Sheffield Hallam University

Outsourcing service provision through step-wise transformation

CLARK, Tony <http://orcid.org/0000-0003-3167-0739> and BARN, Balbir S

Available from Sheffield Hallam University Research Archive (SHURA) at:

https://shura.shu.ac.uk/12061/

This document is the Accepted Version [AM]

Citation:

CLARK, Tony and BARN, Balbir S (2014). Outsourcing service provision through step-wise transformation. In: ISEC '14 7th India Software Engineering Conference, Chennai. New York, ACM, p. 8. [Book Section]

Copyright and re-use policy

See http://shura.shu.ac.uk/information.html

Sheffield Hallam University Research Archive
http://shura.shu.ac.uk

Outsourcing Service Provision Through Step-Wise Transformation

Tony Clark
Middlesex University London, UK
t.n.clark@mdx.ac.uk

Balbir S. Barn
Middlesex University London, UK
b.barn@mdx.ac.uk

ABSTRACT

Component-based development principles promise a flexible approach to system design and implementation. In particular service-based techniques provide a computational model whereby the physical location of components makes no difference to the overall system behaviour. Economic business models for organisations have led to *outsourced services* as an attractive way of reducing costs and allowing a business to focus on its key processes. In the context of business and IT alignment, this raises a problem of how to transform an organization and its enterprise systems so that it can take advantage of an external service, given that in most cases the existing processes will be embedded in many places across the organisation. This paper addresses this problem by proposing a simple component-based simulation language together with transformation rules that can be used to incrementally isolate a service as an external component.

1. INTRODUCTION

Modern software systems are often organised in terms of *components*. Component-based approaches generalise basic object-oriented implementation platforms by allowing large collections of objects to be grouped together and externalised in terms of public interfaces. Such systems execute in terms of messages between components where the distance between message source and target is completely arbitrary. Component-based approaches can be used at different stages of the development life-cycle, at different levels of granularity and can involve different architectural approaches. Elsewhere we have critiqued the relative merits of various architectural styles [10], here we present an short overview of these approaches.

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 [3] when compared to more traditional implementation techniques. The organization of systems into coherent interfaces has been argued [25] as having disadvantages in terms of: extensions; accommodating new business functions; associating single business processes with complex multi-component interactions.

As described in [18] and [23], 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 [21]. 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 [1] 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.'

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 [6].

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.' [16]. 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 [13, 22, 20, 5, 14]: *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*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ISEC '14, February 19-21 2014, Chennai, India
Copyright 2014 ACM 978-1-4503-2776-3/14/02 ...$15.00
`http://dx.doi.org/10.1145/2590748.2590756`.

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 [28] while other leading examples include the Open Group Architecture Framework (TOGAF) [24] and the framework promulgated by the Department of Defense (DoDAF) [27]. 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 [17].

Enterprise Architectures are built to support *use-cases* related to managing and evolving an organization. For example, *directive development* is concerned with developing directives that express *how* a business operates; *business intelligence* describes how a CEO is informed of the state of the organization at any level; *resource planning* involves the allocation of business resources to processes; *impact analysis* covers a variety of analyses used to measure the effect a proposed change has on an organization; *change management* involves describing the context and requirements for changes in any aspect of the business, including the construction of *as-is* and *to-be* analysis and the calculation of the return on investment (ROI) for any proposed change; *regulatory compliance checking* establishes that an organization meets some externally imposed constraints on its operating procedures; *risk analysis* identifies dangers, both internal and external, that can affect the successful operation of the organization; *acquisition and merger* involves the comparison of two organizations to identify their similarities and differences with respect to achieving a goal; *outsourcing* involves the identification of services that can be supplied by an external partner. Supporting these use cases is challenging and requires models that accurately describe relevant aspects of an organization at an appropriate abstraction level.

2. PROBLEM AND CONTRIBUTION

An important EA use-case identified in the previous section is *outsourcing*. This has become increasingly popular across many sectors where it is difficult for an individual firm to master all the knowledge required to perform all business functions [29]. According to [19] outsourcing decisions are taken during the development of a new system architecture and modularity is a key enabler for these decisions: 'A system producer basically faces two alternatives to manage the development of its components: in-house development or outsourcing.'

Business Transformation involves changing the current processes and resources used by a business in order to achieve a goal. Outsourcing is an example of a transformation that must identify those elements of a business that can be given to a service provider. The transformation removes the elements of the business that are no longer required.

Achieving an outsourced business function through business transformation involves a precise understanding of how the *as-is* business operates and producing a *to-be* business that incorporates the service provider. Component-based techniques can help with this since the service provider can be viewed as a component incorporated within the business ecosystem. However, this approach relies on a precise representation of the business as a collection of components and the ability to decompose and refactor the components in order to isolate the service provider.

Matching internal service needs with those provided by an external service is reminiscent of some of the earlier work on component re-use and repositories where research such as that by Cheng et al [7] and Jeng et al [15] under the direction of Betty Cheng presented approaches of using formal specifications utilising pre and post condition specification of operations on components to attempt to perform matching of required components with those stored in external repositories.

