A Method for Enterprise Architecture Alignment

Tony Clark[1], Balbir S. Barn[1], and Samia Oussena[2]

[1] Middlesex University, London, UK
[2] University of West London, UK

Abstract. Business and ICT strategic alignment remains an ongoing challenge facing organizations as they react to changing requirements by adapting or introducing new technologies to existing infrastructure. Enterprise Architecture (EA) has increasingly become relevant to these demands and as a consequence numerous methods and frameworks have emerged. However these approaches remain bloated, time-consuming and lacking in precision. This paper proposes a light-weight method for EA called LEAP and introduces a language for EA simulation that is illustrated with a detailed case study of business change currently being addressed by UK higher education institutions.

Keywords: Enterprise Architecture, Simulation, Alignment.

1 Introduction

Enterprise Architecture (EA) is intended 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 [18]. 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: 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 of systems; checks for quality assurance and compliance; and strategic planning [9,25,20,4,13]. Alignment between business and IT strategy however remains one of the most pressing concerns [6].

EA has its origins in Zachman's original EA framework [34] but has since seen a range of methods introduced along with some specific tool modelling languages such as ArchiMate [12]. Emerging methods while purporting to address EA requirements have themselves posed questions about their efficacy. Because methods have largely been located as part of EA frameworks they do not readily provide the means by which to easily address the need to understand how to change an EA to meet a new requirement. Drilling down, the potential impact and change to an EA required would need to be promulgated as an impact analysis, a sliced view of the EA (of the systems affected), a gap analysis of missing functions and most importantly an equivalence analysis of an existing system and proposed changes. Current methods and frameworks that have largely presented layered architectural models do not necessarily lend themselves to this

E. Proper et al. (Eds.): PRET 2012, LNBIP 120, pp. 48–76, 2012.
© Springer-Verlag Berlin Heidelberg 2012

type of modeling and analysis. Furthermore their bloated and document driven nature presents additional issues of complexity and places significant workloads on enterprise architects and those tasked with managing systems in large organization. In section 2 we discuss and review the various EA methods and frameworks currently available.

Another aspect that has potential to influence the use of an EA to address use cases such as measuring alignment between business and IT, business change or integration of new systems is the different architectural styles that may be prevalent in a single organization. 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 [19] and [26], 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 [22]. The relationship between event driven SOA and EA is described in [2] 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.

Technologies for EA need to support a wide spectrum of business concepts and use-cases. When designing such technologies there are essentially two approaches: top-down or *analytic* and bottom-up or *synthetic*. The former characteristically identifies all potentially distinct categories of feature from the domain with the goal of equipping the user with a richly diverse collection of elements with which to express their models. The approach guarantees to provide a sufficiently expressive language at the expense of precision and orthogonality. The latter characteristically identifies a precisely defined collection of orthogonal concepts with associated semantics; the goal is to achieve precision with respect to a collection of defined use-cases, as opposed to the the more holistic, but imprecise, top-down approach.

There is safety in the analytic approach, it is guaranteed to be complete mostly because of its ambiguity and rich collection of features. However this safety is misleading since the resulting language is not amenable to mechanical processing and rigorous analysis. Therefore, top-down languages are forever consigned to

the early stages of system analysis and design where so-called *sketching* is an important modelling technique.

So how dangerous is the synthetic approach? Certainly it is almost guaranteed to be incomplete since the design of the language must be contextualized with particular use-cases. However, this might lead to a language that is *good enough* for most cases, and as such will have engineering benefits that far outweigh those of a sketching language. In addition, a synthetic language provides a firm basis for iterative language development through the incremental analysis of new use-cases.

EA languages are currently exclusively top-down. They are large and imprecise and therefore almost guaranteed to support any interpretation of a business and associated EA use-case. Typically, an EA language makes distinctions between different views of a business, for example separating business, application and technology layers. The result is a very large number of suspiciously similar features. Our proposal is that this is not necessary, and that the multiplicity of feature variety and separation of views is bogus, at least fundamentally[1].

This paper validates the claim that EA technologies should be synthetic by introducing and using a technology called LEAP to analyse a problem faced by UK universities. The case study involves the specification of an idealized system that meets a new organizational regulation, then shows how the current IT systems can be used to implement the system and finally describes a process by which the two architectures can be aligned. It is precisely because LEAP is a synthetic operational language for architecture simulation that the user can have confidence in the alignment, in contrast to other enterprise modelling technologies. The approach has been used to describe concrete IT systems within our own organization and to indicate appropriate modifications necessary to meet new regulations. Our contribution claim is that LEAP represents a novel approach to EA in that it is a simple, precise and executable technology that offers a different approach to EA analysis with the advantages that come with the ability to analyze and simulate an architecture.

2 Related Work on EA Methods

The nature of EA, that is, its breadth, the range of organizational impact and the inherent complexity of operating at multiple levels (business through to deployment) and technology variations means that it is difficult to arrive at specific understanding of what methodologies are available and the extent of their utilization. As Riege et al point out: '*Although there are isolated EA methods taking the situation of application into account,..., there is no overall landscape of EA methods available*' [25, :p389]. In addition, the practitioner nature of EA also means that well documented methods are difficult to access. In order to establish the current availability and literature surrounding EA methods, key word searches '*Enterprise Architecture Method*' were conducted in Google Scholar and the ACM/IEEE digital libraries. This section presents an overview

[1] How it is presented to a business user is entirely a different matter.

of the current situation and while it is not an exhaustive literature review (the limitations of the paper and its main focus prevents that) it does allow the reader an insight into the state of the art.

Much literature has concentrated on providing descriptions of a number of architecture frameworks. Usefully Steen et al point out: '*Frameworks provide structure to the architectural descriptions by identifying and sometimes relating different architectural domains and the modelling techniques associated with them*' [28, :p6]. Some of the more popular and widely disseminated EA frameworks include:

Zachman's Framework that provides a logical structure for classifying and organizing representations of an EA relevant to specific stakeholders in terms of 36 different types of views [34];

The Reference Model for Open Distributed Processing (RM-ODP) is an ISO/ITU Standard (ITU, 1996) that defines a framework for architecture specification of large distributed systems using five viewpoints on a system and its environment: enterprise, information, computation, engineering and technology. The theoretical basis of the RM-ODP model resides in object oriented principles and service oriented specification and the mapping of the levels to implementation objects [15,24];

The Open Group's framework TOGAF [27] and related frameworks for the Department of Defense (DODAF [33]), Federal processing (FEAF), UK Ministry of Defence (MODAF) [3] provides over-arching structures for supporting a consistent approach for standardizing, planning, analyzing and modelling of architectural system components.

Regardless of the specifics of the framework, as Tang et al note there are common deficiencies such as: (1) the level of detail required in an architecture model is not generally specified; (2) support, specification and management of non-functional requirements is lacking and (3) software configuration modelling is also generally lacking [29]. Although architecture includes notions of design, the objective of architecture is different from design but there are a lack of guidelines to address the case when architectural activity moves into detailed design.

The frameworks discussed above claim independence of any specific method. In addition to the availability of these frameworks, a number of methods aimed at delivering techniques, languages and tools to support EA have also been developed. The ADM method underpinning TOGAF is one exception. Methods have focused on specific aspects of business and IT alignment [30,31] (an oft cited requirement [6]) or they have provided a means of providing analysis tools for understanding EA changes and impact [17,16]. Of note also is the UN/CEFACT modelling method - a UML based approach to design business services that are focused on collaboration with external organizations [14]. Like [11]this method introduces the notion of the extended enterprise architecture that includes external system components (located in other organizations) that require collaboration.

