts = 1)
```

When the student uses the library or hands in coursework, the monitor must increase the number of contacts:

```
context Monitor
  on library(name:String) or office(name:String)
  pre contacts->exists(c | c.name = name)
  post contacts->exists(c |
    c.name = name and c.contacts = c.contacts@pre + 1)
```

The monitor generates an alert when the number of contacts is insufficient and when time limit has been reached or when the student has missed a coursework:

```
context Monitor
  on tick(time:Integer)
  pre time > limit
  with c:Contact in contacts when c.contacts < min_contacts do
  raise alert(c.name)

context Monitor
  on missing(student:String,cw:String)
  pre contents->select(c |
    c.name = student and c.contacts < min_contacts)
  raise alert(student)
```

C. Invariants

The previous sections have defined modules in isolation. Each component contains a model of locally managed data, implements an interface of operations, generates events, and processes incoming events via rules.

The components must work together to enforce a collection of invariants. At the specification level we need not be worried about *how* the components manage the interactions, we just state the conditions that must be satisfied.

Each component has a single instance and therefore, we can associate the name of the component with that instance when writing invariants. The following invariant ensures that the registry, student office and the library are all in-sync when a new student registers (note that not all students need to register with the library but if they do they must have been processed by the registry):

```
Registry.registered->size = StudentOffice.students->size
Registry.registered->size >= Library.students->size
Registry.registered->size = Monitor.contacts->size
```

The number of points of contact must be enforced by an invariant that takes into account the number of registrations, the uses of the library and coursework submission. The total number of coursework submissions is maintained by the student office, however the total number of library interactions is not maintained since the borrowing record is removed when a student returns a book. Fortunately, each component maintains a list of raised and received events via special associations: `in` and `out`. Events are just normal data that can be processed via these associations:

```
Monitor.contacts->forall(c | c.contacts =
  StudentOffice.students->select(s |
    s.name = c.name).submitted->size +
  Library.out->select(e | e.name = c.name)->size)
```

D. Monitoring Connections

The specification of each component given above does not indicate the event connectivity between components. Each component raises a number of event types and may listen to events raised by any number of other components. The next stage in the process is to define listening relationships. Such a relationship can exist between any two components but really only makes sense when one component produces an event and the listening component consumes an event of the same type. Figure 3 shows the listening dependencies between UME components for the UK BA architecture. By placing a dependency between, for example, `Registry` and `Monitor`, we know that whenever the registry produces a `register(student,course)` event it will immediately be available to the monitor.

```
exp ::=
  component (exp*) {      components, monitored
    [state { term* }]     local data (optional)
    [operations { op* }]  methods (optional)
    [rules { rule* }]     event processing (optional)
  }
| fun(arg*) exp           functions
| exp(exp*)               applications
| var                     variables
| atom                    integers, strings, booleans
| state                   local data
| self                    reference
| { exp* }                blocks
| { bind* }               records
| [ exp | qual* ]         lists
| new term                extension
| delete term             deletion
| if exp then exp else exp conditional
| raise term              event generation
| case exp { arm* }       matching
| let bind* in exp        locals
| letrec bind* in exp     recursive locals
| exp <- term             message passing
pattern ::=
  var                     variables
| name(pattern*)          term patterns
| atom                    ints, strings, bools
| name = pattern          pattern binding
| [pattern*]              lists
| pattern:pattern         cons pairs
| ? exp                   predicate
term  ::= name(exp*)
arm   ::= pattern -> exp
bind  ::= pattern = exp
qual  ::= pattern <- exp | ?exp
op    ::= name(arg*) { exp* }
rule  ::= name : pattern* { exp* }
```

Fig. 4: EDA Simulation Language

V. EDA Simulation

Figure 4 shows a *domain specific language* that has been defined to support EDA simulation. It is beyond the scope of this paper to describe the detailed semantics of the language; we give a brief overview and then show how the UME case study is implemented in the language.

An EDA simulation consists of a collection of component definitions. Each component monitors events (essentially the connections shown in figure 3). The local data of a component is represented as a collection of terms of the form `Name(v,...v)`. Components have conventional OO methods that can be invoked synchronously (using `.`) and asynchronously (using `<-`). Component rules match patterns against events raised by monitored components and then perform expressions.

Rule and operation bodies are standard functional language expressions except for: sequenced blocks, local data modification (`new` and `delete`) and events `raise`. Access to the list of all local terms in a component is provided via `state`. List comprehensions are useful to process the local state, for example `[ name | Person(name,age) <- state, ?(age > 65) ]` constructs the names of all the people in the local data that are over `65`.

