Conflicting Perspectives on Architecting Software – In Search of a Practical Solution

SASA BASKARADA & FRANK FURSENKO
School of Computer and Information Science
University of South Australia
Mawson Lakes SA 5095
AUSTRALIA
{Sasa.Baskarada, Frank.Fursenko}@unisa.edu.au

Abstract: - What exactly is software architecture? Even though the phrase "Software Architecture" is currently a buzzword in the software engineering community it is not always a very well understood term. Various different concepts are often depicted as being software architectures and a wide range of different IT professionals are describing themselves as software architects. Software architectures are also often modelled with a range of different techniques, using a variety of different tools and languages. This paper attempts to clarify what software architectures really are as well as to find a practical solution to successfully modelling them.

Key-Words: - Architecture Description Language (ADL), Software Architecture, Software Design, Software Engineering, Unified Modelling Language (UML)

1 Introduction

Software architectures represent high-level views of systems and for that reason allow developers to concentrate on the big picture rather than on low-level details [1]. They are also one of the best approaches to consider non-functional requirements early in the development process [2]. However, there is no standard definition of the term "Software Architecture" [3] and there is no agreement yet on how they should be modelled either. Consequently, software architectures are often described informally (using inappropriate notations). Standardising the notation would reduce the ambiguity and make software architecture modelling easier for practitioners.

A number of Architecture Description Languages (ADLs) already exist, which can be used to model software architectures. However, the software architecture community does not yet agree on what features should be present in an ADL, or precisely what these features should model [4]. Furthermore, there exists no common definition of the term Architecture Description Language either [3]. A large number of ADLs have been proposed as well and each of them takes a particular approach to the modelling of architectures [5]. This is one more reason why a standard modelling notation is needed.

As UML is already widely used for software analysis & design it could also be very useful for software architecture modelling. UML could be a helpful tool for visualising, documenting and modelling of software architectures. Therefore, this paper also investigates if and how UML could be used to effectively model software architectures. Even though UML has already been used to model certain types of software architectures, it was originally designed to model object oriented analysis & design and thus it may not be suitable for the modelling of all aspects of software architectures [6]. Therefore, successful modelling of software architectures may require some extensions to UML. If UML was adapted to model software architectures effectively, developers could use it as a standard language for the modelling of software architectures as well as object-oriented analysis & design. By modelling the CommPOS (Commercial Point Of Sale) software architecture in UML, the paper shows how UML could be used to model component-based software architectures.

2 Software Architectures

Software architectures are a key area in the software engineering discipline, because every system has an architecture. Clements and Northrop state that the term "Software Architecture" is mostly used to describe structural aspects of a software system [7]. They describe software architecture as an extremely important part of system design because it represents the earliest set of design

decisions, which makes software architectures difficult to get right and hard to change [7]. Software architectures represent high-level views of systems and as a result allow developers to concentrate on the big picture rather than on low-level details [1]. They are also one of the best approaches to consider non-functional requirements early in the development process [2]. As the size and complexity of software systems grow, modelling software architectures is becoming ever more important. The architecture depends on the requirements and the design decisions made to satisfy those requirements. A software engineer has to decide on the architecture of a software system in a similar way as a building architect decides on a particular architecture when building a house. The architecture can be newly developed or reused from similar existing systems. Certain types of applications have unique characteristics and share similar structure. Hence, they might conform to the same architecture. In order to speed up development, reduce production cost, control the complexity, elevate abstraction levels, achieve separation of concerns and facilitate software reuse, certain software architectures have been developed for certain types of applications. Therefore, developing applications conforming to particular software architectures potentially increases productivity and reliability. Software architectures provide a template for design and also allow the management to better estimate the costs involved in the project. They aid in building a system by structuring large collections of components (clients, servers, databases, etc). Software architectures focus on the system structure and interaction between different components which is one of the most important aspects of large system design. Bachman et al state that, one cannot hope to build an acceptable system unless the architecture is appropriate and effectively communicated [8]. Clements & Northrop describe software architecture as a vehicle for stakeholder communication, because it provides a common ground for discussing concerns among different stakeholders [7]. Garlan states that software architecture plays an important role in the understanding, reuse, construction, evolution, analysis and management of software systems [9].