There are examples of methods that have a more generic EA purpose. These methods do not focus on typical use cases for EA, instead they are aimed at

addressing the design gap introduced earlier and identified by [29]. An early example is Memo [10] an EA method that introduces a range of visual modelling languages supporting multiple views. The method provides an integrated process model. Some of the approaches proposed could be argued to have been superseded by advances in business process modelling notably with the advent of BPMN (Business Process Modelling Notation) and service oriented architectures. Pereira and Sousa [23] introduce a method that is overlayed on top of the Zachman framework and suggests how specific techniques can be used to develop each of the 36 viewpoints. Integration of artifacts produced for the viewpoints is also suggested. Support for Event modelling is not immediately clear in this method. The SOMA method developed by Arsanjani et al for IBM is an end-end software development life-cycle method that assumes a service oriented architecture style for EA. The method uses concepts of component based design and goal oriented modelling as well as established techniques such as use case modelling to support the design and implementation of EA solutions [1]. While the language and concepts underpinning SOMA have some similarity with the method and technology proposed in this paper, we note that consistent with all the methods reviewed here, there is not immediate clarity on how event modelling is integrated and supported in these methods.

The range and variation of methods in terms of focus and scope strongly supports the case proposed by Riege et al that there is no method that fits all the requirements for EA and instead there is a need for a method engineering approach [25]. They identify three broad contingency factors that should influence the target focus for EA methods: Adoption of advanced architectural paradigms and modelling capabilities; Deployment and monitoring of EA data and services and Organizational penetration of EA. We argue that that methods also need to ensure that they address the key use cases for EA such as business and IT alignment.

Our claim is that many EA use-cases can be addressed using a precisely defined synthetic language compared to the imprecise analytic technologies currently available. To validate this claim we have constructed a technology called LEAP that is briefly introduced in section 3 where it is compared with a leading EA technology. We have applied LEAP to a number of real-world case studies such as that described in section 6, however because these become rather large, section 4 describes a complete LEAP application using a simple example. We do not claim that LEAP will support *every* EA use-case, however our aim to address a series of use-cases and incrementally extend LEAP. This paper argues that LEAP represents a practical technology for Architecture Alignment and our approach is defined in section 5.

3 LEAP

The LEAP language proposes that EA is fundamentally about representing and analyzing data-rich and highly-structured executable systems at different levels

of abstraction. It has been designed as a synthetic language where distinctions between many business concepts are deemed fundamentally irrelevant (domain specific presentation issues being viewed as perfectly respectable sugar). The key concepts in LEAP are:

component. A component is the key structuring concept in LEAP and can be used to represent entities such as physical systems, roles, logical systems, transient elements, and organizational units. Components encapsulate data, behaviour, conditions such as business directives and goals, and can be nested. Components are intended to support a process of step-wise refinement where a business can be expressed as a single component at a high-level of abstraction and where refinement develops a graph of sub-components.

data. Each component defines models of data; shared models support information communication between components. Data is highly structured, including lists and records, to facilitate declarative pattern matching.

functions. LEAP is a functional language. Functions can be attached to components to represent business processes and can be used at any level to parameterize over language features. For example, parameterizing over components supports template patterns.

messages. LEAP execution is performed in terms of messages between component ports. Messages are defined in port interfaces and bear model data. Execution strategies for both SOA and EDA are supported through the construction and use of component architectures based on the same fundamental concepts.

rules. Component behaviour can be specified in terms of rules that match against data in the local database and messages arriving at the component's ports. Rules facilitate complex event processing since a rule may rely on receiving multiple unordered interrelated messages and database changes of arbitrary complexity. Where appropriate, rule collections can be expressed using state machines within components.

conditions. Business goals and directives can be expressed using invariants over the state of a component and its sub-components. Component behaviour can be expressed in terms of pre and post-conditions.

LEAP has a precise semantics in the form of an operational implementation and an associated tool for graphical display and simulation. Our claim is that the features above are necessary and sufficient for a wide range of EA use-cases, and there they are not sufficient, a conservative extension will suffice.

Table 1 shows a comparison between ArchiMate concepts and LEAP concepts. We have chosen ArchiMate because it is arguably the most developed EA notation. The table shows that ArchiMate includes a large number of different elements that can be mapped onto a smaller number of LEAP elements. In fact, ArchiMate includes more elements that those shown because several of the concepts occur as distinct elements in different layers. Our claim is that this mapping provides evidence that EA languages, and ArchiMate in particular,

Table 1. Comparison of Archimate and LEAP

Archimate Concept	**LEAP Concept**
Actor	Component
Application Layer	Components
Artifact	Data Model
Behaviour	Operation,rule, transition
Business Function	Operation, rule, transition
Business Layer	Components
Business Process	Operation, rule, transition
Business Service	Operation, rule, transition
Collaboration	Components
Collective Behaviour	Components
Communication Path	Connections
Concept	Component,Class
Contract	Invariant
Device	Component
External Perspective	LEAP Model
Individual Behaviour	Component
Interaction	Mesage
Interface	Interface
Internal Perspective	LEAP Model
Meaning	Semantics
Network	Component
Product	Data Model
Representation	Data Model
Role	Component
Software	Component
Technology Layer	Components
Value	?

includes redundancy that is difficult to analyze without mapping to a language with precise semantics.

It should be stated that LEAP does not claim to be an Architecture Description Language, although it shares many features with technologies for ADL. Features of LEAP can be used to model both physical and logical aspects of a system including information, roles and organizational units. Furthermore, although LEAP has an operational semantics, there is no support for expressing complex features such as real-time.

LEAP is a text-based language together with a graphical modelling tool. Figure 1 shows the core language features. A LEAP model consists of a collection of a collection of nested component definitions. A component has input and output ports from which it reads and to which it writes messages. Ports are typed with interfaces. Each component manages a database whose tables are defined as classes and associations in a data model. the database can be initialized using a **state** declaration and a component is initialized by the **init** expressions that are evaluated when the component is created. Incoming messages are handled by operations. An operation may update the component's database by adding and removing data; changes in the database are monitored by a collection of rules. When a rules patterns all match the current state of the database, the body of the rule is performed.

```
exp ::=
  cmp
| fun(arg*) exp              functions
| exp(exp*)                  applications
| var                        variables
| atom                       ints,strs,bools
| state                      local data
| self                       reference
| { exp* }                   blocks
| { bind* }                  records
| [ exp | qual* ]            lists
| new term                   extension
| term                       terms
| delete term                deletion
| if exp then exp else exp conditional
| replace pattern with term else exp
| find pattern in exp exp else exp
| case exp { arm* }          matching
| let bind* in exp           locals
| for pattern in exp { exp } loops
| forall pattern in exp { exp } univ quant
| exp <- name(exp*)          message passing
term   ::= name(exp*)
arm    ::= pattern -> exp
bind   ::= pattern = exp
qual   ::= pattern <- exp | ?exp
```

```
cmp ::=                       components
  component [name] {          optional name
    port*                     input/output ports
    [model { element* }]      data models
    [state { term* }]         local data
    [invariants { inv* }]     always hold
    [operations { op* }]      methods
    [rules { rule* }]         event processing
    [init { exp* }            initialization
    (name = exp)*             bindings
  }
port ::=
  port name[(in|out)]: interface { message }
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
op     ::= name(arg*) { exp* }
rule   ::= name : pattern* { exp* }
```

Fig. 1. LEAP Language

4 A Simple LEAP Example

The people of Ruritania adore fruit, especially apples and oranges. However, an increase in the voracious Ruritanian Fruiter Beetle (*Greengrocerous Apostrophorum*) means that availability must be limited so that each Ruritanian can have either apples or oranges, but not both, each day. Typically a Ruritanian fruit shop will sell apples and oranges at separate tills, merging the account at the end of each day. However this makes it difficult to police the fruit quotas. A new system must be implemented that enforces the rules until the beetle can be eradicated.

Our business goal in this case is to enforce the regulations. Our approach is to design an idealized architecture that satisfies the regulation and then to extend the current fruit shop architecture in such a way that it is possible to show how the physical architecture is consistent with the logical architecture. Our claim is that this EA use-case is supported by LEAP because it has a precisely defined behaviour.