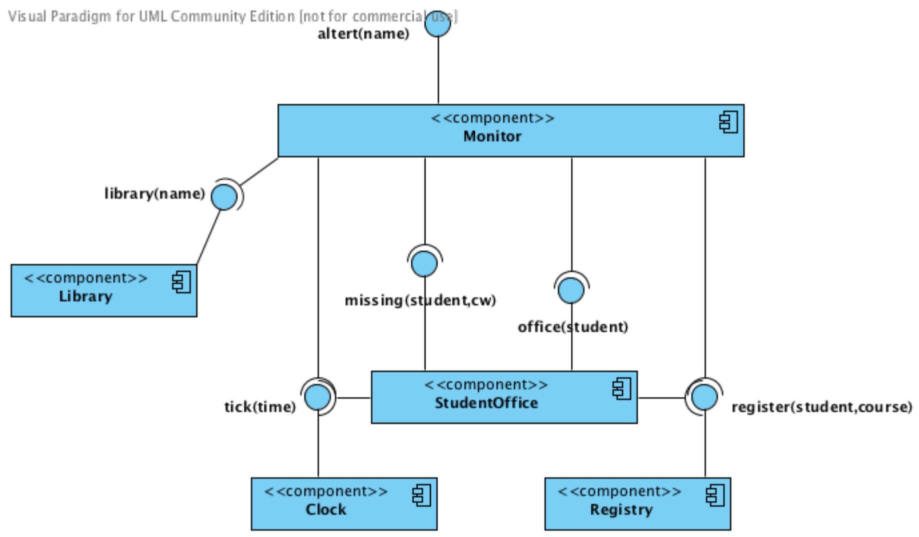


Fig. 3: Component Dependencies

An EDA system consists of a collection of named components that publish their operation interfaces and that monitor events. Events generated by a component C are received by all components that monitor C. Component rule patterns match against received events and against local data. When all patterns have matched the rule is ready to fire; rules are fired in any order.

The registry is defined as a component with no initial state and no rules. The `register` operation allows new students to be added to the local data and raises a registration event. The `students` operation is used to query the registry database:

```
component registry
  operations {
    register(student,course) {
      new Student(student,course);
      raise Register(student,course)
    }
    students() { state }
  }
}
```

The following component defines a library. The local data declares all the available books. The `register` operation checks whether the student exists, if not then a new student is created and a library usage event is raised. The `borrow` and `return` operations are as expected. The library defines an operation `books` that calculates a list of books currently borrowed by a student; this will be used by the user interface component later. The library defines no rules.

```
component library() {
  state {
    Book('b1','Programming') ...
  }
  operations {
    register(student) {
      if student?(student)
      then print(student + ' exists in Library.')
      else {
        new Student(student);
        raise Library(student) }
    }
    borrow(student,book) {
      if student?(student) and book?(book)
      then {
        new Borrows(student,book);
        raise Library(student)
      }
      else print(student + ' cannot borrow ' + book)
    }
    return(student,book) {
      delete Borrows(student,book);
      raise Library(student)
    }
    student?(name) {
      [ s | Student(s) <- state,?(s=name)] != []
    }
    book?(id) {
      [ i | Book(i,n) <- state,?(i=id)] != []
    }
    books(student) {
      [ book | Borrows(s,id) <- state, ?(s=student),
               Book(i,book) <- state, ?(i=id) ]
    }
  }
}
```

The student office component is defined below. It predefines the available courses, modules and assessments as local data. The `coursework` operation queries the currently submitted assessments for a student. The `missed` operation calculates the assessments that are outstanding for all students. The `handin` operation is used to record each new assessment.The rule `time` uses the `for` operation to raise a missing event for each outstanding coursework. The `register` rule initializes a `Student` record in the student office:

```
component student_office(clock,registry) {
  state {
    Course('MBA',[
      Module('Marketing',[CW('MCW1',5),CW('MCW2',8)])])
    Course('Business',[
      Module('Marketing',[CW('B_MCW1',6),CW('B_MCW2',9)])])
    Course('Business and IT',[
      Module('Marketing',[CW('BIT_MCW1',4),CW('BIT_MCW2',6)])])
  }
  operations {
    courseworks(student) {
      [ id | student(s,c,cws) <- state, ?(s=student), id <- cws ]
    }
    missed(time) {
      [ {name->name;id->id} |
           Student(name,studying,cws) <- state,
           Course(course,modules) <- state,
           ?(course=studying),
        Module(mname,assessments) <- modules,
        CW(id,due) <- assessments,
        ?(due<time),
        ?(not(member(id,cws))) ]
  }
  handin(student,module,coursework) {
    case [stud | stud=Student(s,c,cws) <- state, ?(student=s)] {
      [Student(s,c,cws)] -> {
        delete Student(s,c,cws);
        new Student(s,c,cws+[coursework])
      };
      otherwise -> print('handin: cannot find: ' + student)
    };
    raise Office(student)
  }
  for(list,action) {
    case list {
      [] -> true;
      x:xs -> { action(x); for(xs,action) }
    }
  }
}
rules {
  time: Time(n) {
    delete Time(n-1);
    self.for(self.missed(n),fun(x) raise Missing(x.name,x.id))
  }
  register : Register(s,c) {
    new Student(s,c,[])
  }
}
}
```

The monitor component listens to events from all other components. It handles registration events by initializing a `Contact` record. Library and student office events increase the contact count by 1. A `Missing` event occurs when a student has not handed the coursework in on time and raises an alarm. An alarm is also raised when the required number of contacts have not been achieved by the required time.

