Int J Software Informatics, Volume 7, Issue 2 (2013), pp. 137–168 Tel: +86-10-62661040
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org

Email: ijsi@iscas.ac.cn, ijsi2007@gmail.com

Dynamic Reconfiguration of Event Driven Architecture Using Reflection and Higher-Order Functions

Tony Clark and Balbir S. Barn

(Middlesex University, UK)

Abstract Component-based modelling is used as the basis of a number of approaches including Enterprise Architecture and System Architecture Design. Service Oriented Architecture (SOA) is a popular component-based approach but it has been criticised as not being sufficiently flexible. A more flexible alternative is Event Driven Architecture (EDA) that can support Complex Event Processing. Dynamic reconfiguration of component behaviour is attractive because it allows an architecture to be extended and modified in situ without being taken off-line, updated and redeployed. This article shows how higher-order functions and reflection can support dynamic reconfiguration and how this approach is integrated with EDA. The approach is defined as patterns expressed in a component modelling language called LEAP and validated through a case study.
Key words: component; service oriented architecture; event driven architecture; reconfiguration; higher-order functions

Clark T, Balbir B. Dynamic reconfiguration of event driven architecture using reflection and higher-order functions. *Int J Software Informatics*, **Vol.7, No.2 (2013): 137–168. http://www.ijsi.org/1673-7288/7/i157.htm**

1 Introduction

Component based software systems are organized as a collection of hierarchically composed components each of which contains private data and operation implementations. Components communicate by sending messages to each other. Messages may be sent directly to another component or may be sent to an output port that is connected to the input port of one or more target components. A message is a structured value with a name and a collection of arbitrarily complex data values.

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

---

Corresponding author: Tony Clark, Email: t.n.clark@mdx.ac.uk
Received 2012-10-04; Accepted 2012-12-17.

communication between components is performed with respect to a single generic event interface.

There are two important differences between SOA and EDA. Firstly EDA 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. Secondly, EDA can support dynamic extensibility through the introduction of new types of message both produced and consumed by a component. In general, SOA provides static interfaces that require an architecture to be rebuilt when new services are introduced.

Given that EDA and CEP are claimed to be more flexible than SOA, how should we design an architecture 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 software component architecture. The most widely used language for modelling systems is UML, however UML components are based on concepts from SOA and therefore do not support EDA and CEP.

Once designed, how do we maintain such an architecture? As described in Ref. [33] *modifiability* and *changeability* are two important characteristics affecting the maintainability of component-based systems.The requirement for services evolution is described in Ref. [24]. However, there has been little work on how to design an architecture that ensures adequate provision of these characteristics.

The contribution of this paper is to identify the key characteristic features of EDA and to show how they are represented in the architectural modelling language LEAP whose features allow EDA to be expressed as patterns that can be included in new component definitions. In particular we show how EDA with higher-order functions[32] and reflection[18] can facilitate dynamic reconfiguration of an architecture that would not be possible with an SOA based approach.

2 Background

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[4] when compared to more traditional implementation techniques. The organization of systems into coherent interfaces has been argued[36] as having disadvantages in terms of: extensions; accommodating new business functions; associating single business processes with complex multi-component interactions.

2.2 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.'[21].

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[6, 14, 17, 27, 30]: *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; *acquisition and mergers* describing the alignment of businesses and the changes that occur on both when they merge.

EA has its origins in Zachman's original EA framework[39] while other leading examples include the Open Group Architecture Framework (TOGAF)[35] and the framework promulgated by the Department of Defense (DoDAF)[37]. 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[22].

A number of commercial EA analysis and simulation tools are available [16]. 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.

2.3 Event driven architecture

As described in Ref. [26] and Ref. [34], 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[28]. Whilst a complex event based approach to architectural design must take efficiency concerns into account, the primary concern is how to capture, represent and analyse 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 Ref. [2] where a framework is proposed that allows enterprise architects to formulate and analyse 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.'

2.4 Complex event processing

Complex Event Processing (CEP)[12] 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[7].

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[1]. The current state of the art is described in Ref. [31] 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 Ref. [40] 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 Ref. [40] 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[38] describe a language called SASE for processing complex events from RFID devices. The language is based on 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 Ref. [29] 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.

2.5 Component reconfiguration

Batista et al[5] identify two types of run-time reconfiguration in component based systems: *programmed* reconfiguration where changes can be foreseen at design time and *ad-hoc* reconfiguration are changes that cannot be predicted. The authors describe an ADL called Plastik that uses rules and reflection to reconfigure component connections at run-time. The system does not support the kind of ad-hoc reconfiguration described in this article whereby the behaviour of an existing component can change without access to the implementation of the component.

The approach in Ref. [24] is based on *dynamic binding* for service evolution. The approach allows an SOA-based architecture to self-configure. The approach is a design technique based on workflows which could be simulated by the higher-order approach described in this article.

The need for reflection in dynamically reconfigurable systems is described in Ref. [18] where a reflective component model is used in the OpenRec framework to allow submitted scripts to operate on the component architecture. We have chosen to use reflection to provide access to a component's internal data model, however the higher-order features of LEAP could be used to implement the OpenRec approach.

Component reconfiguration is motivated by the need to evolve a component architecture and takes a number of forms. Patterns can be used to express the different types of reconfiguration as described in Ref. [19]. Our article addresses the need to change the behaviour of existing components without stopping, updating and redeploying the system. The LEAP modelling language can support other patterns of reconfiguration such as changing the topology of the component architecture, however these are not considered here.

The dynamic reconfiguration approach described in this article is similar to the *mixin* approach of Frag[41] whereby new behaviour is introduced by adding new superclasses and defining a method lookup mechanism. However we do not require specific language constructs for dynamically changing class based inheritance and therefore consider our approach to be at a finer level of granularity.

2.6 LEAP

LEAP[3, 8-11] is a technology for executable modelling of system architectures. It addresses the requirements and design phases of system development and provides a language for representing systems in terms of components, ports and connectors. LEAP takes the form of a textual language and a tool that is used to construct, execute and display aspects of the models as diagrams.

LEAP execution occurs in terms of message passing between components. A component consumes messages from its input ports and invokes the appropriate internal operation. An operation may change the internal state of the component and produce output messages. Component ports are connected so that messages sent to output ports are transferred to connected input ports.

This article uses parts of the LEAP language that address the design of component architectures in terms of their connectivity and behaviour. The relevant LEAP language sub-set is shown in Fig. 1 where keywords are defined in **bold**, non-alpha characters are terminals except for * that denotes 0 or more occurrences, [and] that denotes optionality (parentheses are occasionally used for grouping where the context should be clear). The non-terminals `o`, (initial lower-case letter) `name`, (initial upper-case letter) `Name` and `const` are assumed. The LEAP concrete syntax uses separators `;` and `,` to separate elements in sequences (such as arguments and record fields) however these are omitted from the definition of the syntax. The key aspects of the language are:

components Component syntax is defined by `cmp`. A component may have parent components whose definitions are included into the child. Each port defined by a component declares an interface of messages that it can process.

information A component defines a model in terms of classes and associations. Each class has a collection of fields. The state of a component is a list of terms that must conform to its declared model. The state of a component is updated using the **new**, **delete** and **replace** commands. A component can define a collection of invariants that are conditions that its state must satisfy. A component contains a **state** clause that can be used to create a starting state.

```
exp ::=                          expressions
  component cmp                  components
| fun(arg*) exp                  functions
| exp(exp*)                      applications
| name                           variables
| const                          ints,strs,bools
| exp.name                       field ref
| exp o exp                      binary exp
| state                          local data
| self                           reference
| { exp* }                       blocks
| { (name -> exp)* }             records
| [ exp | qual* ]                lists
| new term                       extension
| term                           terms
| delete term                    deletion
| if exp then exp else exp       conditional
| replace pattern with exp else exp
| find pattern in exp when exp else exp
| case exp* { arm* }             matching
| let bind* in exp               local defs
| letrec bind* in exp            rec defs
| for pattern in exp { exp } loops
| forall pattern in exp { exp } universal
| exists pattern in exp { exp } exists
| exp <- name(exp*)              async message
| exp.name(exp*)                 sync message

term   ::= Name(exp*)            data
arm    ::= pattern* -> exp       case clause

bind   ::=                       bindings
  name = exp                     variables
  name(name*) exp                functions
| component name cmp             components

qual   ::=                       qualifiers
  pattern <- exp                 selection
| ?exp                           guards

type ::=                         types
  int                            integers
| str                            strings
| bool                           booleans
| void                           nothing
| any                            anything
| [ type ]                       lists
| Name(type*)                    terms
| name                           type vars
| fun(type):type                 functions
```

```
cmp    ::=                       components
  extends exp* {                 parents
    iop*                         i/o ports
    [spec { opspec* } ]          operation specs
    [model { element* }]         data models
    [state { term* }]            local data
    [invariants { inv* }]        always hold
    [operations { op* }]         message handlers
    [rules { rule* }]            data monitors
    [init { exp* }]              initialization
    (name = exp)*                bindings
  }

iop   ::= port name[(in|out)]: inter
inter ::= interface { idef* }
idef  ::= name(name:type*):type

opspec ::= idef {
  pre pattern
  post pattern
  messages exp <- name(exp*)
}

element ::=
  class name { (name:type)* }
| assoc name { name type name type }

pattern ::=
  var                            variables
| Name(pattern*)                 term patterns
| atom                           ints,strs,bools
| name = pattern                 pattern binding
| [pattern*]                     lists
| pattern:pattern                cons pairs
| ? exp                          predicate
| {pattern}                      singleton sets
| pattern U pattern              set union

op     ::= name(arg*) { exp* }
rule   ::= name : pattern* { exp* }
```