4.1 Logical Architecture

The first step in EA Alignment is to define the logical architecture. Typically this will create a single component definition that captures the logical information and behaviour together with any constraints that must be achieved. Figure 2

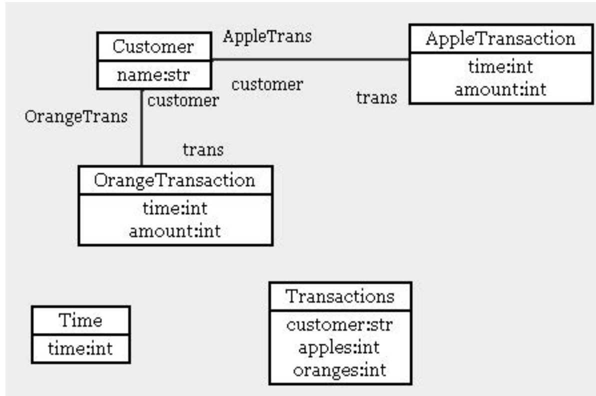

Fig. 2. Greengrocer Information

shows the information managed by each Ruritanian greengrocer. The business goal is specified as the following LEAP invariant:

```
illegal_to_buy_both_products_on_same_day {
  exists OrangeTrans(c,OrangeTransaction(t,_)) in state {
    exists AppleTrans(c,AppleTransaction(t,_)) in state { false }
  }
}
```

Simulation of the logical component is achieved by defining a LEAP component that sends messages to a greengrocers shop component as shown in figure 3. The definition of component `logical_architecture_simulation` includes the definition of the logical architecture named `greengrocers`. The simulation component has a predefined state that contains a sequence of messages each of which has a time, a message name and a sequence of arguments. There are two simulation rules: `send` that fires when there is a message at the current time, and `tick` that fires when there is no message at the current time and when the end of the simulation has not been reached. The `send` rule sends the message to the input port of the greengrocer component. The `tick` rule increments the time and sends a `tick` message to the greengrocer. The logical component is defined in figure 4. The port named `in` can receive messages named `buy_apples`, `buy_oranges` and `tick`. Each message is handled by an operation with the same name. Consider `buy_apples`, it uses the private local operation `get_customer` to select a term from the component's database (named `state`) if it exists or create a new customer-term if it does not. The `buy_apples` operation proceeds by querying the database for the current time and then adding two new database terms that represent an apple transaction.

The `greengrocer` component has a rule named `day` that is run once per day in order to consolidate the accounts. Rules are checked each time a message arrives or when the database changes in a component. In this case all the transactions for the customer in the given day are added up and a new consolidated `Transactions` term is added to the database.

```
component logical_architecture_simulation {
  component greengrocers {
    // Defined elsewhere...
  }
  state {
    Time(0)
    Message(0,'buy_oranges',['fred',10])
    Message(1,'buy_apples',['fred',10])
    Message(2,'buy_oranges',['fred',10])
    Message(3,'buy_apples',['fred',10])
    Message(4,'buy_oranges',['fred',10])
    Message(4,'buy_apples',['fred',10])
    End(5)
  }
  rules {
    send: Time(t) Message(t,m,args) {
      send(greengrocers.in,m,args);
      delete Message(t,m,args)
    }
    tick: Time(t) not(Message(t,_,_)) not(End(t)) {
      delete Time(t);
      greengrocers.in <- tick();
      new Time(t+1)
    }
  }
}
```

Fig. 3. Simulation Component

```
component greengrocers {
  model { // As shown in figure Greengrocer Information ... }
  invariants { // The illegal_to_buy_both_products_on_same_day condition... }
  port in[in]: interface {
    buy_apples(name:str,amount:int):void;
    buy_oranges(name:str,amount:int):void;
    tick():void
  }
  operations {
    buy_apples(customer,amount) {
      let c = get_customer(customer); t = time()
      in new AppleTransaction(t,amount), AppleTrans(c,AppleTransaction(t,amount))
    }
    buy_oranges(customer,amount) { // As above for oranges... }
    get_customer(name) { find Customer(name) in state else new Customer(name) }
    time() { find Time(t) in state { t } else 0 }
    tick() { replace Time(t) with Time(t+1) else new Time(1) }
    addup(l) { case l { [] -> 0; h:t -> h + addup(t) } }
  }
  rules {
    day: Time(t) {
      for Customer(name) in state {
        let oranges = addup([n | OrangeTrans(Customer(name),OrangeTransaction(tt,n)) <- state, ?(tt <= t) ]);
            apples = addup([n | AppleTrans(Customer(name),AppleTransaction(tt,n)) <- state, ?(tt <= t) ])
        in replace Transactions(name,a,o) with Transactions(name,apples,oranges)
           else new Transactions(name,apples,oranges)
      }
    }
  }
}
```

Fig. 4. The Logical Greengrocers Component

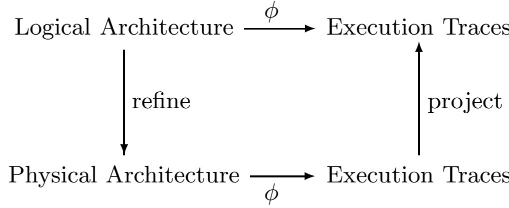

Fig. 5. Architecture Refinement

4.2 Refinement

LEAP has an executable semantics which means that a LEAP model can be mapped to execution traces. Figure 5 is an overview of the approach for architecture alignment. A Logical Architecture, such as that described for Ruritanian `greengrocers` is mapped to execution traces via a semantic function ϕ. A Physical Architecture is constructed using the real-world systems available to the organization leading to a collection of physical execution traces. It remains to show that the physical architecture is *complete* and *consistent*. Completeness is achieved by showing that there is a physical trace for every correct logical trace and consistency is achieved by showing that every physical trace can be projected onto a correct logical trace, subject to preserving key information.

Table 2. Logical Architecture Execution Trace

component	in	state
greengrocers	buy_oranges('fred',10)	
greengrocers	tick()	OrangeTrans(Customer('fred'), OrangeTransaction(0,10)), OrangeTransaction(0,10), Customer('fred')
greengrocers	buy_apples('fred',10)	Transactions('fred',0,10), Time(1),...
greengrocers	tick()	AppleTrans(Customer('fred'), AppleTransaction(1,10)), AppleTransaction(1,10),...
greengrocers	buy_oranges('fred',10)	Transactions('fred',10,10), Time(2),...
greengrocers	tick()	OrangeTrans(Customer('fred'), OrangeTransaction(2,10)), OrangeTransaction(2,10), Transactions('fred',10,10), Time(2), ...
greengrocers	buy_apples('fred',10)	Transactions('fred',10,20), Time(3),...
greengrocers	tick()	AppleTrans(Customer('fred'), AppleTransaction(3,10)), AppleTransaction(3,10), Transactions('fred',10,20), Time(3), ...
greengrocers	buy_apples('fred',10)	Transactions('fred',20,20), Time(4), ...
greengrocers	buy_oranges('fred',10)	AppleTrans(Customer('fred'), AppleTransaction(4,10)), AppleTransaction(4,10), Transactions('fred',20,20), Time(4), ...
greengrocers	tick()	OrangeTrans(Customer('fred'), OrangeTransaction(4,10)), OrangeTransaction(4,10), ...
greengrocers		Transactions('fred',30,30), Time(5), ...

Given that LEAP has a precisely defined semantics (currently implemented as a tool for executing LEAP models), it would be possible to formally establish the refinement criteria. In practice however, it is likely that rigorous inspection of traces will be sufficient to provide confidence of correct refinement. The LEAP tool can produce an XML trace of a model execution. The table shown in table 2 is a symbolic representation of the output for the greengrocer simulation where execution proceeds from top to bottom and repeated state is represented by ellipses.