```
component monitor(clock,registry,student_office,library) {
  state {
    Contacts(4,10)
  }
  operations {
    count(student) {
      case [ c | c=Contact(s,n) <- state, ?(s=student)] {
        [Contact(s,n)] -> n;
        x -> 0
      }
    }
  }
  rules {
    register: Register(s,c) {
      new Contact(s,1)
    }
    library: l=Library(s) c=Contact(s,n) {
      delete l;
      delete c;
      new Contact(s,n+1)
    }
    office: l=Office(s) c=Contact(s,n) {
      delete l;
      delete c;
      new Contact(s,n+1)
    }
    missing: m=Missing(s,cw) {
      delete m;
      raise Alarm(s,'COURSEWORK')
    }
    alarm: Time(t)
          Contact(s,n)
          Contacts(min,t')
          ?(n<min) ?(t=t') {
      raise Alarm(s,'CONTACTS')
    }
    time: Time(n) {
      delete Time(n-1)
    }
  }
}
```

VI. Implementation and Analysis

In order to run, the simulation must be seeded with some events that populate the components. For simulation purposes we define two new components: the `simulator` and the `screen`. The simulator component has a state consisting of a collection of terms each of which is a message to one of the UME components. The message contains a time and a simulation rule matches each clock tick against the time in a message term; when the time matches, the message is sent via the built-in `send` operation:

```
component simulator(clock,monitor) {
  state {
    Send(1,registry,'register',['stud01','MBA'])
    ...
  }
  rules {
    time: Time(n) { delete Time(n-1) }
    step: Time(t) Send(t,target,message,args) {
      send(target,message,args)
    }
    contact: Alarm(s,'CONTACTS') {
      raise Display(s,'red')
    }
    assess: Alarm(s,'COURSEWORK') {
      raise Display(s,'green')
    }
  }
}
```

In addition to sending messages, `simulator` monitors `Alarm` events. Each event contains the name of a student and a tag that describes the context of the alarm. The simulator generates events that describe the level of importance associated with each student: `green` means that the student is giving cause for concern and `red` means that the student may cause UME to fail to achieve a key business goal.

The implementation uses a web-server to display information about students. The information to be displayed is encoded as HTML using terms. Figure 5 shows part of the screen model that is used to encode HTML. Each class is encoded as a term, attributes are encoded as term data and associations are encoded as sub-terms. For example, to produce a screen that contains a single table with two students:

```
Screen(
  Table([
    [Text('student1'),Text('MBA')],
    [Text('student2'),Text('Business')]
  ])
)
```

A `Button` has a label and an action. The action is a function with no arguments; when the button is pressed in a web-browser, the function is called causing any actions in the body of the function to be performed.

A `Div` term acts like a `<DIV>` element in HTML. The record is used to set style attributes for the scope of the entries in the body of the div. The optional record attached to a `Table` performs the same function.

