A Common Basis for Modelling Service-Oriented and Event-Driven Architecture

Tony Clark  
School of Engineering and Information Sciences  
Middlesex University  
London, UK  
t.n.clark@mdx.ac.uk

Balbir S. Barn  
School of Engineering and Information Sciences  
Middlesex University  
London, UK  
b.barn@mdx.ac.uk

ABSTRACT

Component based approaches to Enterprise Architecture (EA) include Service Oriented Architecture (SOA) and Event Driven Architecture (EDA). Model-based approaches to EA support SOA in terms of components and services expressed as interfaces and messages. However, there are few model-based approaches that support EDA even though SOA and EDA are both based on components. UML has components, however there is no support for events and no support for component patterns (or templates). This paper describes a simple extension to UML that supports both SOA and EDA. Components have both operation and event interfaces. The modelling language is implemented using a higher-order simulation language where templates are defined as functions over component definitions. The languages are described using a case study that has been implemented in Java.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous;  
C.0 [Computer Systems Organization]: Systems Architectures

General Terms

Modelling, Enterprise Architecture

Keywords

Service Oriented Architecture, Event Driven Architecture, Model Driven Engineering

1. 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. The approaches differ in details, 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.

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.’ [1].

Several different styles of architecture are possible. 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.

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.

As described in [2] and [3], complex events can be the basis for a style of EA design. EDA replaces 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 [4]. 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 [5] 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.’

Our claim is that system architectures should be based on both EDA and SOA. Therefore our aim is to provide a basis for modelling architectures based on both services and events. In the rest of the paper we will refer to EDA lan-
guages which should be read as a basis for both event and service based architectures. Our contribution is to analyse the requirements for EDA and then to propose a modelling language and a corresponding simulation language. To achieve the most general basis possible and to take advantage of the large number of UML modelling tools available, the modelling language is based on UML. The simulation language is an extension to a general purpose functional language. In particular, the functional language allows us to capture architectural patterns as functions over components which is an essential, but underdeveloped, area of architectural design. The modelling language has been implemented as stereotyped UML elements and the simulation language has been implemented as an interpreter in Java.

The paper is structured as follows. Section 2 reviews complex event processing and EA modelling languages. Section 3 performs a domain analysis on EDA and describes an EDA modelling language. Section 4 describes a case study as an EDA model. Section 5 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 6.

2. EVENT DRIVEN EA

2.1 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 [6] when compared to more traditional implementation techniques. The organization of systems into coherent interfaces has been argued [7] 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.

2.2 Complex Event Processing

Complex Event Processing (CEP) [8] 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 [9].

CEP processes 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.

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 [10]. The current state of the art is described in [11] 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 [12] events can be extracted from services, database, RFID and activities. The events are processed by rules that detect relationships, including temporal, between events in order to transform multiple events over time into business actions. In [12] 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 [13] 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 [14] 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.

3. EVENT DRIVEN MODELLING

An EA aims to represent an organization and its essential elements. However the complexity of the problem domain cannot be adequately addressed by one architectural style. In this paper we have proposed that two specific styles SOA and EDA have between them the necessary concepts that are able to provide more complete representation of EA. This section provides a overview of the unification of EDA and SOA by identifying the common features in section 3.1 and proposing a specialization of UML class models that support the features in section 3.2.

3.1 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.

3.2 An EDA Modelling Language

Section 3.1 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 [15] and [16]. The extension is deliberately minimal in order that existing UML tools can support EDA models in terms of stereotyped and, where OCL cannot be extended, comments. Section 4 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. Components may be nested, a parent always listens to the events raised by its children. 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. 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 : CmpDef pre exp post exp action

The action part of a constraint is used to specify the events 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 [17] as follows:

context C on pattern exp pre exp post exp action

4. CASE STUDY

4.1 Overview

The use of shared services has become an increasingly important strategic driver in the UK. 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 [18] 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 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. 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.

Requirement: 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.

4.2 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 and to generate simulations. 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.

UME makes a list of existing systems that can be used to track student interactions: the Library; Departments; the Registry. Each department has a student office that handles student assignments, and a collection of lecturers. 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.

