Computers and Electrical Engineering 39 (2013) 2400–2412

Contents lists available at SciVerse ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Intention-oriented programming support for runtime adaptive autonomic cloud-based applications ☆

Thar Baker [a], Michael Mackay [a], Martin Randles [a], Azzelarabe Taleb-Bendiab [b,*]

[a] School of Computing and Mathematical Sciences, Liverpool John Moores University, Liverpool, UK
[b] School of Computer and Security Science, Edith Cowan University, Perth, Australia

ARTICLE INFO

Article history:
Available online 23 May 2013

ABSTRACT

The continuing high rate of advances in information and communication systems technology creates many new commercial opportunities but also engenders a range of new technical challenges around maximising systems' dependability, availability, adaptability, and auditability. These challenges are under active research, with notable progress made in the support for dependable software design and management. Runtime support, however, is still in its infancy and requires further research. This paper focuses on a requirements model for the runtime execution and control of an intention-oriented Cloud-Based Application. Thus, a novel requirements modelling process referred to as Provision, Assurance and Auditing, and an associated framework are defined and developed where a given system's non/functional requirements are modelled in terms of intentions and encoded in a standard open mark-up language. An autonomic intention-oriented programming model, using the Neptune language, then handles its deployment and execution.

1. Introduction

The high availability of computing resources is a major requirement in modern systems. To achieve such extremely high availability, systems need to be able to manage and optimise themselves in a broad range of instances; allowing availability with no runtime breaks or offline maintenance. These systems are said to be Eternal Systems (ESs); and the facilitation of such systems is a long-term research goal [1]. Additionally, computation is increasingly becoming highly distributed through large-scale service-oriented computing such as the Internet of Services (IoSs), Internet of Things (IoTs), and Internet of Contents (IoCs) [2]; and also Cloud Computing (CC) [3]. These computing paradigms aim to consistently deliver resources anywhere at anytime. An essential feature of a cloud-based system is its adaptability to changes in user requirements and infrastructure properties [4–6]. Such dynamic capabilities are often derived from an architecture model that describes the application components, services and their interactions, the properties and policies that regulate the composition of the components, and the limitations on the allowable range of control operations.

Such systems are becoming an essential part of the modern socio-economic fabric to such an extent that we are now relying on them for many day-to-day computational services. On the other hand, it has also engendered a range of technical challenges such as how to ensure the systems' and services' availability, adaptability, manageability, assurance and auditability; many of these challenges are under active research. For instance, Bieber and Carpenter in [7] advocate for the shift to a ser-

☆ Reviews processed and recommended for publication to Editor-in-Chief by Associate Editor Dr. Taeshik Shon.
* Corresponding author.
E-mail addresses: t.baker@ljmu.ac.uk (T. Baker), m.i.mackay@ljmu.ac.uk (M. Mackay), m.j.randles@ljmu.ac.uk (M. Randles), a.taleb-bendiab@ecu.edu.au (A. Taleb-Bendiab).

0045-7906/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compeleceng.2013.04.019

vice-oriented software development approach, while [6] discusses the necessity of providing self-management for the virtual resources to run on top of the physical system components, e.g. self-managed services in a cloud computing environment provide continuous management to ensure that user requirements are always met [8]. Alternatively others are focusing on the design principles for dependable software using approaches ranging from conventional software engineering and model-driven development [9], to new nature-inspired methods such as Autonomic Computing [10]. Whilst much progress has been made towards design-time support for dependable software design and management, runtime support requires further research.

This forms the motivation and context for the work presented in this paper, which studies runtime autonomic design principles to support the cloud-based eternal system vision. Specifically, Section 3 focuses on the runtime execution and control of an intention-oriented Cloud-Based Application (CBA) requirements model and an identified set of functional requirements which have been designed into a novel requirements modelling process referred to as Provision, Assurance and Auditing (PAA) [4]. In addition, in Section 4, an associated framework has been defined and developed where a given system's non/functional requirements are first modeled in terms of intentions and then encoded in a standard open mark-up language. An autonomic intention-oriented programming model, using the Neptune language [10], then handles its deployment and execution. Section 5 evaluates the PAA approach; this will be followed by the results in Section 6 and the conclusion in Section 7.

2. Related work

2.1. CBA requirements description languages

Whilst most software applications are designed for a particular purpose, CBAs are, additionally, designed to be utilised on-demand. Thus, CBAs ought to be translated into a set of users/systems *requirements* and *resources*, to manage the de/allocation of these resources based on the requirements. This was the inspiration behind the Distributed Application Description Language (DADL) [11], which is used to describe the behaviour and needs of distributed applications that are used via a cloud-computing infrastructure. It also defines the available resources, and what can be optimally allocated to the users. It is an extension to the configuring and deployment features of the SmartFrog framework [12], which provides orchestration capabilities to start and stop sub-systems and resources automatically.

Durra [13] is another description language, which is utilised to link tasks to available resources, process the output and makes a decision accordingly; this is not available in DADL or SmartFrog. However, since this language requires high level programming skills, it is unsuitable for use by non-specialists. This was the incentive for proposing new development requirements' modelling approaches that can design and refine the requirements in terms of *why*, *how* and *what* the system has been developed for. Since a requirement itself can be seen as a goal, Goal-Oriented Requirements Engineering (GORE) [14] was proposed to tackle the above issues through different activities (e.g. goal elicitation, refinement, and analysis). However, there is a clear lack, in GORE, of features to tackle the '*why*' part of the requirements definition [14]. Thus, GORE has been supplemented by the Distributed Intentionality (i^*) modelling language [15], which pays more attention to this issue. i^* is used in the early phases of requirements definition to model agents' dependencies, e.g. a service depends on another service to fulfil specific tasks via a formal requirements definition language, Formal Tropos [16]. This is itself an extension to i^*, which includes architectural design requirements. In 1990, Knowledge Acquisition in autOmated Specification (KAOS) [14] emerged to combine the different levels of requirements modelling (Goal, Object, and Operation). It aimed to gain a formal description of the goals' conflicts that occur during the requirements definition processes rather than the system requirement itself. This led to the design of a more powerful requirements engineering tool called GRAIL [17] to support KAOS conceptually through a graphical editor. In other words, it represents an intermediate environment between the KAOS model and the final system, as shown in Fig. 1.

