Open access • Journal Article • DOI:10.1145/3282517.3282545

The ALI Architecture Description Language — Source link

Umaima Haider, John D. McGregor, Rabih Bashroush

Institutions: University of East London, Clemson University

Published on: 02 Jan 2019 - ACM Sigsoft Software Engineering Notes (Association for Computing Machinery)

Topics: Architecture description language, Software architecture description, View model, Requirements traceability and Information system

The ALI Architecture Description Language

Umaima Haider
Dept. of Engineering & Computing
University of East London
London
UK
uhaider@clemson.edu

John D. McGregor
School of Computing
Clemson University
South Carolina
USA
johnmc@clemson.edu

Rabih Bashroush
Dept. of Engineering & Computing
University of East London
London
UK
r.bashroush@qub.ac.uk

Abstract

Architecture Description Languages (ADLs) have emerged over the past two decades as a means to abstract details of large-scale systems in order to enable better intellectual control over the complete systems. Recently, there has been an explosion in the number of ADLs created in the research community. However, industrial adoption of these ADLs has been rather limited. This has been attributed to various reasons, including the lack of support of some ADLs for: variability management, requirements traceability, architectural artefact reusability and multiple architectural views. To overcome these limitations, this paper is a report on ALI, an ADL that was designed to complement existing work by adding mechanisms to address the aforementioned limitations. The ALI design principles, concepts, notations and formal semantics are presented in this paper. The notation is illustrated using two distinct case studies, one from the information systems domain – an Asset Management System (AMS); and another from the embedded systems domain - a Wheel Brake System (WBS).

Categories and Subject Descriptors

D.3.3 [**Software and its engineering**]: Software system structures – software architectures

General Terms

Design, Language.

Keywords: Software architecture, Architecture description languages

1. Introduction and Motivation

Within the software engineering community, the concept of software architecture started to emerge as a distinct discipline in 1990 [1] which led to an explosion of interest during the 1990s and 2000s, referred to as the "Golden Age of Software Architecture" [2]. Today, software architecture is growing from its adolescence in research laboratories to the responsibilities of maturity, which was predicted by Shaw [3] over a decade ago. But that does not mean that the time for research, innovation, and enhancement is past. In fact, it brings an additional responsibility to show not just that ideas are promising (adequate grounds to continue research) but also that they are effective (indispensable grounds to move into practice) [3]. In other words, it is a coupling between ongoing research and actual application to make new ideas practical. For this reason, software architecture has drawn considerable attention from both academia and industry.

The increasing complexity of software and the critical nature of its use are driving a rapid maturation of the field of software architecture. According to Garlan [4], a critical issue in the design and construction of any complex software system is its architecture; that is, its organization as a collection of interacting elements – modules, components, services, etc. Thus, a well-designed architecture ensures the quality and longevity of a software system. A number of approaches exist that can describe a software architecture, ranging from formal notations (e.g. ADLs), semi-formal (e.g. UML) and informal (e.g. boxes and lines, videos, etc.).

Architecture Description Languages (ADLs) are currently considered to be viable tools for formally representing the architectures of systems at a reasonably high level of abstraction to enable better intellectual control over the systems [5]. An ideal ADL is considered to be both human-readable and machine-readable. An ADL must be simple, understandable, encompassed by multiple architectural views and syntactically flexible. With regards to this, Lago et al. [6] presented a general framework of requirements for the next generation architectural languages by taking into account current architectural needs of both the academic and industrial worlds.

Over the past two decades, a number of ADLs [7] have been developed as compared to the number of ADLs reported in [8], [9] but the majority of the problems have not been resolved. Among those, a frequent problem is that ADLs have gained wide acceptance in the research community as a means of describing system designs but their current industrial adoption level is still reported to be as low as before with some exceptions, for example, in the embedded systems domain [10-13]. This could be due to a number of reasons identified in [14-16], including the mismatch between their strengths and the needs of practitioners.

Many existing ADLs tend to focus on a specific aspect of a system (e.g. system structure), or be geared towards a particular application domain (e.g. embedded systems). While domain specific notations can be well tailored to serve particular application area needs, today's systems (and system-of-systems) cross traditional design boundaries, where software persists across various layers (e.g. Cyber-physical systems, Smart Cities systems, etc.). Thus, to be able to use an ADL in such domains, it would need to have the flexibility and expressiveness that allows it to stretch beyond a single application domain.

There has recently been an increase in the usage of variability mechanisms at the architectural level (e.g. to represent product families or runtime system adaptation). Variability management allows a) the development and evolution of different versions of software and product variants, b) planned reuse of software artefacts, and c) well-organized instantiation and assessment of architecture variants [17]. An ADL with the capability to capture and express such complex variability exhibited in software systems would empower architects to build and model more sophisticated systems.

To overcome these aforementioned limitations, we propose ALI - an **A**rchitecture Description **L**anguage for **I**ndustrial Applications [18]. ALI has been designed to be a comprehensive language, suited for different types of systems, from individual systems, to product lines, and system-of-systems. A major goal of ALI is to provide a blend of flexibility and formalism. Flexibility means it is easier to use and informative enough to convey the needed information to the stakeholders involved in the architecting phase. Formalism, on the other hand, paves the way for developing better tool support and automated analysis. ALI is designed to be highly customisable to provide support for a wide range of application domains. ALI is built on existing literature, and takes into consideration the latest recommended guidelines [4, 6, 16] and characteristics [14].

The remainder of this article is organised as follows: Section 2 discusses the current literature while highlighting the limitations of existing work. Section 3 presents the design principles behind ALI and the high-level (abstract) description of ALI in the form of a conceptual model. Section 4 covers the details of the language, visiting the different constructs in the ALI notation. Section 5 describes ALI structural and behavioural semantics. Section 6 illustrates, through two case studies, the ALI ADL. Evaluation of the case studies and the analysis are then discussed in Section 7. Section 8 discusses the study limitations. Finally, Section 9 concludes the paper by summarizing the important points and outlining future research directions.

2. LITERATURE REVIEW

Several ADLs [7] have been developed over the past two decades. Existing literature is critically analysed in the next subsection, with limitations discussed in the following one.

2.1 Critical Appraisal of Existing Work

Since the early 90's, a thread of research on formal architecture description languages (ADLs) has evolved. Many different ADLs have been proposed in the literature for modelling architectures both within a particular domain, and as general-purpose architecture modelling notations.

All the classical ADLs (also considered first generation ADLs [19]) compared and analysed by Medvidovic and Taylor [8] were conceptually based on structural architecture modelling features (components, connectors, interfaces and architectural configuration). Another ADL survey was conducted by Clements [9] in the same era. Some of the second generation ADLs have been compared in [20] but it covers a very limited number of characteristics of the languages.

Reflecting on existing literature, it is interesting to note that very few ADLs were originated in industry. Described below are the three widely cited ones.

AADL [10] (Architecture Analysis & Design Language) derived from the MetaH [21] ADL, is a SAE standard formal modelling language for describing software and hardware system architectures and uses a component-based notation for the specification of task and communication. It provides precise execution semantics for system components, such as threads, processes, memory, and buses. All external interaction points of a component are defined as *features*. Data and events *flow* through and across multiple components. The AADL *Behavioural annex* describes nominal component behaviour and the *Error annex* describes flows in the presence of errors.

Koala [12] is a component oriented ADL based on key concepts from Darwin [22]. Basically, it was designed with the aim of achieving a strict separation between component and configuration development in order to reuse software components in many different configurations for different product variants, while controlling cost and complexity.

EAST-ADL [11] defines an approach for describing automotive electronic systems through an information model that captures engineering information in a standardized form, provides separation of concerns and embraces the de-facto architecture of automotive software – AUTOSAR [23]. It covers a variety of aspects -functions, requirements, variability, software components, hardware components and communication.

Although these ADLs come from different industries, they all relate to the embedded systems domain. AADL and EAST-ADL emerged from the avionics and automotive industries and are currently widely used in their respective domains. Koala, on the other hand, was developed within the consumer electronics domain, though its use has not seen the same proliferation as the previous two. These industrial ADLs, that are limited (perhaps by design) to the embedded systems application domain, would not be suitable to model cross-domain applications (e.g. Cyber-Physical applications encompassing embedded as well as Information systems).

On the academic side, a large number of ADLs have been proposed, each characterised by slightly different conceptual architectural elements; different syntax or semantics; varying emphasis on a single view (structural or behavioural) or operational domain such as embedded system; or for specific analysis techniques.

Below are some of the ADLs developed in academia:

- ACME [24] is a general purpose ADL proposed as an architectural interchange language.
- Darwin is a *declarative* ADL which is intended to be a general purpose notation for specifying the structure of distributed systems composed from diverse component types using diverse interaction mechanisms [22].
- UniCon [25] creates a useful, pragmatic and extensible test-bed that would allow the architectural abstractions used by practitioners (such as pipes, filters, objects, clients and servers) to be captured and reasoned about in a systematic manner.
- xADL[26], an XML based architecture description language, is defined as a set of XML schemas and has been designed to use the standard XML infrastructure and to be easily extensible using standard XML-Schema extension mechanisms.
- C2 is a component- and message-based ADL which simplifies the definition of architectures following the Chiron-2 ("C2") style [27].
- Rapide [28] is an event-based concurrent object-oriented language specifically designed for prototyping architectures of distributed systems.
- WRIGHT [29] is designed with an emphasis on analysis of communication protocols and provides formal semantics for an entire architectural description by extending Communicating Sequential Processes (CSP) [30]. Wright has been extended, termed Dynamic WRIGHT [31], with the ability to handle foreseen dynamic reconfiguration aspects of architecture.