The term itself “Software Architecture” seems to be overused at the moment (it is a buzzword). Therefore, it is used to describe various concepts, many of which are not software architectures at all. There is still little consensus on software architecture terminology, representation and methodology [7], which makes the modelling ever more difficult. For that reason, software architectures are often described informally (using inappropriate notations), which can make them open to interpretation and hard to understand.

There are two basic approaches to architecting software systems. The top-down approach divides a large problem into a number of sub-problems, which can then be newly implemented or solved by reusing existing components. The second approach is bottom-up and it requires implementing new components or reusing existing ones to compose a system. Real life situations require the use of both of these approaches. This is an important point because software architectures can be seen as being composed of components, connectors and configuration. We can get those by decomposing the system (or we can use the bottom-up approach to compose a system out of existing components, connectors and configurations). This paper shall therefore also investigate if any UML diagrams could be used to represent software architectures by modelling components, connectors and configurations.

2.1 Components

Software architectures are often described as models of components and interconnections among these components. Medvidovic & Taylor describe a component in an architecture as a unit of computation or a data store (a familiar example is a Unix process) [10]. They maintain state, perform operations and exchange messages with other components [6]. Each component has a type, constraints associated with it, and an interface through which it interacts with other components. Components can also vary greatly in size and even a software architecture layer can be considered a component. Egyed & Kruchten suggest the use of UML class diagrams for modelling of component based architecture [11]. However, this paper considers other UML diagrams as well (component diagram seems like a more appropriate solution).

2.2 Connectors

Connectors model component interactions and the rules governing those interactions. They transmit messages between components. Simple interactions can be achieved through method calls and global variables and more complex interactions include database access and client-server applications. Egyed & Kruchten suggest the use of UML aggregations, associations, dependencies and generalisation as connectors [11]. However this paper explores other more appropriate options. UML

connectors could also have stereotypes or constraints associated with them as well.

2.3 Configurations

Configurations are instances of components and connectors, describing semantics of a software architecture and placing constraints on component interaction. Medvidovic & Taylor describe configurations as connected graphs of components and connectors that describe architectural structure [10].

2.4 Architectural Views

Clements et al state that modern software architecture practice embraces the concept of architectural views, and Bachmann et al state that documenting software architecture is primarily about documenting the relevant views [12], [8]. Views are essentially abstractions, each with respect to different criteria [7]. Many different views can be used to model software architectures, and Kruchten describes the "four plus one" approach [13]. He identifies four main views of software architecture plus a fifth view that ties the other four together. Bachmann et al describe these views as [8]:

- **Logical View** – behavioural requirements and the services the system should provide to its end users.
- **Process View** – performance, system availability, concurrency, distribution, system integrity and fault tolerance.
- **Development View** – the actual software models.
- **Physical View** – system availability, reliability, performance and scalability.

However, one of the most commonly used views in software architectures is the layered view [8].

2.4.1 The Layered View

Bachmann et al describe a layer as a collection of software units such as programs or modules that may be invoked or accessed [8]. It is mostly represented as vertically arranged rectangles. Layering divides the software structure into discrete units (presentation layer, application layer, data layer, etc.). Each layer provides functionality, which is independent from any other layers. Therefore, a layer could also be considered a component in software architecture modelling. Each layer also has an interface, which can be used by other layers. Bachmann et al define an interface as a boundary across which two independent entities meet, interact, or communicate with each other [14]. Therefore, as long as layer's interface is not changed, a layer can be modified without affecting any other layers. Bachmann et al also state that UML has no built in primitive corresponding to a software architecture layer, but layers could be defined as a stereotype of a UML package [8].

3 Architecture Description Languages (ADLs)

In an effort to standardise the modelling of software architectures, a number of Architecture Description Languages (ADLs) have been developed (ACME, C2, Rapide, Wright, etc.). ADLs are not programming languages, but rather languages used to model software architectures. They attempt to formalise the modelling of architectural structure and behaviour. ADLs endeavour to make models of software architectures more understandable and enable a greater degree of analysis although some ADLs are more generic then others (some are specialised to particular domains). Tools are also available, for many ADLs, which support visual representation and analysis.