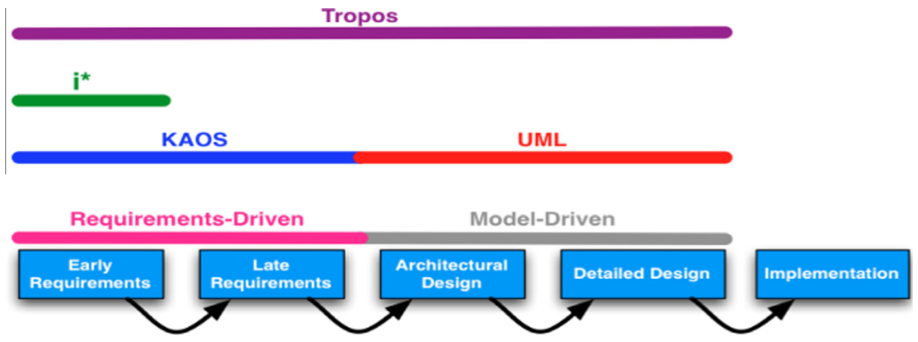

Fig. 1. System design phases vs. requirements engineering methods.

2.2. Cloud-of-clouds

Since the dynamic unlimited scale-up/down/out of resources can be achieved via the integration of different clouds, cloud-of-clouds [18], several cloud providers started using new tools to automate the process of managing and maintaining their systems as self-managed services [8]. The automated nature of the self-managed services in a cloud-of-clouds can be clearly seen when the cloud platform is self-adaptive and self-managed to accept new resources, with different sets of configurations and settings from other clouds. As such, self-managed services will enable cloud-of-clouds to possess exceptional capabilities to automatically manage and control the applications running on the cloud [6]. Aneka [19] is a cloud integration platform, which was developed to harness computing resources on demand by integrating several heterogeneous clouds, constructing what is known as interCloud [20]. It allows users to express their needs by using only the *pre-available* APIs in the Aneka Software Development Kit (SDK) or uploading *existing* applications to the cloud [19]; there is no other way of adding new components into the application. CometCloud has also been proposed as an autonomic computing engine for cloud environments [21], which is based on a decentralised XML based coordination form, to support a highly dynamic Cloud Computing infrastructure. It integrates public and private cloud networks to provide Cloud-bursting and Cloud-bridging such as CometPortal [21]. However, in the current version of CometCloud, the users cannot amend their changeable requirements and service demands within the cloud environment.

3. Conceptual design

In this section, a novel requirements' modelling process referred to as (PAA) and its associated framework are presented. The recently developed Neptune language [10] is used to generate the associated component compositions for the required application. Based on the idea of dynamic software evolution [3] – PAA is intended to facilitate seamless autonomic CBA evolution, and allows the user to modify the system's intention model via the PAA WikiEditor. The intention model, as shown in Fig. 2, consists of a set of processes that describe the desired business behaviour. Some of these processes, if not all, are modularised into sub-processes that include a set of tasks needed to achieve the requested behaviour. This is in contrast to Business Process Execution Language (BPEL), which does not yet support the sub-process notion [22]. The processes and tasks interpretation, via PAA Neptune Framework, will be described in more detail in Section 4.

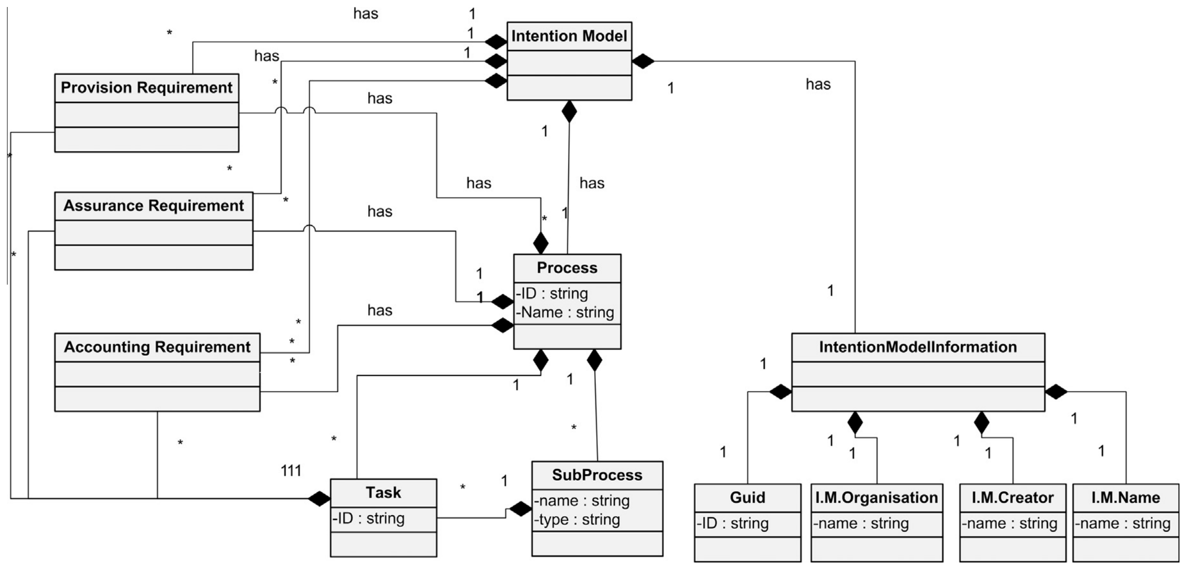

Fig. 2. Intention meta-model.

3.1. PAA framework

Fig. 3 shows the modules within PAA Neptune Framework, as well as a typical flow through the system, when presented with an intention model to be executed. Each module encapsulates its own discrete function so that the complete process can be made. When an intention is introduced to PAA Neptune Framework, there are two paths for the execution (*a* or *b*). In both ways, the first module receiving the intention model is the *Interpreter*. The interpreter is responsible for interpreting the

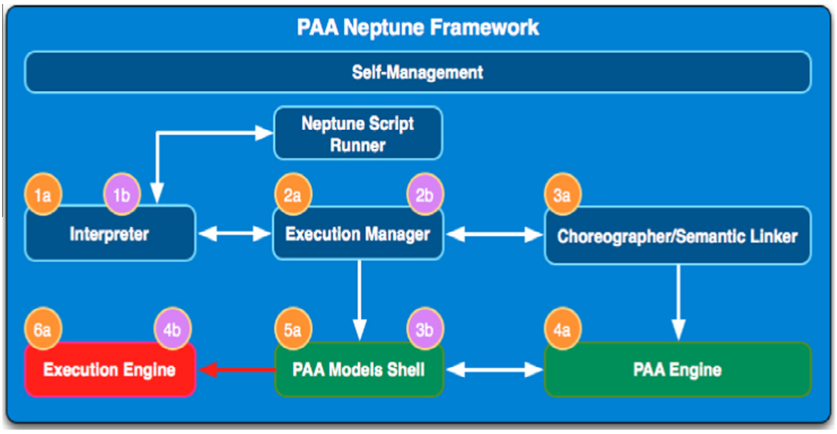

Fig. 3. PAA neptune framework.

intention and its components via two sub-modules: the *Process Interpreter* and *Logic Interpreter*. The process interpreter interprets the flow of the processes in the intention model (depending on the results gained from the logic interpreter) and the conditions via the logic interpreter. It then maintains the current state of the process in the intention model such that statements given in the logic model can be evaluated, and the flow through the model decided.