According to the ANSI/IEEE 1471-2000 standard, structural and behavioural viewpoints are the two most important and frequently used viewpoints for architectural description. The specification of each viewpoint with their entities is elucidated in [19, 32]. A great challenge for an ADL is being able to describe static and dynamic software architectures from structural and behavioural perspectives.

ADLs like ACME [24], Aesop [33], Aspectual-ACME [34], Darwin [22], Koala [12], MontiArc^{HV} [35], UniCon [25], Weaves [36] and xADL [26] were focused largely on the structural concerns of software architecture. On the other hand, some ADLs covered both behavioural and structural specifications, including: AADL [10], ABC/ADL [37], ADLARS [13], ADML [38], C2 [27], CBabel [39], EAST-ADL [11], LEDA [40], MetaH [21], PrimitiveC [41], PRISMA [42], Rapide [28], SOADL [43], xADL [26], XYZ/ADL [44], vADL [45], WRIGHT [29, 31], Zeta [46], π-ADL [19] and π-SPACE [47]. Only a few ADLs cover behavioural aspects, such as Monterey Phoenix [48], yet, these ADLs do not treat behaviour as a first class citizen (tend to be merged with structural description).

Additionally, most of these languages (except [22, 29]) define structural elements using their own bespoke notation. Some ADLs (such as AADL) have a core language that contains constructs for representing structural and behavioural aspects of the architecture. Some used different processes to define the behavioural description. For example, Rapide describes behaviour through partially ordered event sets (or "posets"); Wright uses CSP with minor extensions; LEDA, PRISMA, SOADL, vADL, π-ADL and π-SPACE use the π-calculus.

Generally, the overall architectural structure of ADLs focuses on the basic component, connector and system paradigm. All ADLs that have been analysed so far treat components as first class citizens, but in some languages [10, 12, 13, 21, 22, 28, 35, 40, 45, 49-54] there is no notion of connectors as first class citizens. Connectors are not even defined. This does not mean that we cannot create a useful language without first class connectors. There are viable and potentially useful architectural languages that have been created without them, like [10, 12, 22, 28]. [36, 38, 40, 45, 46, 51, 52, 55, 56] do not support an architectural configuration as a first class element. Neither connector nor architectural configuration was considered first class citizens in [40, 45, 51, 52]. Both types of ADLs, the ones that support connectors, and the ones that do not support connectors, tend to impose architectural decisions on the architect (by forcing the architect to have a connector or not). The need here is for more flexibility to allow architects to decide whether a connector is need or not based on the system under development (rather than an ADL restriction).

There are few second generation academic ADLs that focus mainly on the behavioural modelling in a slightly different way as compared to traditional ADLs. Monterey Phoenix [48] is an ADL in which behaviour of the system is defined as a set of events (event traces) with two basic relations: precedence and inclusion. Different types of patterns (such as alternative, optional, etc.) are defined in the form of an event trace that occurs in a transaction. But they lack the unique visual notation for each of these event patterns. A schema is defined as a set of transactions that includes all possible event traces. It can be tedious to understand (especially visually) and sometimes becomes more complicated when it is encapsulated with several pattern types in a single schema, particularly, in the case of large-scale and complex systems.

PrimitiveC-ADL [41] is a component-based language that modifies the application architecture by subdividing components into subsystems of static and dynamic elements. A *design pattern* typically shows relationships and interactions between components' dynamic behaviour parts. The *decision policy* proposes the use of a *design pattern* and the application of the *decision policy* depends on a *scenario*. The main problem in [41] and other ADLs [19, 45] is that while they define the behaviour of the system within a component or in their configuration, behavioural elements are not explicitly defined. In other words, it provides a single view of the system which is not suitable for a large-scale industrial system where component behaviour varies enormously. In that case, component definition becomes complex and it is difficult to differentiate static and dynamic parts.

AspectLEDA [57] is an ADL that provides behavioural specification of the system using the UML use case and activity diagrams by adopting the Aspect-Oriented (AO) approach. Each use case diagram represents a component that constitutes the system and its interactions are expressed in the form of sequence diagrams. Subsequently, a sequence diagram for every use case contains by default an aspect component as each use case is extended with an aspect. In other words, it describes the interactions among components visually via a UML sequence diagram with its dependency on an AO approach. Looking at this, component interactions need to be more elaborative in a sense by considering component interfaces (or ports) that are involved in the interaction which would be helpful to design complex systems.

Another major element that needs attention with regards to ADLs, is the concept of variability. This is a very important and critical area when it comes to its use in the architectural description, especially in large-scale industrial applications [13, 58]. Variability is the ability to design for a planned set of changes for deployment in specific contexts [17]. It facilitates the development of different variants of a system architecture. Variability is largely taken into account in the architecture and design phase of software engineering [59]. Although there are several ADLs where variability has been studied, variation is specific to describing a set of related products as in a software product line (SPL). Among the ADLs are: PL-AspectualACME [60], ADLARS, EAADL [61], LightPL-ACME [62], vADL, and DSOPL [63]. Other ADLs that consider variability as a separate entity are: MontiArc^{HV} and Δ-MontiArc [64].

Software architecture typically plays a key role as a bridge between requirements and implementation [4]. In terms of ADLs, a challenge in bridging this gap is how to trace feature (requirements) into the architecture description particularly, into each architectural element. So far, in the research literature, ADLARS and LightPL-ACME are the only two ADLs that made an attempt to capture the relationship between the system's features and the architectural structures. Both assumed a feature model as a precursor to the architecture design process and were limited to specifying a product line.

It is worth mentioning that there are few ADLs that try to represent different aspects and domains in the architecture by presenting it in the form of different versions. Each focuses on a particular aspect/domain. For instance, ACME has been extended to AspectualACME with its descendant PL-AspectualACME, LightPL-ACME, Cloud-ADL [65] and ADML; MontiArc [66] to MontiArc^{HV}, Δ-MontiArc and MontiArcAutomaton [67].

There is a framework known as ByADL (Build Your ADL) [68] that supports a software architecture team in defining their own ADL by allowing software architects to (i) extend existing ADLs with domain specificities, new architectural views, or analysis aspects, (ii) integrate an ADL with development processes and methodologies, and (iii) customise an ADL. Basically, it takes the meta-model of the ADL to be extended as an input.

Overall, a common pitfall for the discussed ADLs is their limited ability to support large-scale real-life applications. Some possible reasons behind this are discussed in [14, 16]. Limitations are further discussed in the next section.

2.2 Limitations within Existing ADLs

After critically analysing the existing ADL literature, particularly around scalability and uptake (industrial adoption), it was evident that only ADLs that were originated in industry saw some level of industrial adoption. This has been attributed to potential misalignment between practitioner needs and the academic focus [16].

Below, we summarise some of the main limitations identified in ADLs that emerged from academic research, but failed to achieve any notable industrial adoption:

L1: ***Limited support for variability management***
To manage the size and complexity of industrial systems, and with the current trend of delaying architectural decisions as much as economically feasible (and the shift of variability from hardware to software), it is valuable to have the capability of modelling variability adequately in the architecture design.

L2: ***No explicit mechanism to link requirements to architectural artefacts***
Requirements traceability has emerged as a main objective in industry. Yet, without the support for capturing such relationships at the architecture description stage, the link between requirements and implementation becomes difficult to establish and maintain. For example, although AADL does not support modelling such relationships natively, tools such as AADL's OSATE provide such mechanisms outside the core ADL.

L3: ***Application domain dependency***
As can be seen from the previous section, many ADLs are tailored for a particular application domain, with embedded systems having the majority of such systems. However, given the way today's systems are evolving with the rise of the Internet of Things (IoT), Cyber-Physical Systems (CPS), and Smart Cities to name a few, for an ADL to be capable of modelling a complete solution, it needs to cross cut multiple application domains.

L4: ***Restrictive syntax***
Many ADLs impose a strict syntax and design principles on the architect (e.g. layered model, network model, etc.). Building ADLs in such a way allows the ADL designer to provide various automated architectural analysis. However, from a practitioner perspective, the last thing needed is to be forced to reason about the system in a specific way.

L5: ***Lack of support for architectural artefact reusability***
Existing ADLs have been designed to support the abstraction of details; however, support for architectural artefact reuse across multiple projects is lacking. While architecture reuse has seen some success in specific domains, e.g. the automotive domain using AADL [10], the granularity of reuse remains relatively small. In order to support large-scale reuse, ADLs would need to provide mechanisms to capture some degree of variability in the description of artefacts (and their interfaces) to enable redeployment in multiple contexts.

L6: ***Overloaded architectural views***
Given that one of the main benefits of having an overall system architecture description is to use it as a communication vehicle among the various stakeholders, not all the information captured within the architecture relate to every stakeholder. Accordingly, ADLs providing one or two architectural views tend to suffer from information overload. The importance of having multiple architectural views has also been highlighted in [32].

L7: ***Focus on structure more than behavioural architectural aspects***
The structural description of a system changes less frequently compared to the behavioural description because systems can serve different objectives with the same structural description. In other words, the structural description can encapsulate more than one behavioural description. Yet, it can be said that most ADLs still overlook the importance of behavioural description. While it is viewed as a major construct in some [10, 28], it is not covered in many [25, 29, 36], with fewer ADLs supporting the representation of behavioural architectural knowledge graphically [69].

In the following sections, we discuss the ALI ADL [18], which builds on initial work done in [70], to overcome these limitations.

3. DESIGN PRINCIPLES

Analysing the practitioner's needs as discussed in [16, 71], and in the literature review as discussed in Section 2, our goal is to design a language that is simple enough to be usable in an industrial setting, yet formal enough to adequately support automated analysis and other essential functionalities (extensibility, evolution, traceability etc.), though the latter is outside the scope of this paper.