4.2.1 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:

```
<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) 
```

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) 
```

4.2.2 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.

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

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::borrow(student: String, book: String) 
pre students->exists(s | s.name = student) and 
post students->exists(s | s.name = student) 
raise borrow(student, book) 
```

```
context Library::borrow(student: String, book: String) 
pre students->exists(s | s.name = student) and 
post students->exists(s | s.name = student) 
raise borrow(student, book) 
```
4.2.3 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:

4.2.4 Departments

Each UME department has a student office and a number of academics. The student office manages student assignments and the lecturers have tutorial meetings with students; both of these count as points of contact. Late coursework is an indicator that things are going wrong for a student. A department is an example of how components nest:

A department contains a component that implements the student office and a collection of components that represent the information system that academics use to manage student meetings. The office maintains a database of all courses including modules and coursework. 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:

```
class StudentOffice

context register(name: String, course: String)
post students->exists(s | s.name = name 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:

```
class StudentOffice

context handin(cw: String)
post students->exists(s | s.name = name and s.submitted->includes(cw))
raise office(name)
```

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:

```
class StudentOffice:

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

A lecturer may have a meeting with a student. The meeting is registered on the department’s tutorial information system and raises an event:

```
class Lecturer:

context meeting(name: String)
raise meeting(name, student)
```

Since the Department component contains lecturer and office component it will automatically receive any events that they raise. The department conflates the contact messages into a single Department event:

```
class Department:

context on office(student) or meeting(lecturer, student)
raise department(student)
```

4.2.5 The Monitor

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

```
class Monitor

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

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

```
class Monitor

context 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:
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:

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

4.3 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 may have several instances, although it is often the case in a component based architecture that each component is deployed once. Therefore, when writing invariants, we will associate a component name with its set of instances, and in the case of a singleton set with the instance itself. 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):

```plaintext
Registry.registered->size =
Department->iterate(d | d.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, tutorial meetings and coursework submissions. 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:

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

5. EVENT DRIVEN SIMULATION

Figure 2 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:

```plaintext
exp ::= component [name] (exp) (components, monitored [state (term)]] [operations (op*]) [rules (rule*]) methods (optional)
bind* fields
)
| fun(arg*) exp functions
| exp(arg*) 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 (arms) catching
| let bind in exp locals
| letrec bind in exp recursive locals
| exp <- term message passing
| exp . name field reference
pattern ::= var
| name(pattern*) term patterns
| atom ints, strings, booleans
| name = pattern term binding
| [pattern*] lists
| pattern:pattern cons pairs
| pattern,pattern alternatives
| ? exp predicate

```

Figure 2: EDA Simulation 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. The local data of a component is represented as a collection of terms of the form Name(A, ...). 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.

Component expressions contain local state, operations, rules and fields. An important feature of the language is that it allows component expressions to be nested, i.e. components can contain children and functions can return components. The latter feature allows a pattern of components to be implemented as a function over collections of components that can be instantiated by applying the function to arguments. This feature is called component templates and is used to implement departments the UME architecture.

An EDA system consists of a collection of named compo-
ments 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') Book('b2','Management') }
  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) <- name ] } != []
    book?(id) { [ i | Book(i,n) <- id ] } != []
    books(student) {
      [ book | Borrows(s,book) <- state, ?(s=student), Book(i,book) <- state, ?(i=id) ]
    }
  }
```

Each department defines a student office that manages assignments. Unlike the library the student registry, there are many departments, therefore many student offices, and the number of departments may change over time. No matter how many offices there are, they all perform the same tasks and only differ in terms of the courses that are available.

We take a template approach to defining departments where a function is defined whose parameters represent the variable elements of a component and where the body of the function returns the component. A department component is defined using three templates: mk_academic that creates an academic; mk_student_office that maps a list of courses to a component; mk_department that maps a department name, a list of courses and a list of academic’s names to a department.