Transformation and refactoring approaches to component-based systems are often based on an analysis of the interfaces or require detailed understanding of programming-language semantics [4, 26]. However, if we are to work at the level of abstraction required by EA and achieve outsourcing through component-based refactoring, then there must be a precise, but implementation independent language that supports approaches such as SOA and EDA without requiring detailed knowledge of message protocols or run-time platforms.

Our hypothesis is that it is possible to take a component-based approach to outsourcing in terms of precisely defined decomposition operators that isolate the service provider through transformation and refinement. Our primary contribution is to define this approach using μLEAP which is a simple, abstract, executable component language and to show that the approach can be used to outsource a simple service. μLEAP represents a new refinement of our existing LEAP technology by the embedding of μLEAP as a domain specific language in the LISP based platform Racket[1].

3. CASE STUDY

The University of Middle England (UME) is under pressure to reduce costs. It has been in contact with a number of service providers in the UK Higher Education sector and has found a company that offers a service to manage registration and tuition fees for students.

Unfortunately, UME currently distributes student registration information around the institution. An academic department holds information about the fees for the individual courses that it offers, and also holds information about whether a student has paid the tuition fees. The UME finance department also manages information about whether students have paid their fees. Although the finance function knows about courses and departments, they do not currently hold any information about tuition fees which are set at departmental level. Each academic department also manages a list of active staff members, whilst the finance department knows about the staff grades for all members of department that have ever been employed by UME.

UME would like to modify its internal architecture so that it can take advantage of the service provider. This will involve changing how both departments and finance operates so that the provider manages registration and payment of tuition fees, informing a department when each student can officially start to study. Payroll, is to remain as a collaboration between the academic departments and finance.

4. μLEAP

The language LEAP and its associated toolset [9, 2, 11, 10, 8] has been developed to support the design, analysis and simulation of component-based systems. The LEAP approach aims to reduce such systems to a small number of orthogonal features. The full LEAP language contains

[1]`http://racket-lang.org`

```
p =                               programs           r =              rules
    d ... e ...                                        (rule
d =                               definitions            (when p p)
    (def name e)                                         (become e e)
|   (def name (name ...) e)                              e ...)
e =                               expressions        p =              patterns
    const                                                (= name p)
|   var                                              |   name
|   ()                                               |   _
|   (component e r ... e ...)                        |   const
|   (Name e ...)                                     |   ()
|   (fun (name ...) e)                               |   (p . p)
|   (block d ... e ...)                              |   (p ...)
|   (e e ...)                                        |   (∈ p p)
|   (@ e e ...)                                      |   (union p p)
|   (case e (p e) ...)
```

Figure 1: μLEAP Syntax

many features that enable the language to integrate with a tool-set, including the ability to produce diagrams directly from the LEAP component models.

This paper uses our previous experience with LEAP as a platform for validating our hypothesis that outsourcing can be achieved using a compositional and transformational component based approach. As such we do not need the complete LEAP language and therefore we use a sub-set called μLEAP as defined in figure 1. The rest of this section provides an overview of μLEAP and the remainder of paper uses the language to implement and analyse the case study.

μLEAP data items are: constants (numbers, strings, bools), lists, terms, functions and components. Boolean constants are **tt** and **ff**. μLEAP is embedded as a domain specific language in Racket and uses Lisp-style lists consisting of the empty list `()` and cons pairs `(h . t)`. A term `(F v ...)` consists of a *functor*, `F`, which is a name starting with a capital letter, followed by a sequence of values. The following is a term representing a person with an age and a name: `(Person 34 "fred")`. A function `(fun (x y) (+ x y))` is a value that can be applied to a collection of args and returns a value.

The key data value in μLEAP is the *component*. Components behave as independently executing state transition machines. The state of a component can be any value and the transitions are defined by a collection of rules. Messages sent to a component are added to an internal message queue. Each rule uses pattern matching against the current state of the component and its message queue. Each time the component changes, the rules are examined in turn. The first rule that matches, *fires* producing a new component-state and message queue and potentially sending messages to other components.

Typically, component messages are terms where the term-functor corresponds to the name of the message and the term-elements are data items passed in the message. Rules examine the head of the message queue. The following is a component that is initialised with a starting integer, a limit and a target component. It is sent `Inc` messages that increments the integer until the limit is reached at which point the target component is sent a `Go` message and before re-initialising:

```
1 (component (list 0 limit target)
2   (rule
3     (when (limit limit target) ((Inc) . messages))
4     (become (list 0 limit target) messages)
5     (target (Go)))
6   (rule
7     (when (current limit target) ((Inc) . messages))
8     (become (list (+ current 1) limit target) messages)))
```

The program shown above provides examples of several key μLEAP features. The state of the component is a list of three elements. Line 1 uses the built-in function `list` to construct a list of elements. A 3-element list is constructed as `(list x y z)` which is equivalent to the expression: `(: 1 (: 2 (: 3 ())))`, where `:` is the function that maps two values (typically a list-head and a list-tail) to a pair.