To achieve this goal, we present the set of principles, which were used to drive the development of the ALI notation in order to address the limitations pointed out in the previous section. Additionally, this section provides an overview of ALI's conceptual model.

3.1 Principles

Six general principles guided the creation of the ALI ADL:

P1: Variability management
Software architects need adequate support for dealing with variability in designing their system architecture. As stated in [72], it is essential for the architect to have suitable methods for handling (i.e., representing, managing and reasoning about) variability. From an architectural description perspective, variability is a concern of multiple stakeholders, and in turn affects other concerns. So,

variability needs to be treated in a similar way to other essential functionalities of the architectural language.

Our proposed ALI ADL treats variability as a first class citizen and manages it as an integral part of the language. It provides the ability to manage variability not only in the overall system architecture description but also in the design of individual architectural elements – interfaces, connectors and components. This is done with the help of a simple if/else structure concept along with the keywords "**supported**" and "**unsupported**". Additionally, ALI supports variability management in its behavioural description using the same if/else structure (details in Section 4.1.12).

P2: Requirement traceability

Tracing requirements from the problem space (specification) into the solution space (implementation) provides a valuable tool for architects to help validate the produced system against its set of objectives. However, for such an end-to-end traceability to work, there needs to be continuity in capturing relevant information at all development stages, including architecture.

ALI supports requirements traceability by supporting the linkage of end-user features (see Section 4.1.2) directly to architectural elements (components, connectors, patterns etc.) using first order logic (to allow for complex dependencies).

Additionally, ALI supports 'conditioning' the behavioural aspects of the system to external parameters (see Section 4.1.10). Such conditions can also be used to change the behaviour of the system given various external requirements.

P3: Cross application domain modelling

With the emergence of new paradigms such as the Internet of Things (IoT) and Cyber-Physical Systems (CPS), architecture descriptions are now faced with the challenge of encompassing multiple application domains. For example, if we consider the Smart Cities scenario, systems in this application domain will entail the applications running sensor platforms (IoT devices), communication gateways, databases (BigData infrastructure), and Information Systems that deliver end-user services. While the architecture of the sensor systems can be modelled using embedded-system oriented ADLs, these ADLs do not necessarily lend themselves to representing the architecture of Information Systems. Thus, there is a need for new generation ADLs that are capable of modelling system across traditional design boundaries.

ALI supports cross-application domain modelling by introducing flexibility in the notation design at different levels. For example, ALI allows for the creation of custom interface types using a dedicated notation. The case studies discussed show 'port' like interfaces, as well as 'WSDL' interfaces.

P4: Balance formality and flexibility to better support the design process

During the early stages of the design process, architects tend to sketch things at a very high level, using mere lines and boxes. At that stage, for example, it is difficult to start talking about the details of interfaces between components or what meta information to capture about each architectural element.

As the system development process progresses, and as more details are captured about the system, specific details in relation to architectural elements can then be discussed and modelled. Thus, an ADL needs to allow some flexibly at the initial stages of the design process, and at the same time, provide the required formality when details are available.

For example, ALI achieves this balance between formality and flexibility by allowing architects to work with undefined interface types. When such informal interfaces are used, no automated analysis would be possible. Once the design is mature, and interface types are created, ALI could then provide an array of verification checks (as specified in the interface type description).

P5: Increase architectural artefact reusability

Architectural artefacts tend to be tailored to particular system requirements. Accordingly, very few ADLs discuss the concept of artefact reusability across multiple systems. Yet, in real-life, it is more often than not we are faced with similar architectural challenges that could be solved using architectural artefacts we have in existing projects. ALI supports the concept of large-grain reusability by allowing architectural artefacts to be made configurable based on selected sets of features.

For example, components in ALI can be customised based on which features are selected for a particular component, and the values of these features. Similarly, connectors and interfaces can also be parameterised using feature sets. By mapping the feature set of the source domain (where the component is taken from) and the feature set of the destination domain (where the component is going to be deployed), the component can adapt to the new environment (further details in Section 4.1.6).

P6: Multiple architectural views

As systems increase in size and complexity, and as more and more stakeholders take interest in system development (product managers, architects, end-users 'in the loop', etc.), the information captured within an architecture description is expanding. Accordingly, in order to minimise information overload, and sacrifice abstraction for completeness, the need for multiple architectural views catering for different stakeholders is becoming an important feature of an ADL.

ALI is designed with the concept of multiple architectural views, where each view corresponds to a stakeholder (or stakeholder group) and addresses a different set of concerns (see Section 4.1 for textual and Section 4.2 for graphical descriptions).

3.2 Conceptual Model

A *conceptual model* is a high-level description of how a system is organized and operates [73]. The aim of a conceptual model is to express the meaning of terms and concepts used by domain experts and to find the correct relationships between different concepts. Several notations [74-77] exist that are used to describe the conceptual model. Among those, UML (*Unified Modelling Language*) [74] is a widely used and comprehensive notation (ISO/IEC 19501).

Fig. 1 shows the conceptual model of ALI. The *reference architecture* represents the overall system description. The reference architecture is made up of *arrangements* and *viewpoints*. Arrangements represent the structural (static) description of the system and are composed of *components* and *connectors*, which communicate through *interfaces*.

Viewpoints are sets of *transaction domains* that pertain to a common concern (e.g. car ignition system). Transaction domains represent the behavioural (dynamic) aspects of the system, and are composed of sets of *transactions* that together serve a particular system feature (e.g. user/key validation). Transactions are expressed in terms of sets of *events* that achieve a system functionality (e.g. key authorisation). And events are the basic communication mechanism between components (e.g. key code update event).

Conditions are parameters that represent external (to the system) environmental aspects that could impact the system behaviour. The architecture description is parameterised using these conditions, which can be either true or false. Different combinations of conditions and their values, called *scenarios*, can be used to test and adapt the behaviour of the system to various contexts.

Finally, a *product architecture* can be derived from a reference architecture using a *product configuration*. Product configurations represent desired *features*, and their values, for a specific product in a product line.

In the next section, the details of these constructs, along with the notation used to describe them, are discussed.

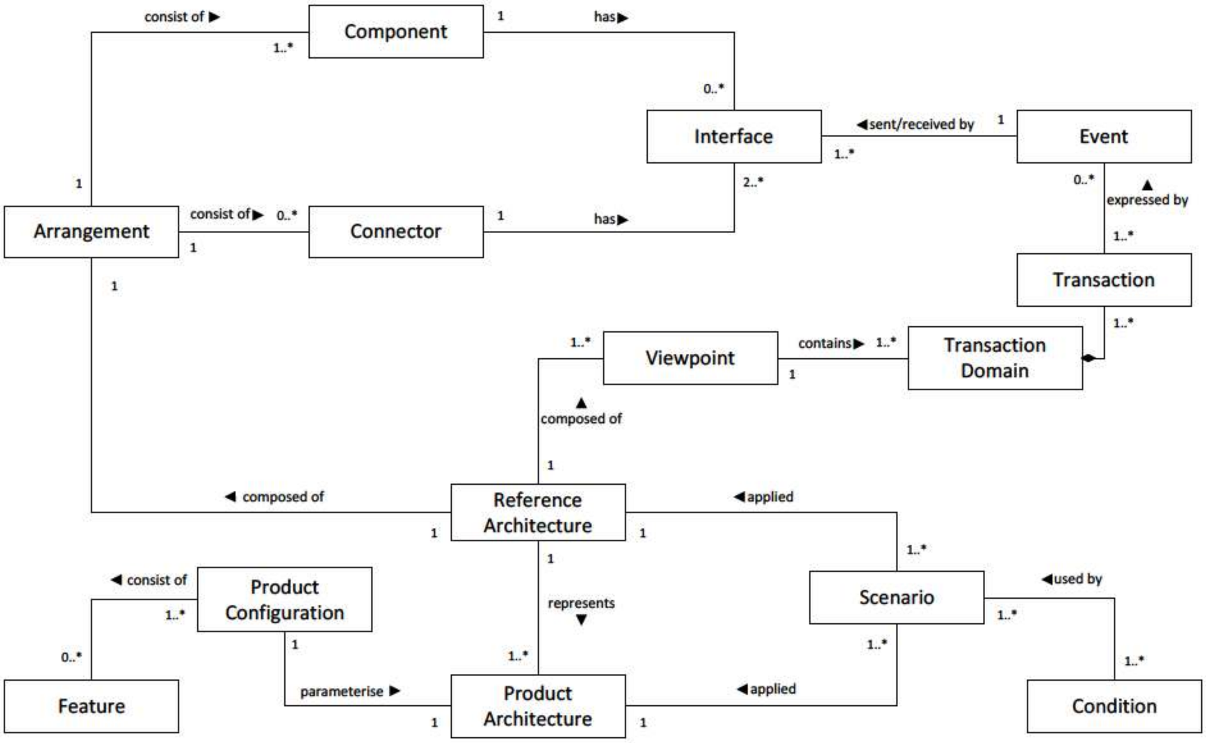


Fig. 1. ALI conceptual model

4. Constructs and Notations

Woods and Bashroush [71] observed that architectural languages constructed with complex or obscure syntactical notations are rarely used correctly. Generally, practitioners avoid adopting complex languages into their development process especially in large scale systems where it becomes tedious to handle and understand. Malavolta et al. [16] conducted an industrial survey which shows that architectural languages need to be simple and intuitive enough to communicate the right message to the stakeholders involved in the architecting phase along with some formality in order to drive analysis and other automation tasks. The ALI notation is designed accordingly, where the syntax (graphical and textual) is kept simple and flexible, yet formal enough to conduct automated analysis.