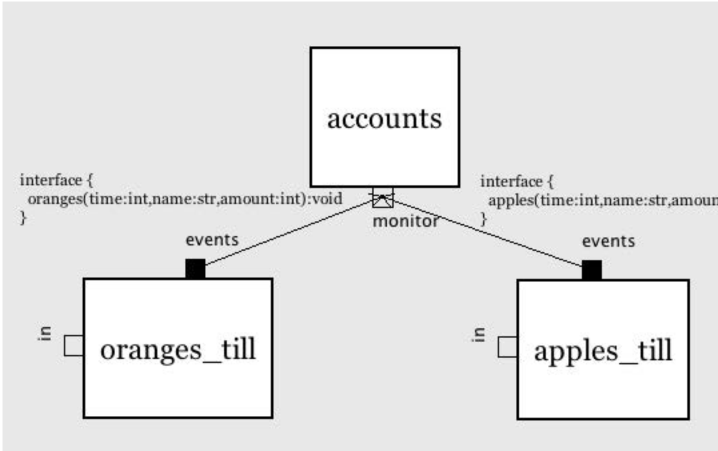

Fig. 6. Ruriatian Greengrocers: Physical Architecture

4.3 Physical Architecture

A physical architecture must reflect the systems available to an organization. All Ruritanian Greengrocers must, by law, implement separate tills for apples and oranges. Therefore, they have separate IT systems that must be consolidated in order to implement the new regulations. The consolidation is achieved by a new IT system, called `accounts`, as shown in figure 6. Notice that the two tills have output ports that produce events; a third party component can monitor the events in order to detect changes. The simulation component must be modified slightly to reflect the physical architecture as shown in figure 7. The two till components are almost identical, therefore they are candidates for *template patterns*. LEAP can represent template patterns by abstracting a function over a component definition. Each till behaves exactly the same except for the type of produce being sold, therefore the physical architecture simulator defines the `make_till` operation defined in figure 8. Now it is possible to create both types of till using the function:

```
component physical_architecture_simulation {
  state {
    Time(0)
    Message(0,oranges_till.in,'buy',['fred',10])
    Message(1,apples_till.in,'buy',['fred',10])
    Message(2,oranges_till.in,'buy',['fred',10])
    Message(3,apples_till.in,'buy',['fred',10])
    Message(4,apples_till.in,'buy',['fred',10])
    Message(4,oranges_till.in,'buy',['fred',10])
    End(5)
  }
  rules {
    send: Time(t) Message(t,p,m,args) {
      send(p,m,args);
      delete Message(t,p,m,args)
    }
    tick: Time(t) not(Message(t,_,_,_)) not(End(t)) { replace Time(t) with Time(t+1) }
  }
  operations {
    time() { find Time(t) in state else 0 }
  }
  init {
    connect(apples_till.events,accounts.monitor);
    connect(oranges_till.events,accounts.monitor)
  }
}
```

Fig. 7. Physical Architecture Simulation

```
make_till(type) {
  component {
    model {
      class Customer { name:str }
      class Transaction { type:str; time:int; amount:int }
      assoc Trans { customer Customer trans Transaction }
    }
    port in[in]: interface {
      buy(name:str,amount:int):void
    }
    port events[out]: interface {
      buy(type:str,time:int,name:str,amount:int):void
    }
    operations {
      buy(customer,amount) {
        let c = get_customer(customer)
        in {
          new Transaction(type,time(),amount);
          new Trans(c,Transaction(type,time(),amount));
          events <- buy(type,time(),customer,amount)
        }
      }
      get_customer(name) {
        find Customer(name) in state else new Customer(name)
      }
    }
  }
}
```

Fig. 8. The `make_till` Operation

```
component physical_architecture_simulation {
  operations {
    make_till(type) { ... }
  }
  oranges_till = make_till('oranges')
  apples_till = make_till('apples')
  ...
}
```

The accounts component monitors the events created by the till components, must consolidate the accounts and must detect fraud when it occurs. The physical definition of the new system is shown in figure 9. The execution trace for the physical architecture is shown in table 3. At the end of the trace, the system reports fraud because the customer has attempted to buy apples and oranges on the same day.

It remains to show that the physical architecture is correct with respect to the logical architecture. In an ideal world we would formally prove this to be the case. However, in a practical setting, where architectures may be large and where expertise with formal methods is limited, we argue that is more realistic to be able to use rigorous argument through inspection, to provide confidence of correctness. Since LEAP provides modular components, operational semantics and execution traces, it is possible to generate execution data that can be inspected off-line.

```
component accounts {
  model {
    class Transactions { customer:str; apples:int; oranges:int }
  }
  port monitor[in]: interface {
    buy(type:str,time:int,customer:str,amount:int):void
  }
  operations {
    buy(type,t,customer,amount) {
      case type {
        'apples' -> new Apples(t,customer,amount);
        'oranges' -> new Oranges(t,customer,amount)
      }
    }
  }
  rules {
    record_apples: Apples(t,customer,amount) not(Oranges(t,customer,_)) {
      replace Transactions(customer,apples_bought,oranges_bought) with
        Transactions(customer,apples_bought+amount,oranges_bought)
      else new Transactions(customer,amount,0)
    }
    record_oranges: Oranges(t,customer,amount) not(Apples(t,customer,_)) {
      replace Transactions(customer,apples_bought,oranges_bought) with
        Transactions(customer,apples_bought,oranges_bought+amount)
      else new Transactions(customer,0,amount)
    }
    fraud: Oranges(t,customer,_) Apples(t,customer,_) {
      print('FRAUD: ' + customer + ' at time ' + t + ' ' + state)
    }
  }
}
```

Fig. 9. Physical Definition for accounts

In the case of the Ruritanian Greengrocers system, we need to show that execution traces such as those shown above satisfy correctness. Consider transforming the physical trace into the logical trace. The two till components can be transformed into their logical counterpart by re-introducing type information. The accounts component is almost equivalent to the information held in the logical component and can be trivially transformed. Messages that purchase apples and oranges can be transformed by reintroducing type information, and `tick` messages can be introduced when the time changes. Therefore, we argue that the physical trace is consistent with the logical trace.

It remains to show that every correct logical execution has an equivalent physical execution. To see this we argue as follows. Every `buy_oranges` and `buy_apples` message is translated to a `buy` message that targets the appropriate till component. The effect of these messages on the tills and accounts has the desired effect. The `tick` messages are removed, but they occur when no other messages are being processed and have the same effect in the physical architecture.

5 An Approach to Architecture Alignment

In this section we introduce our approach to using LEAP for Architecture Alignment. The motivation for developing a method to support EA is driven by our hypothesis that existing methods are large, cumbersome, and are not based on precisely defined concepts. Where methods have used modeling languages such as ArchiMate they are constrained by orthodox layering approaches (business layer, functional layer, deployment layer and so on) that prevent rigorous equivalence analysis. Our proposed method also uses existing techniques to identify key information, but then represents it using a precisely defined simulation language. Figure 10 provides an overview of our proposed method.

Consistent with most approaches to EA methods where there is need to describe *as-is* and *to-be* models, there are two streams of activity which converge at key stages. The *to-be* analysis stream includes activities to **Model Requirements**. We do not prescribe how you might wish to derive the requirements in order to produce a model of requirements but as our method is based on UML-style modelling, models will include artifacts such as business information models, process models and business use case models. Existing method approaches such as Catalysis [8] and its derivatives [7] could be used for developing information models whilst recommended approaches for process modeling could include Ould's approach [21].

In parallel to the **Model Requirements** step, the activities in the **Collate Physical Architecture** stage will bring together existing descriptions of systems and their configurations. Our experience of such descriptions are large pictorial based documentation captured using drawing tools such as Powerpoint. A key output of this stage is a description of the systems that exist in the organization. We recommend capturing the description of each system as a UML Component to aid the migration to later stages of the method. Again, the method

Table 3. Physical Architecture Execution Trace

id	in	state
oranges_till	buy('fred',10)	
oranges_till		Trans(Customer('fred'), Transaction('oranges',0,10)), Transaction('oranges',0,10), Customer('fred')
accounts	buy('oranges',0,'fred',10)	
apples_till	buy('fred',10)	
accounts		Transactions('fred',0,10), Oranges(0,'fred',10)
apples_till		Trans(Customer('fred'), Transaction('apples',1,10)), Transaction('apples',1,10), Customer('fred')
accounts	buy('apples',1,'fred',10)	...
oranges_till	buy('fred',10)	...
accounts		Transactions('fred',10,10), Apples(1,'fred',10), Oranges(0,'fred',10)
oranges_till		Trans(Customer('fred'),Transaction('oranges',2,10)), Transaction('oranges',2,10), ...
accounts	buy('oranges',2,'fred',10)	...
apples_till	buy('oranges',2,'fred',10)	...
accounts		Transactions('fred',10,20), Oranges(2,'fred',10), Apples(1,'fred',10), Oranges(0,'fred',10)
apples_till		Trans(Customer('fred'),Transaction('apples',3,10)), Transaction('apples',3,10), ...
accounts	buy('apples',3,'fred',10)	...
accounts		Transactions('fred',20,20), Apples(3,'fred',10), Oranges(2,'fred',10), Apples(1,'fred',10), Oranges(0,'fred',10)
oranges_till		Trans(Customer('fred'),Transaction('oranges',4,10)), Transaction('oranges',4,10), ...
apples_till	buy('fred',10)	...
accounts	buy('oranges',4,'fred',10)	...
apples_till		Trans(Customer('fred'),Transaction('apples',4,10)), Transaction('apples',4,10),...
accounts	buy('apples',4,'fred',10)	Transactions('fred',20,30),...

does not prescribe new approaches, it leaves it to the practitioner to determine how to produce the artifacts required.