The `screen` component is defined below:

```
component screen(clock,simulator) {
  operations {
```

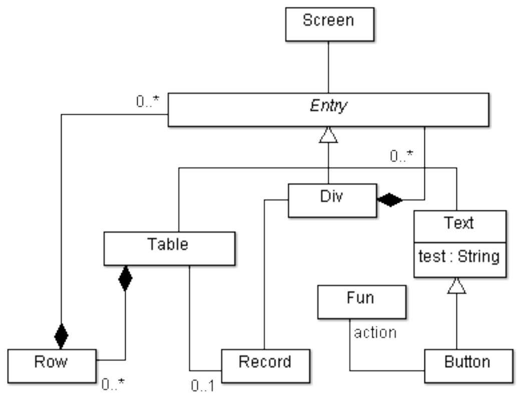


Fig. 5: Screen Model

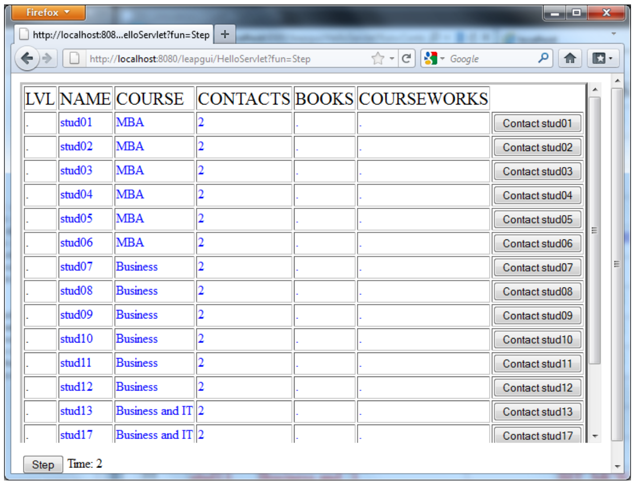
```
screen() {
  Table([[students()],[Table([[step(),ticker()]])]])
}
text(n,c,s) {
  Div({style->'font-size:'+n+'px; color:'+c+';'},[Text(s)])
}
headers(str) { text(22,'black',str) }
display(name,str) {
  case [ c | Display(s,c) <- state, ?(name=s) ] {
    ['red'] -> text(16,'red',str);
    ['red','green'] -> text(16,'red',str);
    ['green','red'] -> text(16,'red',str);
    ['green'] -> text(16,'green',str);
    otherwise -> text(16,'black',str);
  }
}
level(name) {
  case [ c | Display(s,c) <- state, ?(name=s) ] {
    ['red'] -> Text('!!!');
    ['red','green'] -> Text('!!!');
    ['green','red'] -> Text('!!!');
    ['green'] -> Text('???');
    otherwise -> Text('.')
  }
}
courseworks(student) {
  display(student,list(student_office.courseworks(student)))
}
cols() {
  ['LVL','NAME','COURSE','CONTACTS','BOOKS','COURSEWORKS']
}
student_details(name) {
  [level(name),
   display(name,name),
   display(name,course),
   display(name,monitor.count(name)),
   books(name),
   courseworks(name),
   Button('Contact ' + name, fun() {
     delete Display(name,'red');
     self <- update_display(name)
   })]
}
students() {
  let body = [ student_details(name) |
               Student(name,course) <- registry.students() ]
      head = [map(header,cols())]
  in Table(head+body)
}
books(student) {
  display(student,list(library.books(student)))
}
list(l) {
  case l {
    [] -> '.';
    [t] -> t;
    t:ts -> t + ', ' + (list(ts))
  }
}
ticker () { Text('Time: ' + time()) }
```

```
  time() {
    case [ t | Time(t) <- state ] {
      [t] -> t;
      [] -> 0
    }
  }
  step() { Button('Step',fun() clock <- tick()) }
  server() {
    case [ s | Server(s) <- state ] {
      [s] -> s
    }
  }
  update_display() {
    server() <- display(Screen(screen())) }
  tick(server) {
    new Server(server);
    server <- display(Screen(screen()))
  }
 }
 rules {
  tick_screen: t=Time(n) {
    delete Time(n-1);
    self.update_display()
  }
 }
}
```

The life-cycle of `screen` is as follows. When the web-server first connects to `screen` it calls `tick` and supplies a web-server `server`. The server is stored in the local data of `screen` as the term `Server(server)` and is accessed using the function `server()`. When `screen` wants to update web-server, it sends a message `browser <- display(s)` where `s` is a screen (a term-instance of the model in figure 5). The initial screen contains a table of student information and a button created by the operation `step()`; the button-action is a function that sends a `tick()` message to the clock causing a `Tick(t)` event to be raised and received by all components monitoring `clock`. The rule `tick_screen` is fired when `screen` receives a `Tick(t)` event and causes the web-server to be supplied with a new screen via the `update_display()` operation.