Patterns are used in rules and in **case**-expressions. A pattern matches a value and may contain variables that are bound to sub-values as a result of a successful match. Rules use patterns in the **when** clause to match against the current state and message queue of a component. A **case**-expression matches a value against a sequence of patterns and evaluates the expression associated with the pattern. Structured patterns match pairs, and lists, The pattern `(union p q)` treats a list `l` as a set and matches `p` and `q` against any pair of exhaustive set partitions of `l`. The membership pattern `(∈ p q)` is equivalent to `(union (p) q)`.

The component defines two rules (lines 2 and 6). The first rule has a guard on line 3 that matches against the current state and message queue. The state-pattern must match a list of three elements where the first two elements are the same *i.e.*, the limit has been reached. The message-pattern matches a list of elements headed by a term whose functor is `Inc` and without term-elements. The tail of the list is any value and is matched against the variable `messages`.

The first rule performs a transition to a new component state involving a list of three elements whose first element is `0` (the component has re-initialised), and where the message has been consumed. The action performed by the rule (line 5) sends a message `(Go)` to the component `target`. Messages are sent to components by applying them to a sequence of arguments. Message passing may be *synchronous* or *asynchronous*. Asynchronous message passing starts with **@**, as in `(@ target (Go))`. In that case, the message is sent and the expression immediately returns the value `nothing`. A synchronous message omits the **@**, sends the message to the target component and waits for the return value. The target component will handle the message using its own rules, and the return value is provided by the corresponding rule-action. The second rule (line 6) matches when the limit has not been reached. In this case the current value is incremented and no action is taken.

Both components and functions are first-class data values in μLEAP. This feature is important since it used to implement a component combinator $\oplus$ that allows *as-is* architectures to be decomposed into a collection of atomic components that can be re-factored in order to identify and isolate a service provider. As a result, μLEAP has the flavour of a functional programming language and can support many of the patterns that are typical of that idiom. For example:

```
(def map (f l)
  (case l
    (() ())
    ((h . t)
     (: (f h)
        (map f t)))))
```

is a function that sends each message in the list `l` to the function (or component) `f`, building a list of results. An asynchronous version of this is:

```
(def @map (f l)
  (case l
    (() (block))
    ((h . t)
     (block
      (@ f h)
      (@map f t)))))
```

The function `map` is used in the case study below, as is the function `find` that uses a predicate `p` to select an element in a list:

```
(def find (p l)
  (case l
    (() (error "no element"))
    ((h . t)
     (case (p h)
       (tt h)
       (ff (find p t))))))
```

5. REFACTORING APPROACH

Our proposal is that we can use a flexible component-based approach to decompose and refactor an architecture in order to isolate a component that corresponds to a service provider. Typically an *as-is* architecture will contain large complex components where information and behaviour are distributed, and possible duplicated, amongst many different components. Therefore, any approach must allow large complex components to be decomposed and represented as a composition of much simpler components that can be re-organised.

Consider a business that operates a software component that manages two counters:

```
(def business
  (component (list n m)
    (rule
      (when (n m) ((IncN) . ms))
      (become (list (+ n 1) m) ms))
    (rule
      (when (n m) ((DecM) . ms))
      (become (list n (- m 1)) ms))))
```

Suppose that a service provider offers an interface that increments numbers at a very reasonable price. The business would like to outsource that portion of its business that increments the counter whilst it retains that which decrements the counter. In this example the service provider component is obvious. Suppose that the binary operator $\oplus$ maps two components to a single combined component. Given such an operator, `business` can be redefined as:

```
(def internal
  (component m
    (when m ((DecM) . ms))
    (become (- m 1) ms)))

(def service
  (component n
    (when n ((IncN) . ms))
    (become (+ n 1) ms)))

(def business (⊕ internal service))
```

Given this basic idea, the approach is described as follows: Suppose that a business can be defined using a collection of components A, B and C and that there is a service provider D that roughly corresponds to B. Using infix notation and being clear about associativity, the business is: $A \oplus (B \oplus C)$. Notice that B is buried inside the expression, *i.e.*, the business. If, $\oplus$ is associative then the architecture can be redefined as: $(A \oplus B) \oplus C$. If $\oplus$ is commutative then the architecture is equivalent to: $(B \oplus A) \oplus C$. Now associativity can be used to isolate B: $B \oplus (A \oplus C)$. If we can define a correspondence ϕ between B and D, then it is possible to replace B by the service provider: $\phi(D) \oplus (A \oplus C)$.

In a realistic situation, it will be unlikely that existing components can be decomposed in such a straightforward manner. Further analysis leads to the following requirements for the composition operator $\oplus$:

[**R1**] A single component may send messages to itself. If a component `C` is decomposed into `D` $\oplus$ `E` then the independent components `D` and `E` must be able to refer to the whole.

[**R2**] It is likely that several sub-components will be removed when moving from an *as-is* architecture to a *to-be* involving a service provider. Therefore, the operator $\oplus$ should be both associative and commutative, or there should be variants that can be selective used as appropriate. For example if a component `A` can be decomposed into (`B` $\oplus$ `C`) $\oplus$ (`D` $\oplus$ `E`) where the service provider corresponds to `B` and `E`, then it should be possible to transform the expression into (`C` $\oplus$ `D`) $\oplus$ (`B` $\oplus$ `E`) thereby isolating the service provider.