The **Configure Physical Architecture** step slices a description of an EA to determine what system components are likely to be impacted by emerging requirements. Techniques that can be used to support this impact analysis includes use case maps [5]. A use case map is simply a trace of path of causal sequences of events across a set of system components representing an EA. The events are triggered by a business use case identified in the **Model Requirements** step.

Alternative approaches that could be used in this step include the use of CRC to help identify those system components that are (collaboratively) responsible for delivering a business use case [32]. The key output from this activity is an artifact expressed in system components that includes all the EA system elements that will be subject to some impact as a result of the emerging requirements.

Up to now, the steps in the method have utilized well-established notations and techniques. The subsequent steps in stage 2 incorporate an integrated set of concepts from SOA and complex event processing.

The **Define Physical Enterprise Architecture (L-EA)** step is aimed at defining a slice of the existing physical architecture that we know will be subject to impact from new requirements. The slice emerged from the **Configure**

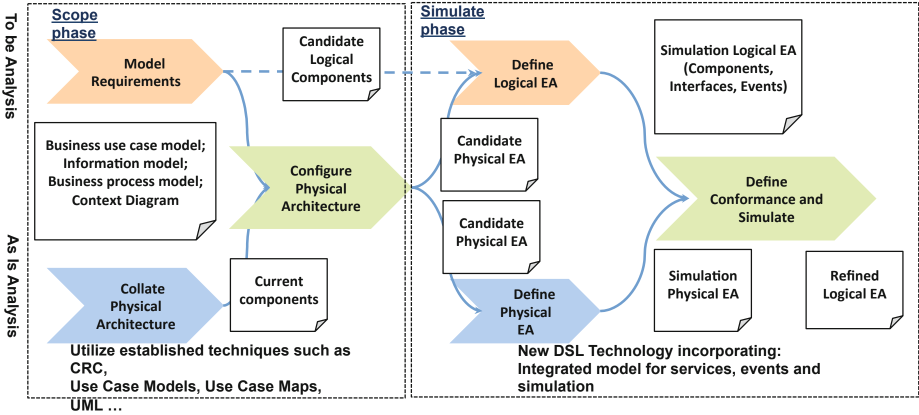

Fig. 10. Method Overview

Physical Architecture step and like the Logical EA step is now expressed in our DSL using concepts such as components, ports, rules and events.

The **Define Logical Enterprise Architecture (L-EA)** step produces a model based description of a target logical EA - that is - the system components, information structures and constraints that are required as a result of the Model Requirements step. Where appropriate the Logical EA may use candidate logical components from the Configure Physical Architecture step.

The Logical EA (L-EA) uses our integrated concepts derived from SOA and complex event modeling so the L-EA is expressed as components offering services, raised events, requested services and monitored events. Dependencies between components are thus expressed in terms of services request and fulfillments and event management.

The **Conformance** step uses simulation to produce and visualize results. The logical architecture describes what is required and the physical *to-be* architecture defines how existing systems can be used to satisfy the requirements. It remains to validate the physical architecture by showing that the behavior conforms to the requirements. If the simulation produces the same output when it is run with both the logical and physical EA definitions then we claim that they are *aligned*. Such an approach presents a practical solution that is geared toward EA practitioners.

6 Case Study

Having outlined the method and technology, this section presents a genuine requirement faced by IT directors in UK higher education institutions to deliver key information sets (KIS) to applicants deciding on which course and which university to chose for study at undergraduate level.

Higher education institutions (HEI) in the UK are faced with a challenging and dynamic business environment where public funding of HEIs has been

reduced by up to 70%. This lost funding is being replaced by the introduction of a new student fees regime beginning in 2012 following a bill introduced in the UK Parliament in November 2010. The UK Government is of the view that students will require key information in order to make informed decisions regarding the selection of courses and institutions. Currently this information is not readily available in a consistent and easily accessible form. Consequently the Higher Education funding body (HEFCE) is coordinating the specification of the required information and how it is to be made available and at what time.

HEFCE produces KIS data at a given census date each year. In order to be included in KIS, each university must register with both the NSS and DHLE government agencies before the census date. KIS information consists of NSS data, teaching and learning data from each university, financial data from each university (including university owned and private accommodation costs), employability data from the DHLE agency.

The NSS data is completed by students via a web portal. The details of the information go to the NSS agency and the university is informed of the completion for their records. Private property prices within the geographic area around the university are captured by monitoring RSS feeds from property companies.

7 Applying LEAP to the Case Study

Section 4 has shown a detailed, but simple example of using LEAP for architecture alignment. The example shows how LEAP is used to capture the top-level information structures and invariants that arise from a business requirement, how LEAP can be used to represent an architecture of interacting components based on existing IT systems, and then how alignment is established through simulation and rigorous argument.

Section 5 has outlined a pragmatic approach to Architecture Alignment that can be based on a range of technologies. This section follows the method using LEAP. Since the case study is quite large we will present an overview and include samples of the implementation where these are illustrative of our approach.

7.1 Step 1: Model Requirements

Figure 11 shows the information model that supports the KIS requirements. The model is taken directly from the LEAP tool that represents the requirements as a single top-level component. Each University has a number of students. Information is maintained on the cost of both University owned accommodation and private accommodation in the area. A student studies a course and optionally completes an NSS return in their third year of study; the NSS form allows students to comment on the quality of the University's provision of teaching and learning in terms of questions such as: '*Do you agree that you receive prompt feedback on formative assessments?*'.

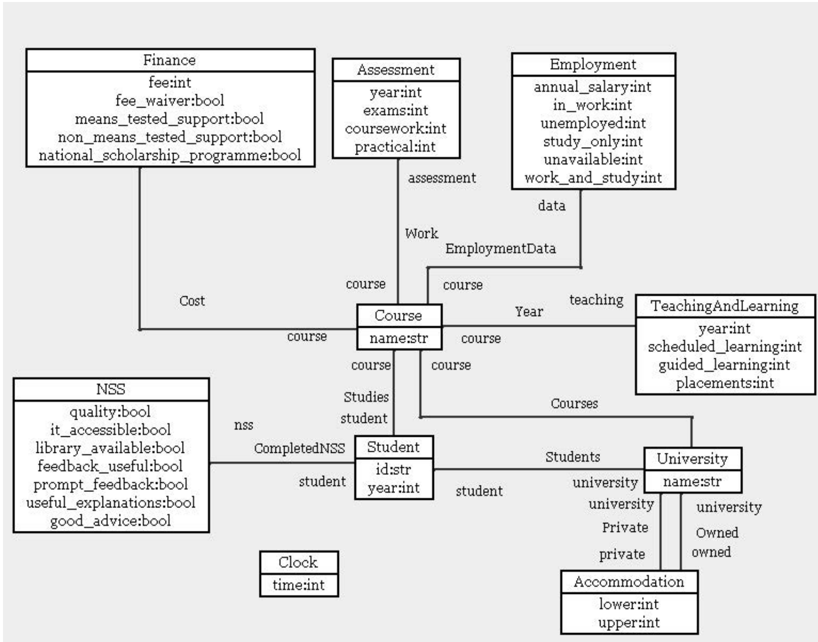

Fig. 11. KIS Information Structure

Each course is delivered in terms of scheduled, guided and practical teaching and learning components, and assessed in terms of exams, courseworks and practicals. Information is maintained nationally about employment statistics for particular courses, such as the salary of graduates and the percentage who are in work or unemployed 6 months after graduation. Each HE course in the UK has a cost and may involve various forms of financial support.