The `screen` component uses the `screen()` operation to produce a table containing the following entries, each of which is returned using a separate operation: student details; the step-button; the current time. The `student_details(name)` operation is used to produce a table row for the student with the supplied name. The `display` operation is used to construct text that has a colour defined by the current alarm level for a student. The query operations `student_office.courseworks(student)`, `monitor.count(name)`, `registry.students()`, `library.books(student)` are used to construct each row. Where information is a list of strings, the operation `list(l)` is used to concatenate the strings and separate them with commas.

The implementation is written in Java and has an architecture as shown in figure 6. A standard web-server runs a servlet labelled GUI that processes a term-encoding of the model in figure 5. When the GUI starts, it supplies a handle to itself (`tick(server)`) to the screen component. The `screen` component informs GUI of a new screen `server <- display(Screen(...))` and prompts the clock to advance under user control via the button-action `tick()`.

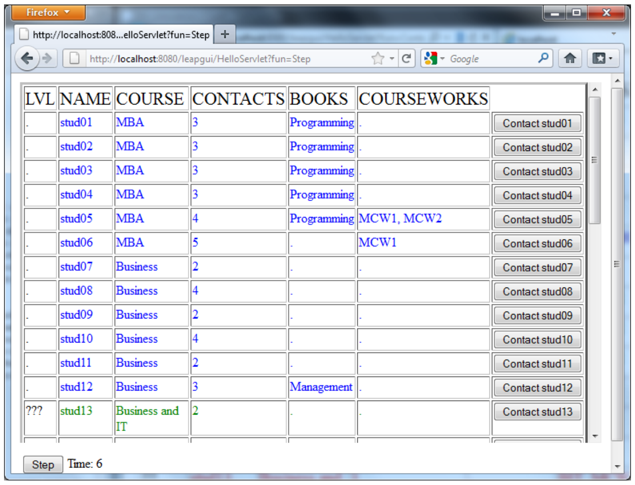


http://localhost:8080/leapgui/HelloServlet?fun=Step

LVL	NAME	COURSE	CONTACTS	BOOKS	COURSEWORKS	
.	stud01	MBA	2	.	.	Contact stud01
.	stud02	MBA	2	.	.	Contact stud02
.	stud03	MBA	2	.	.	Contact stud03
.	stud04	MBA	2	.	.	Contact stud04
.	stud05	MBA	2	.	.	Contact stud05
.	stud06	MBA	2	.	.	Contact stud06
.	stud07	Business	2	.	.	Contact stud07
.	stud08	Business	2	.	.	Contact stud08
.	stud09	Business	2	.	.	Contact stud09
.	stud10	Business	2	.	.	Contact stud10
.	stud11	Business	2	.	.	Contact stud11
.	stud12	Business	2	.	.	Contact stud12
.	stud13	Business and IT	2	.	.	Contact stud13
.	stud17	Business and IT	2	.	.	Contact stud17

Step Time: 2

(a) Student Registration

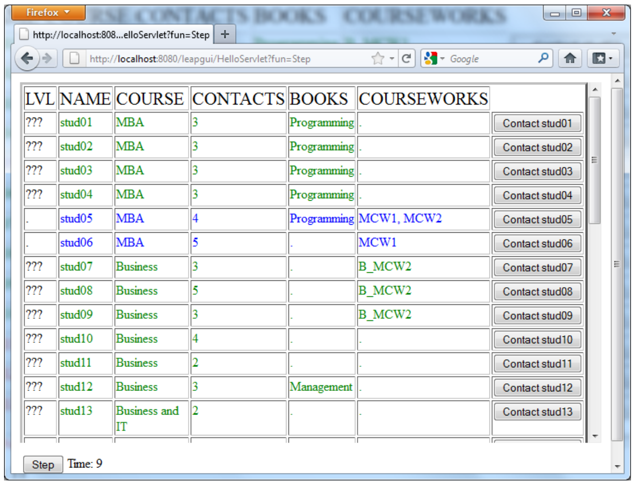


http://localhost:8080/leapgui/HelloServlet?fun=Step

LVL	NAME	COURSE	CONTACTS	BOOKS	COURSEWORKS	
.	stud01	MBA	3	Programming	.	Contact stud01
.	stud02	MBA	3	Programming	.	Contact stud02
.	stud03	MBA	3	Programming	.	Contact stud03
.	stud04	MBA	3	Programming	.	Contact stud04
.	stud05	MBA	4	Programming	MCW1, MCW2	Contact stud05
.	stud06	MBA	5	.	MCW1	Contact stud06
.	stud07	Business	2	.	.	Contact stud07
.	stud08	Business	4	.	.	Contact stud08
.	stud09	Business	2	.	.	Contact stud09
.	stud10	Business	4	.	.	Contact stud10
.	stud11	Business	2	.	.	Contact stud11
.	stud12	Business	3	Management	.	Contact stud12
???	stud13	Business and IT	2	.	.	Contact stud13

Step Time: 6

(b) Missing Coursework

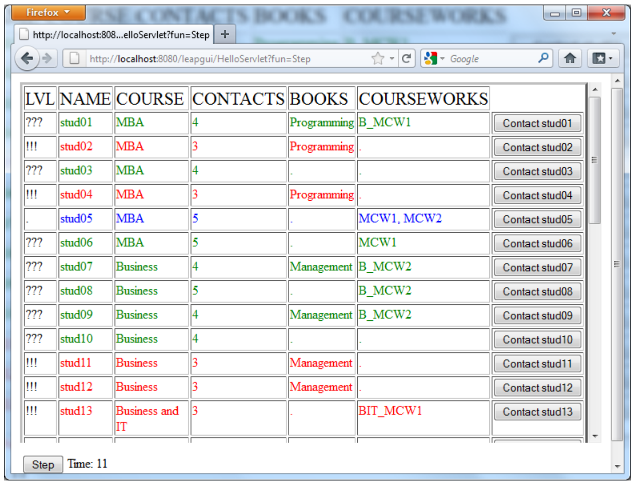


http://localhost:8080/leapgui/HelloServlet?fun=Step

LVL	NAME	COURSE	CONTACTS	BOOKS	COURSEWORKS	
???	stud01	MBA	3	Programming	.	Contact stud01
???	stud02	MBA	3	Programming	.	Contact stud02