The interpreter also has a strong connection with the *Neptune Script Runner,* which is called by the interpreter when there is a scripting code in the intention model that should be executed to return results so the intention's task can make a decision. For example, if the user added a new Neptune function to extend certain behaviour for the application, then the runner will be called directly by the interpreter to execute the new function and return its results. Then it can resume the interpretation of the rest of the intention model using `parseOutput`, and `parseInput` methods. Fig. 4 shows the connection type between the runner and Neptune objects.

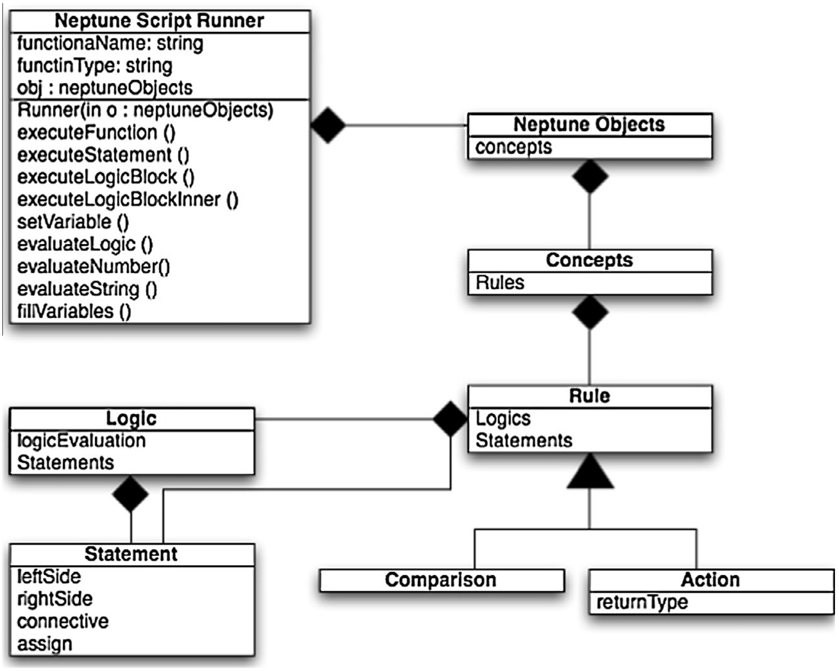

Fig. 4. Neptune script runner and objects model.

The concepts within Neptune objects are composed of rules that themselves can reference concepts, located as part of the collection that constitutes the Neptune object. Consequently, action and comparison are two specialised rules that represent the concept assignment behaviour. After the interpretation, the intention is sent to the *Execution Manager* junction to choose either path (*a*) or (*b*) based on GUID's value. If the user is new (GUID is null), the execution manager chooses path *a* and asks the choreographer module to start matching the tasks in the intention model to Neptune Based Language Objects (NBLOs) via the *Semantic Linker* [4,23]. Those NBLOs include the core Neptune functionalities and thus each one has certain functions that should be executed when that NBLO is called. Otherwise, if the user has previously used the application, then the execution manager will take path *b*. To further aid the assurance of the intention model, the *Assurance Checker* of PAA focuses on the intention process/task Validation & Verification (V&V); for example making sure each process is uniquely named to avoid overlap among processes. It also deals with Authentication (verifying the identity of an entity) and Authorisation (whether a requesting entity will be allowed access to an object) in the application. Concept Aided Situation Prediction Action (CA-SPA) [4] constructs are used in PAA to specify policies for controlling access to system concerns and to ascertain the authorisation. Efficient and timely auditing is critical in order to respond quickly in managing and responding to faults and failures in a competitive business environment. Thus, the *Auditing/Accounting Generator* is responsible for creating the auditing/accounting model that allows users to make fast well-informed decisions in real time and collect information on resource usage for the purpose of auditing, billing, or cost allocation in the cloud environment. This generator creates an XML file, which includes three important auditing attributes (Events, Logs, and Monitoring) [23].

4. Implementation details

4.1. Intention interpretation

Each task in the intention model is built up of three main attributes: An *ID* of the task, *input* to the task (except for the start point) and *output* to the next task. Thus, Listing 1 shows that the task interpretation code works in the same order: checks the ID of the task (line 2), processes the input (line 3), and produces the output (line 4).

```
1.  1. foreach (XmlNode node in nodes) {   DataColl d = new DataColl();
2.  d.id = node.Attributes["id"].InnerText; //now we can get the input & output
3.  d.inputs = parseInputs(node);
4.  d.outputs = parseOutputs(node);       store.datas.Add(d); }
5.  XmlNodeList nodesO = node.SelectNodes("moveto");
6.   foreach (XmlNode nodeO in nodesO) {  //ok Output i = new Output();
7.  i.moveID = nodeO.InnerText;
8.  if (nodeO.Attributes.Count != 0)
9.  {       i.result = nodeO.Attributes["result"].InnerText; }
10. a.Add(i);      }       return a;       }
```

Listing 1. Parse Inputs.

The tail of the intention model is preserved for adding `neptunescript`, as shown in Listing 2. This part of the intention model is needed when the user expresses a new behaviour to the application by adding its Neptune code. This capability highlights the intention model's novel contribution since the other CBA requirements description languages that were described in Section 2.1, do not allow the user to add new behaviour(s) to the system at runtime. The Neptune function should be called through the `NeptuneFunction` attribute followed by the name of the new function (*line 1*). The returned value is mainly used in the `result` attribute (*line 5*). If the result was 1, then `action2` should be taken, if not, `action1` is chosen. In this way, Neptune function(s) and behaviour(s) can be added and modified at runtime during the execution of the application.

```
1.  <question id="Decision1" NeptuneFunction="processSwelling">
2.  <input type="text">  <validation><![CDATA[]]></validation>
3.  <message><![CDATA[]]></message> </input>
4.  <moveto result="1">Action2</moveto> <moveto result="0">Action1</moveto>
5.  </question> <neptunescript><![CDATA[ function processSwelling as number
6.  { if (NeptuneProcess.Swelling == 1)     {    return 1;      }
7.          else    {       return 0;      }} ]]></neptunescript>
```

Listing 2. Question task and neptune added function.

4.2. Intention to provision

After the intention interpretation is finished, the execution manager starts to decide what the next module of PAA Neptune Framework is, as described in Section 3.1, by checking the GUID identifier value, as shown in Listing 3. If the GUID is null

```
1.  if (node.Attributes["GUID"].InnerText ==  null)
2.  { Choreographer ();
3.  node.Attributes["GUID"].InnerText = Guid.NewGuid().ToString();
4.  } else  //if guid not null PAA_models_shell ();
```

Listing 3. Execution manager junction.