[**R3**] The internal interfaces used by an organisation are likely to be different to those provided as a service. Since components and messages are first class data in μLEAP it is straightforward to rename messages. For example, given a component `C` with a message interface `(M)` that is to be replaced with a service component whose message interface is `(N)`, but otherwise has the same behaviour then we can wrap a renaming: `C[NM]` which is shorthand for:

```
(component ()
  (rule
    (when () ((N) . ms))
    (become () ms)
    (c (M))))
```

[**R4**] An organisation will consist of a collection of components and will execute in terms of message traces $m \in M$. The state of the organisation is represented by the aggregate state of the components Σ. If the organisation is broken down into finer grained components that are composed using $\oplus$ then the structure of the aggregate state is Σ' and the execution traces are $m' \in M'$. However, it should be possible to define a mapping $\phi : \Sigma' \to \Sigma$ such that executions can be mapped:

$$\begin{tikzcd}
\Sigma' \arrow[r, "m'"] \arrow[d, "\phi"'] & \Sigma' \arrow[d, "\phi"] \\
\Sigma \arrow[r, "\phi(m')"'] & \Sigma
\end{tikzcd}$$

If the diagram shown above commutes then it means that the execution of the *to-be* organisation is consistent with the *as-is* execution after re-namings, changes in state composition, and modifications to messages are taken into account.

The approach to outsourcing is to be supported by a range of operators that behave like $\oplus$ and _[__] with well understood properties. In our case study we use two operators that are defined below. In order to satisfy **R1**, it must be possible for each component-part to refer to the whole component. This is achieved by defining *pre-components* as functions that map a whole-component (`self`) to a component-part. The pre-components are combined using the operators as described below. Given a pre-component, it is transformed into a whole-component using `new`:

```
(def new (pre-component)
  (def whole-component (pre-component whole-component))
  whole-component)
```

The pre-component composition operator is defined as follows:

```
(def combine (v1 v2)
  (case (cons v1 v2)
    (((Fail) . (Fail)) (Fail))
    ((Fail) . v)         v)
    ((v      . (Fail))   v)
    ((_ . _)         (error)))

(def ⊕ (c1 c2)
  (fun (self)
    (block
      (def o1 (c1 self))
      (def o2 (c2 self))
      (component (list o1 o2)
        (rule
          (when s (m . ms))
          (become s ms)
          (combine (o1 m) (o2 m)))))))

(def ϵ (self)
  (component ()
    (rule () (m . ms))
    (become () ms)
    (Fail)))
```

The $\oplus$ operator combines pre-components that handle synchronous messages (*i.e.*, those that contain rules that return results. The function `combine` ensures that the two components `o1` and `o2` are independent and only one can return a result for a given message (or both fail). This ensures that the operator $\oplus$ is a *commutative monoid*:

$$X \oplus Y = Y \oplus X$$
$$X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$$
$$X \oplus \epsilon = X = \epsilon \oplus X$$

Although the case study in this paper uses $\oplus$, in practice there is likely to be a family of operators that exhibit different transformation properties. For example, we may combine two components that exclusively deal with asynchronous messages. In that case the following associative and commutative operator can be used:

```
(def ⊗ (c1 c2)
  (fun (self)
    (block
      (def o1 (c1 self))
      (def o2 (c2 self))
      (component ()
        (rule
          (when s (m . ms))
          (become s ms)
          (@ o1 m)
          (@ o2 m))))))
```

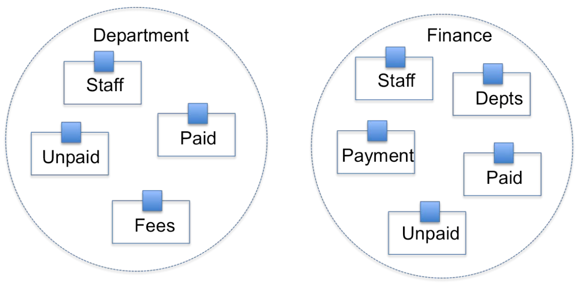

Figure 2: Overview of the *As-Is* Architecture

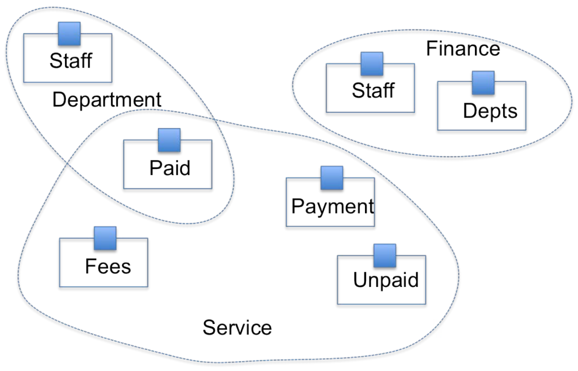

Figure 3: Overview of the *To-Be* Architecture

6. WORKED EXAMPLE

Section 3 describes a case study that involves two interacting components. The approach described in this paper is a method for representing a component architecture and subsequently transforming it in order to isolate a component corresponding to a service provider.

Figure 2 shows the decomposition of the finance and academic departments into separate aspects. The academic department manages staff, students registered on courses (with both paid and unpaid tuition fees) and the cost of the fees for each course. The finance department manages all staff grades, students who have paid and the process of payment.