7.2 Step 2: Define L-EA and Simulation

Figure 12 shows the outline of the logical architecture simulator. All component structure in the architecture has been flattened and has been represented as information. For example, in the physical architecture we will be required to implement components for universities and for HEFCE.

The logical architecture will also contain a number of invariants that must be maintained when the logical architecture is refined to become a physical architecture. For example, the values of various fields must be unique and percentage values must add up to 100. The most important invariant that follows from the business requirement is that when the time reaches a specific point, all the information necessary to construct the KIS report must be available. Given this requirement, any mapping from logical to physical must provide KIS data no matter how distributed the data becomes.

```
component kis_logical {
  component kis {
    model { // KIS Information Structure... }
    port in[in]:interface {
      make_university(name:str,courses:void):void;
      register_student(university:str,student:str,course:str):void;
      accommodation(university:str,lower:int,upper:int):void;
      owned(university:str,lower:int,upper:int):void;
      complete_nss(student:str,quality:bool,it_accessible:bool,library_available:bool,...):void
    }
    operations {
      make_university(name,courses) {
        let u = new University(name)
        in for Course(course_name,t1,t2,t3,a1,a2,a3,employment,finance) in courses {
          let c = new Course(course_name)
          in // create and link instances of the information model...
        }
      }
      register_student(university,student,course) {
        find u=University(university) in state {
          find Courses(u,Course(course)) in state {
            let s = new Student(student)
            in new Students(u,s), new Studies(s,Course(course))
          } else error('no course ' + course)
        } else error('no university ' + university)
      }
      complete_nss(student,q,i,l,f,p,u,g) { ... }
      accommodation(university,lower,upper) { ... }
      owned(university,lower,upper) { ... }
    }
  }
  state {
    Time(0)
    Message(0,kis.in,'make_university',['middle england',[
      Course('Computer Science', TeachingAndLearning(1,60,40,0),..., Assessment(1,100,0,0),...,
        Employment(20000,50,20,20,10,0), Finance(9000,true,true,true,true))
    ]])
    Message(1,kis.in,'register_student',['middle england','fred','Computer Science'])
    ...
    Message(2,kis.in,'complete_nss',['fred',true,true,true,true,true,true,true])
    Message(3,kis.in,'accommodation',['middle england',500,1000])
    Message(4,kis.in,'owned',['middle england',500,1000])
    ...
    End(100)
  }
  rules {
    send: Time(t) Message(t,p,m,args) { send(p,m,args); delete Message(t,p,m,args) }
    tick: Time(t) not(Message(t,_,_,_)) not(End(t)) { replace Time(t) with Time(t+1) }
  }
}
```

Fig. 12. Logical Architecture Simulation

7.3 Step 3: Collate Physical-EA

The next step of our method involves reviewing the current physical *as-is* system architecture. Most organizations have a systems overview which is used as the input to this step. The result is an understanding of the current capability of the organization in terms of systems, interfaces, information and events.

The context for the physical EA includes external systems. When generating KIS data, all universities must work with the following external systems: students use the **web** to complete NSS reports; employability information is maintained by the **dhle**; the NSS forms are collated by **nss**; an RSS **property** feed provides information on the cost of accommodation at regular intervals; **hefce** manages the KIS process.

We will use the University of Middlesex (Mdx), London, UK as the basis for our case study. Space limitations prevent us from providing a complete description of the Mdx physical architecture, however it is consistent with most UK HEIs and includes systems for registry, an asset management system that includes a sub-system for university accommodation, an examinations database, a library system, a financial management system called PAFIS, a teaching and learning system called OASIS, an alumni management system, a student portal, and a staff portal.

7.4 Step 4: Configure Physical-EA

The next step of the method analyses the physical system of an organization and takes an appropriate *slice* to produce just those systems that will be involved in the required *to-be* architecture. In the case of supplying KIS data, we know that Mdx will need to provide student, accommodation, teaching and learning, assessment and financial information. Therefore, the P-EA will not include the alumni or library management systems.

7.5 Step 5: Define Physical-EA

Figure 13 shows a physical architecture model for KIS including two universities. The physical architecture distributes the information structures and invariants across multiple components and uses component-nesting within the university components to drill down to particular IT systems. For the purposes of simulation, the multiple universities are constructed using template patterns.

The simulation is driven by the `hefce` component that triggers an event when it is time to construct the KIS reports. The simulation uses pattern matching across multiple events within `hefce` to determine when all of the information has been received as shown in figure 14. Of particular interest are the rules defined by `hefce`. When both registration events are received in any order from NSS and DHLE then `hefce` registers the university. The `kis_run` rule detects when it is time for KIS reporting and sends messages to all universities, to DHLE and to NSS. The replies from these messages are received in an arbitrary order and update the `hefce` database; the `kis` rule detects when updates have occurred (again in any order) and creates the KIS report for each university.

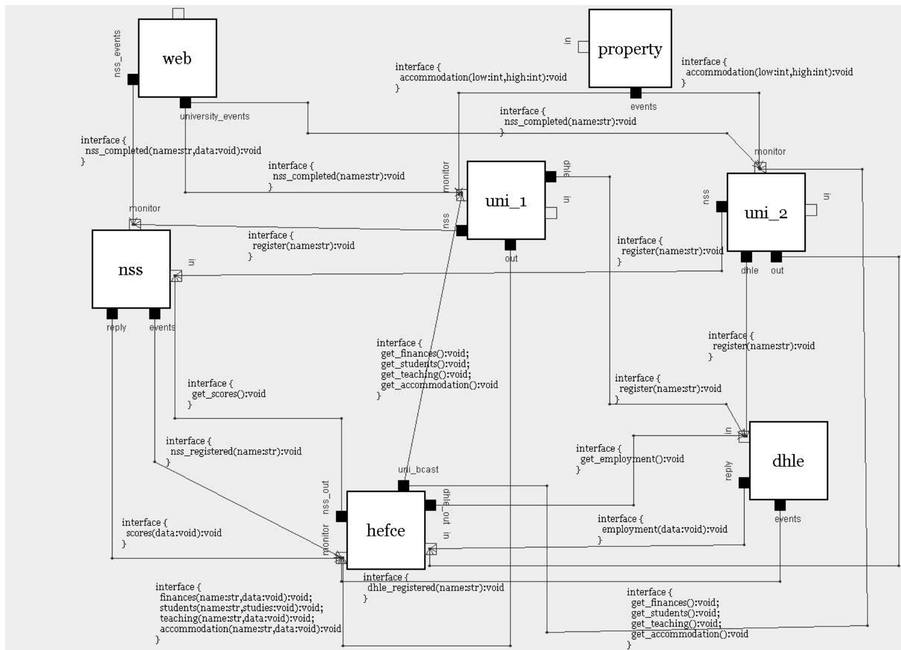

Fig. 13. Physical Architecture

7.6 Step 6: Conformance

Our EA design method produces both a logical and a physical architecture description using the LEAP simulation language. The logical architecture describes what is required and the physical *to-be* architecture defines how existing systems can be used to satisfy the requirements. It remains to validate the physical architecture by showing that the behaviour aligns to the requirements.

In general, conformance can be established using a number of approaches. The context defines a collection of system executions in terms of messages, events and state changes. It is possible to use inspection-based techniques to show that all required executions are handled appropriately by the physical architecture.

Our KIS physical architecture simulation model included Mdx IT systems that managed information on students, finance, property and academic teaching and learning. The invariants for the logical model were translated into equivalent conditions over the physical architecture.