An academic is created using the following template that maps a name to a component. The component provides an operation that records a tutorial meeting between the lecturer and a student and raises an event:

```
mk_academic(n)
  component () {
    operations {
      student_meeting(student) { raise Meeting(name,student) }
    }
  }
```

The student office template is defined below. 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, and the missed rule raises an event when the component detects that a student has missed a coursework deadline:

```
mk_student_office(courses) {
  component (clock,registry) {
    operations {
      coursework(student) {
        [ id | Coursework(s,n,id,t) <- state, ?(s=student) ]
      }
      students() { state }
      missed(time) {
        [ (n->name)->id | Student(n,...) <- state, Course(course,modules) <- courses, ?(course=studying), Module(name,assessments) <- modules, CW(id,due) <- assessments, ?(due<time), ?(not(member(id,cws))) ]
      }
      handin(student,module,coursework) {
        new Coursework(student,module,coursework);
        case [ s | Student(n,...) <- state, ?(student=n) ] {
          [Student(s,c,cws)] -> {
            delete Student(s,c,cws);
            new Student(s,c,cws+coursework)
          }
        }
        raise Office(student)
      }
      for(list,action) {
        case list {
          [] -> true;
          x:xs -> (action(x); for(xs,action) )
        }
      }
      rules {
        missed: Time(t) {
          delete Time(t-1);
          self.for(self.missed(t),fun(x) raise Missing(x.name,x.id))
        }
        register : Register(student,course) {
          case [ x | Course(x,ms) <- courses, ?(x=course) ] {
            [ x ] -> new Student(student,course,[x])
          }
        }
      }
    }
  }
```

Each department is created using the following emplate that maps a department name, list of courses and list of academic names to a component. The component offers operations that query the courseworks for a student, hand-in an assignment, record a tutorial, and select an academic. The component monitors the events produced by the school office and academic sub-components and the rules map child-events to department-events:

```
mk_department(n,courses,academic_names) {
```
component(self, academics, self, school_office) {
  operations{
    courses(courses) { school_office.courses(courses) }
    handin(student, module, coursework) {
      school_office.handin(student, module, coursework)
    }
    meeting(student, academic) {
      (get_academic(academic)).student_meeting(student)
    }
    get_academic() {
      case [a | a <- academics, ?((a.name) = n)] { [a] -> a }
    }
  }
  rules{
    meeting: Meeting(lec, student) { raise Department(lec) }
    office: Office(student) { raise Department(lec) }
    missing: Missing(student, cu) { raise Missing(student, cu) }
    school_office = mk_student_office(courses)
    academics = { mk_academic() | n <- academic_names } name = n
  }
}

UME decides to try the new architecture with two departments: Computer Science and the Business School. Although each department has different courses and academics, the student monitoring system is created using the sme templates that are defined above:

computer_science = mk_department('Computer Science',
  Course('Business and IT',
    Module('Marketing', [CW('BIT_MCW1',4), CW('BIT_MCW2',6)]),
    Course('Business and Informatics',
      Module('Marketing', [CW('BI_MCW1',3), CW('BI_MCW2',9)])) ),
  ['Dr Piercemuller', 'Prof Plumb'])

business_school = mk_department('Business School',
  Course('Business and IT',
    Module('Marketing', [CW('BM_MCW1',5), CW('BM_MCW2',8)]),
    Course('Business',
      Module('Marketing', [CW('B_MCW1',6), CW('B_MCW2',9)])),
    Course('Business and Marketing',
      Module('Marketing', [CW('BM_MCW1',5), CW('BM_MCW2',11)])) ),
  ['Prof Inglenook', 'Dr Who'])

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 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.

6.2 HTML

The implementation uses a web-server to display information about students. The information to be displayed is encoded as HTML using terms. Figure 3 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('s1'), Text('MBA')], [Text('s2'), 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.

6.3 The Screen Component

The screen component is defined below:

component screen(clock, simulator) {
  operations{
    ...}

  rules{
    ...}
}

6. IMPLEMENTATION

The components defined in the previous section respond to messages and events. This section describes how the simulation is controlled (section 6.1), presented to the user (sections 6.2 and 6.3) and how the language is implemented (section 6.4).