(line 1), the intention model will be sent to the *choreographer module* and the execution manager will assign a new GUID value to the intention model, as shown in line 3. Otherwise, the *PAA Models Shell module* will be called.

After that, the choreographer starts the *semantic linker module*. The NBLOs linker, within the semantic linker, is responsible for linking the tasks' types to the available NBLOs registered in the NBLOs controller. Once a match is found, calls to the Neptune core functionalities, contained within the selected NBLO, are produced to provide a solution to the given task. Thus, the semantic linker needs input in the form of task types, which it links to the available NBLOs. Once a task is interpreted and linked, the *composer* starts assembling the NBLOs and sends them to *PAA Engine*. The NBLOs are defined as ASP.Net Web User Controls (WUCs) using the *ascx* format and C# class component. The NBLOs include the core Neptune functionality and thus each one has certain functions that should be executed when that NBLO is called. For example, Listing 4 shows that if the `multiplechoicelist` NBLO was called then the NBLOs controller will load the `multiplechoicelist` NBLO with its elements to the `PlaceHolder`.

```
if (x.Attributes["type"].Value == "multiplechoicelist") {// nblos controller
Control c = LoadControl("nblo/multiplechoicelist.ascx"); // task interpreter
((neptune_nblo_OptionList)c).Question =   x.Attributes["question"].Value;
((neptune_nblo_OptionList)c).Name = x.Attributes["name"].Value;
((neptune_nblo_OptionList)c).ID = x.Attributes["nbloID"].Value;
XmlNodeList options = x.SelectNodes("option");
foreach (XmlNode xl in options)
{   ((neptune_nblo_OptionList)c).Options.Items.Add(xl.InnerText); }
 (neptune_nblo_OptionList)c).Options.SelectedIndex = 0;
PlaceHolder1.Controls.Add(c);}
```

Listing 4. (`multiplechoicelist`) NBLO.

4.3. Assurance support

In order to ensure that the adaptation is executed correctly at runtime, the adaptation approach, together with the underlying adaptation tool kit, platform, and policy, should provide dedicated techniques and facilities to overcome any emergent problems such as modifying non-functional properties, replacing one service with another, or executing different composition fragments. PAA has added a new runtime adaptive assurance technique by allowing the injection of new assurance rules and policies to the application according to encountered failures. As PAA produces systems as processes and tasks, and components in the form of intention descriptions, it has applied and implemented assurance concepts in two complementary methods: (i) non-functional conditions attached individually to each process as a task in the intention, via assurance requirements, to ensure its correct interpretation, and (ii) Concept Aided-Situation Prediction Action (CA-SPA) assurance policies. In both cases the assurance conditions will be added in the form of a Neptune Script to the intention model to add legitimacy to the processes or the whole intention model. To illustrate this, Listing 5 shows the first method and the Provision, Assurance and Accounting requirements, which have been added at runtime to validate the process entries.

```
define ValidPreShippingOrder as PAA
{    provision { order as PetOrderDetails }
  assurance {    order.ShippingAddress is VALID;
             order.SelectedPet      is INSTOCK;
             order.ShippingDate    is NULL;  }
accounting {/*error handling or logging */} }
```

Listing 5. Assurance validation and verification.

In the second case however, the new assurance CA-SPA policy will be used to monitor not only a process or task within the intention model but rather the entire intention behaviour. In previous works [4,24], the authors have shown that the application's users can use the developed PAA WikiEditor tool to modify business process models at runtime. The assurance rule injected into the process intervenes to assure the validation and verifies the process correctness whereas the CA-SPA rule injected into the intention model intervenes to assure the quality and safety of the adapted intention and guarantee correctness of the new changes. From a security perspective, the *Authentication, Authorisation and Accounting* (*AAA*) *mechanism* provided in the assurance checker is used [25] to specify the application security behaviour such as the addition of access control policy to certain processes. Authorisation in the AAA framework consists of the enforcement of role-based interactions with the system and/or changes to the system.

4.4. Auditing support

The PAA Neptune modelling approach is intended to create fully adaptive cloud-based applications, ensuring the ES characteristics of these applications and allowing the user to modify the entire application and its behaviour at runtime, using PAA WikiEditor. As the provision generator starts work on a provision model, the auditing/accounting generator will start developing the auditing model. Creation of a new auditing model starts by loading the initial blank Auditing.xml file, which in the first instance has only the XML document header. It also provides an updating counter (updatecounter), which, in the first instance, starts with value 0 and is incremented each time the user creates and/or updates the intention or provision models to record the total number of model versions that have been created or modified by the user. It also provides a current view counter (`currentview`), which represents the current intention/provision model number, which starts with default value 1 that represents the initial intention model number currently loaded. The maximum number of backed-up intention models, that can be saved and displayed in the auditing model is currently limited to 10, thus, when ten intention model versions have been generated/modified, and the user needs to create a new version, the oldest version (`id = ‘‘1’’`) will be removed from the auditing model and the newest one put at the top of the stack, i.e. a first in first out policy. Listing 6 shows an example of the auditing model with one new modified intention.

```
<update_info updatecounter="1" currentview="2"> <update id="1"
intention="intention201083111628.xml" date="2010/08/03" time="11:16:28"
ip_address="192.168.239.1" ip_address_list="InterNetwork=150.204.48.163"
host_Name="cmptsham" /> <update id="1" intention="intention201083111628.xml"
date="2010/08/03" time="11:16:28" ip_address="192.168.239.1"
ip_address_list="InterNetwork=192.168.192.1" host_Name="cmptsham" /> <update id="1"
intention="intention201083111628.xml" date="2010/08/03" time="11:16:28"
ip_address="192.168.239.1" ip_address_list="InterNetwork=192.168.245.1"
host_Name="cmptsham" /> <update id="1" intention="intention201083111628.xml"
date="2010/08/03" time="11:16:28" ip_address="192.168.239.1"
ip_address_list="InterNetwork=192.168.239.1" host_Name="cmptsham" /> </update_info>
```

Listing 6. Auditing model after the adaptation.

The auditing model also provides more information such as intention model modification date and time, and the new (modified) intention model name (concatenated from the date and time). In addition, to limit the misuse of PAA WikiEditor to modify the intention/provision and track, the provenance of the modification (the user's IP address), is also provided by the auditing model via (`ip_address, ip_address_list`) as shown in Listing 6. Moreover, the user's hostname is given in the auditing model via (`host_Name = ‘‘cmptsham’’`) in the above listing.