In the following sections, we discuss the main constructs of the ALI notation. Where applicable, we demonstrate the concepts using parts of the Asset Management System (AMS). AMS is used to manage investment portfolios of financial instruments such as equities and commodities (see Section 6.1 for further details).

4.1 ALI Textual Notation

The ALI textual notation is designed based on the principles defined in Section 3.1. The textual notation is made up of 14 main constructs as listed in Table 1.

These constructs are discussed in the following sections.

4.1.1 Meta Types

The *meta types* section provides a formal syntax for capturing meta-information of the architectural element (e.g. components, connectors, etc.). A meta type is defined by the information it encompasses. The information is stored in fields, where each field has a name (*tag*) and a data type (text, number, etc.). The following example defines a meta type called `Meta_ServerEquity`:

```
meta type Meta_ServerEquity {
      tag creatorID, intention: text;
      tag cost, version*: number;
      tag last_updated: date;
  }
```

In this example, "meta type" is a keyword which is used to start a meta type definition. `Meta_ServerEquity` is the name of the meta type being specified. Each meta type contains a set of tags each of which is either textual, numeric, date, enumeration or character. Five tags are defined in the above example: two textual, two numeric and one date. The asterisk "*" on the `version` tag indicates that it is an optional tag.

TABLE 1

ALI Constructs

Construct	Description
Meta types	Provide an extensible mechanism for capturing architectural meta-information
Features	The system features are catalogued
Interface templates	Specifies a dedicated notation for creating categories of interface types
Interface types	Architectural interfaces are defined
Connector types	Architectural connectors are defined
Component types	Architectural components are defined
Pattern templates	Reusable architectural design patterns are defined
Product configurations	Provides the feature combinations that characterise individual products
Events	The events that flows within a system are defined
Conditions	The system behavioural conditions (architecture parameterisation) are catalogued
Scenarios	Behavioural scenarios are defined (sets of conditions to represent a runtime scenario)
Transaction domains	Provides the behavioural interactions within a particular system domain
Viewpoints	Different behavioural viewpoints are defined
System	The overall system architecture is described

Once meta types are specified, *meta objects* conforming to these types can be created and attached to architectural elements throughout the architectural description. These meta objects provide an area for appending additional information related to these elements.

Below is an example of a meta object that conforms to the meta type defined in the example above.

```
meta: Meta_ServerEquity {
     creatorID: "Martin005";
     intention: "Acts as a mediator to manage
                 equity portfolio";
     cost: 5,000;
     version: 1.3;
     last_updated: 20-02-2017;
}
```

A meta object could also be a combination of more than one meta type. To enhance the language flexibility, it is also possible to create meta objects that do not conform to any meta type. However, little automated analysis can be performed on such informally described data. The reason for allowing objects with no type is to enable architects to sketch what they initially think could be relevant meta-information, then, once confirmed, they create the appropriate meta types to ensure conformance.

The formal specification of meta information allows for easier CASE tool development to harness these meta objects and conduct automated analyses (e.g. cost/benefit analysis, project timing/scheduling, etc. depending on what type of meta information is available). Other meta information could include: design decisions, component compatibility, etc. which when extracted and formatted using proper CASE tools, allow automated architecture documentation to be achieved on-the-fly.

In general, it is expected that the meta types will be created once and used repeatedly across the different systems developed by the same enterprise. In order to make sure that critical information is always provided within an architecture description, a project management team (or any other stakeholder) may first identify the standard set of information required (tags), and then provide it to architects for conformance. The flexibility in the syntax also allows the architects to augment this information with fields (tags) that they may want to use internally within the architecture team.

4.1.2 Features

The feature description notation provides a catalogue of the system *features* (mandatory, optional or alternative) used within the system. The feature definition comprises of:

- *alternative names*: In many cases, different teams within the development process address a feature with different names. This sub-section of the feature definition keeps track of the different names (if any) that are used to address the same feature. This property will keep track of the system features and reduce redundancy.
- *parameters*: A feature can carry different types of parameters -textual, numerical and Boolean. Though, not all features would be parameterised.

Below is an example of how features are defined in ALI:

```
features {
  Equity: {
     alternative names: {
          Designer.FI1, Developer.Ey,
          Evaluator.F11;
       }
     parameters: {
          {Equity_Type = text};
        }
       }

  Equity_Derivative: {
     alternative names: {
          Designer.ID1, Developer.SD,
          Evaluator.F14;
       }
     parameters: {
          {Derivative_Type = text,
           Premium_Period = text,
           OTC = boolean};
       }
      }
   // similarly other features can be defined
 }
```

In the example above, `Equity_Derivative` was defined showing that it is referred to as `ID1` by the design team, `SD` by the development team, and `F14` by the evaluation team. The feature encompasses three parameters, two textual and one Boolean.

In ALI, system features are defined in a stand-alone catalogue as shown above. The catalogue serves as an adapter between any feature modelling technique used and the architecture description, making ALI independent of any particular feature modelling technique.

4.1.3 Interface Templates

The *interface template* notation provides a framework that allows the description of multiple interface type categories within a system description. The idea behind this is to create a set of common interface templates (e.g. WSDL, RMI, etc.) needed within an application domain once, and reuse them in different projects. These interface templates can be used as a specification in defining the *interface types* of the system, either explicitly (as explained in the next section) or in the *component type* definition (Section 4.1.6). This template specification can also be reused outside the defined system depending upon the design requirement as per principle P5 in Section 3.1.

The interface template definition is divided into three main sections:

- *provider syntax definition*: where the syntax of the provider interface is specified using a subset of the JavaCC [78] notation. JavaCC (Java Compiler Compiler) is an open source notation that allows the definition of grammars using EBNF style syntax [79]. The JavaCC specification can then be compiled to produce a parser for a particular interface definition.
- *consumer syntax definition*: where the syntax of the consumer interface is specified using a subset of the JavaCC notation.
- *constraints*: where the interface connectivity constraints are specified. These include:
  - *Should match*: here the terms (identified in the below syntax definition sections using the JavaCC notation) that should match between two interfaces to be considered compatible (allowed to bind) are identified.
  - *Binding*: comprises of three different fields: 1) *multiple* -a Boolean value that states whether multiple binding is allowed on this interface; 2) *data size* -range of the data that can pass through this interface by providing the maximum and minimum values; and 3) *max connections* – maximum number of simultaneous connections allowed on the interface.
  - *Factory*: This is a Boolean value that states whether the interface is a factory or not. A factory interface means that when a connection request is received on this interface, a new instance is created to handle that particular request while the factory interface continues to listen to new incoming requests. Example: server socket interfaces in Java are factories. On the other hand, C++ sockets do not support factory functionality by default.
  - *Persistent*: This is a Boolean value that indicates a persistent interface (the internal data of the interface component is kept unchanged after the current connection has ended) when set to true and indicates a transient interface (internal data is reset to initial values when the current connection is terminated) when set to false.

An interface template description begins with the keyword "interface template" followed by the interface template name such as `MethodInterface` as in the example below:

```
interface template MethodInterface {
  provider syntax definition: {
    "Provider"":"
     "{"
        {"function" <FUNCTION_NAME>
          "{"
             "impLanguage" ":" <LANGUAGE_NAME> ";"
             "invocation" ":" <INVOCATION> ";"
             "parameterlist" ":" "("
                          [<PARAMETER_TYPE> {","
                          <PARAMETER_TYPE}] ")" ";"
             "return_type" ":" <RETURN_TYPE> ";"
            "}" }
     "}"
    }
  consumer syntax definition: {
    "Consumer"":"
        "{"
           "Call" ":" <INVOCATION> "("
                       [<PARAMETER_TYPE> {","
                        <PARAMETER_TYPE}] ")" ";"
        "}"
   }

  constraints: {
        should match: {INVOCATION_NAME =
                  .INVOCATION_NAME,PARAMETER_TYPE}
        binding: {
             multiple: true;
             data_size: [50KB, 500MB];
             max_connections: 5;
     }
        factory: false;
        persistent: false;
       }
     }
     }
```

For further details about the notation used for specifying the interface template syntax, please refer to JavaCC [78].

It is important to clarify here that the interface template definition is not meant to be read by humans, but rather created once and then read by CASE tools that would verify the interface descriptions and connections made throughout the architecture definition.

4.1.4 Interface Types

The *interface type* notation provides a set of pre-defined interface types that are created in conformance to the definition of an *interface template*, described in the previous section. Interface types can be (re)used in the design of architectural elements (components and connectors) throughout the system description (design principle P5).

An interface type definition begins with the keyword "interface type" as in the example below:

```
interface type {
  ArithmeticOperation: MethodInterface {
        Provider: {
         function Addition
                        {
                          impLanguage: Java;
                          invocation: add;
                          parameterlist: (int);
                          return_type: void;
                        }
         function Subtraction {…}
         function Multiplication {…}
        }
        Consumer: {
           Call: getValue (long_int);
        }
}

  AverageOperation: MethodInterface {
        Provider: {
         function Average
                        {
                          impLanguage: Java;
                          invocation: average;
                          parameterlist: (int);
                          return_type: void;
                        }
         }
        Consumer: {//nothing consumed}
    }

   NumericOperation: MethodInterface {
        Provider: {
         function GetValue
                        {
                          impLanguage: Java;
                          invocation: getValue;
                          parameterlist: (void);
                          return_type: long_int;
                        }
        }
        Consumer: {
         Call: add (long_int);
         Call: subtract (long_int);
         Call: multiply (long_int);
         Call: average (long_int);
        }
     }
… //similarly other interface types can be
defined
}
```

Each interface type is defined by a unique name followed by the *interface template* name, to which it conforms. In the example above, `ArithmeticOperation` performs basic mathematical operations to calculate the portfolio value based on the value it consumed and `AverageOperation` calculates the portfolio value by using the average formula strategy. The interface type `NumericOperation` consumes values returned from these interface types and provides them to the other interface. They all conform to the interface template `MethodInterface` defined in the previous section. We can also define other interface types that conform to other *interface templates* such as WSDL, RMI, etc.