These languages were developed by a wide range of researchers, hence there is no common definition of the term architecture description language (ADL), and that there is no standard ADL [3], [4]. Furthermore, each ADL takes a particular approach to modelling (each ADL addresses a particular problem domain), which means that most ADLs can only be used to model a particular set of architectures. Dashofy et al state that research and experimentation in software architectures have resulted in an overabundance of ADLs [4]. These ADLs are mostly developed and used in academic circles and they haven't yet gained much acceptance in practitioners' community. ADLs are not used to a large extent by the developers mostly because there is a no standard ADL and the syntax is also fairly complex. There is also no agreement on which exact features an ADL must support, but there is an acceptance that they have to provide notations for modelling software architecture components, connectors and configurations.

4 Unified Modelling Language (UML)

The Unified Modelling Language (UML) is an object oriented analysis and design language. It is a family of notations that has become a standard for developing software artefacts. It has also found application in modelling non-software artefacts, such as business processes, human workflows, and

non-code development artefacts [15]. It is widely used in the software engineering community and there are many Computer-Aided Software Engineering (CASE) tools available that support UML modelling (ArgoUML, Enterprise Architect, MagicDraw UML, etc.). Rational Rose, a popular graphical software modelling tool, uses the Unified Modelling Language (UML) as its primary notation [11]. UML is a very large language which is defined by the Object Management Group (www.omg.org) [16]. The current UML specification document has more than 700 pages. It is a family of notations that includes use case diagrams, class diagrams, object diagrams, interaction diagrams, activity diagrams, statechart diagrams, component diagrams, and deployment diagrams. UML has become a standard language used for object oriented system design. It is widely accepted in the software engineering community and having it as a standard language for the modelling of software architectures would make software architecture modelling easier for developers and practitioners. UML notation is also a lot easier to understand than ADL notations.

4.1 UML Extension Mechanisms

Raw UML is suitable for modelling of some aspects of software architectures, but fails in modelling others [2]. Therefore, successful modelling of software architectures in UML may require some extensions to UML. UML defines an extension mechanism, which allows for adoption to specialised needs and those mechanisms allow for extensions of UML core concepts as well. The extension mechanism can also be used to add new modelling elements. UML could be adapted to model software architectures in two ways (using lightweight extensions or heavyweight extensions).

4.1.1 Lightweight Extensions

Lightweight extensions allow for adaptation of UML semantics without changing the UML meta-model. This is the preferred way of extending UML because lightweight extensions do not change the language itself. The existing CASE tools are also compatible with those types of UML extensions.

Stereotypes are perhaps the best known UML extension mechanism. They can be used to extend UML by specialising the meaning of elements. They may also introduce new additional elements and are represented as text within guillemots (‹‹stereotype ››). Stereotypes can be applied to almost all UML elements including classes, attributes, operations, associations, etc.

Tagged values are Key-Value pairs denoting characteristics of an element. They are represented as text within braces ({Key = Value}). Tagged values define additional properties for model elements and they can be used in conjunction with stereotypes.

Constraints are conditions or restrictions placed on elements. They are represented as text within braces ({constraint}). Constraints provide a mechanism for extending the semantics by adding new rules.

Object Constraint Language (OCL) is part of UML and it is a formal language mainly used to express UML constraints. OCL expressions do not have any side effects, which means, the evaluation of OCL expressions cannot alter the state of the system. If OCL rules are formally specified in a profile, then a modelling tool may be able to enforce these rules and help the person doing the modelling.

4.1.2 Heavyweight Extensions

UML can also be extended by modifying the UML meta-model. The meta-model is a UML model that specifies the abstract syntax of UML models [9]. It represents the structure and semantics of UML and all UML models are instances of the UML meta-model. Extending the meta-model would result in a new notation (it would change the language) and therefore it would also introduce incompatibilities with existing UML compliant CASE tools [13]. Therefore, this paper suggests the use of lightweight extensions to UML for modelling of software architectures.