Once the aspects of each major component have been identified, the approach creates a *to-be* architecture as shown in figure 9 where the aspects have been reallocated. Some aspects are handled by the service provider and have been removed from the *as-is* component. Others, such as recording students as having paid for their courses are shared between the service provider and the business.

Section 6.1 uses μLEAP to represent the *as-is* case study architecture. Section 6.2 describes the service provider as a μLEAP component and section 6.3 uses the approach to

```
(def department
 (component
  (list
   (Staff (list (Lecturer "Dr Piercemuller")))
  (Courses (list (Course "computer science" 9000 ()))))
  (rule
   (when ((Staff ls) cs) ((GetStaff) . ms))
   (become (list (Staff ls) cs) ms)
   ls)
  (rule
   (when
    (ls (Courses (∈ (Course c f ss) cs)))
    ((Register s c) . ms))
   (become
    (list
     ls
     (Courses (: (Course c f (: (Student s ff) ss)) cs)))
     ms))
  (rule
   (when
    (ls (= cs (Courses (∈ (Course _ _ (∈ (Student s _) _)) _))))
    ((HasStudent s) . ms))
   (become (list ls cs) ms)
   tt)
  (rule
   (when s ((HasStudent _) . ms))
   (become s ms)
   ff)
  (rule
   (when
    (ls (= cs (Courses (∈ (Course c _ (∈ (Student s _) _)) _))))
    ((RegisteredFor s) . ms))
    (become (list ls cs) ms)
    c)
  (rule
   (when
    (staff (= cs (Courses (∈ (Course c f _) _))))
    ((GetFee c) . ms))
   (become (list ls cs) ms)
   f)
  (rule
   (when
    (ls (Courses (∈ (Course c f (∈ (Student s _) ss)) cs)))
    ((Paid s p) . ms))
   (become
    (list
     ls
     (Courses (: (Course c f (: (Student s p) ss)) cs)))
     ms))))
```

Figure 4: As-Is Department Component

apply a step-wise transformation to the *as-is* architecture in order to produce a *to-be* architecture containing the service provider component.

6.1 As-Is Architecture

The current system is implemented using two components. The first component is used to manage a department within the university and the second deals with finance. This section gives a simple implementation of both components using μLEAP.

Figure 4 shows a component `department` that manages a department. The state of the component consists of a list of lecturers and a list of courses. Each course is a term of the form `(Course name fees students)` where `students` is a list of student terms of the form `(Student name paid?)`.

The department component defines rules that process the following messages:

`(GetStaff)` that returns the list of lecturers in the department.

`(Register student course)` that adds a student record to the appropriate course. The student is marked as having fees outstanding.

`(HasStudent name)` returns true when the department has a named student.

`(RegisteredFor student)` returns the name of the course that the named student is studying.

`(GetFee course)` returns the fee associated with the named course.

`(Paid student paid?)` informs the department that the student has paid something towards their tuition fees. The boolean value `paid?` determines whether the fees are full paid or not.

```
(def finance
 (component
  (list
   (Departments (list department))
   (Staff (list (Grade "Dr Piercemuller" 10)))
   (Students ())
   (Courses (list (Course "computer science" department))))
  (rule
   (when ((Departments ds) gs ss cs) ((Payroll) . ms))
   (become (list (Departments ds) gs ss cs) ms)
   (map
     (fun (d)
       (map (fun (l) (finance (Pay l))) (d (GetStaff))))
     ds))
  (rule
   (when
    (ds (Staff (∈ (Grade n g) ls)) ss cs)
    ((Pay (Lecturer n)) . ms))
   (become
    (list ds (Staff (: (Grade n g) gs)) ss cs) ms)
   ; make payment
   )
  (rule
   (when (ds gs (Students ss) cs) ((Register s) . ms))
   (become
    (list ds gs (Students (: (Student s ff) ss)) cs) ms))
  (rule
   (when
    ((Departments ds) gs (Students (∈ (Student n _) ss)) cs)
    ((Pay n a) . ms))
   (become
    (list
     (Departments ds)
     gs
     (Students (: (Student n (>= a f)) ss))
     cs)
    ms)
   (def ds (find (fun (d) (d (HasStudent n))) ds))
   (def course (department (RegisteredFor n)))
   (def fee (department (GetFee course)))
   (department (Paid n (>= a f))))))
```

Figure 5: As-Is Finance Component

Figure 5 shows the finance component of the university. The state consists of four elements: (1) a list of the departments in the university; (2) a list of the all university staff and their job-grades; (3) a list of student records of the form `(Student name paid?)`; (4) a list of terms that associate course names with departments.

The finance component has an interface that handles the following messages:

`(Payroll)` which causes all of the staff of the university to be paid. This involves iterating over all of the departments, getting the staff in each department, looking

```
(def provider
 (component
  (list
   (Courses (list (Course "computer science" department 9000)))
   (Students ()))
  (rule
   (when (cs (Students ss)) ((Register n c) . ms))
   (become (list cs (Students (: (Student n c ff) ss))) ms))
  (rule
   (when ((= cs (Courses (∈ (Course c d f) _)))
          (Students (∈ (Student s c _) ss)))
         ((Pay s a) . ms))
   (become
    (list cs (Students (: (Student s c (>= a f)) ss))) ms)
    (case (>= a f)
     (tt (d (Register s c)))))))
```

Figure 6: Service Provider

up their job-grade and making an payment based on the grade.