Figure 1. LEAP language

lists Lists are used extensively in LEAP to structure data. As in Lisp, a list is constructed from a head and a tail since this facilitates recursive processing of structured data. List processing is supported by list comprehensions, for example:

`[ x + y | x <- [1,2], y <- [3,4]]` is the list `[4,5,5,6]`. Guards can be used: `[ x + y | x <- [1,2], ?  odd(x), y <- [3,4], ?even(y)]` is the list `[5]`.

functions LEAP supports higher-order functions. Functions can be dynamically created and passed as argument to other functions. In particular, a component contains a collection of operations that are named functions invoked when messages are processed.

pattern matching LEAP supports pattern matching in many language constructs. In particular there are a number of expressions such as **for** and **exists** that use pattern matching to process lists of terms. This is particularly useful when processing the list of terms that is a component's state.

execution LEAP execution occurs by passing messages. A message is synchronous or asynchronous (although we will use only asynchronous messages in this article). In addition, a component contains state monitoring rules that match patterns against the current state of a component. The rules run each time the state changes.

LEAP models execute by processing messages. Each component has a collection of input ports containing a queue of messages. Each system cycle, an input message is removed from one of the input queues for each component. The message is processed by invoking the appropriate operation defined by the component. The operation has access to the state of the component represented as a list of terms. The operation may update the state by adding or removing terms. The operation may call other operations and also produce messages that are added to the queue of named output ports on the component. Once the operation has completed, a component's rules are checked against its current state. A rule is enabled if its patterns match the state. At most one enabled rule is selected to run. The body of a rule can perform the same actions as an operation. Finally, once all components have consumed a single message, output queues are flushed by transferring messages using the connectors from output ports to input ports.

2.7 Case study

Our case study for evaluating our approach to dynamic component configuration based on EDA and CEP 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[13] 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* 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.

`http://www.jisc.ac.uk/media/documents/techwatch/jisc_ea_pilot_study.pdf`

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 the rest of the article describes how EDA can provide a basis for a flexible dynamically extensible system architecture.

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.

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.

UME performs an analysis of existing systems and processes, whether manual or automated, in order to comply with the regulations. New systems are designed in order to ensure that all possible contact points with students are recorded. The list of systems is as follows:

registry A student must register for a course before they can start studying. This in itself does not constitute a point of contact because a student must have also paid for the course before they become a UME *legal citizen*.

finance A student must pay for their studies. The UK government has recently introduced a new fee structure and students must pay 9K. Since both payment and registration must occur in any order, this constitutes an opportunity for CEP to help.

library When a student has registered and paid for their studies, they may use the library facilities. Activities such as borrowing and returning a book constitute a point of contact. After the proposed system has been implemented, UME notices that it could also be used to detect problem students who have not presented sufficient points of contact *and* who have a fine outstanding in the library. This situation requires a system reconfiguration and provides an opportunity for us to show EDA with higher-order functions and reflection can support dynamic architecture reconfiguration without a static rebuild.

department Each course of study includes coursework with associated deadlines. When a student hands in coursework this constitutes a point of contact. After installation of the proposed system UME notices that students who miss coursework deadlines could also be monitored by the system. Again, this presents an opportunity to show how EDA with higher-order functions can support dynamic reconfiguration.

monitor The proposed system includes a new IT system that collects points of contact for students. The monitor will also provide support for registering a student as a *problem citizen.*

3 Problem and Contribution

Most languages for component architectures and EA modelling are based on SOA which leads to a system model that can be checked statically before deployment. However this makes dynamic reconfiguration difficult since typically the system must be taken off-line, reconfigured statically, and then re-deployed.

EDA based systems provide a more flexible approach since they reduce multiple messages into a single type of event by reifying the message name and arguments as data. In principle, this allows dynamic reconfiguration because there is essentially only a single message data type: `Event`.

However, the introduction of new events requires two additional extensions: the conditions under which the event is raised and the behaviour that is used to handle the event when it is received. Our approach to address this issue is to propose that EDA based systems should include higher-order functions. This addition allows handlers to be registered with components for raising new types of event and for handling them. Higher-order functions have been shown elsewhere to support a wide diversity of dynamic execution patterns, and as such can encode more structured approaches such as mixins.

The addition of higher-order functions to EDA raises a question of how to extend a third party component that provides an extension interface but does not expose its internal data representations. Our proposal is to introduce data reflection so that a component can be requested to produce a meta-representation of its internal state. New event producer and consumer functions can be defined to work on this representation without further knowledge of the internals of the component they extend.

Finally, we acknowledge that the EDA approach forces components to be dynamically typed since messages are reduced to a single data type: `Event`. This can be criticised as undesirable since a system cannot be statically determined to be type safe. Our approach allows both SOA and EDA to co-exist and we envisage it being deployed in a way that allows short-term EDA-based dynamic reconfigurations to be assimilated into statically checked SOA-based messages at regular medium-term system upgrades.

This section describes our overall approach and includes the key features of our contribution. Section 3.1 shows how SOA-based message passing can be specified in LEAP and how pre and post-conditions can also be used to specify events. Section 3.2 shows how LEAP supports CEP. Section 3.3 shows how our proposal for dynamic reconfiguration using higher-order functions is implemented in LEAP. Section 3.4 shows how reflection works in LEAP and its relationship to dynamic reconfiguration.

3.1 Specification and messages

Both SOA and EDA based systems rely on messages being sent between components using named ports. LEAP directly supports components, ports and connectors for this purpose. Operations that are defined by an LEAP component `c` are invoked when a corresponding message is consumed from the head of an input queue owned by `c`. The operation is performed and may result in a number of messages being sent to output ports defined by `c`. Where an output port is connected to an input port, messages are transferred and the output ordering is preserved.

Modelling component behaviour is supported by operation specifications defined in terms of pre and post conditions. The pre-condition and post-condition are both boolean expressions defined in terms of the component's state. A pre-condition defines a predicate over the state of the component before the operation is performed and the post-condition defines a predicate over the state both before *and* after the operation has been performed.

The state of an LEAP component is a list of terms **`state`** whose types are given by models in the component definition. Additionally, in the post-condition, **`state@pre`** is used to refer to the state before the operation completes. Therefore, it is possible to use pre- and post-conditions to define a requirement for a particular state change when an operation is performed without having to express the mechanism by which the change happens.

In addition, an operation may produce messages, therefore an operation specification includes a clause that defines a predicate over the component's output ports in terms of the messages that have been produced.

The standard language for modelling operation specifications is the Object Constraint Language (OCL) defined as part of UML. OCL is used to specify class operations and consequently can define predicates over the state of instances of the class and those instances that can be reached by traversing links from the object. OCL also includes a mechanism for expressing predicates over messages sent and received by an object[15]. The language for message specification is more expressive than that in LEAP, however they address different aims which makes LEAP more appropriate to SOA and EDA. OCL is class-based and LEAP is component-based. UML classes have a single input and output port whereas, SOA requires multiple named ports as provided by LEAP. A UML class corresponds to a database table and its instances correspond to rows in that table, therefore OCL expresses conditions in terms of rows (although `allInstances` can be used to range over all rows in a table). LEAP components define models containing multiple classes and a component's **`state`** contains all the contents of all tables making it easier to express predicates that range over multiple tables.