5. Evaluation example: system design basis

Microsoft's PETSHOP application was selected, as a case study application in this paper, as it was designed to provide a blueprint for building *n-tier* applications in .Net to highlight its abilities and the effectiveness of the language. The design of a PETSHOP application involves abstracting out the User Presentation Layer (UPL), Business Logic Layer (BLL), and Data Access Layer (DAL) to a level that ensures separation of the various logic concerns. PETSHOP allows users to browse, search and purchase from an online catalogue of pets within a common web front-end interface. In this way, PETSHOP forms publicly available, best-practice system architecture for e-commerce and provides a rich architecture that can thus be used to interpret the benefits of modelling and developing systems using PAA.

5.1. Designing adaptation scenario

By weaving the behaviour of the *order* process in PETSHOP at runtime, such that a new process is generated and injected into the application to introduce new behaviour, we can highlight many of the design and adaptation abilities of PAA and contrast this against the current MS design. Practically, introducing new behaviour here shows the strengths of using open abstracted XML-based models and how the application intentions are used to weave in new behaviour autonomously to ensure the eternal availability of the system. The same adaptation can then be attempted in the MS PETSHOP codebase using traditional techniques available in *.Net* to compare both approaches. Comparisons between the relative difficulties in implementing such solutions is a difficult task, as judging one approach as 'easier' than another, to instigate an adaptation, is dependent largely on the experience and technique of the developer. In contrast, ascertaining whether an adaptation took place at runtime or at design time is inherently simple; if no re-deployment, re-uploading and re-debugging of the application occurred in a traditional design time, then it can be considered to be a runtime adaptation. However, the associated cost of providing runtime adaptation must also be considered, as runtime adaptation is advocated for its use to produce reliable, self-aware systems.

5.2. Scenario implementation

In PETSHOP, the process of performing an order is split between the UPL and BLL, e.g. as the user selects pets through the user interface and classes execute to access the store and ensure pets are available. Data is retrieved to inform the process via discrete .Net web pages (e.g. `OrderProcess.aspx`) and when the user has finished the work and clicked the relevant button, the trigger code contained within these .Net pages instigates the move from the current process to another one. The execution of the process then passes to the BLL to interpret the data, and store this within the database in the DAL. Once stored, the logic moves the execution to produce another page for the user. As such, the classes within the BLL can be said to change the state of the process by signifying that a task is complete, and that data is added to the DAL structures, as these are the side-effects of its actuation. Similarly, pages within the UPL can be said to introduce data to the state of the process, and signify that the task of producing data is complete. As such, any amendments to weave a new behaviour to the application should be done in BLL through the amendments of its classes. However, in autonomic software, consumption behaviour is only available with the actual implementation and the source of the services/components is hidden. To overcome the above defects, PAA method adds a new intermediate abstraction layer, Meta-Data Layer (MDL) between the UPL and BLL to produce a new 4-tier architecture, Fig. 5. The new MDL provides the PETSHOP intention model as a way to introduce runtime weaving and a new adaptive method of accessing the BLL contents. As such the MDL model does not relate directly to the contents (classes) of the services described, but rather provides a method to access it.

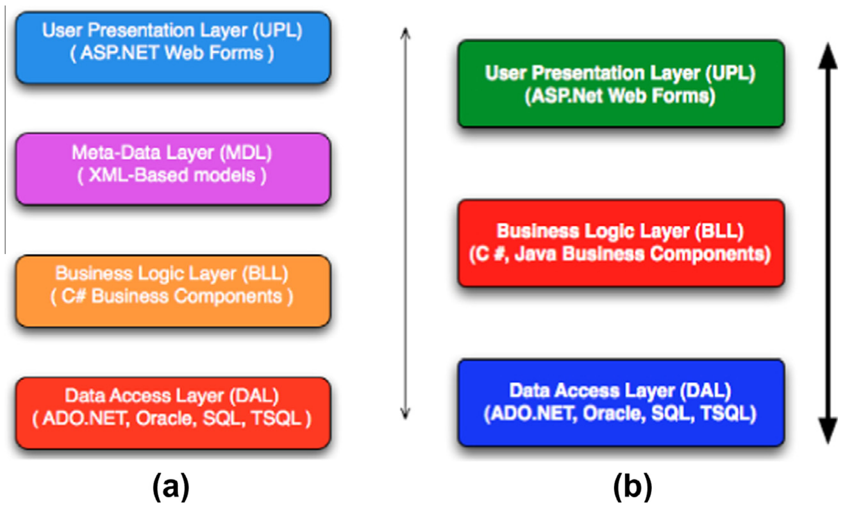
User Presentation Layer (UPL) (ASP.NET Web Forms)
Meta-Data Layer (MDL) (XML-Based models)
Business Logic Layer (BLL) (C# Business Components)
Data Access Layer (DAL) (ADO.NET, Oracle, SQL, TSQL)
(a)
User Presentation Layer (UPL) (ASP.Net Web Forms)
Business Logic Layer (BLL) (C #, Java Business Components)
Data Access Layer (DAL) (ADO.NET, Oracle, SQL, TSQL)
(b)

Fig. 5. 4-Tiers architecture against 3-tiers.

5.3. Adding PAA support to PETSHOP

This scenario considers the PETSHOP order process so PAA support will be written into this process to attain PAA goals while ensuring they do not lose their meanings and behaviour. Listing 7 shows the new shipping components within the BLL and the UPL written in NeptuneScript. In this way, `nbloOrderShipping` describes the method `getShipping` within the UPL as adding the `ShippingAddress`, ShippingPostcode and `ShippingBand` features to the `orderID` element within the state of the process. Similarly, `nbloSaveOrderShipping` adds the feature `shipping` to an element `Database` to describe the fact that operationally, the method `ProcessShipping` in the BLL saves shipping data to the database. Both actuation calls are deemed synchronous as the methods `getShipping` and `ProcessShipping` should complete before continuation of the process flow. Values are set for the features by way of the return value of the underlying component. As such, `ShippingAddress` will take the return value of the `ShippingAddress` object returned by the method `getShipping`. Other values can be specified by the `as` operator in Neptune, such that the values of the return type can be reflected.

```
define nbloOrderShipping with NString ordered {
purpose {feature ShippingAddress  to orderID;
feature ShippingPostcode to orderID;
feature ShippingBand     to orderID;}
Actuation {// call the presentation layer
call BaseLanguage.Csharp("orderPL.dll","getShipping",orderID,sync);}}
define nbloSaveOrderShipping with NString orderID  // logic layer component
{ Purpose { feature orderID.Shipping to Database;}
Actuation {call
BaseLanguage.Csharp("orderLL.dll","processShipping",orderID,sync);}}
```

Listing 7. New NBLO definition.