`(Pay lecturer)` looks up the job-grade of the lecturer and makes the payment.

`(Register student)` adds a new student record. The student is marked as having tuition fees outstanding.

`(Pay student amount)` informs finance that the student has made a payment. The tuition fee is requested from the appropriate department and the student record is updated accordingly.

6.2 Service Provider

Figure 6 shows an implementation of the service provider component. The provider manages the financial aspect of student registration and therefore manages a state that is a list of course and student terms. A course term has the form `(Course name department fees)` and a student has the form `(Student name course paid?)`. When a student registers with the provider they are marked as owing tuition fees and when the pay the correct amount their status changes, and the appropriate department is informed.

6.3 Transformation

Our proposition is that we can take the *as-is* architecture described in section 6.1, decompose it using the operator ⊕, transform the resulting tree of pre-components and then show that the service provider defined in 6 can be isolated in the *to-be* architecture.

The first step is to decompose the *as-is* architecture. Consider the `department` component. It consists of two different aspects: *staff* and *courses.* The staff can be defined as a separate pre-component:

```
(def department-staff (self)
 (component
  (list (Lecturer "Dr Piercemuller"))
  (rule
   (when staff ((GetStaff) . ms))
   (become staff ms)
   staff)
  (rule
   (when s (m . ms))
   (become s ms)
   (Fail))))
```

From now on the rule that produces `(Fail)` will be omitted from pre-components since it is always the last rule. The courses information is slightly more structured since it contains two aspects, the tuition fees and the students. Therefore the tuition fees are identified as a separate pre-component:

```
(def department-courses-fees (self)
 (component (list (Course "computer science" 9000))
  (rule
   (when
    (= courses (∈ (Course name fee) _)) ((GetFee name) . ms))
   (become (list courses) ms)
   fee)))
```

The students information relates to students that have paid their fees and those that have not. Therefore we can separate these issues out. The students that have paid their fees are maintained by a pre-component that manages a list of courses. Each course contains a list of students who have paid their tuition fees:

```
(def department-paid-students (self)
 (component (list (Course "computer science" ()))
  (rule
   (when (∈ (Course c ss) cs) ((AddPaid s c) . ms))
   (become (: (Course c (: (Student s) ss)) cs) ms))
  (rule
   (when
    (∈ (Course c (∈ (Student s) ss)) cs)
    ((HasPaidStudent s) . ms))
   (become (: (Course c (: (Student s) ss)) cs) ms)
   tt)
  (rule
   (when cs ((HasPaidStudent s) . ms))
   (become cs ms)
   ff)
  (rule
   (when
    (∈ (Course c (∈ (Student s) ss)) cs)
    ((RegisteredFor s) . ms))
   (become (: (Course c (: (Student s) ss)) cs) ms)
   c)))
```

The pre-component `department-unpaid-students` is virtually the same as that shown above except it handles additional messages for `(HasUnpaidStudent)` and `(RemoveUnpaid)`.

Having defined all of the pre-components it is possible to compose them to produce a single component for a department. Firstly, a pre-component `department-students` that manages students is defined as follows:

```
(def student-extension
  (fun (self)
    (component ()
      (rule
        (when s ((Paid student) . ms))
        (become s ms)
        (self (AddPaid student (self (RegisteredFor student))))
        (self (RemoveUnpaid student)))
      (rule
        (when s ((HasStudent name) . ms))
        (become s ms)
        (or (self (HasPaidStudent name))
            (self (HasUnpaidStudent name)))))))

(def department-students
  (⊕ (⊕ department-paid-students department-unpaid-students)
     student-extension))
```

Next, the courses and staff can be added:

```
(def pre-department
  (⊕ department-staff
     (⊕ department-courses-fees department-students)))
```

Finally, a department is created by:

```
(def department (new pre-department))
```

The finance component is decomposed into pre-components as shown in figure 7.

```
(def finance-departments (self)
 (component
  (list department)
  (rule
   (when ds ((GetDepartments) . ms))
   (become ds ms)
   ds)))

(def finance-staff (self)
 (component
  (list (Grade "Dr Piercemuller" 10))
   (rule
    (when gs ((Payroll) . ms))
    (become gs ms)
    (map
     (fun (d)
      (map (fun (l) (self (Pay l))) (d (GetStaff))))
     (self (GetDepartments))))
   (rule
    (when (∈ (Grade n g) gs) ((Pay (Lecturer n)) . ms))
    (become (: (Grade n g) gs) ms))))

(def finance-students-paid (self)
 (component ()
  (rule
   (when ss ((AddPaid s) . ms))
   (become (: (Student s) ss) ms))))

(def finance-students-unpaid (self)
 (component ()
  (rule
   (when ss ((Register s) . ms))
   (become (: (Student s) ss) ms))
  (rule
   (when (∈ (Student s) ss) ((RemoveUnpaid s) . ms))
   (become ss ms))))

(def finance-payment (self)
 (component ()
  (rule
   (when () ((Pay n a) . ms))
   (become () ms)
   (def d (find (fun (d) (d (HasStudent n)))
             (self (GetDepartments))))
   (def c (d (RegisteredFor n)))
   (def f (d (GetFee c)))
   (case (>= a f)
    (tt (d (Paid n)))))))

(def pre-finance
  (⊕  (⊕ finance-staff finance-departments)
      (⊕ finance-payment
         (⊕ finance-students-paid finance-students-unpaid)))))

(def finance (new pre-finance))
```