4.1.5 Connector Types

Like many other ADLs, such as ACME [24], Aesop [33], CBabel [39], EAST-ADL [11], UniCon [25], WRIGHT [31] and π-ADL [19], to name a few, connectors are considered first class citizens in ALI.

A connector type definition begins with the keyword "connector type" followed by the connector type name and is divided into three sections.

```
connector type Calculator_Equity
  {
    features: {
       MTM_Price_Method: "Share prices matched
                           with market price",
       Company_Price_Method: "Unlisted share price
                           of an individual company",
       Weighted_Average_Method:
             "Portfolio Valuation is done on the
              basis of average share price";
     }
    interfaces: {
       valueport1: ValueOperation;
       valueport2: ArithmeticOperation;
       valueport3: AverageOperation;
       valueport4: NumericOperation;
     }
    layout: {
      connect valueport4 and valueport1;
      if (supported(MTM_Price_Method ||
             Company_Price_Method))
        {connect valueport1 to valueport2;
         connect valueport2 to valueport4;}
     else if (supported(Weighted_Average_Method))
         connect valueport3 to valueport4;
     }
}
```

- *features:* a set of optional/alternative features used to parameterise a connector type. By changing feature values, a connector can be reconfigured to be deployed in different products (based on feature availability and parameter values). The configuration is achieved using if/else structures and the keywords "**supported/unsupported**" to link features to the connector definition.
- *interfaces*: where the connector interfaces are defined along with their interface types. These resemble the input/output ports of the connector. Basically, interfaces are instances of *interface types* that are defined in accordance to *interface templates*.
- *layout*: The layout section describes the internal configuration (structure/arrangement) of the connector. It demonstrates how the connector interfaces are connected internally, that is, how the traffic (information) travels internally from one interface to another. This syntax introduces a high level of configurability to the connector definition which provides better support for defining configurable product and product line architectures. Two types of configurations are allowed between connector interfaces, namely:
  - *uni-directional connections (to)*: which specify that the data from one interface goes to another interface. This is done using the keywords: "**connect**" and "**to**". Example: `connect valueport1 to valueport2` in `Calculator_Equity` outputs the data on the `valueport1` interface to the `valueport2` interface.
  - *bi-directional connection (and)*: which specify that the data can travel in both directions between two interfaces. This is done using the keywords: "**connect**" and "**and**". Example: `connect valueport4 and valueport1` in `Calculator_Equity` outputs the data on the `valueport4` interface to the `valueport1` interface and vice versa.

Additionally, the keyword "**all**" can be used to connect a connector interface to all other interfaces of the connector using a bi-directional or unidirectional communication. For example, "`connect all to all`" can be used to create bi-directional connections among all ports. We can also have "`connect valueport1 to all`" which makes the input on interface `valueport1` available as output on all other interfaces of the connector.

Lastly, ***meta objects*** can be attached to connector types by simply defining the meta object (as explained in Section 4.1.1) inside the connector type definition (anywhere between the start and end brackets.

4.1.6 Component Types

The *component type* definition is divided into three main sections:

- *features:* a set of optional/alternative features that make up a component type. The purpose and definition of this section is exactly similar to the concept of features defined in the *connector type* (see Section 4.1.5). That is, it provides the capability to reuse components in multiple products and systems by varying feature values (*product configurations*).
- *interfaces*: which specify the different interfaces used by the component. The interfaces section is divided into two sections, *definition* where new interfaces can be created from scratch; and *implements* where already defined interfaces can be reused (interfaces implemented here are instances of *interface types*).
- *sub-system*: where the internal structure (sub-system) of the component is described. The sub-system section is divided into three sections:
  - *components*: where the different sub-components included within the component are defined.
  - *connectors*: where the different connectors used in connecting sub-components are defined.
  - *arrangement*: where the way in which sub-components are connected is described. To allow flexibility, ALI provides three different methods that can be used to connect components:
    a. *Using connectors*: where a connector mediates the connection between two or more components. This is done using the keywords: "`connect`".
       Example: `connect component.interface1 with connector.interface1.`
    b. *Direct binding*: where component interfaces are bound directly without the use of a connector. This is done using the keywords: "**bind**". For example: `bind component1.interface1 with component2.interface1.`
    c. *Using patterns*: where predefined connection patterns can be used to connect a set of components according to a selected architectural pattern (see Section 4.1.7).

For example, a *component type* description for a portfolio equity valuator (which calculates the portfolio value based on the valuation method requested by the fund manager) begins with the keyword “component type” followed by the component type name `Portfolio_EquityValuator` as shown in the example below.

```
component type Portfolio_EquityValuator
  {
   meta: Meta_Valuator, Meta_ShareTradeData {
    /* demonstrates meta object comprises of two
       meta types */
    acceptance_value: “any numerical value”;
    value_approximation: “2 significant figures”;
    curreny_acceptance: “all top international
         trading currencies that exists in stock
         exchange”;
    last_request: 18-01-2017;
    intention: “to calculate the portfolio value
         on the basis of current business day
         trading”;
    }

  features: {
    E_Share: “Type of equity in financial
              instruments”,
    MTM_Rate_Method: “Share prices matched with
                     market price”,
    Company_Rate_Method: “Unlisted share price of
                         an individual company”,
    Weighted_Average_Value_Method:
               “Portfolio Valuation is done on the
                basis of average share price”;
    }

  interfaces: {
   definition: {
       //No need to define any interfaces
   }
   implements:{
      // MethodInterface interface template
      NumercialValue: NumericOperation;
      if (supported (MTM_Rate_Method ||
                    Company_Rate_Method))
        //WSDL interface template
        PriceStatus: ValueData;
      if (supported
                (Weighted_Average_Value_Method))
        CalculationMessage: PortfolioMessenger;
    }
  } //end of interfaces
```

```
sub-system: {
 components {
  PValueProcessor<false, false, false, true,
              true, true>: Portfolio_Processor;
  if (supported(E_Share)) {
   if (supported(MTM_Rate_Method)&&
      unsupported(Weighted_Average_Value_Method))
     MTMValuator<true, false, false, false>:
                              EquityCalculator;
   else if (supported(Company_Rate_Method))
     CRValuator<false, true, false, false>:
                              EquityCalculator;
   else
     WeightedValuator<false, false, true, false>:
                              EquityCalculator;
   }
  }
 connectors {
  HTTP_EMarket<MTM_Rate_Method, Company_Rate_
             Method, false>: HTTP_EquityValuator;
  if (supported(MTM_Rate_Method) &&
      unsupported(Weighted_Average_Value_Method))
    Cal_MTM<true, false, false>:
                             Calculator_Equity;
  else if (supported(Company_Rate_Method))
    Cal_CR<false, true, false>:
                             Calculator_Equity;
  else {
    HTTP_VProcessor<true, false>: HTTP_Equity;
    HTTP_CalWAV<false, false, true>:
                         HTTP_EquityCalculator;
    Cal_WAV<false, false, true>:
                            Calculator_Equity;}
 }
 arrangement {
  // connecting components using connectors
  connect PValueProcessor.CalculationMessage with
                        HTTP_VProcessor.msgport2;
 connect my.CalculationMessage with
                        HTTP_VProcessor.msgport1;
 if (supported (MTM_Rate_Method ||
             Company_Rate_Method)){
   connect PValueProcessor.PriceStatus with
                         HTTP_EMarket.valueport1;
   connect my.PriceStatus with
                        HTTP_EMarket.valueport2;}
 if (supported(MTM_Rate_Method) &&
     unsupported(Weighted_Average_Value_Method)){
   connect PValueProcessor.CalculationValue with
                              Cal_MTM.valueport1;

   connect MTMValuator.OperationalValue with
                              Cal_MTM.valueport2;
    connect MTMValuator.OperationalValue with
                                Cal_MTM.valueport2;
    connect my.NumericalValue with
                                Cal_MTM.valueport4;}
  else if (supported(Company_Rate_Method)) {
    connect PValueProcessor.CalculationValue with
                                  Cal_CR.valueport1;
    connect CRValuator.OperationalValue with
                                  Cal_CR.valueport2;
    connect CRValuator.OperationalValue with
                                  Cal_CR.valueport2;
    connect my.NumericalValue with
                                 Cal_CR.valueport4;}
  else {
    connect PValueProcessor.AverageRequest with
                           HTTP_CalWAV.messageport1;
    connect WeightedValuator.AverageMessage with
                           HTTP_CalWAV.messageport2;
    connect WeightedValuator.AverageValue with
                                 Cal_WAV.valueport3;
    connect my.NumericalValue with
                                Cal_WAV.valueport4;}
     } // end of arrangement section
   } // end of sub-system section
} // end of component type
```

The example above shows how the component configuration can change depending on what features are supported. The keyword "**my**" is used to reference the component's own interfaces as opposed to sub-component interfaces (similar to the use of "**this**" in some programming languages).