5.4. Expanding system functionality

Expanding the system's behaviour at runtime can be achieved in one of the two ways: (i) inject the new behaviour code in the right location in the intention model, if it is previously written and available; (ii) If the code is not available, then the user should first write and inject a new NBLO relating to the new component for performing a new task. If the NBLOs of the new process are already available in the application, then users only need to add the code shown in Listing 8 to the original PET-SHOP intention model, between the Order Shipping and Order Process code. In this case, the semantic linker will look for the NBLOs to perform the behaviour.

```
<!-- Check Delivery --> <action id="Check Delivery">
<input type="text"> <message><![CDATA[Checking your order's delivery in
process...]]></message> </input> <ui page="checkDelivery.aspx"/>
<moveto result="[Not set]">Order Process</moveto>
</action> <!-- Order Process XML Code-->
```

Listing 8. The new added XML code behaviour.

Otherwise, if the required NBLO is not available, then new NBLO code should be written at runtime, and added at the bottom of the original intention model in the ⟨neptunescript⟩ part. The new nbloCheckDelivery code is shown in Listing 9.

```
define nbloCheckDelivery with NString orderID
{ purpose { feature isUKBased to orderID;}
Actuation { // call the presentation layer call BaseLanguage.C#
("checkage.dll","AddressCheck.InUK",orderID,sync);}}
```

Listing 9. Neptune check delivery NBLO code.

The intention model of the Order Process can now be updated by adding the code needed for the delivery check which is provided by a method AddressCheck.InUK in a new assembly checkage.dll, thereby not affecting the original compilation of the assemblies.

6. Results

Measuring PAA System's performance and CPU consumption against MS PETSHOP are the two major comparison metrics selected for use here. Averages of 10,000 process executions were made along with 100 concurrent accesses of visitors to both systems to compare their performance in a heavy traffic situation. Since the original MS PETSHOP did not include any need for behaviour modification process, its response time is much faster and the CPU consumption will be less than the other systems, which have adaptation abilities. As such, for comparison purposes, the same test was also performed between a modified PETSHOP implementation with the adaptation characteristic and PAA PETSHOP. In this way, a benchmark for adaptation of any form in both applications could be determined. Fig. 6a depicts the performance relationship between the two applications in the first instance of the execution.

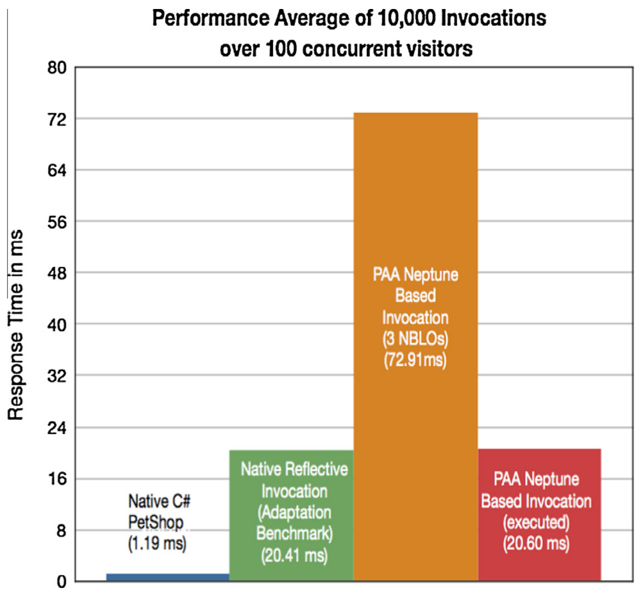

Fig. 6a. Performance relationships.

The results indicate that the overhead of using PAA, the amber rectangle, is significant upon the first execution of the process. This is due to the semantic linker having to process the requested NBLO descriptions to perform the task. After the semantic linking is done and the tasks have been provided and linked, the execution time also introduces a slight overhead in terms of computational performance over the adaptation benchmark. The overhead is introduced by having to dispatch Neptune to perform the executions, and for the semantic linking and provision of the auditing service.

In terms of CPU consumption, Fig. 6b shows the comparison results of executing the original (namely native) PETSHOP with the modified PETSHOP (with adaptation support) and PAA PETSHOP. It is obvious that the Native PETSHOP needs lower CPU than the modified PETSHOP, as the former does not support runtime adaptation. However, the overhead of PAA PETSHOP over the modified PETSHOP is also relatively worse as it needs approximately double the amount of CPU needed by the modified PETSHOP. This is expected and justified because of the extra interpretation actions that occur at runtime via use of PAA Neptune Framework. However, after the runtime interpretation execution is finished, the CPU usage is massively reduced (red vertical bar) to about half that needed by PAA system in the amber vertical bar. It should be noted at this stage that the design and implementation of PAA framework is a prototype to provide full runtime adaptation to the cloud-based applications and to ensure the application's ES vision, rather than being optimised for system performance.

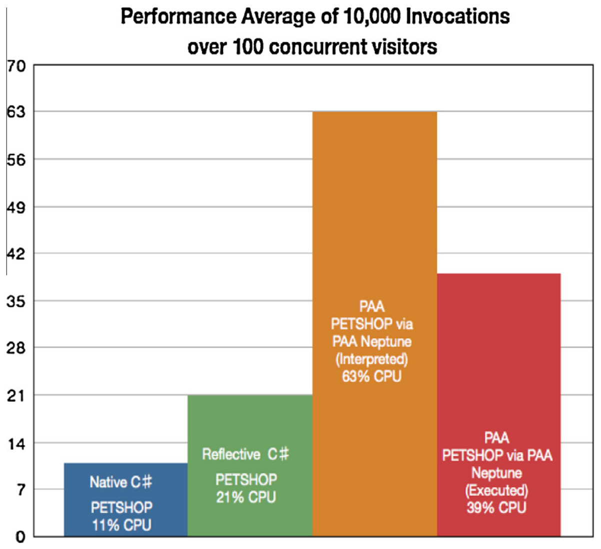

Fig. 6b. CPU consumption relationships.