In EDA each component produces and consumes events, so for example when a student registers for a course this event is made available to all components that are listening for it. Each operation must include along with the specification of any required state change, the conditions under which the `registers` event is raised. The following example shows how specifications and implementations are defined separately in LEAP:

```
 1  component table {
 2    model {
 3      class X { id:int; ys:[Y] }
 4      class Y { id:int }
 5    }
 6    port messages[in]: interface { register(x_id:int,y_id:int):void }
 7    port events[out]: interface { registered(x_id:int,y_id:int):void }
 8    spec {
 9      register(x_id:int,y_id:int):void {
10        pre ?not(exists X(x_id,{Y(y_id)} U _))
11        post ?exists X(x_id,{Y(y_id)} U _)
12        messages events <- registered(x_id,y_id)
13      }
14    }
15    operations {
16      register(x_id,y_id) {
17        find X(x_id,ys) {
18          find Y(y_id) in ys { } else {
19            replace X(x_id,ys) with X(x_id,Y(y_id):ys) else {};
20            events <- registered(x_id,y_id)
21          }
22        } else {
23            new X(x_id,[Y(y_id)]);
24            events <- registered(x_id,y_id)
25        }
26      }
27    }
28  }
```

The component `table` is used to manage associations between `X`'s and `Y`'s. Each `X` is associated with several `Y's` and they both have identifiers (lines 2-5). Messages to register a `Y` against an `X` are received as `register(x_id,y_id)` (line 6). An event `registered` is generated when a successful registration is completed (line 7).

The specification for registration is given in lines 8-14. This is defined independently of the implementation and a component may contain one or the other, or both. The pre-condition (line 10) uses pattern matching to check that no association exists. The pattern `X(x_id,p)` matches any `X` in the state with the given identifier and whose list of `Y`'s matches the pattern `p`.

A set pattern {`p`} **U** `q` is a useful device to match any element of a list against `p` such that the rest of the list matches `q`. In the case of line 10 this is used to check to see that there is no `X` that already contains a `Y` with the supplied identifier (the pattern _ means *don't care*).

The post-condition (line 11) requires the registration to have taken place and line 12 requires the registration event to have been produced on the output port.

The implementation for `register` is given in lines 16-26. In this case the **find** expression is used twice, the first time to extract an `X` with the supplied identifier and the second time to extract a `Y`. In both cases suitable actions are taken if the **find** fails to locate a matching term.

3.2 Complex event processing

SOA-based architectures cannot easily support CEP that involves detecting patterns in multiple events that can occur at different times. Figure 2 shows the simplest CEP architecture. Components `a` and `b` both produce independent events that must be processed by `merge`. In general, an event contains arbitrarily complex data and `merge` must detect corresponding pairs of events. In the simple example, each event contains an identifier and `merge` must match events with corresponding identifiers. The LEAP definition of `a` is as follows (`b` is an equivalent definition):

```
component a {
  port events[out]: interface { a(id:int):void }
}
```

The definition above declares that `a` has a single output port named `events` that produces messages of the form `a(i)` where `i` is an integer.

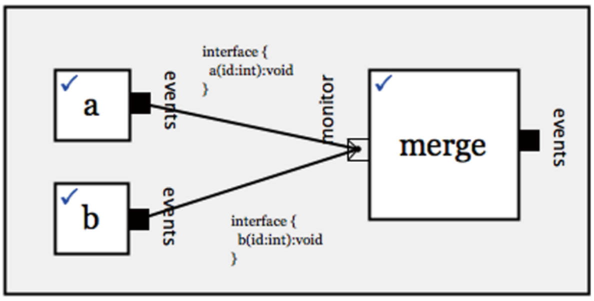


Figure 2. Complex event processing

An SOA approach to the definition of `merge` will require an internal database that manages all the incoming messages. When a new message is received, the database must be searched for a previous message whose data matches. Since, in general, an activity may depend on any number of incoming messages this can lead to a significant combinatorial overhead.

CEP abstracts from the housekeeping details necessary to manage the incoming messages within `merge` and uses declarative rules over a simple internal database as shown in the following LEAP definition:

```
 1   model {
 2     class A { a:int }
 3     class B { b:int }
 4   }
 5   port monitor[in]: interface {
 6     a(id:int):void;
 7     b(id:int):void
 8   }
 9   port events[out]: interface { merged(id:int):void }
10   operations {
11     a(id) { new A(id) }
12     b(id) { new B(id) }
13   }
14   rules {
15     merge: A(id) B(id) {
16       events <- merged(id)
17     }
18   }
19 }
```

The database is defined in lines 2-5 and consists of two types of term. Line 3 defines a class of terms of the form `A(i)` and line 4 defines a class of terms of the form

`B(i)`. The port named `monitor` is declared to process messages named `a` and `b`, and the port `events` is declared to produce messages of the form `merged(i)`.

The operations on lines 11 and 12 are invoked each time the corresponding messages are received at the `monitor` input port. They simply update the database with a new term of the appropriate type.

The rule defined on line 15 is called `merge` and has a pattern on line 15 that matches against the current state of the database. The repeated use of the variable `id` in the rule must be consistently matched against terms in the database. Therefore, for the rule to be satisfied, it must match against two terms of different types and which agree on the identifier value. The rule will be matched against all possible matching pairs, but any given pair will only match once. The body of the rule on line 16 sends a `merged` message to the `events` port.

The definition of the `merge` component above uses rules to abstract away from the housekeeping details that would otherwise be necessary if an SOA approach was used to match corresponding input messages.

3.3 Dynamic reconfiguration

SOA based architectures are based on ports with pre-defined interfaces describing the messages produced and consumed at each port. When connectors are used to associate an output port with an input port, the interfaces must match otherwise the system runs the risk of a run-time error where a message cannot be associated with an operation that handles it.

This approach makes a system type-safe and is appropriate for system architectures where the functionality is known in advance. However, there may be occasions where system functionality must be extended. In an SOA based approach, this requires the system to be shut down, new interfaces defined and the system to be rebuilt and redeployed. Such a process will guarantee type safety, but may not always be possible. For example, not all components in a system may be under local control in which case they cannot be shut down. Furthermore, locally controlled components may be relied on by external components and shutting them down may not be an option.

In such situations EDA is a more attractive approach because the interfaces used to produce and consume messages are fixed and new types of message can easily be introduced because message types are represented in data. Having introduced a new message type, the operation that handles it by a component must be provided. Again, an SOA based approach would require static system reconfiguration offline. However an EDA approach *together with* higher-order functions allows dynamic reconfiguration by registering event handlers via messages.

Consider the case where the UME Library is modelled as a component. Using SOA we might fix the Library interface so that it produces a message for student contact each time a student borrows and returns a book. However, at some later stage we might introduce library fines and we are interested in detecting when students have outstanding fines because this is an indication, together with a low contact point score, of a problem citizen. An EDA approach with higher-order functions will allow us to dynamically add in the fine event and to add a handler to the monitoring system without shutting either system down. At some later stage when it is safe to take the systems off-line, the events can be translated to SOA-style messages and operations.

Event processing is performed using a model of events. LEAP allows one component to extend others, the extension relation occurs at definition time and includes a copy of the parent into the child. The following component is used as a simple basic for modelling events:

```
component event_model {
  model {
    class Event {
      name:str;
      args:[any]
    }
  }
}
```

The following component `event_consumer` defines how a simple dynamically configurable event processing component can be defined in LEAP:

```
 1  component event_consumer extends event_model {
 2    model {
 3      class EventHandler {
 4        name:str;
 5        arity:int;
 6        handler:fun(Event(str,[any])):any
 7      }
 8    }
 9    port monitor[in]: interface { raise(event:Event(str,[any])):void }
10    port handlers[in]: interface {
11      define_handler(name:str,arity:int,handler:fun(Event(str,[any])):any):void
12    }
13    operations {
14      raise(event) {
15        case event {
16          Event(name,args) ->
17            find EventHandler(name,arity,handler) when arity = #(args) {
18              handler(event)
19            } else print(’no handler defined: ’ + state)
20        }
21      }
22      define_handler(name,arity,handler) {
23        new EventHandler(name,arity,handler)
24      }
25    }
26  }
```

An event handler (lines 3-8) contains the name of the event, the number of arguments and a handler. The type of the handler (line 6) is a function that maps an event to an LEAP value.