???	stud03	MBA	3	Programming	.	Contact stud03
???	stud04	MBA	3	Programming	.	Contact stud04
.	stud05	MBA	4	Programming	MCW1, MCW2	Contact stud05
.	stud06	MBA	5	.	MCW1	Contact stud06
???	stud07	Business	3	.	B_MCW2	Contact stud07
???	stud08	Business	5	.	B_MCW2	Contact stud08
???	stud09	Business	3	.	B_MCW2	Contact stud09
???	stud10	Business	4	.	.	Contact stud10
???	stud11	Business	2	.	.	Contact stud11
???	stud12	Business	3	Management	.	Contact stud12
???	stud13	Business and IT	2	.	.	Contact stud13

Step Time: 9

(c) Warning Signs



http://localhost:8080/leapgui/HelloServlet?fun=Step

LVL	NAME	COURSE	CONTACTS	BOOKS	COURSEWORKS	
???	stud01	MBA	4	Programming	B_MCW1	Contact stud01
!!!	stud02	MBA	3	Programming	.	Contact stud02
???	stud03	MBA	4	.	.	Contact stud03
!!!	stud04	MBA	3	Programming	.	Contact stud04
.	stud05	MBA	5	.	MCW1, MCW2	Contact stud05
???	stud06	MBA	5	.	MCW1	Contact stud06
???	stud07	Business	4	Management	B_MCW2	Contact stud07
???	stud08	Business	5	.	B_MCW2	Contact stud08
???	stud09	Business	4	Management	B_MCW2	Contact stud09
???	stud10	Business	4	.	.	Contact stud10
!!!	stud11	Business	3	Management	.	Contact stud11
!!!	stud12	Business	3	Management	.	Contact stud12
!!!	stud13	Business and IT	3	.	BIT_MCW1	Contact stud13

Step Time: 11

(d) Business Goal in Jeopardy

Fig. 7: Simulation

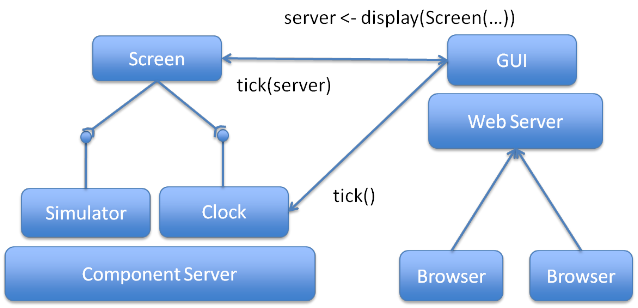
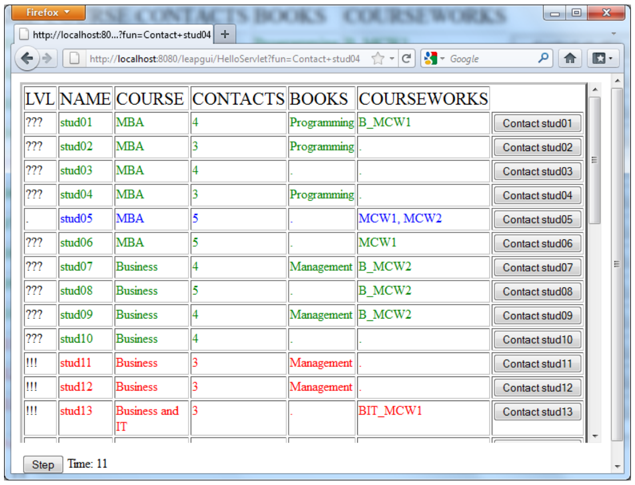


Fig. 6: Implementation Architecture

A sequence of simulation snapshots is shown in figure 7; each snapshot is a browser screen-shot generated after several clicks of the `Step` button. Figure 7a shows the situation just after students have registered for their courses and have eagerly used the library. Figure 7b occurs a little later and shows that student `stud13` has missed a coursework deadline. Figure 7c shows the situation just before the UK BA deadline; students `stud05` and `stud06` are hitting coursework deadlines, however all other students have missed at least one deadline. Figure 7d shows the situation immediately after the deadline has passed. The system has flagged all students who have had insufficient contacts with UME. Notice that since the deadline for `MCW2` has passed, student `stud06` has changed status.

The buttons on the right hand side of the screen allow the administrator to override students who are flagged as having insufficient contacts. Figure 8 screen-shot shows the situation where students `stud02` and `stud04` have been contacted and their status has been reset (although their course-works remain outstanding).

The simulation can be used by UME to analyze whether or not the proposed architecture achieves the business goal of complying with the UK BA regulations. It can determine whether all appropriate events are recorded, whether there is redundancy in the system, and whether existing IT systems are sufficient. Once complete, a simulation can be implemented by mapping it onto concrete IT systems and business processes.