4.1.7 Pattern Templates

The *pattern template* notation in ALI allows the definition and use of architectural patterns. They are first defined and then (re)used throughout the architecture by calling the pattern template needed. The pattern template definition takes the interfaces to be connected as an argument and is defined in a similar way to the definition of functions (methods) in programming languages. A pattern template definition comprises of:

- *pattern name*: a unique pattern name.
- *arguments*: a set of interfaces to be connected. Single interface and/or arrays of interfaces can be passed as arguments. The minimum and maximum number of interfaces passed can be specified as arguments for arrays of interfaces.
- *definition*: the description of how the interfaces are to be connected (the pattern). The syntactical notation used for defining patterns is very simple and provides support for:
  - *connecting interfaces*: uses syntax similar to that used in the connections section of the connector type definition (discussed in Section 4.1.5).
  - *defining loops*: to allow for connecting arrays of interfaces. The syntax used here is similar to the syntax used in most programming languages for creating *for* loops. The point to be noted is that the arrays of interfaces start at index 1 and not at 0 (like in most programming languages).

Below is an example that defines `Client_Server` pattern:

```
pattern templates:
  {
   Client_Server (server : MethodInterface,
                  clients [1…N] : MethodInterface)
    {
     for ( i = 1 ; i <= N ; i++ )
       connect clients[i] and server;
    }
  }
```

In this example, the `Client_Server` pattern takes as an argument one interface `server` of template `MethodInterface`, and an array of interfaces called clients (with `[1..N]` meaning at least one `client` interface) of template `MethodInterface`. The pattern is defined as: for all N `clients` interfaces, create a bi-directional connection with the `server` interface (see Section 4.1.5 on the use of the keywords: "**connect**", "**and**", and "**to**" for connecting interfaces).

4.1.8 Product Configurations

A *product configuration* is a set of features, along with their values, representing a particular product configuration (this is also called *product feature set* in Software Product Line Engineering). Product configurations can be used to generate specific products from the parameterised reference architecture. Below is an example product configuration for an `Equity_Share_Derivative` product.

```
product configurations {
  Equity_Share_Derivative: {
    Equity {Equity_Type = long};
    Equity_Share = true;
    Equity_Derivative {Derivative_Type = options,
                       Premium_Period = 1year,
                       OTC = false};
    Share_Sector {Holdings = 100,
                  Total_Share_Value = 1,550,
                  Share_Sector_Category =
                     (banking, pharmaceutical)};
  }
  … // similarly other products can be defined
}
```

4.1.9 Events

Events are abstractions of actions performed during the execution of the system, such as a message transmission from one component to another. In ALI, events are defined using a unique name, along with the interface templates they travel to and from. Below is an example of how to define events:

```
events {
  ValuationRequest: <WSDL, WSDL>;
  RequestValuationDetails: <MethodInterface,
                                   MethodInterface>;
  CalculateValue: <WSDL, WSDL>;
  …
}
```

In the above example, `ValuationRequest` is an event that flows between two `WSDL` interfaces. It is also possible for events to travel from, and to, more than one interface template. In this case, interface templates are listed within parentheses and separated by commas as shown in the example below.

```
Inform: <(MethodInterface, WSDL),
         (MethodInterface, WSDL)>;
```

4.1.10 Conditions

Conditions are used to parameterise the system description to make it adapt to certain environmental conditions. Every set of conditions (a scenario) can then be used to simulate a certain environmental situation (e.g. failure, market changes, etc.). These can be used to test the way the architecture definition can adapt to different operational changes (design principle P2). Conditions are defined with a unique name along with a simple textual description. Below is an example definition of four different conditions.

```
conditions {
  PriceChanged: "Change in share price";
  PriceUnchanged: "No change in share price";
  ShareTrade: "Buying/Selling of shares";
  Exchange_Traded: "Shares listed in stock
                    exchange";
   …
}
```

4.1.11 Scenarios

Scenarios are basically collections of different conditions, along with their values, which together can simulate a certain operational scenario. Below is an example scenario description.

```
scenarios {
   P.RevaluatingPC: {
      Description: "Revaluating portfolio due to
            change in share price with no trading";
      Parameterisation: {
                           PriceChanged = true;
                           PriceUnchanged = false;
                           ShareTrade = false;
                          }
   }
  … // similarly other scenarios can be defined.
}
```

In the above example, scenario `P.RevaluatingPC` demonstrates the portfolio revaluation when there is a change in share price only. It encapsulates three *conditions* (defined in the previous section) in which one is true and two are false. Scenarios can be very useful when comparing different architectural configurations. Scenarios serve as switches, and as such, do not support parametrisation (which can be achieved in other parts of the notation, such as transaction domains).

4.1.12 Transaction Domains

Transaction domains represent the behavioural aspects of the system. Each transaction domain comprises a set of components and connectors within a system that work together to achieve some system functionality (e.g. portfolio evaluation). Within a transaction domain, various transactions are defined, each describing a particular system function or feature (e.g. valuation processing, MTM valuation, etc.). Transactions are defined in terms of event flows.

The *transaction domain* definition is divided into two main sections, *contents* which lists the components and connectors included in a transaction domain; and *transactions* which describes the transactions encompassed in the transaction domain. Each transaction is defined in terms of the events that flow to achieve the transaction, and the description of the event flow (*interactions*). Table 2 summarises the textual notation used in defining *interactions* within a transaction.

TABLE 2

ALI TRANSACTION DOMAIN TEXTUAL NOTATION

Notation	Meaning
Component.Interface	Component name with interface name
* Component	External component (or system)
Event	Event name
Event/Connector	Event traveling on connector
TRANSACTION	Transaction name
sends receives from to	Keywords describing the path of an event
if/else	Alternation (OR Fork)
|	Alternation (OR Join)
,	Concurrency (AND)
[…]	Multiple simultaneous interactions (concurrency)
(…)	Grouping of events
;	Interaction termination

Below is an excerpt of the `PortfolioValuation` transaction domain definition.

```
transaction domain PortfolioValuation
 {
   /* Meta objects can be attached as discussed
     earlier in Section 4.1.1 */
  contents:
    {
     /*provides the list of components and
       connectors involved in this transaction
       domain*/
     components: {Portfolio_GUI, UI_Server,
       EquityDb, Job_Processor, Value_Processor,
       Market_Share_Data, Equity_Market_Data,
       *Stock_Market, *Company_Financial_Account,
       UI_Price_Server,Portfolio_Value_Calculator,
       *P/L_System}
     connectors: {HTTP_GUI, HTTP_Status,
       HTTP_Processor, HTTP_ExMRate,
       HTTP_ExCRate, HTTP_CRate, HTTP_Price,
       HTTP_External, Cal_Processor,
       DB_VProcessor}
    }
transactions:
      {
       VALUATIONREQUEST: {…}
       VALUATIONUPDATE: {…}
       MTMVALUATION: {…}
       UNLISTEDVALUATION:
         {
          events: {RequestPrice, CurrentPrice}
          interactions:
            {
             *Company_Financial_Account receives
                         RequestPrice/HTTP_ExCRate;
             *Company_Financial_Account sends
                       CurrentPrice/HTTP_ExCRate to
                       UI_Price_Server.PriceStatus;
              UI_Price_Server.PriceStatus sends
                           CurrentPrice/HTTP_CRate;
            }
        }
       REVALUATION: {…}
       VALUATIONPROCESS:
        {
          events: {Inform, RequestPriceList,
                RequestPrice, CurrentPrice}
          interactions:
            {
             if (supported(Equity_Share)) {
               if (PriceUnchanged)
                VALUATIONUPDATE receives
                         Inform/ODBC_Processor from
                                  VALUATIONREQUEST;
               else {
                if (supported(MarkToMarket_Method
                          && (Exchange_Traded))
                 MTMVALUATION receives
                  RequestPriceList /HTTP_Processor
                        from VALUATIONREQUEST;
                else
                 UNLISTEDVALUATION receives
                  RequestPrice/HTTP_ExCRate from
                  VALUATIONREQUEST;}
              }
          [REVALUATION receives
          CurrentPrice/HTTP_ExMRate from
          MTMVALUATION |
          REVALUATION receives
          CurrentPrice/HTTP_CRate from
          UNLISTEDVALUATION];
         } //end of interaction
      } //end of transaction
  } //end of transactions section
 } } //end of transaction domain
```

Given the way interaction domains represent event flows, graphical representations (discussed in Section 4.2) tend to work much better in expressing complex flows.

4.1.13 Viewpoints

Viewpoints in ALI represent collections of transaction domains that relate to a particular stakeholder. A viewpoint definition includes: a unique name, description and a list of related transaction domains. Below is an example viewpoint definition for `PortfolioInvestment`.

```
viewpoints {
   PortfolioInvestment: {
      Description: "Investment made into the
                    Portfolio";
      Transaction Domain: {PortfolioValuation,
                           PortfolioRebalance,
                           PortfolioStrategy};
     }
    … //similarly other viewpoints can be defined
 }
```