The event processor defines two ports. The port `monitor` (line 9) is used to receive incoming events. In order to implement the EDA architecture, all participating components must implement either an output port for messages of type `raise(Event(str,[any]))` if they are an event producer, or an input port for messages of the same type if they are an event consumer. Of course a component may be both an event producer and consumer. In order to monitor an event producer, an event consumer connects to the appropriate output port.

The port named `handlers` (lines 10-12) provides a mechanism to dynamically register handlers with the event consumer. The `define_handler` message will create a new handler (line 23).

The `raise` operation (lines 14-21) processes an incoming event. Pattern matching is used to find an event handler with the appropriate name and arity (line 17). Once found, the handler is invoked by applying it to the event (line 18). The reaction to a missing handler will depend on the application and architectural design issues. It may be ignored or there may be a reply mechanism contained in the event. In this case we print a message (line 19).

3.4 Reflection

The previous section has described how dynamic reconfiguration of event consumption can be supported if the architectural framework supports higher-order functions and implements an EDA approach to architecture. This section describes how reflection can be used to support dynamic reconfiguration of event production.

Each component includes a private state whose type is defined by a model. In general, the data maintained in the state will have a variety of implementation formats making it very difficult to introduce a monitor that generates a previous unforeseen event based on a predicate over the hidden component state.

Although the components may have hidden private data formats, it is possible to translate the data into a single value model that incorporates type information. This process is called *reflection* and provides a universal format for processing data. In the case of deployed systems the universal format is likely to be encoded as XML.

Reflection can be used to inject data monitors into components that conform to the EDA architectural style so that a component can produce events in unforeseen circumstances. For example, it may become desirable for the Library component to raise events when a fine is imposed on a reader. In an SOA environment this will involve static reconfiguration. Reflection translates the Library state into a universal format and higher-order functions can be used to inject a handler that processes data values expressed in this format.

Each LEAP component contains a state that is implemented as a list of terms. The type of each term is given by a class defined in a model owned by the component. The model is private to the component and therefore the data cannot be processed by another component unless it shares the same model.

Although data types are private to LEAP models, LEAP provides a reflection operator `lift` (and a corresponding inverse called `intern`) that can be used to translate data values into a single universal format whose model is shared by all components. For example, a component might define a class called `Student` that allows it to manipulate terms of the form `Student('fred')`. The reflection operator transforms the term as follows: `lift(Student('fred')) = Term('Student', [Str('fred')])`.

The following component `event_producer` defines how state can be monitored using reflection:

```
component event_producer extends event_model {
  model {
    class Monitor {
      handler:fun([Term(str,[any])]):[Event(str,[any])]
```

```
 5      }
 6    }
 7    port monitors[in]: interface {
 8      define_monitor(handler:fun([Term(str,[any])]):[Event(str,[any])]):void
 9    }
10    port events[out]: interface {
11      raise(event:Event(str,[any])):void
12    }
13    rules {
14      change: ok {
15        process_change()
16      }
17    }
18    operations {
19      define_monitor(handler) { new Monitor(handler) }
20      process_change() {
21        let terms = [ lift(t) | t <- state ]
22        in for Monitor(m) in state {
23             for event in m(terms) {
24               self.events <- raise(event)
25             }
26           }
27      }
28    }
29  }
```

The component above defines a class called `Monitor` that is used to manage data monitors. A monitor is a handler function that maps a sequence of terms (in universal format) to a sequence of events. The idea of a monitor is that it receives the state of a component, as reflected terms, when the state changes. The monitor produces a sequence of events based on the changes. For example, if a borrowing has been flagged as incurring a fine then the monitor should produce an appropriate event.

A new monitor is added using the `monitors` port (lines 7-9) and an event is produced using the `events` port (lines 10-12).

Rules are used to monitor state changes in an LEAP component. A rule contains a collection of patterns, if the patterns match the current component state then the rule is enabled. Each execution cycle selects a rule that is ready to fire and executes the body of the rule.

Lines 14-16 show a very simple rule. The pattern is **ok** which always matches, and therefore the rule is enabled for any state change. The body of the rule calls the local `process_change` operation.

Each update to the component is processed by `process_change` defined on lines 22-29. The reflection operator guarantees that the data supplied to the monitors is in a format that the monitor function can process. Each monitor is applied to the reflected terms (line 25) and the resulting events are raised (line 26).

4 Modelling with EDA

Our proposal is that architectural modelling is usually based on an SOA based approach and as such message interfaces are static. This leads to difficulties when an architecture is to be dynamically reconfigured. An EDA approach is much more flexible because it provides a universal communication mechanism for components to produce and monitor events. This is achieved at the expense of a loss of static interface checking, however the two approaches are not incompatible and it is possible to view EDA as being used to guarantee dynamic reconfiguration where updates are translated to SOA on each major system upgrade. This is equivalent to the ability of certain programming languages to accommodate dynamic changes via an interpreter and where code is incrementally compiled, for example Eiffel's *Melting Ice* technology[25].

This section shows how the UME case study can be modelled in LEAP using EDA. The models are not complete, but they exhibit the key features of the approach. Section 4.1 shows the component architecture from two perspectives. Firstly, components are categorized into event producers and consumers and then the components are connected in order to monitor events.

Section 4.2 defines the components. Our intention is to show a range of LEAP features including specification, implementation, SOA and EDA and therefore the component definitions use a mixture of approaches.

Section 4.3 shows how the EDA mechanisms can be used to achieve dynamic reconfiguration.

4.1 Architecture

Figure 3 shows an LEAP inheritance hierarchy for the case study components. Each component is either an event producer or an event consumer or both. Where the behaviour of a component is known (for example `clock`) it is appropriate to fix its behaviour as a producer or consumer, however there is no reason why, in order to future-proof the system, each component should not inherit from both `event_producer` and `event_consumer`.

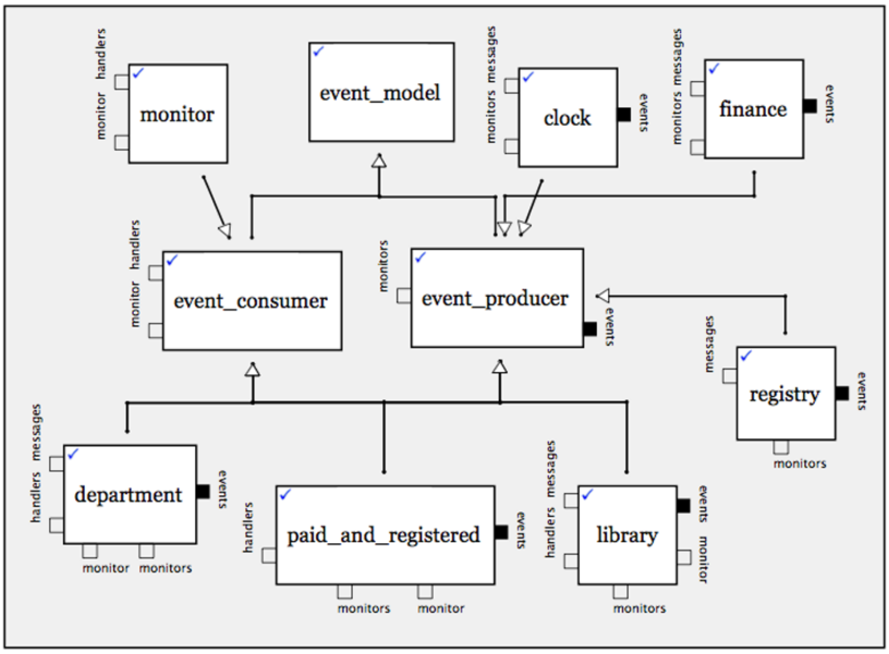


Figure 3. Component inheritance

Figure 4 shows how components are connected in order to monitor events. Each component displays a collection of ports as boxes: a white box is an input port and a black box is an output port. The ports are named (in this case the names are inherited from either `event_producer` or `event_consumer`). In an EDA architecture, components have ports for defining monitors and handlers, a single port called `events` for producing events, and a single port called `monitor` for receiving events.