5 UML as an ADL

We have highlighted the problem that there is still extensive disagreement about what software architecture really is and how to model it. There is also no agreement yet on what precisely an ADL is either. Each ADL takes a particular approach to the modelling of software architectures and there is a broad variety of ADLs (there is no standard language). One of the solutions to these problems is to use UML as an ADL. If UML is to be used as an ADL, UML would have to take its own approach to the modelling of software architectures. It would not make sense to try to imitate all the features of existing ADLs, but to select the features which are considered as absolutely necessary in an ADL. There seems to be consensus in the research community that an ADL has to be able to at least model components, connectors and configurations. Therefore, if UML is to be used as an ADL, it would have to be able to model those successfully.

Various UML diagrams could be used to model software architecture components, however UML explicitly provides Component diagrams for component modelling. UML Component diagrams describe the organisation of components in the system. Various other diagrams (class diagrams, interaction diagrams, etc.) could also be used to represent components visually. UML state diagrams can be used to represent component states and package diagrams can be used to represent groupings of components (packages can also be used to model architectural layers by grouping various components). UML interfaces, which are collections of operations (Class element with the interface stereotype applied), can be used to specifically model component interfaces. UML associations or dependencies can be used to model connectors. The choice would depend on the type of the connector and on the diagram used to represent software architecture components. As this paper is concentrating on the use of component diagrams for representation of components, dependencies are going to be used to model software architecture connectors. UML constraints are going to be used to specify constraints on component interaction. These constraints will be specified in the Object Constraint Language (OCL). Pre and post conditions will be specified in OCL as well.

5.1 Extensions to UML based on ADLs

Research done by Medvidovic et al suggests some lightweight extensions to UML based on C2, Rapide and Wright [6]. The UML extensions suggested by the authors attempt to provide UML with all the features that these ADLs support. Extending UML based on their work essentially creates a new ADL, which contains all the features of C2, Rapide and Wright combined. Having all these extensions may not be necessary, because each one of those ADLs takes a different approach to the modelling of software architectures. C2, Rapide and Wright are essentially not compatible with each other in the first place and if UML is to be used as an ADL, UML would not have to be compatible with all of them either. Pérez-Martínez states that the only solution to modelling the C3 architectural style (C3 is derived from C2) in UML is to extend the UML meta-model [17]. In reality extending the UML meta-model would not be practical because the resulting language would not be compatible with existing CASE tools. Therefore, this paper suggests lightweight extensions (stereotypes, constraints and tagged values) to UML for representation of software architectures.

6 CommPOS Software Architecture

In order to investigate UML's ability to model software architectures, we have used UML to model the software architecture of the Commercial Point Of Sale System (CommPOS v2.0). CommPOS is quite a complex system (10000+ lines of code, 100+ classes), which was developed at the University of South Australia. The system was developed in JAVA/J2EE and includes features like online access, PDA access, inventory management, networking functionality, business rule administration, etc. It supports various pricing strategies and it provides interfaces to external accounting software and credit authorisation services (e.g. VISA). CommPOS is a large system with a non trivial software architecture and this paper shows an approach to successfully modelling it in UML. In order to model CommPOS architecture, the system needed to be decomposed into components. Since components can vary in size (depending on the granularity level), there was not only one correct way to model CommPOS architecture. However, by examining the system we could identify following components:

- **JBoss** – Java application server used for J2EE support.
  - **Data** – Enterprise Java Beans (EJB), which reside in JBoss.
  - **Ordering** – Java Servlet, which resides in JBoss (used for web access).
- **GUI** – The presentation layer.
- **Store** – The physical store.
- **Register** – A register in a store.
- **Kitchen** – Server displaying orders placed with the Ordering Servlet.
- **Services** – Services factory.
  - **Inventory** – Service returned by the factory.
  - **Accounting** – Service returned by the factory.
  - **Credit Authorisation** – Service returned by the factory.
- **Pricing Strategy** – Pricing strategy factory (returns pricing strategies).
- **Business Rules** – Used to specify business rules.

UML Component elements have been used to represent various system components and UML Interfaces (class elements with the interface stereotype applied) have been used to specify the interfaces of these components. Stereotypes were used to further specialise the meaning of individual elements and OCL was used to specify constraints on component communication. UML dependencies were used to model connectors. Figure 1 shows component interfaces, which were specified in a UML class diagram.