Figure 7: Finance Decomposition

6.4 Outsourcing

The previous section has decomposed the *as-is* architecture as a collection of components combined using $\oplus$. We can use the properties of these operators to transform the pre-components in order to isolate the service provider. It is unlikely that the transformations will produce a component that is in one-to-one correspondence with the service provision component, however a mapping ϕ can be defined between the states of the isolated pre-component and the service provider component and used to to establish equivalence.

The first step is to transform the pre-components. To make this more concise, we rename the pre-components as shown in figure 8. Therefore, a department pre-component is:

$$A \oplus (B \oplus ((C \oplus D) \oplus E)$$

A	`department-staff`
B	`department-courses-fees`
C	`department-paid-students`
D	`department-unpaid-students`
E	`student-extension`
F	`finance-staff`
G	`finance-departments`
H	`finance-payment`
I	`finance-students-paid`
J	`finance-students-unpaid`

Figure 8: Pre-Component Labels

Given the properties of $\oplus$ we can transform this expression into the following pre-component:

$$(B \oplus (D \oplus E)) \oplus (A \oplus C)$$

The finance department pre-component is defined as follows:

$$((F \oplus G) \oplus (H \oplus (I \oplus J)))$$

Again, using the properties of the pre-component combination operator the expression can be transformed into:

$$(F \oplus G) \oplus (H \oplus (I \oplus J))$$

Through transformation, we have isolated two elements of pre-components that can be combined to produce a new component: $(B \oplus (D \oplus E)) \oplus (F \oplus G)$, producing the following system defined by three pre-components instead of two:

$$\begin{array}{lcl} \texttt{pre-department} & = & (A \oplus C) \\ \texttt{pre-finance} & = & (F \oplus G) \\ \texttt{pre-service} & = & (B \oplus (D \oplus E)) \oplus (H \oplus (I \oplus J)) \end{array}$$

We must now establish that the new system configuration has equivalent behaviour to the *as-is* architecture and that the `pre-service` component captures the behaviour of the service provider as defined in figure 6.

Equivalence is established by defining state mappings preserve the behaviour between systems. Since μLEAP states and messages are explicitly represented in each component, equivalence can be established through inspection of the definitions and reasoned argument. The state of the *to-be* architecture has three elements corresponding to the separate components. The form of the state is given as type signatures.

```
1 ((Lecturer String)... (Course String (Student String)...))
2 ((Grade String Integer)... Department ...)
3 ((Course String Integer)... (Course String (Student String)...)...)
```

A *to-be* department (1) has a state containing lecturers and courses with paid-up students. A *to-be* finance component (2) manages the staff and their grades and contains a collection of departments. The *to-be* model of the service-provider has a state (3) containing the same course occurring in three different aspects: fees, paid students and unpaid students.

Taken as a whole, the *to-be* state can be mapped to the *as-is* state. For example, the paid and non-paid students that are now managed by the service provider can be mapped back to the states in the finance and academic departments. Therefore, no information has been lost. Furthermore, if we exercise the *to-be* system by processing messages, we find that the state changes are consistent with respect to the mapping.

```
(def department
 (component
  (list
   (Staff (list (Lecturer "Dr Piercemuller")))
   (Courses (list (Course "computer science" ()))))
  (rule
   (when ((Staff ls) cs) ((GetStaff) . ms))
   (become (list (Staff ls) cs) ms)
   staff)
  (rule
   (when
    (ls (Courses (∈ (Course c ss) cs))) ((Register s c) . ms))
   (become
    (list
      ls
      (Courses (: (Course c (: (Student s) ss)) cs))) ms))))

(def finance
 (component
  (list
   (Departments (list department))
   (Staff (list (Grade "Dr Piercemuller" 10)))
  (rule
   (when ((Departments ds) gs) ((Payroll) . ms))
   (become (list (Departments ds) gs) ms)
   (map
    (fun (d)
     (map
      (fun (l) (finance (Pay l)))
     (d (GetStaff))))
   ds))
  (rule
   (when
    (ds (Staff (∈ (Grade n g) gs))) ((Pay (Lecturer n)) . ms))
   (become (list ds (Staff (: (Grade n g) gs))) ms))))
```

Figure 9: To-Be Architecture

If we now reverse the decomposition based on the definitions of `pre-department` and `pre-finance` that have been achieved by transformation and refinement above, the *to-be* architecture is produced as shown in figures 9 and 6, with components `department`, `finance` and `provider` as required.