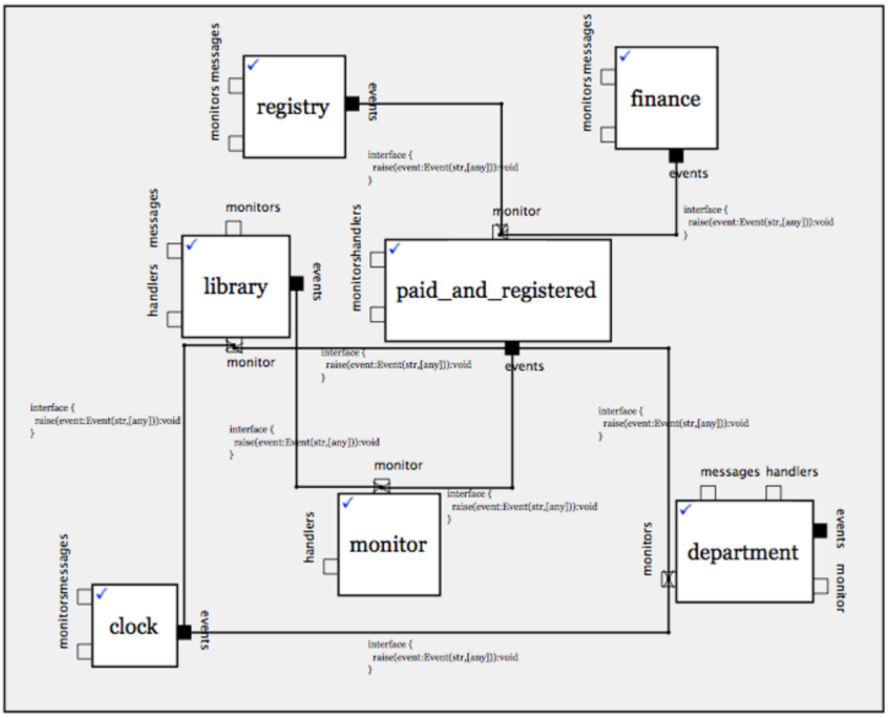


Figure 4. Event driven architecture

As shown in the diagram, the `monitor` component receives and merges events. The `paid_and_registered` component is used to perform CEP in terms of events from `registry` and `finance`. The `clock` is used to inform both the `library` and the `department` of the current time in order that the library can import fines and the department can manage coursework deadlines.

4.2 Components

The `clock` component provides an example of a simple event producer:

```
component clock extends event_producer {
  model {
    class Time { time:int }
  }
  port messages[in]: interface { tick():void }
  spec {
    tick():void {
      pre Time(t0)
      post Time(t1) ?(t1=t0+1)
      messages events <- raise(Event('time',[t1]))
    }
12    }
13    operations {
14      tick() { replace Time(n) with Time(n+1) else new Time(1) }
15    }
16    init {
17      self.monitors <- define_monitor(
18        fun(events)
19          find Term('Time',[Int(t)]) in events {
20            [Event('time',[t])] }
21          else [])
22    }
23  }
```

The message `tick` is provided via a static SOA based interface. Lines 6-12 provide an example of how LEAP operation specifications can be used to require an event is raised as a result of a state change. The event is raised by a monitor that is registered in the **init** block in lines 16-22. Since the only change that can happen to a clock is that time advances, the handler function (lines 18-21) expects to match a term pattern and extract the time `t` and to produce a single event `Event('time',[t])`.

Of course, using a state monitor as shown above is not the only way in which events can be produced. It is possible to generate them directly as shown in the definition of `registry` below:

```
 1  component registry extends event_producer {
 2    model { class Student { name:str; course:str } }
 3    port messages[in]: interface { register(s:str,c:str):void }
 4    spec {
 5      register(s:str,c:str):void {
 6        pre Student(s,_)
 7      }
 8      register(s:str,c:str):void {
 9        pre not(Student(s,_))
10        post Student(s,c)
11        messages events <- raise(Event('registered',[s,c]))
12      }
13    }
14    operations {
15      register(s,c) {
16        find Student(s,_) { }
17        else {
18          new Student(s,c);
19          events <- raise(Event('registered',[s,c]))
20        }
21      }
22    }
23  }
```

Lines 4-13 show how an operation specification is divided into multiple clauses with different pre-conditions. In the first case (lines 5-7) a student already exists and does not need to be registered twice. In the second case (lines 8-12) the student does not exist and should be added to the database and an appropriate event is raised.

The definition of `register` shows how an `event_producer` can simply generate events directly without registering a handler. Of course this is less flexible since a handler could be removed whereas the event raised by `register` is hard-wired into the component.

The `finance` component also hard-wires event production. It is interesting because it does not directly translate each state change to an event since a student may pay the fees in instalments. The event will only be generated when the student has paid at least `9K`:

```
component finance extends event_producer {
  tuition_fee = 9000
  model {
    class Student { name:str; fee:int }
  }
  port messages[in]: interface { pay(s:str,i:int):void }
  operations {
    pay(s,i) {
      find Student(s,n) when n = tuition_fee {
        error(’student already paid: ’ + s)
      } else find Student(s,n) when n < tuition_fee {
          if (n + i) >= tuition_fee
          then payment_complete(s)
          else replace Student(s,n) with Student(s,n+i) else new Student(s,n+i)
      } else if i >= tuition_fee
        then payment_complete(s)
        else new Student(s,i)
    }
    payment_complete(s) {
      replace Student(s,n) with Student(s,tuition_fee) else new Student(n,tuition_fee);
      events <- raise(Event(’paid’,[s]))
    }
  }
}
```

When both `registry` and `finance` produce events, the student can start to use the facilities of UME. The detection of multiple unordered events constitutes CEP and is provided by `paid_and_registered`:

```
 1 component paid_and_registered extends event_consumer,
 2 event_producer {
 3   model {
 4     class Registered { name:str; course:str }
 5     class Paid { name:str }
 6   }
 7   rules {
 8     ready: Paid(s) Registered(s,c) {
 9       events <- raise(Event(’contact’,[s]));
10       events <- raise(Event(’registered’,[s,c]));
11       events <- raise(Event(’paid’,[s]))
12     }
13   }
14   init {
15     self.handlers <- define_handler(’registered’,2,fun(e) case e {
16       Event(_,[s_name,c_name]) -> new Registered(s_name,c_name)
17     });
18     self.handlers <- define_handler(’paid’,1,fun(e) case e {
19       Event(_,[s_name]) -> new Paid(s_name)
20     })
21   }
22 }
```

Event processing is provided by two handlers defined in lines 14-16 and 17-19 respectively. In the first case the handler detects an event of the form `Event` `('registered', [student,course])` and in the second case an event of the form `Event('paid', [student])`. In both cases the information is added to the local database where the rule `ready` (lines 7-11) detects the changes. The rule can only become enabled when both patterns match. When it fires the appropriate events are produced.

The `library` component provides an external interface that records borrowings (we omit the other library functions). In order to use the library, a student must have paid their fees. In addition, the library monitors the time via clock events in order to process fines. We leave the implementation of fine processing until section 4.3 as an example of dynamic reconfiguration:

```
component library extends event_consumer, event_producer {
  model {
    class Reader { name:str; borrows:[Borrow] }
    class Borrow { book:str; time:int }
    class Time { time:int }
  }
  port messages[in]: interface {
    borrow(r_name:str,b_name:str):void
  }
  operations {
    borrow(r_name,b_name) {
      find r=Reader(r_name,bs) {
        replace r with Reader(r_name,Borrow(b_name,time()):bs);
        events <- raise(Event('contact',[r_name]))
      } else error('no reader: ' + r_name)
    }
    time() {
      find Time(n) { n } else 0
    }
  }
  init {
    self.handlers <- define_handler('paid',1,fun(e) case e {
      Event(_,[s_name]) -> new Reader(s_name,[])
    });
    self.handlers <- define_handler('time',1,fun(e) case e {
      Event(_,[t]) -> replace Time(_) with Time(t) else new Time(t)
    })
  }
}
```

A department maintains information on its courses, the students registered for the courses, the modules on each course and the coursework deadlines for each module. When a student completes a coursework it is registered so that it is possible to determine which student has coursework outstanding.