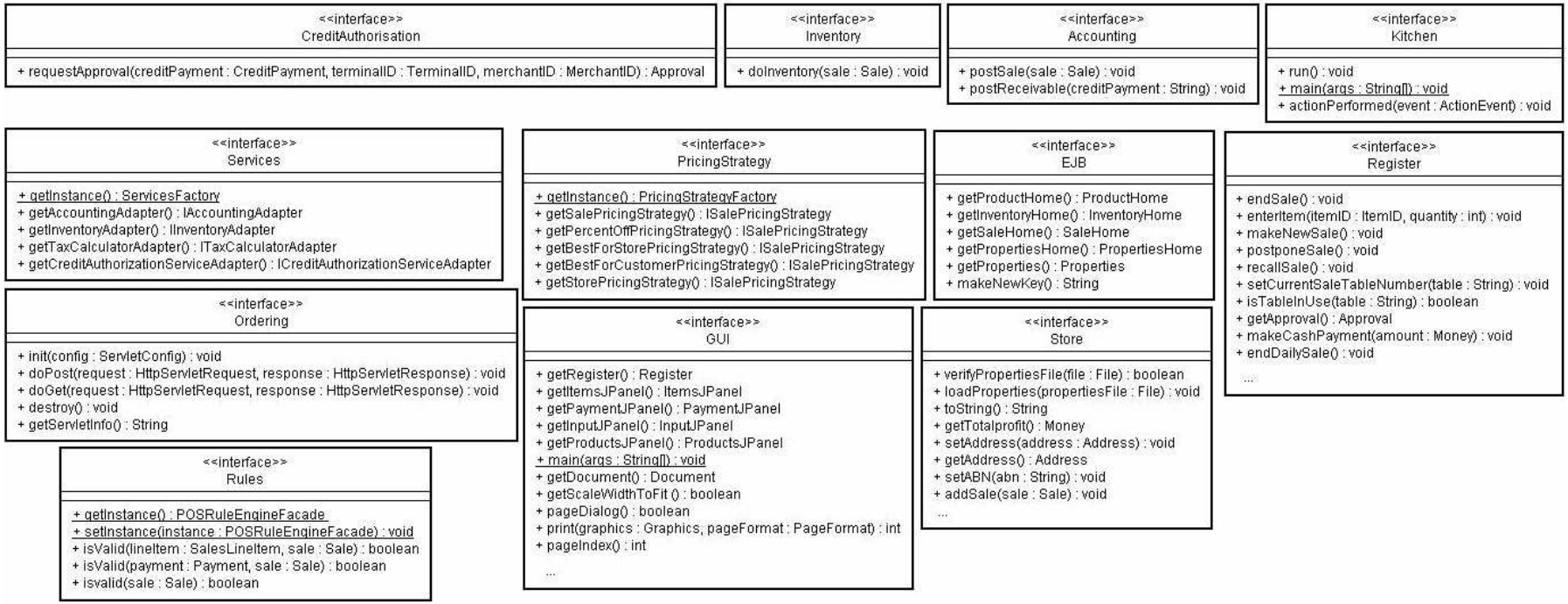

Fig.1 Component Interfaces.