7. ANALYSIS AND CONCLUSION

This paper has identified a use-case of component-based systems and Enterprise Architecture in particular: organizational transformation in order to outsource business functionality. Such a use-case is intrinsic to any notion of business and IT alignment as the requirement to move from an *as-is* architecture to an *to-be* architecture leads to a problem: how to analyse and transform the architecture in order to achieve confidence that the resulting business, based on the new service, is equivalent to the existing business.

Our contribution is to identify and implement a process for achieving outsourced services that is based on component decomposition and transformation. Our claim is validated by implementing the process using an abstract, higher-order, executable component language μLEAP and using the implementation to address a simple case study. The process requires more detailed elaboration outside the scope of this paper, for example to include discussions on how candidate components for outsourcing may be identified. For example, outsourcing any operation needs to be done in a business context of strategy and goals of an organisation. Our previous work provides an indication of such a direction of travel [11].

Whilst we have shown that the approach can be implemented, the case study is the basis for much further work. μLEAP is executable and therefore has an operational semantics, and all of the examples shown in the paper have been implemented as μLEAP executable models. The models have been run against test data that supports the claims that have been made for the architectural transformations. A precise analysis of component-based architectures is likely to require a more declarative semantics than that presented here; the form of the semantics and the ability to use it as a basis for reasoning about architecture is left as further work.

In addition, μLEAP lacks a type system that would help to validate claims of component equivalence. As identified earlier in this paper, the $\oplus$ pre-component composition operator is likely to be one of many and therefore more elaborate case studies will be required in order to identify alternatives.

This paper presents a technological basis for an approach, as such it lacks a methodological framework within which the technology can be used. Given an as-is enterprise, there must be some guidance regarding its representation as a μLEAP model. Existing component-based approaches might be used here, however given the aim of using refinement and transformation using operators, the subsequent identification and isolation of an outsourced component is likely to be facilitated by judicious choice of representation for as-is structures. Our hypothesis is that use-cases can be used to identify that *slice* of an organisation that is pertinent to a collection of related business functions and that the resulting collection of information and behaviour can be represented as a single μLEAP component and subsequently refined into multiple components that is in correspondence with the as-is internal structure.

Although the approach as described here is textual, it could be supported by graphical languages such as UML where a high-level view of an organization is described as a collection of components via diagrams and μLEAP is used to define the internal details for simulation. Such an approach would require a UML profile to allow components to be expressed as a combination of sub-models using the operators defined in this paper.

Our case study has been used to validate the technology but lacks validation through results that establish its practicality. For example, without guidance, how easy is it to get an organisation of components that makes subsequent transformation difficult or impossible? Given the size of an organisation, is the technology too detailed, leading to problems of maintenance and availability of expertise? These areas are left for further work.

8. REFERENCES

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

[2] Balbir S. Barn and Tony Clark. Goal based alignment of enterprise architectures. In Slimane Hammoudi, Marten van Sinderen, and José Cordeiro, editors, *ICSOFT*, pages 230–236. SciTePress, 2012.

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

[4] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

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

[6] A. Buchmann and B. Koldehofe. Complex event processing. *IT-Information Technology*, 51(5):241–242, 2009.

[7] Yonghao Chen and Betty HC Cheng. Formalizing and automating component reuse. In *Tools with Artificial Intelligence, 1997. Proceedings., Ninth IEEE International Conference on*, pages 94–101. IEEE, 1997.

[8] T. Clark, B. Barn, and S. Oussena. LEAP: a precise lightweight framework for enterprise architecture. In Arun Bahulkar, K. Kesavasamy, T. V. Prabhakar, and Gautam Shroff, editors, *ISEC*, pages 85–94. ACM, 2011.

[9] Tony Clark and Balbir Barn. Goal driven architecture development using leap. *Enterprise Modelling and Information Systems Architectures*, 8(1):40–61, 2013.

[10] Tony Clark and Balbir S. Barn. A common basis for modelling service-oriented and event-driven architecture. In Sanjeev K. Aggarwal, T. V. Prabhakar, Vasudeva Varma, and Srinivas Padmanabhuni, editors, *ISEC*, pages 23–32. ACM, 2012.

[11] Tony Clark, Balbir S. Barn, and Samia Oussena. A method for enterprise architecture alignment. In Erik Proper, Khaled Gaaloul, Frank Harmsen, and Stanislaw Wrycza, editors, *PRET*, volume 120 of *Lecture Notes in Business Information Processing*, pages 48–76. Springer, 2012.

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

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

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

[15] Jun-Jang Jeng and Betty HC Cheng. Specification matching for software reuse: a foundation. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 97–105. ACM, 1995.

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

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

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

[19] Juliana Hsuan Mikkola. Modularity, component outsourcing, and inter-firm learning. *R&D Management*, 33(4):439–454, 2003.

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

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

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

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

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

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

[26] Hironori Washizaki and Yoshiaki Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer Programming*, 56(1-2):99 – 116, 2005. New Software Composition Concepts.

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

[28] J. Zachman. A framework for information systems architecture. *IBM systems journal*, 38(2/3), 1999.

[29] Francesco Zirpoli and Markus C Becker. The limits of design and engineering outsourcing: performance integration and the unfulfilled promises of modularity. *R&d Management*, 41(1):21–43, 2011.