LEAP provides a diagram format for component data models as shown in Fig.5. The component is defined below:

```
component department extends event_consumer, event_producer {
  model {
    class Course { name:str; students:[Student]; modules:[Module] }
    class Student { name:str; submitted:[Coursework] }
    class Module { name:str; courseworks:[Coursework] }
    class Coursework { name:str; deadline:int }
  }
  port messages[in]: interface { submit(name:str,cw:str):void }
```

```
 9    operations {
10      registered(name,course) {
11        find c=Course(course,students,modules) {
12          find Student(name,_) in students {
13            error('cannot register twice: ' + name)
14          } else replace c with Course(course,Student(name,[]):students,modules)
15        } else error('no course: ' + course)
16      }
17      submit(s_name,cw_name) {
18        find c=Course(c_name,
19                {Student(s_name,submitted)} U students,
20                {Module(m_name,{cw=Coursework(cw_name,deadline)} U courseworks)} U modules) {
21          replace c with Course(c_name,Student(s_name,cw:submitted):students,c.modules)
22          else error('cannot replace');
23          events <- raise(Event('contact',[s_name]))
24        } else error('cannot submit')
25      }
26    }
27    init {
28      handlers <- define_handler('registered',2,fun(e) case e {
29        Event(_,[s_name,c_name]) -> registered(s_name,c_name)
30        else {}
31      })
32    }
33  }
```

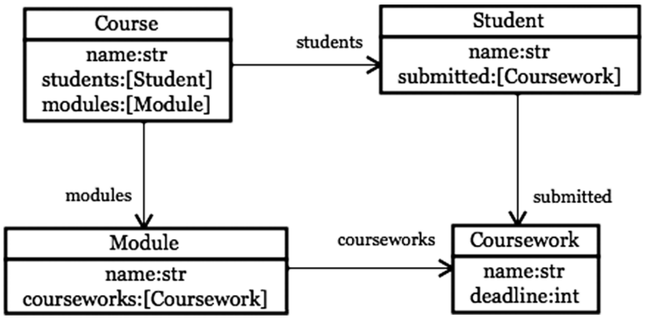


Figure 5. Department model

The `registered` event is monitored by a handler (lines 28-31) and records the new student. The external interface called `messages` provides an operation for submitting coursework. The pattern matching in lines 18-20 shows how the appropriate student and coursework is selected from the course which is replaced with an updated term (line 21).

The final component is the `monitor` that is used to collect the `contact` events that are raised by the other components. The monitor component manages a count of the contact points as required by the UKBA regulations. When a student does not collect sufficient contact points by a given time they are considered to be a `problem`:

```
component monitor extends event_consumer {
  model {
    class Student { name:str; contacts:int; is_problem:bool }
  }
  init {
    self.handlers <- define_handler('contact',1,fun(e) case e {
      Event(_,[s_name]) ->
        replace Student(s_name,n,p)
        with Student(s_name,n+1,p)
        else new Student(s_name,1,false)
      else {}
    })
  }
}
```

4.3 Dynamic reconfiguration

Our claim is that EDA and higher-order and reflective features support the dynamic reconfiguration of system architectures. In particular it allows new handlers and event generators to be added without having to have access to the hidden type information within a component.

This section shows how dynamic configuration can be achieved for the UME case study. The first step is to extend the `monitor` component with a handler for problem student events. This allows new events to be received and to update the local database (we shall assume knowledge of the data type in this extension and relax this assumption in the following examples):

```
monitor.handlers <- define_handler('problem',1,fun(e) case e {
  Event(_,[s_name]) ->
    replace Student(s_name,n,_)
    with Student(s_name,n,true)
    else new Student(s_name,1,true)
  })
```

Now consider the library component. Assuming that we cannot know the exact representation of data within the library, but since monitors use a general purpose *reflected* representation for data, the monitor can be added by sending the `library` component a message:

```
1  library.monitors <- define_monitor(fun(terms)
2    find Term('Time',[Int(t)]) in terms {
3      find Term('Reader',[Str(r),Cons(Term('Borrow',[_,Int(t')]),_) ]) in terms
4      when t > t' + 2 {
5        [Event('problem',[r])]
6      } else []
7    } else [])
```

In the example above, we assume that fines are imposed after 2 time units have elapsed (perhaps these are weeks) and that while there is a student with an outstanding fine, the `problem` events will continue to be generated.

Note the pattern matching over the terms in lines 2 and 3 above. This shows how the `lift` operator has produced a representation for terms, integers, lists and strings. In particular, an LEAP list is reflected as follows:

```
lift([1,2,3]) = Cons(Int(1),Cons(Int(2),Cons(Int(3),Nil)))
```

A more extensive example involves extending the `department` component to produce events when a coursework deadline has been missed. In this case the reflected data representation must be processed using pattern matching by a collection of auxiliary operations:

```
department.monitors <- define_monitor(fun(terms)
  find Term('Time',[Int(t)]) in terms {
    for Term('Course',[_,students,modules]) in terms {
      process_modules(t,modules,students)
    }
  } else {})

process_modules(time,modules,students) {
  case module {
    Cons(Term('Module',[_,courseworks]),tail) -> {
      process_courseworks(time,courseworks,students);
      process_modules(time,tail,students)
    }
  }
}
process_courseworks(time,courseworks,students) {
  case courseworks {
    Cons(c=Term('Coursework',[_,Int(deadline)]),tail) -> {
      if deadline > time
      then process_students(c,students);
      process_courseworks(time,tail,students)
    }
  }
}
process_students(coursework,students) {
  case students {
    Cons(Term('Student',[Str(name),completed]),tail) -> {
      if not(member(coursework,completed))
      then events <- raise(Event('problem',[name]));
      process_students(tail)
    }
  }
}
member(element,list) {
  case list {
    Cons(element,_) -> true;
    Cons(_,rest)   -> member(element,rest)
    else false
  }
}
```

5 Displays

We have shown how reflection can be used to support dynamic reconfiguration of components and to allow hidden data types to be externalized. In addition, reflection can be used to provide a generic way of displaying the state of a component. The basic idea is to define a mapping from reflected data to a display model. LEAP defines a simple display model shown in Fig. 6 when a screen is sent as a message to a GUI component: `gui.in <- display(screen)`, the screen is displayed. Button terms use functions as call-backs where the functions produce new screens.

This section shows how a simple `displayable` component can be defined that uses reflection to convert the state of any component into a screen. The `displayable` component can be used as the parent of any LEAP component. We will then show a series of screen shots showing the a simulation of the case study that have been generated by an extended version of `displayable`.

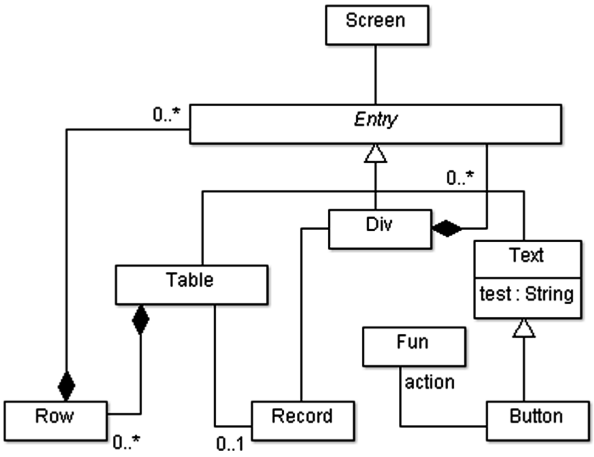


Figure 6. Screen model

Figure 7 shows a model that is used to encode a collection of terms as a table. The state of a component is a list of terms in any order. We would like to display the state as a collection of tables corresponding to the classes in the component's model. In order to do this we must gather terms of the same type together and to deal with references between tables.

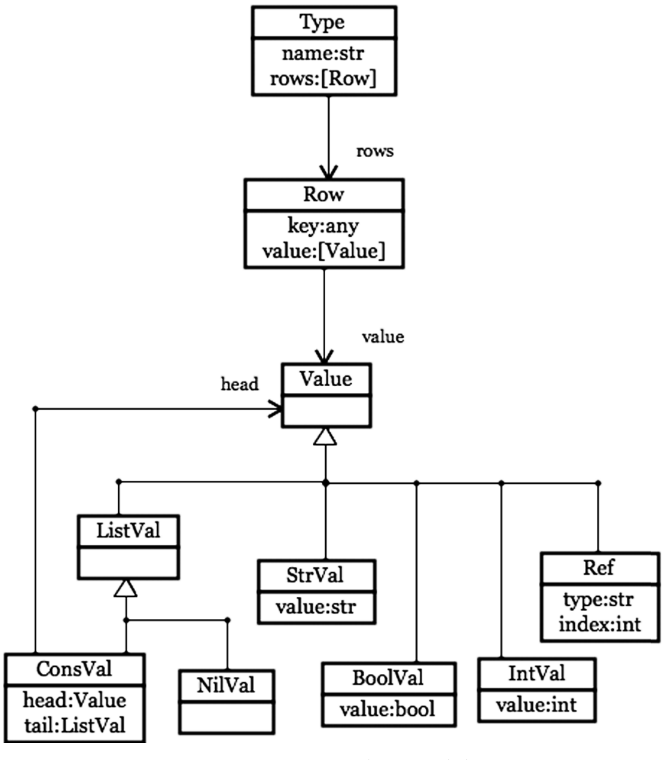


Figure 7. Display model

Consider the state `[A(1,B(2)),A(2,B(1))]`. This will be represented as a sequence of types:

```
[Type('A',[Row(k1,[IntVal(1),Ref('B',0)]),Row(k2,[IntVal(2),Ref('B',2)])]),
 Type('B',[Row(k3,[IntVal(2)]),Row(k4,[IntVal(1)])])]
```

Each type contains a list of rows where each row is an instance of the type. A row has a unique key (for example `k1`) and a sequence of values. A value is either atomic, a list, or a reference to another instance. A reference `Ref(type,index)` indexes a row in a type.