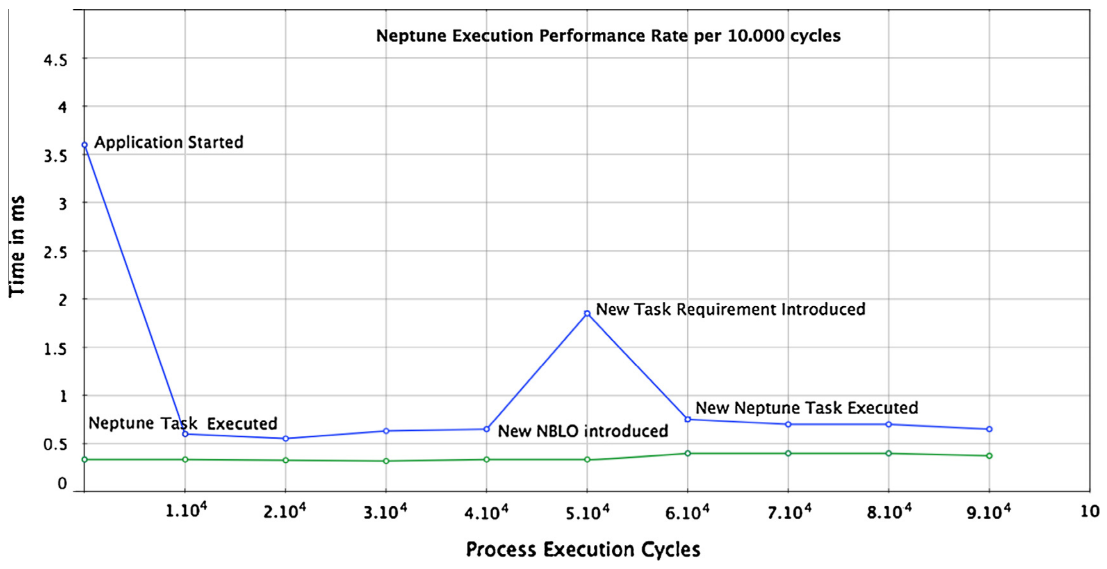

Fig. 7. PAA PETSHOP vs. MS PETSHOP.

6.1. *PAA performance analysis*

Another test was conducted to monitor the stability of PAA PETSHOP during its operation in comparison to the original MS PETSHOP. This time, the runtime adaptation ability of PAA was tested by introducing new behaviour to PAA PETSHOP and thus to validate its performance. As is known, introducing new behaviour at runtime has a direct performance impact due to further interpretation needed by the semantic linker and it could be contrasted against the performance impact of the same behaviour in MS PETSHOP. Fig. 7 shows the two systems executed to produce the same behaviour over a time-scale of cycles of a process execution. The blue line shows that the performance at start-up execution time is worse for PAA as there are numerous requirements that should be interpreted and linked at once. After this time however, PAA looks stable, although the performance is slightly worse than the green (MS) one until it reaches 50,000 executions.

At 50,000 executions cycles, a new task requirement is introduced to the intention model via a new NBLO. As new semantic linking needs to be done at runtime, performance is reduced such that the time to complete the process increased to nearly 2 ms. Since there is only one new requirement introduced at this point, the performance impact (the blue line) is much less than that of the first initiation at cycle 0, where many new requirements were introduced all at the starting time. After the linking and execution, performance returns to a new standard, slightly slower than the behaviour between 10,000 and 50,000 cycles. This is due to the added time needed to execute the new action by PAA Neptune framework. On the other hand, there is a slight increase in execution time of the original MS PETSHOP (represented in green line) from 50,000, due to the new behaviour introduced to the PETSHOP code to be executed.

6.2. *Hot swapping and hot plugging testing*

Hot swapping and hot plugging testing have been accomplished in this paper to ensure system's performance stability on different tasks provided by the new system. The response time of swapping a service/component with another one, and adding a new component at runtime, have been evaluated to give a better indication of the proposed approach's capabilities. As shown in Fig. 8, at 30,000, the performance is reduced due to the hot-swapping introduction that needs some semantic linking and matching to establish the requested NBLOs for the new component/service. The hot swapping in PAA is done through the intention model modification, as explained in Section 4. Nevertheless, the time that PAA PETSHOP needs to accomplish the hot swapping is slightly over the time needed by Microsoft PETSHOP. The next crest in Fig. 8 shows the hot plugging in PAA PETSHOP, which needs more time than the time needed for the hot swapping due to introducing a new component to the system. In this case, the execution manager, the semantic linker and the execution engine need more time to fulfil the tasks and create the new function. It should be noted that after executing the hot swapping and hot plugging, PAA PETSHOP performance is improved again. This result shows a reasonable parity, on the overall system performance, with different executed tasks.

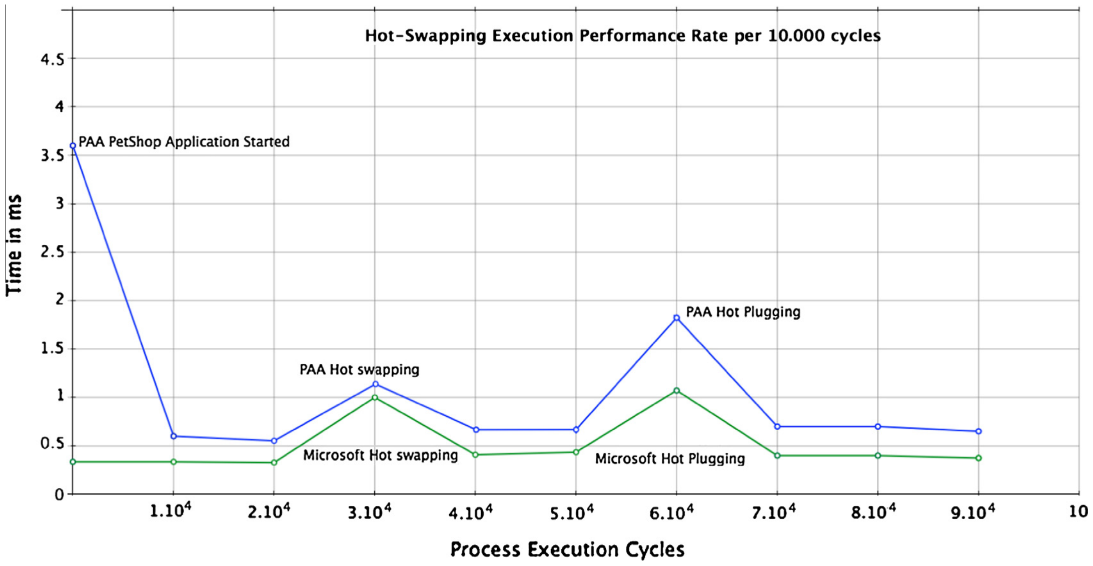

Fig. 8. Hot swapping/plugging testing of PAA PETSHOP vs. Microsoft PETSHOP.

7. Conclusion

This paper demonstrated the feasibility of developing intention models to support the dynamic runtime adaptation of a restricted range of cloud-based applications to model emergent requirements. The approach has been applied to a Grid Computing type scenario, as proof of concept: The move to a wide range of Cloud applications and cloud-of-clouds, ought to proceed in a natural manner without much difficulty. Nevertheless, this is left as future work. Current approaches were explored to requirements description and cloud-of-clouds as the basis to design and develop PAA framework. The major components of PAA were described, and how extensions could be supported through PAA WikiEditor. Moreover, the operation of the framework has been demonstrated through the example of Microsoft PETSHOP and shown this approach to be scalable for introducing new requirements. Specifically, this work has continued the advances shown in [4,23] by extending the comparison of a PAA mediated runtime dynamic execution to that of a standard runtime execution, whilst still maintaining comparability with Microsoft's original/non-dynamic benchmark.