Firefox
http://localhost:80...?fun=Contact+stud04
http://localhost:8080/leapgui/HelloServlet?fun=Contact+stud04
Google

LVL	NAME	COURSE	CONTACTS	BOOKS	COURSEWORKS	
???	stud01	MBA	4	Programming	B_MCW1	Contact stud01
???	stud02	MBA	3	Programming	.	Contact stud02
???	stud03	MBA	4	.	.	Contact stud03
???	stud04	MBA	3	Programming	.	Contact stud04
.	stud05	MBA	5	.	MCW1, MCW2	Contact stud05
???	stud06	MBA	5	.	MCW1	Contact stud06
???	stud07	Business	4	Management	B_MCW2	Contact stud07
???	stud08	Business	5	.	B_MCW2	Contact stud08
???	stud09	Business	4	Management	B_MCW2	Contact stud09
???	stud10	Business	4	.	.	Contact stud10
!!!	stud11	Business	3	Management	.	Contact stud11
!!!	stud12	Business	3	Management	.	Contact stud12
!!!	stud13	Business and IT	3	.	BIT_MCW1	Contact stud13

Step Time: 11

Fig. 8: Manual Override

A. Conclusion

Event Driven Architecture is a style of system design that lacks modelling notation. We have analyzed EDA, identified its characteristic features, and have proposed a modelling notation and associated simulation language. The modelling language is defined as a conservative extension to UML so that existing tools can use stereotyped UML elements to support component models. The meta-model for the language and its associated semantics has been defined. The simulation language shares the semantic domain of the modelling language and therefore EDA simulations can be viewed as a refinement of EDA models. We have implemented the simulation language in Java and used it to implement models taken from a representative case study.

The simulation language has been implemented as a Java interpreter[2]. Each component is deployed on a server that handles communication via sockets; so that each component can be written in the language or can be a wrapper around a pre-existing IT system. Any data, including functions and components, can be sent via messages; Java serialization is used to pass information and global naming is used to manage component identities. The intention is that the language will be extended to allow the migration of system architectures, firstly using complete simulation, then gradually incorporating existing components via wrappers and finally introducing new components.

There is currently no tool support for our approach, however we intend to investigate the use of the Eclipse Modelling Project to implement a UML profile and editor support for the language through XText. Other features of EA modelling include business goals and directives and we intent to extend the language to include these features.

References

[1] D. Barry, *Web services and service-oriented architecture: the savvy manager's guide*. Morgan Kaufmann Pub, 2003.