Given such a table of types, we can produce a screen that is suitable for displaying on an LEAP GUI:

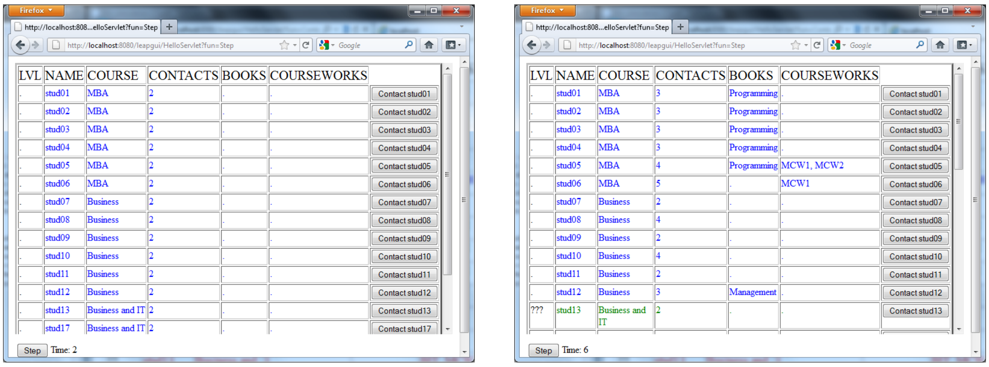
```
Screen(Table([
  [Text('A'),Text('1'),Text('B(0)')],
  [Text('A'),Text('2'),Text('B(1)')],
  [Text('B'),Text('2')],
  [Text('B'),Text('1')]
]))
```

The component `displayable` provides a general purpose mechanism for translating a component state to a screen as shown above. It is partially defined below:

```
component displayable {
  operations {
    display_state(gui,terms) {
      values_to_table([lift(t) | t <- terms],[],fun(value,tables) {
        gui.in <- display(Screen(tables_to_entry(tables)))
      })
    }
  }
}
```

It provides an operation `display_state` that can be called by any component that extends `displayable`. The `display_state` operation expects two arguments `gui` a component to send a `display` message to, and the terms to be displayed. Each term is reflected using `lift` and is supplied to an operation `values_to_table` that maps reflected terms to a table and is defined in appendix A along with `tables_to_entry` that maps tables to a screen entry.

The `displayable` component is very simple and just displays a component as a simple table of its contents. However, the reflection mechanism and higher-order functions allow a more general version of `displayable` to be defined that allows display handlers to be dynamically added. Such an extension can be used to define an extension to `monitor` for the UME case study leading to the simulation snapshots shown in Fig. 8; each snapshot is a browser screen-shot generated after several clicks of the `Step` button. Figure 8(a) shows the situation just after students have registered for their courses and have eagerly used the library. Figure 8(b) occurs a little later and shows that student `stud13` has missed a coursework deadline. Figure 8(c) 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 8(d) 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.


Firefox
http://localhost:808...elloServlet?fun=Step
http://localhost:8080/leapgui/HelloServlet?fun=Step
Google
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

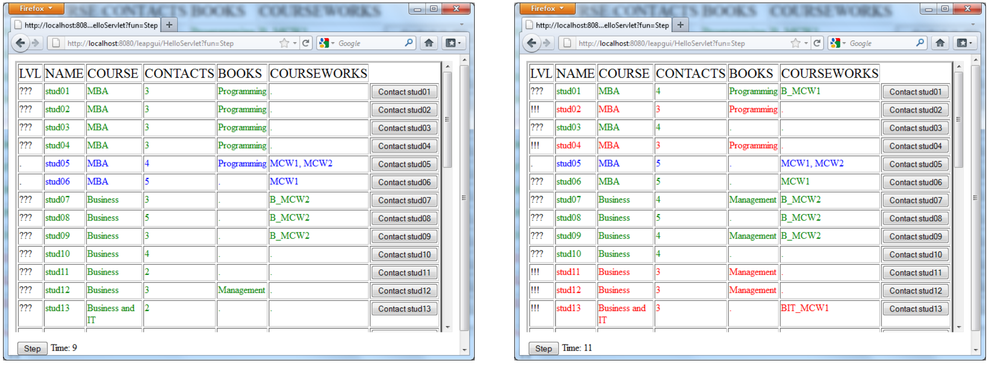
Time: 2

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

Time: 6

(a) Student registration

(b) Missing coursework


Firefox
http://localhost:808...elloServlet?fun=Step
http://localhost:8080/leapgui/HelloServlet?fun=Step
Google
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

Time: 9

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

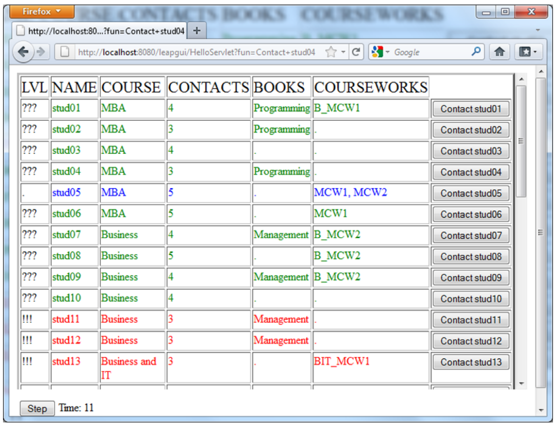
Time: 11

(c) Warning signs

(d) Business goal in Jeopardy

Figure 8. Simulation

The buttons on the right hand side of the screen allow the administrator to override students who are flagged as having insufficient contacts. Figure 9 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).


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

Time: 11

Figure 9. Manual override

6 Conclusion

We have argued that an SOA-based approach to system architecture provides useful static checking but that this benefit makes dynamic extensibility of component behaviour difficult or impossible. An EDA-based approach to system architecture has several benefits including CEP and can support dynamic reconfiguration of an architecture topology because the interfaces are all reduced to a single type of item: `Event`. However, EDA and CEP alone cannot easily support dynamic extension of component behaviour. We have shown that higher-order functions and reflection together with EDA and CEP can achieve this.

Such an approach gains extensibility, but loses static type verification of message interfaces supported by SOA. This article has not provided a solution to this problem, however it would be possible to use reflective capabilities to check that dynamically allocated monitors and handlers matched the events and data items available at the point of registration.

In addition the extra computational overhead of pattern matching events may make this approach unsuitable for real-time systems. However, we would expect that such an overhead would be minimal compared to the time taken to process messages which is inherent in a distributed SOA-based system.

We have validated our approach using the LEAP executable modelling language and shown that it works for a small-scale architecture. Real-time issues notwithstanding, we see no reason why this approach would not scale effectively. An interesting future direction is to show how LEAP models can be deployed as systems on industrial scale component-based technology platforms.

The LEAP language supports both SOA (through statically typed ports and connectors) and EDA (through rules and higher-order functions). As described in Ref. [20] there are advantages and disadvantages of both these approaches; indeed Ref. [20] claims that there a numerous benefits of having an architecture that supports the coexistence of operations and events. Of course, neither approach is without its flaws; those of SOA have been described in the introduction, and EDA can suffer from performance and reliability problems. An approach that incorporates both, could represent the best of both worlds especially if there is a migration route from one to the other. This is an area for further work.