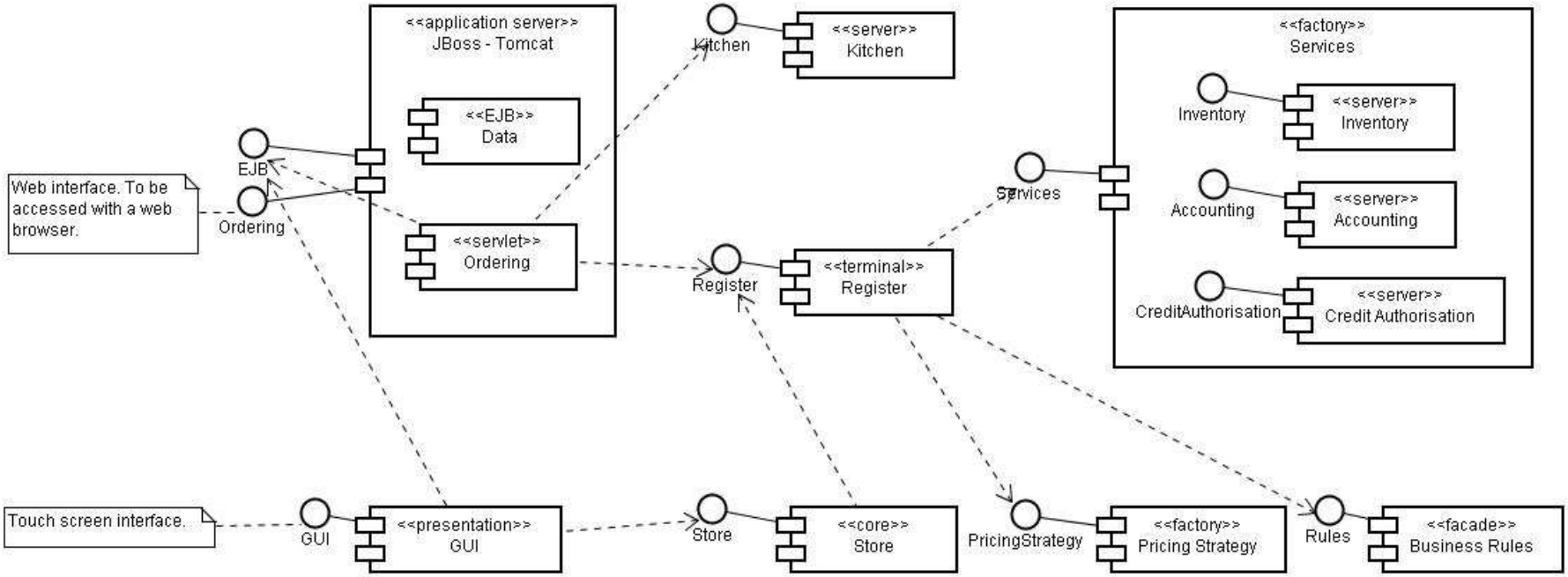

Fig.2 CommPOS Software Architecture.

Next, we modelled the components and connectors using a UML Component diagram (Fig.2). It has to be noted though that ADL connectors are much more powerful than UML dependencies. A dependency is simply a unidirectional relation between two components whereas an ADL connector can have functionality of its own and can be of arbitrary multiplicity (connecting many-to-many).

Finally, we used OCL to specify constraints as well as pre conditions and post conditions. Included bellow, are few example statements (relating to Fig.1 and Fig.2).

```
context  Store::setABN(newABN:String)
pre      newABN.length == 11
post     self.abn = newABN

context  PricingStrategy::getInstance()
post     result = pricingStrategyFactory

context  Register::makeNewSale()
pre      sefl.sale.isComplete == true
post     self.sale.isComplete = false and
self.sales.size = self.sales.size@pre + 1

context  Store
inv:     self.register <> null

context  Register
inv:     self.catalog.size > 0

context  Accounting
inv:     external <> null or local <> null
```

7 Conclusion

This paper has presented a practical approach to the modelling software architectures in UML by modelling components, connectors and configurations. It has done this by using UML to successfully model the software architecture of CommPOS. We believe that the approach presented is sufficiently clear and simple, as to be of use to software engineering developers and practitioners. An additional advantage of this approach is that it uses UML, which is a language that is widely used in the software engineering community, thereby increasing the degree of comprehension of architectural models.

The paper has also identified existing issues in contemporary software architecture research and modelling. There is no consensus in the research community on what software architectures really are and how to represent them. Therefore, software architectures are often represented informally by using inappropriate notations. They are also often represented as views (an example is the layered view). ADLs take the approach to the representation of software architectures by modelling components, connectors and configurations (although many ADLs provide many other features as well).

Many researchers have tried to tackle these issues by making their own definitions and creating their own ADLs for the modelling of software architectures. That has resulted in a large number of different ADLs, each taking a different approach to the modelling of software architectures and each having a different syntax and semantics. There is no standard ADL yet and therefore ADLs haven't gained much acceptance in the developers' community. Some research has already been done on using UML to model software architectures, but it focuses on using UML to mimic the features of existing ADLs [6], [17]. Existing ADLs are not compatible with each other in the first place and if UML is going to be used as an ADL, it wouldn't have to be compatible with other ADLs either.