4.1.14 System

Finally, the *system* notation describes the overall product (or product line) architecture. It uses very similar notation to the component description section with some minor changes. For example, the system description provides support (using asterisk "*") for linking components to external (to the system) components or systems. Additionally, a system description includes a listing of viewpoints. Below is an example system description.

```
system {
   components {
    Portfolio_GUI<>: PortfolioAMS_GUI;
    UI_Server<false, false, false>:
                         Portfolio_EquityUIServer;
    Job_Processor<false, false, false, false,
               false, false>: Portfolio_Processor;
    EquityDb<true, false, false, false, false,
                             false>: AMS_EquityDb;
 if(supported(Equity_Share)){
  // portfolio valuation
  Value_Processor<false, false, false, true,
             true, false>: Portfolio_Processor;
  if(supported(MarkToMarket_Method) &&
      unsupported(Share_Company_Method)){
   Market_Share_Data<false, false, false, true,
                     false, false>: AMS_EquityDb;
   *Stock_Market;}
  else {
   *Company_Financial_Account;
    UI_Price_Server<false, false, true>:
                       Portfolio_EquityUIServer;}
  }
 Equity_Market_Data<false, false, false, true,
                     true, false>: AMS_EquityDb;
 Portfolio_Value_Calculator<true,
     MarkToMarket_Method, Share_Company_Method,
             false>: Portfolio_EquityValuator;
 *P/L_System;
}
connectors {
  HTTP_GUI<true, false, false, false>:
                          HTTP_AMSUserInterface;
  HTTP_Processor<true, false>: HTTP_Equity;
  DB_VProcessor<false, false>:
                           ODBC_EquityPortfolio;
  HTTP_Status<false, false, false, false>:
                          HTTP_AMSUserInterface;
  if(supported(Equity_Share)){
   if(supported(MarkToMarket_Method))
     HTTP_ExMRate<false, true, false>:
                            HTTP_ExternalSystem;
   if(supported(Share_Company_Method)){
     HTTP_ExCRate<false, false, true>:
                            HTTP_ExternalSystem;
     HTTP_CRate<false, true, false>:
                           HTTP_EquityValuator;}
   HTTP_Price<true, true, false>:
                            HTTP_EquityValuator;
   Cal_Processor<false, false, false>:
                              Calculator_Equity;
   HTTP_External<false, false, false>:
                           HTTP_ExternalSystem;}
  }
}
 arrangement {
  //similar to component type arrangement
  /* an snippet below demonstrates some of the
     connections related to revaluation
     transaction that have been presented
  visually in Figure 2 */
  if( supported(Equity_Share)){
   …
```

```
    connect Portfolio_Value_Calculator.
     NumericalValue with
  Cal_Processor.valueport4;
    connect Value_Processor.OperationalValue with

  Cal_Processor.valueport1;
    connect Value_Processor.NotificationMessage
                  with
  HTTP_External.messageport1;
    connect *P/L_System with

  HTTP_External.messageport2;
    connect Value_Processor.NotificationMessage
                 with
  HTTP_Processor.messageport1;
    connect UI_Server.NotificationMessage with

  HTTP_Processor.messageport2;
    /* similarly other connections can be made
       via connectors */
   }
   viewpoints {
     PortfolioInvestment;
   }
}
```

4.2 ALI Graphical Notation

Many of the existing ADLs such as AADL [10], ACME [24], Aesop [33], MontiArc^{HV} [35], Darwin [22], Koala [12], UniCon [25] and π-ADL [19], provide both textual and graphical notations, though none provide a behavioural graphical notation. Yet, in some cases, the need for such graphical behavioural representation was argued, e.g. using Use Case Maps with ADLARS [13, 69]. ALI provides graphical notations for structural and behavioural aspects of the systems.

In order to create a consistent graphical notation that was expressive and easy to use, the theoretical guidance in [80] was followed. Furthermore, lessons learned from Woods and Bashrouh in [71], designing a large-scale architectural description within an industrial context, were also taken into consideration. The following sections discuss ALI's graphical notation.

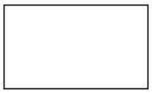
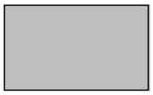
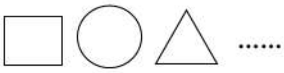
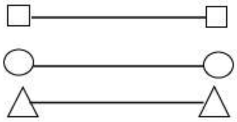
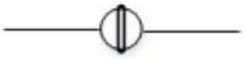
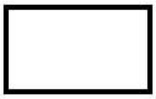
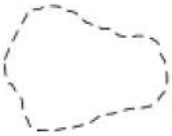
4.2.1 Structural Notation

ALI provides an extensible visual notation for its structural description. Table 3 illustrates the meaning of the symbols used to specify architectural structures in ALI. It is possible to extend the notation to represent components by introducing other graphical objects (e.g. a cylinder to represent a database component) that architects identify with or already use in certain application domains.

Fig. 2 represents the structural description of the `Revaluation` transaction (part of the `PortfolioValuation` transaction domain defined earlier in Section 4.1.12 in textual format). The same notation can be used to describe the whole system.

TABLE 3

ALI GRAPHICAL STRUCTURAL NOTATION

Symbol	Name (Meaning)
	Component
	External Component (or System)
	Interfaces (different shapes represent different interface templates)
	Connectors representing different interface templates
	Direct Binding (no connector)
	Transaction
	Transaction Domain

4.2.2 Behavioural Notation

4.2.2.1 Event Traces

In ALI, *event traces* constitute the graphical representation of *transactions*, described textually in Section 4.1.12. Table 4 provides the detailed description of the symbols used to design event traces. Some of the symbols used are adopted from the UML Activity diagram [74], with added notation to represent concurrency (based on some extended concepts from Petri Nets [81]).

Fig. 3 shows an example of the graphical behavioural representation of the transaction domain `PortfolioValuation` (this maps to the textual representation provided in Section 4.1.12). The example demonstrates the *transactions* that occur in `PortfolioValuation` along with the interactions that take place in the `VALUATIONUPDATE`, `MTMVALUATION` and `UNLISTEDVALAUTION` transactions.