We have demonstrated that our approach can dynamically reconfigure services, handlers and events. Our proposal is that higher-order functions (and components) offer a rich language for the design and simulation of applications that require reconfiguration. A higher-order approach support a wide variety of structural and behavioural patterns in program-based systems, including mixins[41], which gives us confidence that other approaches to reconfiguration can be encoded using configurations of functions and components.

However, we have yet to investigate aspects such as consistency, availability, co-existence and quality of service (QoS) as described in Ref. [23]. This is an area for future work.

References

[1] Agrawal J, Diao Y, Gyllstrom D, Immerman N. Efficient pattern matching over event streams. Proc. of the 2008 ACM SIGMOD International Conference on Management of Data. ACM.

2008. 147–160.

[2] Assmann M, Engels G. Transition to service-oriented enterprise architecture. Software Architecture. 2008. 346–349.

[3] Barn B, Clark T. Goal based alignment of enterprise architectures. In: Hammoudi S, van Sinderen M, Cordeiro J, eds. ICSOFT. SciTePress. 2012. 230–236.

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

[5] Batista T, Joolia A, Coulson G. Managing dynamic reconfiguration in component-based systems. In: EWSA 2005 2 nd European Workshop on Software Architectures. Springer, 2005. 1–17.

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

[7] Buchmann A, Koldehofe B. Complex event processing. it-Information Technology, 2009, 51(5): 241–242.

[8] Clark T, Barn B. Event driven architecture modelling and simulation. In: Gao J, Lu X, Younas M, Zhu H, eds. SOSE. IEEE. 2011. 43–54.

[9] Clark T, Barn B. A common basis for modelling service-oriented and event-driven architecture. In: Aggarwal SK, Prabhakar TV, Varma V, Padmanabhuni S, eds. ISEC. ACM. 2012. 23–32.

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

[11] Clark T, Barn B, Oussena S. A method for enterprise architecture alignment. In: Proper E, Gaaloul K, Harmsen F, Wrycza S, eds. PRET, volume 120 of Lecture Notes in Business Information Processing. Springer. 2012. 48–76.

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

[13] Deeson E. The e-revolution and post-compulsory education–by boys, jos & ford, peter. British Journal of Educational Technology, 2008, 39(4): 750–750.

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

[15] S. Flake and W. Mueller. Formal semantics of OCL messages. Electron. Notes Theor. Comput. Sci., November 2004, 102: 77–97.

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

[17] Henderson J, Venkatraman N. Strategic alignment: Leveraging information technology for transforming organizations. IBM systems Journal, 1993, 32(1).

[18] Hillman J, Warren I. An open framework for dynamic reconfiguration. Proc. of the 26th International Conference on Software Engineering. ICSE '04. IEEE Computer Society. Washington, DC, USA. 2004. 594–603.

[19] Hnětynka P, Pláil F. Dynamic reconfiguration and access to services in hierarchical component models. In: Gorton I, Heineman G, Crnkovic I, Schmidt H, Stafford J, Szyperski C, Wallnau K, eds. Component-Based Software Engineering. volume 4063 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg. 2006. 352–359.

[20] Juric M. Wsdl and bpel extensions for event driven architecture. Inf. Softw. Technol., October 2010, 52(10): 1023–1043.

[21] Lankhorst M. Introduction to enterprise architecture. Enterprise Architecture at Work. The Enterprise Engineering Series. Springer Berlin Heidelberg, 2009.

[22] Lankhorst M, Proper H, Jonkers J. The Anatomy of the ArchiMate Language. International Journal of Information System Modeling and Design, 1(1).

[23] Li W. Qos assurance for dynamic reconfiguration of component-based software systems. IEEE Trans. on Software Engineering, , May-June 2012, 38(3):658 –676.

[24] Liu L, Xu J, Russell D, Davies J, Webster D, Luo Z, Venters C. Dynamic service integration for reliable and sustainable capability provision. Int. J. Systems Science, 2012, 43(1): 79–96.

[25] Meyer B. Programming languages and tools. Touch of Class. Springer Berlin Heidelberg. 2009. 321–360.

[26] Michelson B. Event-driven architecture overview. Patricia Seybold Group, 2006.

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

[28] Overbeek S, Klievink B, Janssen M. A flexible, event-driven, service-oriented architecture for orchestrating service delivery. IEEE Intelligent Systems, 2009, 24(5): 31–41.

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

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

[31] Robins D. Complex event processing. http://tinyurl.com/b2km9b8, 2010.

[32] Sestoft P. Higher-order functions. Programming Language Concepts. 2012. 77–91.

[33] Sharma V, Biliyan P. Maintainability analysis of component based systems. International Journal of Software Engineering and its Applications, 2011, 5(3): 107–118.

[34] Sharon G, Etzion O. Event-processing network model and implementation. IBM Systems Journal, 2008, 47(2): 321–334.

[35] Spencer J et al. TOGAF Enterprise Edition Version 8.1. 2004.

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

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

[38] Wu E, Diao Y, Rizvi S. High-performance complex event processing over streams. Proc. of the 2006 ACM SIGMOD International Conference on Management of Data. ACM. 2006. 407–418.

[39] Zachman J. A framework for information systems architecture. IBM Systems Journal, 1999, 38(2/3).

[40] Zang C, Fan Y. Complex event processing in enterprise information systems based on rfid. Enterprise Information Systems, 2007, 1(1): 3–23.

[41] Zdun U. Tailorable language for behavioral composition and configuration of software components. Computer Languages, Systems & Structures, 2006, 32(1): 56–82.

Appendix A Display Operations

```
value_to_table(value,tables,cont) {
  case value {
    Str(s) -> cont(StrVal(s),tables);
    Int(i) -> cont(IntVal(i),tables);
    Bool(b) -> cont(BoolVal(b),tables);
    Cons(h,t) -> value_to_table(h,tables,fun(h_value,tables)
      value_to_table(t,tables,fun(t_value,tables)
        cont(ConsVal(h_value,t_value),tables)));
    Nil -> cont(NilVal,tables);
    Term(type,values) -> term_to_table(type,values,tables,cont)
    else error('cannot process: ' + value)
  }
}
values_to_table(values,tables,cont) {
  case values {
    [] -> cont([],tables);
    value:rest -> value_to_table(value,tables,fun(v,tables)
      values_to_table(rest,tables,fun(vs,tables)
        cont(v:vs,tables)))
  }
```

```
}
table_lookup(type,tables) {
  case tables {
    [] -> Type(type,[]);
    Type(type,rows):_ -> Type(type,rows);
    _:rest -> table_lookup(type,rest)
  }
}
remove(x,l) {
  case l {
    x:rest -> rest;
    y:rest -> y:(remove(x,rest));
    [] -> []
  }
}
pos(x,l) {
  case l {
    [] -> error(’not contained’);
    x:_ -> 0;
    _:l’ -> 1 + pos(x,l’)
  }
}
term_to_table(type,values,tables,cont) {
  let entry = table_lookup(type,tables)
  in case entry {
       Type(type,l=({Row(values,translated_values)} U rest)) ->
         cont(Ref(type,pos(values:translated_values,l)),tables)
       else
          let tables = remove(entry,tables)
          in case entry {
               Type(type,rows) ->
                 let i = length(rows)
                 in values_to_table(values,tables,fun(translated_values,tables)
                      cont(Ref(type,i),Type(type,rows+[Row(values,translated_values)]):tables))
          }
     }
}
tables_to_entry(tables) {
  Table([Text(name):row_to_entry(values) | Type(name,rows) <- tables, Row(_,values) <- rows ])
}
row_to_entry(values) {
  [ Text(value_to_string(value)) | value <- values ]
}
proper_list(value) {
  case value {
     NilVal -> true;
    ConsVal(_,t) -> proper_list(t)
    else false
  }
}
list_to_string(list) {
  letrec elements_to_string(l) {
    case l {
      NilVal -> ’’;
      ConsVal(x,ConsVal(y,l’)) -> (value_to_string(x)) + ’,’ + elements_to_string(ConsVal(y,l’));
      ConsVal(x,NilVal) -> value_to_string(x)
    }
  }
  in ’[’ + (elements_to_string(list)) + ’]’
}
value_to_string(value) {
  case value {
    StrVal(s) -> s;
    IntVal(i) -> i;
    BoolVal(b) -> b;
    Ref(t,i)  -> t+'('+i+')';
    ConsVal(h,t) ->
      if proper_list(value)
      then list_to_string(value)
      else (value_to_string(h))+':'+(value_to_string(t));
    NilVal -> '[]'
  }
}
```