8 Future Work

Further research into using UML for the modelling of software architecture behaviours and dynamic software architectures is needed. This paper did not specifically address behavioural aspects of software architectures found in many ADLs. Even though UML state diagrams could be used to represent states of individual components, they may not suffice for the modelling of inter-component behaviours. Various other diagrams, object diagrams, activity diagrams, sequence diagrams and communication diagrams could possibly also be used to model architecture behaviour. Another area requiring further research is the representation of dynamic software architectures in UML. Some ADLs (Rapide for example) allow for the modelling of software architectures in which the number of components, connectors and configurations may vary over time.

References:

[1] Medvidovic, N, Rosenblum, DS & Taylor, RN, 'A Language and Environment for Architecture-Based Software Development and Evolution', in *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, USA, 1999, pp. 44-53.

[2] Andersson, J, 'Issues in Dynamic Software Architectures', in *Proceedings of the 4th International Software Architecture Workshop ISAW-4*, Limerick, Ireland, 2000, pp. 111-114.

[3] Rumpe, B, Schoenmakers, M, Radermacher, A & Schurr, A, 'UML+ROOM as a standard ADL?', in *Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems ICECCS '99*, Las Vegas, Nevada, United States, 1999, pp. 43-53.

[4] Dashofy, EM, Hoek, A & Taylor, RN, 'An infrastructure for the rapid development of XML-based architecture description languages', *in Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, USA, 2002, pp. 266-276.

[5] Medvidovic, N & Rosenblum, D, 'Domains of Concern in Software Architectures and Architecture Description Languages', in *Proceedings of the 1997 USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, 1997.

[6] Medvidovic, N, Rosenblum, DS, Redmiles, DF & Robbins, JE, 'Modeling software architectures in the Unified Modeling Language', *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 1, 2002, pp. 2-57.

[7] Clements, PC & Northrop, LM, 'Software Architecture: An Executive Overview', CMU/SEI-96-TR-003, ESC-TR-96-003, Software Engineering Institute, Carnegie Mellon University, 1996.

[8] Bachmann, F, Bass, L, Carriere, J, Clements, P, Garlan, D, Ivers, J, Nord, R & Little R, 'Software Architecture Documentation in Practice: Documenting Architectural Layers', CMU/SEI-2000-SR-004, Software Engineering Institute, Carnegie Mellon University, 2000.

[9] Garlan, D, 'Software architecture: a roadmap', *in Proceedings of the 22nd international Conference on Software Engineering, The Future of Software Engineering ICSE2000*, Limerick, Ireland, 2000, pp. 91-101.

[10] Medvidovic, N & Taylor, RN, 'A Classification and Comparison Framework for Software Architecture Description Languages', *IEEE Transactions on Software Engineering*, vol. 26, no. 1, 2000, pp. 70-93.

[11] Egyed, A & Kruchten, PB, 'Rose/Architect: A Tool to Visualize Architecture', in *Proceedings of the 32nd Annual Hawaii Conference on Systems Sciences*, vol. 8, 1999, p. 8066.

[12] Clements, P, Garlan, D, Little, R, Nord, R & Stafford J, 'Documenting software architectures: views and beyond', in *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, United States, 2003, pp. 740-741.

[13] Kruchten, P, 'The 4 + 1 View Model of Architecture', *IEEE Software*, vol. 12, no. 6, 1995, pp. 42-52.

[14] Bachmann, F, Bass, L, Clements, P, Garlan, D, Ivers, J, Little, R, Nord, R & Stafford, J, 'Documenting Software Architecture: Documenting Interfaces', CMU/SEI-2002-TN-015, Software Engineering Institute, Carnegie Mellon University, 2002.

[15] Kruchten, P, Kozaczynski, W & Selic, B, 'ICSE 2001 workshop on describing software architecture with UML', *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 6, 2001, pp. 78-79.

[16] Object Management Group, viewed 20 September 2004, <http://www.omg.org>.

[17] Pérez-Martínez, JE, 'Heavyweight extensions to the UML metamodel to describe the C3 architectural style', *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 3, 2003, p. 5.