Significant limitations were identified during experimenting the PAA. First – the use of PAA WikiEditor leads to new runtime errors where the user uploads an incompatible intention model rather than adapting the original one; resulting in the system being down due to unexpected error. Thus, developing rules for uploading a new intention model is a matter for future research; however, uploading incompatible intention model will be managed by the assurance model by displaying an error message to the user and re-load the original model. Moreover, the users' spelling mistakes when writing new code in the intention model at runtime are unacceptable as semantic linker is unable to link the requested task(s).

The above evaluation/results are not based on real-world application but rather a comparison of the proposed approach against Microsoft benchmark. A thorough investigation of PAA demonstrating the results in a real-world context is planned future work.

References

[1] Buyya R, Yeo CS, Venugopal S. Market-oriented cloud computing: vision, hype, and reality for delivering IT services as computing utilities. In: 10th IEEE international conference on high performance computing and, communications; 2008. p. 5–13

[2] Vermesan O, Harrison M, Vogt H, Kalaboukas K, Tomasella M. The internet of things – strategic research roadmap. Cluster of European Research Projects on the Internet of Things, CERP-IoT; 2009.

[3] Vaquero LM, Rodero-Merino L, Buyya R. Dynamically scalling applications in the cloud. Computer communication review, vol. 41, no. 1. New York (USA): ACM Press; 2011. pp. 45-52.

[4] Baker T, Hussien A, Randles M, Taleb-Bendiab A. Supporting elastic cloud computation with intention description language. In: The 11th annual postgraduate symposium on the convergence of telecommunications, networking and broadcasting; 2010.

[5] Abbadi IM. Middleware services at cloud application layer. In: IWTMP2PS'11, proceedings of second international workshop on trust management in P2P systems; 2011.

[6] Abbadi IM. Middleware services at cloud virtual layer. In: Proceedings of 11th international conference on computer and information technology (CIT); 2011. p. 115–120.

[7] Bieber G, Carpenter J. Introduction to service-oriented programming (rev 2.1). OpenWings Whitepaper; 2001. p. 1–13.

[8] Abbadi IM, Alawneh M. A framework for establishing trust in the cloud. Comput Electr Eng J 2012;38(5):1073–87.

[9] France R. Rumpe B. Model-driven development of complex software: a research roadmap, in the future of, software engineering FOSE07; 2007. p. 37–54.

[10] Miseldine P, Taleb-bendiab A. Neptune: supporting semantics-based runtime software refactoring to achieve assured system autonomy. In: 4th International conference on autonomic and trusted computing (ATC-07), IEEE organisation; 2007. p. 1–15.
[11] Mirkovic J, Faber T, Hsieh P, Malayandisamu G, Malavia R, DADL: distributed application description language. USC/ISI technical report # ISI-TR-664.
[12] Goldsack P, Guijarro J, Loughran S, Coles A, Farrell A, Lian A, et al. The smart frog configuration management framework. ACM SIGOPS operating systems review; 2009.
[13] Barbacci MC, Weinstock CB, Wing JM. Durra: language support for large-grained parallelism. In: Proc. international conference on parallel processing and applications; 1987.
[14] Lapouchnain A, Goal-oriented requirements engineering: an overview of the current research. In: 12th Asia-Pacific, software engineering conference (APSEC'05); 2005. p. 3–3.
[15] Yu E. Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of ISRE'97: 3rd IEEE international symposium on, requirements engineering; 1997. p. 226–35.
[16] Mylopoulos J, Castro J, Kolp M. Tropos: a framework for requirements-driven software development. In: Information systems engineering: state of the art and research themes. Springer; 2000. p. 261–73.
[17] Darimont R, Delor E, Massonet P, Van Lamsweerde A. GRAIL/KAOS: an environment for goal-driven requirements engineering. In: Proceedings of the 19th international conference on, Software engineering ICSE98; 1997. p. 612–3.
[18] Bessani A, Correia M, Quaresma B, Andre F, Sousa P. DepSky: dependable and secure storage in a cloud-of-clouds. In: Proceedings of the European systems conference eurosys'11, ACM, Austria; 2011.
[19] Vecchiola C, Chu XC, Mattess M, Buyya R. Aneka—integration of Private and public clouds. Cloud computing principles and paradigms, Willy, USA; 2011.
[20] Buyya R, Ranjan R, Calheiros RN. InterCloud: utility-oriented federation of cloud computing environments for scaling of application services. In: Proceedings of the 10th international conference on algorithms and architectures for parallel processing (ICA3PP 2010), South Korea. Springer: Germany; 2010. p. 328–36.
[21] Kim H, Abdelbaky M, Parashar M. CometPortal: a portal for online risk analytics using CometCloud. In: IEEE international conference communication technology and application, vol. 20; 2009.
[22] Kloppmann M, Koenig D, Leymann F, Pfau G, Rickayzen A, Schmidt P, et al. WS-BPEL extension for sub-processes-BPEL-SPE", A joint white paper by IBM and SAP; 2005. p. 1–17.
[23] Baker T, Taleb-Bendiab A, Randles M. Auditable intention-oriented web applications using PAA auditing/accounting paradigm. In: Proceeding of the international conference on techniques and applications for mobile commerce: proceedings of TAMoCo 2009. IOS Press; 2009. p. 61–70.
[24] Baker T, Taleb-Bendiab A, Al-jumeily D. Assurance support for full adaptive service based applications. In: Proceedings of the 12th international conference on computer modelling and, simulation; 2010. p. 1–7.
[25] Mitton D, St. Johns M, Barkley S, Nelson D, Patil B, Stevens M, et al. Authentication, authorisation, and accounting: protocol evaluation. RFC editor; 2001.

Thar Baker is a Senior Lecturer of Networked Systems and Security at LJMU. He finished his PhD in Distributed Software Modelling, form LJMU (2010). His research topics include Computer Networks and Distributed Computing. He has been involved as a member of leading conferences in Cloud Computing and has served as a referee for a number of scientific journals and conferences.

Michael Mackay is a Lecturer in the School of Computing and Mathematical Sciences at LJMU. He has been involved in a number of national and international research projects and his main research interests include IPv6 transitioning, Mobile IP and network mobility, QoS, IPTV streaming and caching and CDNs, Grid networking and Cloud Computing.

Martin Randles is a Researcher, Senior Lecturer and Programme Leader for Software Engineering at LJMU. Through completing an honours degree in Mathematics from Manchester University; wide experience in industry; an MSc. (distinction) in Computing and Information Systems and a PhD, his research interests include the application of mathematical techniques to computing, control/management of large-scale complex systems and autonomic computing.

Azzelarabe Taleb-Bendiab is professor of Computer Science at the School of Computer, Security and Information Science, ECU, Australia. His research interest is in the areas of autonomic software engineering for cloud computing. He has published widely in the area and is a current member of several conferences' organising committees in this area of research.