[2] G. Wang and C. Fung, "Architecture paradigms and their influences and impacts on component-based software systems," 2004.

[3] L. David, "The power of events: an introduction to complex event processing in distributed enterprise systems," 2002.

[4] A. Buchmann and B. Koldehofe, "Complex event processing," *it-Information Technology*, vol. 51, no. 5, pp. 241–242, 2009.

[5] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 147–160.

[6] D. Robins, "Complex event processing," 2010.

[7] C. Zang and Y. Fan, "Complex event processing in enterprise information systems based on rfid," *Enterprise Information Systems*, vol. 1, no. 1, pp. 3–23, 2007.

[8] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 407–418.

[9] A. Paschke, A. Kozlenkov, and H. Boley, "A homogeneous reaction rule language for complex event processing," *Arxiv preprint arXiv:1008.0823*, 2010.

[10] M. Lankhorst, "Introduction to enterprise architecture," in *Enterprise Architecture at Work*, ser. The Enterprise Engineering Series. Springer Berlin Heidelberg, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01310-2_1

[11] M. Ekstedt, P. Johnson, A. Lindstrom, M. Gammelgard, E. Johansson, L. Plazaola, E. Silva, and J. Lilieskold, "Consistent enterprise software system architecture for the cio - a utility-cost based approach," in *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*, 2004.

[12] C. Riege and S. Aier, "A Contingency Approach to Enterprise Architecture Method Engineering," in *Service-Oriented Computing–ICSOC 2008 Workshops*. Springer, 2009.

[13] K. Niemann, *From enterprise architecture to IT governance: elements of effective IT management*. Vieweg+ Teubner Verlag, 2006.

[14] T. Bucher, R. Fischer, S. Kurpjuweit, and R. Winter, "Analysis and application scenarios of enterprise architecture: An exploratory study," in *10th IEEE International Enterprise Distributed Object Computing Conference Workshops, 2006. EDOCW'06*, 2006.

[15] J. Henderson and N. Venkatraman, "Strategic alignment: Leveraging information technology for transforming organizations," *IBM systems Journal*, vol. 32, no. 1, 1993.

[16] J. Zachman, "A framework for information systems architecture," *IBM systems journal*, vol. 38, no. 2/3, 1999.

[17] J. Spencer *et al.*, *TOGAF Enterprise Edition Version 8.1*, 2004.

[18] D. Wisnosky and J. Vogel, "DoDAF Wizdom: A Practical Guide to Planning, Managing and Executing Projects to Build Enterprise Architectures Using the Department of Defense Architecture Framework (DoDAF)," 2004.

[19] M. M. Lankhorst, H. H.A. Proper, and J. Jonkers, "The Anatomy of the ArchiMate Language," *International Journal of Information System Modeling and Design*, vol. 1, no. 1.

[20] B. Michelson, "Event-driven architecture overview," *Patricia Seybold Group*, 2006.

[21] G. Sharon and O. Etzion, "Event-processing network model and implementation," *IBM Systems Journal*, vol. 47, no. 2, pp. 321–334, 2008.

[22] S. Overbeek, B. Klievink, and M. Janssen, "A flexible, event-driven, service-oriented architecture for orchestrating service delivery," *IEEE Intelligent Systems*, vol. 24, no. 5, pp. 31–41, 2009.

[23] M. Assmann and G. Engels, "Transition to service-oriented enterprise architecture," *Software Architecture*, pp. 346–349, 2008.

[24] C. Hall and P. Harmon, "The, enterprise architecture, process modeling, and simulation tools report," *BPTrends. com*, 2007.

[25] J. Cheesman and J. Daniels, *UML components*. Addison-Wesley, 2001.

[26] D. Lienhart, "Softbench 5.0: The evolution of an integrated software development environment," *HEWLETT PACKARD JOURNAL*, vol. 48, pp. 6–7, 1997.

[27] A. Barros, G. Decker, and A. Grosskopf, "Complex events in business processes," in *Business Information Systems*. Springer, 2007, pp. 29–40.

[28] E. Deeson, "The e-revolution and post-compulsory education–by boys, jos & ford, peter," *British Journal of Educational Technology*, vol. 39, no. 4, pp. 750–750, 2008.

[2]http://www.eis.mdx.ac.uk/staffpages/tonyclark/Software/leap.zip and http://www.eis.mdx.ac.uk/staffpages/tonyclark/Software/leapgui.zip