The logical architecture simulation was driven using a sequence of messages that registered courses, registered students, completed NSS forms, and provided property prices. The same messages were used to drive the physical architecture and the results were observed using the LEAP tooling. Figure 15 shows part of the output where LEAP produces HTML. The simulation proceeds by generating clock ticks in response to button clicks. The simulation output shows KIS data rendered as a collection of gui components including a pie-chart and a histogram.

```
component hefce {
  state { KIS_Census(5) }
  port uni_bcast[out]: interface {
      get_finances():void;
      get_students():void;
      get_teaching():void;
      get_accommodation():void
  }
  port in[in]:interface {
    finances(uni:str,data:void):void;
    students(uni:str,data:void):void;
    scores(data:void):void;
    employment(data:void):void;
    teaching(uni:str,data:void):void;
    accommodation(uni:str,data:void):void
  }
  port nss_out[out]: interface { get_scores():void }
  port dhle_out[out]: interface { get_employment():void }
  operations { // updates to database corresponding to input messages ... }
  rules {
    university: NSS_Registered(name) DHLE_Registered(name) { new University(name) }
    kis_run: Time(n) KIS_Census(n) {
      uni_bcast <- get_finances();
      uni_bcast <- get_students();
      uni_bcast <- get_teaching();
      uni_bcast <- get_accommodation();
      dhle_out <- get_employment();
      nss_out <- get_scores()
    }
    kis: University(name)
         Students(name,studies)
         Finance(name,finance)
         Teaching(name,tdata)
         Accommodation(name,adata)
         NSS(data)
         Employment(edata) {
      let filtered_nss = [ nss_data | NSS(student_name,nss_data)       <- data,
                                      Studies(Student(student_name),_) <- studies ];
          filtered_employment = [ Employment(c,d) | Employment(name,c,d) <- edata ]
      in // construct report
    }
  }
```

Fig. 14. The HEFCE Simulation

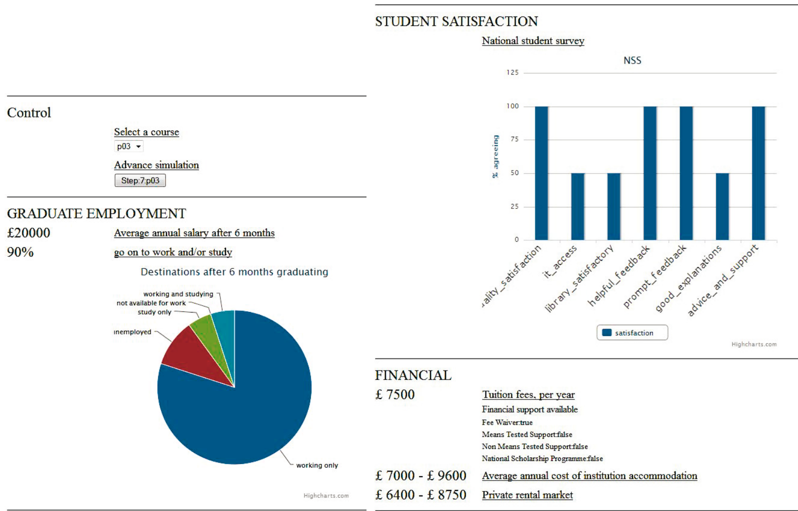

Fig. 15. Part of the KIS Simulation Output

In all cases, no invariants were violated and the output from the logical and physical simulations was identical.

Finally, if we require total confidence in conformance then we need to resort to formal methods such as model checking and theory proving. For large systems such as those found in EA, formal methods are often impractical in terms of complexity. That said, a formal semantics for LEAP is an area for future development in order to investigate whether formal methods could help.

8 Discussion and Further Work

Enterprise Architecture remains a confusing and constantly evolving collection of methods and frameworks which are generally characterized by an expansive outlook, lack of precision, a focus on diagrams and an emphasis on document management. The result is that existing approaches are difficult to analyze and process. This paper has presented an effort to pin down important EA use cases of managing change and better understanding the impact of changing requirements on existing technical architectures of an organization.

We have proposed a synthetic language for EA called LEAP and contrasted it with a leading analytic language called ArchiMate. Our claim is that the large collection of EA features in ArchiMate are not orthogonal and can be mapped to a much smaller collection in LEAP. This claim is validated through a real-world case study although it remains as further work to compare the resulting LEAP simulation with the equivalent ArchiMate models. Furthermore, we do not claim that analytic languages such as ArchiMate are redundant since they

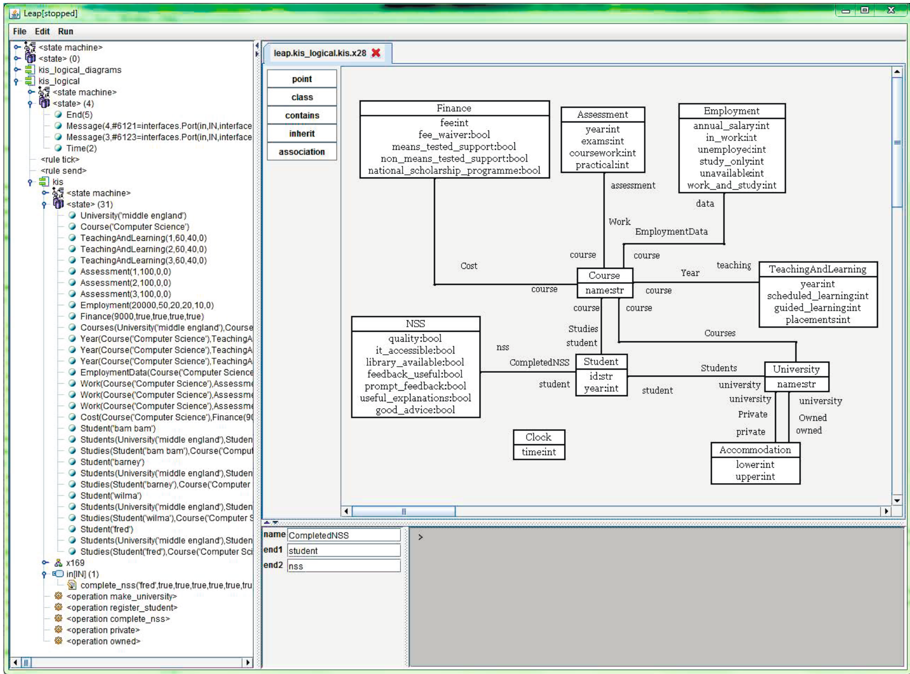

Fig. 16. Leap Tool

are domain-specific and present features in terms recognizable to a business analyst; however, LEAP could be used as a basis for EA language precision through mappings such as that described in table 1. In this way the broad EA could be captured by ArchiMate and then simulated and analysed using LEAP by making decisions about how each ArchiMate concept maps onto elements in LEAP and by introducing procedural and structural detail where required. Simulation results and analysis performed in terms of LEAP can then be presented back to the analyst using corresponding ArchiMate concepts.

It remains to show that the features of LEAP are necessary and sufficient for EA construction and analysis. Our primary concern in this paper is to provide examples of LEAP *in action* for simulation and to show that an operational semantics leads to scope for analysis that is more rigorous than that supported by other methods. Our guiding principle for the definition of LEAP features has been providing a *synthetic* language for EA by identifying low-level precisely defined concepts that can be freely combined. LEAP allows an organization to be modelled as a single component or as a highly-structured collection of collaborative service-oriented and event-based components. An organization can be modelled multiple times from different perspectives and the relationships can be analysed, thereby providing scope for step-wise refinement and a route to reconfiguration including migration to a SOA-based architecture. As such, we

claim that LEAP is highly expressive, but more empirical work is required to establish the claim that it is both necessary and sufficient.

Our claim goes further by proposing that EA languages should be executable wherever that makes sense. EA aims to address features of organizations; organizations are systems that *operate* in terms of structure, resources and information. LEAP provides a simple and universal basis for representing these EA characteristic features without introducing unnecessary distinctions between otherwise fundamentally identical concepts. This claim has been validated by applying LEAP to a real-world EA case study in order to address a typical EA use-case: Architecture-Alignment. We have shown that an operational semantics can be used in a practical sense to build confidence that two different architectural descriptions of the same system are equivalent.