6.1 The Simulator Component

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:

```
class 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.
Server \(\text{server}\) first connects to screen as the term Server(server) and is accessed using

\[
\text{screen}() \{ \text{Table}([\text{depts()}], [\text{Table}([\text{step()}, \text{ticker()}])]) \} \\
\text{Div}([\text{style} \rightarrow \text{`font-size: `; color: `+px; color:`+c++;`}], [\text{Text}(s)]) \\
\text{headers}(\text{str}) \{ \text{Text}(22, \text{`black', str}) \} \\
\text{display}(\text{name}, \text{s}) \{ \\
\text{case} \text{[ c \mid \text{Display}(\text{s}, \text{c}) \rightarrow \text{state}, \text{?}]} \{ \\
\text{['red'], ['red', 'green'], ['green', 'red']} \rightarrow \text{Text('!!!');} \\
\text{['green']} \rightarrow \text{Text('????');} \\
\text{otherwise} \rightarrow \text{Text('??')}} \\
\} \\
\text{level}(\text{name}) \{ \\
\text{case} \text{[ c \mid \text{Display}(\text{s}, \text{c}) \rightarrow \text{state}, \text{?}]} \{ \\
\text{['red'], ['red', 'green'], ['green', 'red']} \rightarrow \text{Text('!!!');} \\
\text{['green']} \rightarrow \text{Text('????');} \\
\text{otherwise} \rightarrow \text{Text('??')}} \\
\} \\
\text{coursesworks}(\text{stud}) \{ \\
\text{display}(\text{stud, list}([\text{coursesworks}(\text{stud}) \mid \text{d < departments}])) \\
\} \\
\text{student_details}(\text{name}) \{ \\
\text{[\text{level}(\text{name}), \text{display}(\text{name}, \text{name}), \text{display}(\text{name}, \text{course}), \text{display}(\text{name}, \text{monitor}.\text{count}(\text{name})), \text{books}(\text{name}), \text{coursesworks}(\text{name})],} \\
\text{Button('Contact ' + name, \text{fun})} \{ \\
\text{delete \text{Display}(\text{name}, \text{'red'})} \\
\text{self < update\text{display}(\text{name})} \} \\
\} \\
\text{depts()} \{ \\
[\text{Table}([\text{Text}(\text{department}.\text{name})]), [\text{students}(\text{department})] \mid \text{department < departments}] \\
\} \\
\text{students}(\text{department}) \{ \\
\text{let body }= [\text{student_details}(\text{name}) \mid \text{Student}(\text{name}, \text{course}) <\text{- department}.\text{school_office}.\text{students}()] \\
\text{in Table}([\text{map}(\text{headers}, \text{cols}) \mid \text{body}) \\
\} \\
\text{books}(\text{stud}) \{ \text{display}(\text{stud, list}([\text{library}.\text{books}(\text{stud})])) \} \\
\text{list()} \{ \\
\text{case} \text{l = [i \rightarrow `'; [t] \rightarrow t; t:ts \rightarrow t + ', ' + (\text{list}(ts))]} \\
\} \\
\text{ticker()} \{ \text{Text('Time: ' + time())} \\
\text{time()} \{ \text{case} \text{[ t | \text{Time}(t) < state ] \{ [t] \rightarrow t; [] \rightarrow 0 \}} \\
\text{step()} \{ \text{Button('Step', \text{fun}) \text{ clock }<\text{ticker}()} \\
\text{server()} \{ \text{case} \text{[ s | \text{Server}(s) < state ] \{ [s] \rightarrow s \}} \\
\text{update\text{display}()} \{ \text{server() < display(Screen(screen()))} \} \\
\text{tick(s)} \{ \text{new Server(s); s < display(Screen(screen()))} \} \\
\} \\
\text{rules} \{ \\
\text{tick\_screen: t\text{Time}(n) \{ \text{delete Time(n-1); self.update\text{display}()}} \\
\} \\
\text{departments} = [\text{computer_science, business\_school}] \\
\text{cols} = [\text{`LVL', `NAME', `COURSE', `CONTACTS', `BOOKS', `CWK']}
\]

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 3). 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 update\_display.

6.4 Implementation Architecture

The implementation is written in Java and has an architecture as shown in figure 4. A standard web-server runs a servlet labelled GUI that processes a term-encoding of the model in figure 3. 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().

A sequence of simulation snapshots is shown in figure 5; each snapshot is a browser screen-shot generated after several clicks of the Step button. Figure 5a shows the situation just after students have registered. Figure 5b occurs a little later and shows that student stud02 has handed in coursework on time but that student stud04 has missed the assignment deadline. Figure 5c shows the situation just after the UK BA deadline; students stud03 and stud04 have insufficient contacts to meet the regulations; student stud01 has sufficient contacts, but has missed a deadline; stud02 is doing fine. Figure 5d shows the situation where a member of the academic team has contacted stud04 directly and has used manual override to change the status.

6.5 Summary

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 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

\(^2\)http://bit.ly/vjNzX7
messages; Java serialization is used to pass information and global naming is used to manage component identities.

7. REFERENCES