We claim that LEAP represents a contribution to industrial EA because it takes a pragmatic approach to introducing precision in EA. Current EA languages lie at the *sketching* end of the development life-cycle. This is valuable, but is not amenable to automated analysis. At the other end of the spectrum lies formal methods and their associated tools, however it is not clear that there is any evidence that such a formalized approach to EA would be tractable given the size and complexity of the systems involved. LEAP lies in-between on this spectrum by supporting diagrams for the key features of an architecture, a high-level programming language for the details and a semantics that, in principle, does not rule out formal analysis in the future.

How would Industry use LEAP? Our experience with KIS and other case studies we have developed, is that the ability to create a simulation of part of an organization is very valuable. LEAP does not require components to map on to physical resources and organizational units, it is intended that features of an EA application, whether tangible or intangible, can be expressed in terms of components, data, rules, operations, constraints, state-machines, and messages. A novel feature of LEAP is that both components and operations are higher-order which means that it is easy to capture template patterns, as shown in the case of the tills in the greengrocer example and the universities in the KIS case study. In another example not reported here, we have used template patterns to capture the life-cycle of a customer record as a component.

Therefore, Industry can use LEAP to produce simulations of architectures at any level of abstraction, and the operational nature of LEAP makes it practical to compare the same system developed to different levels of detail. Since the information is represented as LEAP models, it is possible to generate artifacts from them, including code, although this is something to be investigated as future work. In addition, since components are encapsulated, our intention is to allow LEAP to interface to existing systems, thereby providing a means to migrate an existing architecture by simulating the new components and gradually replacing them with new IT systems.

Our use of LEAP for the KIS analysis at Mdx has shown that existing University information systems can support the new HEFCE regulations subject to being able to provide the appropriate interfaces and supporting the information

models defined in the simulation. Given a simulation, the mapping from LEAP to real information systems is straightforward. LEAP tooling supports single stepping through the simulation together with snapshots of the simulation state as shown in figure 16.

LEAP does not claim to be a universal technology for EA, however, as described above, we have taken a fundamentally different approach to the design of a language for EA compared to that provided by current systems. Since LEAP is a synthetic language it is necessarily limited, however it provides a basis on which to test the hypothesis that the our proposed concepts are sufficient for a wide variety of EA use-cases and, where it is found lacking, our claim is that the required extensions will be orthogonal and precisely defined where possible. Current limitations include the ability to express and manage the refinement of business goals and to express non-functional system requirements (such as cost and risk). We have started a process of consultation with Industry in order to understand how these features need to be represented and processed.

References

1. Arsanjani, A., Ghosh, S., Allam, A., Abdollah, T., Ganapathy, S., Holley, K.: Soma: A method for developing service-oriented solutions. IBM Systems Journal 47(3), 377–396 (2008)
2. Assmann, M., Engels, G.: Transition to Service-Oriented Enterprise Architecture. In: Morrison, R., Balasubramaniam, D., Falkner, K. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 346–349. Springer, Heidelberg (2008)
3. Biggs, B.: Ministry of defence architectural framework (modaf). IEE Seminar Digests 43 (2005)
4. Bucher, T., Fischer, R., Kurpjuweit, S., Winter, R.: Analysis and application scenarios of enterprise architecture: An exploratory study. In: 10th IEEE International Enterprise Distributed Object Computing Conference Workshops, EDOCW 2006 (2006)
5. Buhr, R.J.A., Casselman, R.S.O.: Use case maps for object-oriented systems, vol. 302. Prentice Hall (1996)
6. Chan, Y.E., Reich, B.H.: IT alignment: what have we learned? Journal of Information Technology 22(4), 297–315 (2007)
7. Cheesman, J., Daniels, J.: UML components. Addison-Wesley (2001)
8. D'Souza, D.F., Wills, A.C.: Objects, components, and frameworks with UML: the catalysis approach. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
9. 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. In: Proceedings of the 37th Annual Hawaii International Conference on System Sciences, HICSS 2004 (2004)
10. Frank, U.: Multi-perspective enterprise modeling (memo) conceptual framework and modeling languages. In: Proceedings of the 35th Annual Hawaii International Conference on System Sciences, HICSS 2002, pp. 1258–1267. IEEE (2002)

11. Goethals, F.G., Snoeck, M., Lemahieu, W., Vandenbulcke, J.: Management and enterprise architecture click: The fad (e) e framework. Information Systems Frontiers 8(2), 67–79 (2006)
12. The Open Group. Archimate technical standard (2008), http://www.opengroup.org/archimate/
13. Henderson, J.C., Venkatraman, N.: Strategic alignment: Leveraging information technology for transforming organizations. IBM Systems Journal 32(1) (1993)
14. Huemer, C., Liegl, P., Motal, T., Schuster, R., Zapletal, M.: The development process of the un/cefact modeling methodology. In: Proceedings of the 10th International Conference on Electronic Commerce, p. 36. ACM (2008)
15. ITU. Basic reference model of open distributed processing - part 1: Overview and guide to use. ITU Recommendation X.901 | ISO/IEC 10746-1. ISO/ITU (1994)
16. Johnson, P., Johansson, E., Sommestad, T., Ullberg, J.: A tool for enterprise architecture analysis. In: 11th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2007, pp. 142–142. IEEE (2007)
17. Johnson, P., Lagerström, R., Närman, P., Simonsson, M.: Enterprise architecture analysis with extended influence diagrams. Information Systems Frontiers 9(2) (2007)
18. Lankhorst, M.: Introduction to enterprise architecture. In: Enterprise Architecture at Work. The Enterprise Engineering Series. Springer, Heidelberg (2009)
19. Michelson, B.M.: Event-driven architecture overview. Patricia Seybold Group (2006)
20. Niemann, K.D.: From enterprise architecture to IT governance: elements of effective IT management. Vieweg+ Teubner Verlag (2006)
21. Ould, M.A.: Business Process Management: a rigorous approach. British Informatics Society Ltd. (2005)
22. Overbeek, S., Klievink, B., Janssen, M.: A flexible, event-driven, service-oriented architecture for orchestrating service delivery. IEEE Intelligent Systems 24(5), 31–41 (2009)
23. Pereira, C.M., Sousa, P.: A method to define an enterprise architecture using the zachman framework. In: Proceedings of the 2004 ACM Symposium on Applied Computing, pp. 1366–1371. ACM (2004)
24. Raymond, K.: Reference model of open distributed processing (rm-odp): Introduction. In: IFIP TC6 International Conference on Open Distributed Processing, pp. 3–14. Chapman and Hall, Brisbane (1995)
25. Riege, C., Aier, S.: A Contingency Approach to Enterprise Architecture Method Engineering. In: Feuerlicht, G., Lamersdorf, W. (eds.) ICSOC 2008. LNCS, vol. 5472, pp. 388–399. Springer, Heidelberg (2009)
26. Sharon, G., Etzion, O.: Event-processing network model and implementation. IBM Systems Journal 47(2), 321–334 (2008)
27. Spencer, J., et al.: TOGAF Enterprise Edition Version 8.1 (2004)
28. Steen, M.W.A., Strating, P., Lankhorst, M.M., ter Doest, H., Iacob, M.E.: Service-oriented enterprise architecture. In: Stojanovic, Z., Dahanayake, A. (eds.) Service Oriented Systems Engineering, Hershey, pp. 132–154 (2005)
29. Tang, A., Han, J., Chen, P.: A comparative analysis of architecture frameworks. In: 11th Asia-Pacific Software Engineering Conference, 2004, pp. 640–647. IEEE (2004)
30. Wegmann, A., Balabko, P., Lê, L.S., Regev, G., Rychkova, I.: A method and tool for business-it alignment in enterprise architecture. In: CAiSE 2005 Forum. Citeseer (2005)

31. Wegmann, A., Regev, G., Rychkova, I., Lê, L.S., De La Cruz, J.D., Julia, P.: Business and it alignment with seam for enterprise architecture. In: 11th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2007, p. 111. IEEE (2007)
32. Wirfs-Brock, R., Wilkerson, B., Wiener, L.: Designing object-oriented software, vol. 13. Prentice Hall, Englewood Cliffs (1990)
33. Wisnosky, D.E., 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)
34. Zachman, J.A.: A framework for information systems architecture. IBM Systems Journal 38(2/3) (1999)