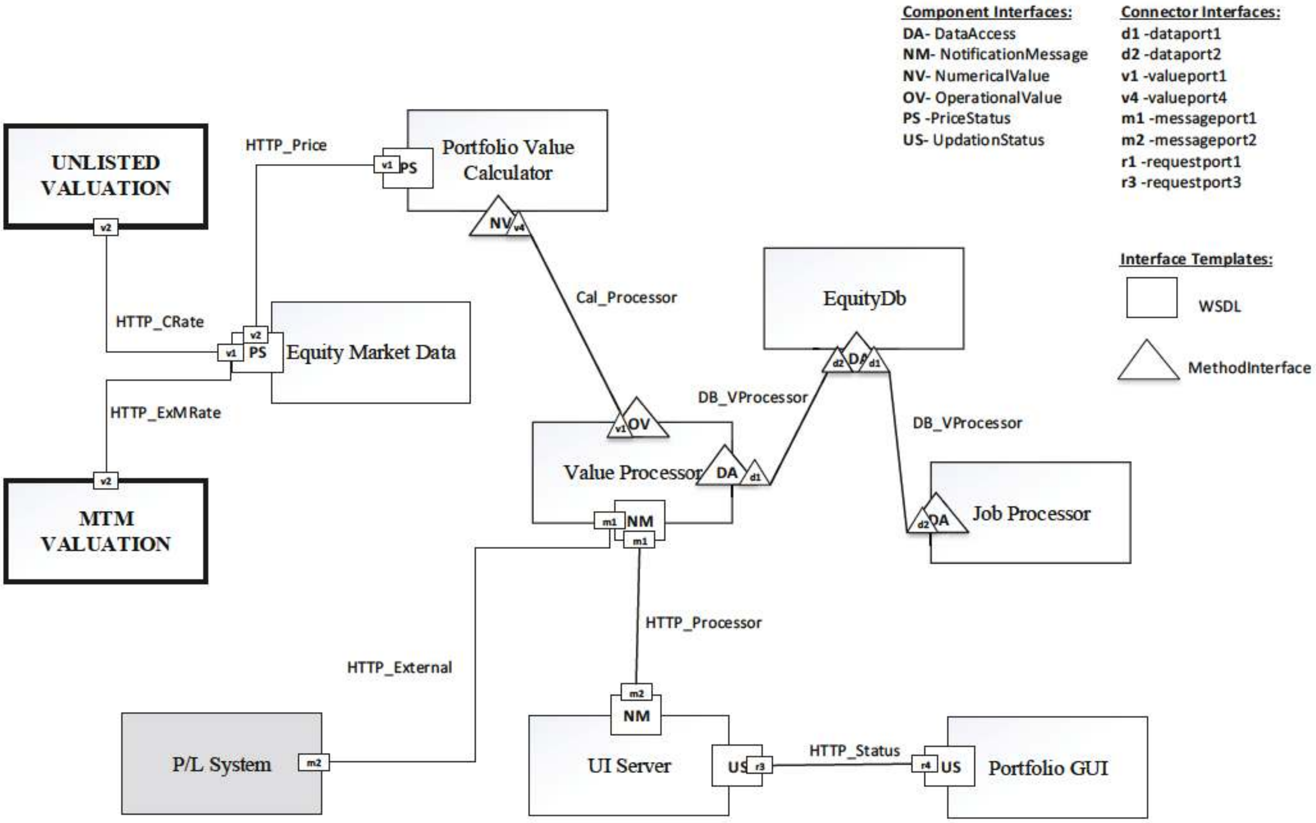


Fig. 2. Graphical structural representation for transaction `Revaluation`

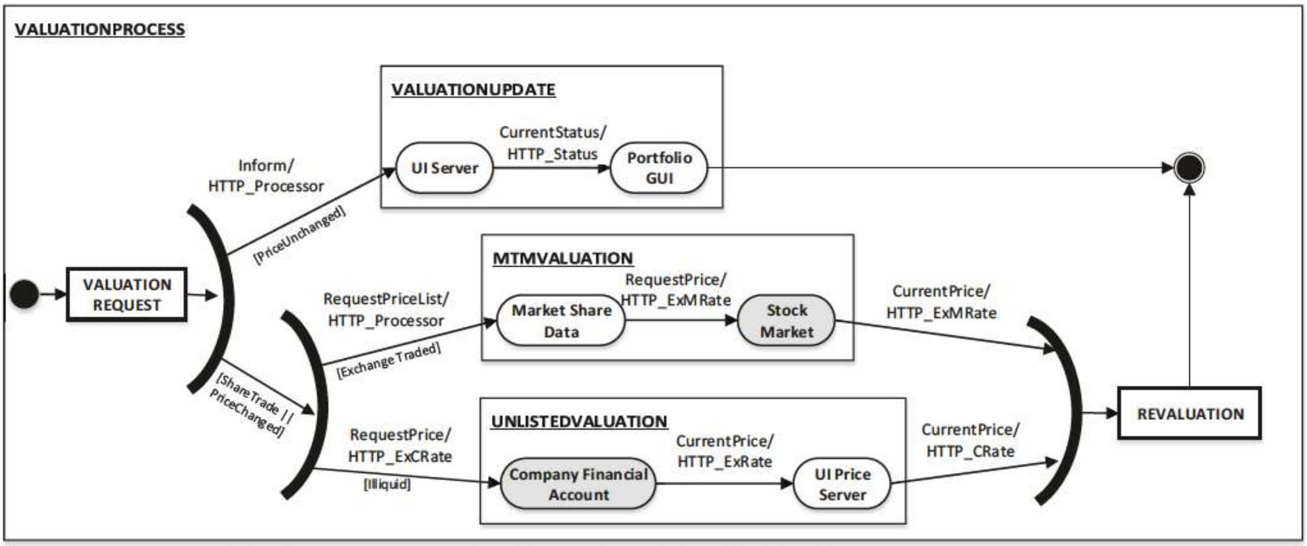
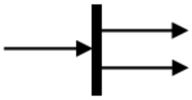
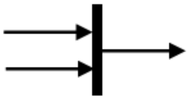
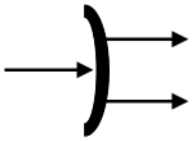
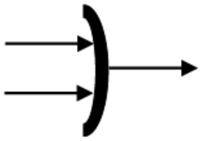
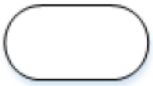
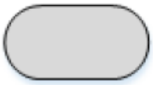
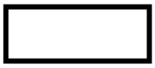


Fig. 3. Graphical behavioural representation of transaction domain `PortfolioValuation`

TABLE 4

ALI EVENT TRACES NOTATION

Symbol	Name	Meaning
●	START	A node that starts the interaction in an event trace by a component that invokes an event.
◉	END	A node that stops the interaction of all the transactions in an event trace.
⊗	FINAL	A node that terminates the interaction of the transaction.
EventName/Cr* →	Event Flow	The direction of an event flow from one component to another component, specifying the event name and the connector* being traversed.
	AND Fork	A source component sending two or more **concurrent** events to destination components.
	AND Join	A destination component receives two or more concurrent events from source components. This blocks until all events are received before progressing.
	OR Fork	A source component sends one or more events to destination components. Selection of the destination components can be linked to system *conditions* and *features*.
	OR Join	A destination component receives any of the events from any one of the source components (non-blocking) as soon as it arrives (without waiting for all expected events).
	Component	A component within the system that sends/receives events.
	External Component/ System	A system (or component) outside the system that communicates with our system.
	Transaction	Transaction is a package containing a set of interactions. It can be nested wherever required in another transaction.

* means optional i.e. if connection is made using connectors (see Section 4.1.6)

4.2.2.2 Component Interaction

This section provides the graphical notation used to describe the interactions of an individual component. While event traces model the complete event flow path, component interactions focus on modelling the interactions of a particular component (focus on components rather than events). For this, UML Sequence diagrams [74] are used to model component interactions. Sequence diagrams are known to many architects and are comprehensive to model handshakes, timing, etc.

Fig.4 shows the component interaction diagram for the component `UIServer` in the transaction domain `PortfolioValuation` (defined textually in Section 4.1.12, with interactions described in Fig. 3).

The squares at the top of the sequence diagram represent component interfaces. White squares represent the interfaces of the component being model (in this case `UIServer`), and greyed squares represent external interfaces (of other components `UIServer` is communicating with).

Having discussed the ALI notation, the next section defines the semantics behind the notation.

5 SEMANTICS

In this section, we discuss the ALI notation semantics [82]. We start by discussing the semantics of the structural notation, then the behavioural notation. It is worth noting that proofs of correctness and completeness of the semantics are beyond the scope of this paper.

The following notation convention is used in the subsequent two sections: i for interface, Ct for component, Cr for connector and e for event. The name of each element (where applicable) is indicated in the subscript. For example, i_A denotes interface A.

5.1 Semantics of Structural Notation

In this section, we discuss the semantics of the structural notation, namely covering: components, connectors and interfaces.

For simplicity, a component is a finite set of n interfaces:

$$i \in Ct = \{1i_A, 2i_B, ..., ni_Z\} \qquad (5.1.1)$$

Different combinations of interfaces in a component can occur depending on the *feature(s)* supported in its specification. All possible occurrences can be defined as:

$$Ct = \wp(i) \qquad (5.1.2)$$

The notation $\wp(i)$ refers to the power set of the set of interfaces of a component. It also includes the null/empty set (∅) which relates 0…* relationship between the interface and the component as explained in the conceptual model section (see Section 3.2 and Fig. 1).

Similarly, a connector is a finite set of n interfaces:

$$i \in Cr = \{2i_B, ..., ni_Z \mid i \geq 2\} \qquad (5.1.3)$$

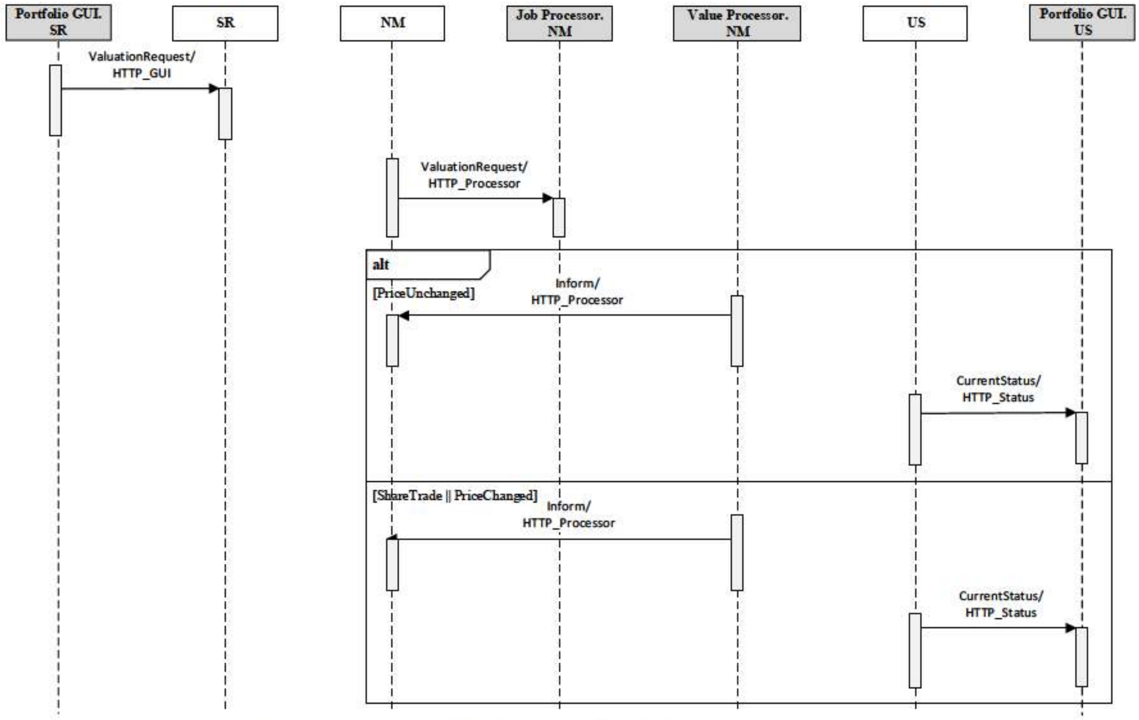


Fig. 4. Component `UIServer` interactions in transaction domain `PortfolioValuation`

The number of interfaces in a connector must be at least two to form a connection between two components.

The following are the naming rules:

Rule 1: The names of all the components must be unique within a system.

$$Ct_A \cap Ct_B = \varnothing, A \neq B \tag{5.1.4}$$

Rule 2: The names of all the connectors must be unique within a system.

$$Cr_A \cap Cr_B = \varnothing, A \neq B \tag{5.1.5}$$

Rule 3: The names of all the interfaces of a component must be unique.

$$\forall i : Ct = \{\forall x \in Ct, \forall y \in Ct \mid x \neq y\} \tag{5.1.6}$$

Rule 4: The names of all the interfaces of a connector must be unique.

$$\forall i : Cr = \{\forall x \in Cr, \forall y \in Cr \mid x \neq y, \mid x \mid \geq 2\} \tag{5.1.7}$$

5.2 Semantics of Behavioural Notation

In this section, we discuss the semantics of the behavioural notation of ALI by translating the notation into formal specification theory, namely CSP [30]. The main construct of the behaviour notation in ALI is the process that defines the interactions of a transaction in a transaction domain.

Using the CSP process notation, where (X:A → P(X)) [pronounced "X from A then P of X"], we transform interaction (*Itn*) into:

$$Itn = \langle e_1.Cr_A^{\dagger} : Ct_s.i_A \rightarrow Ct_r.i_B \rangle \tag{5.2.1}$$

To recall, an interaction in ALI is an event flowing via connector or via direct binding from one component to another component (see Section 4.1.6 for in-depth description). Equation 5.2.1 describes an interaction in terms of CSP as: event (e_1) via connector (Cr_A) from sender component (Ct_s) on its interface (i_A) goes to the receiver component (Ct_r) on its interface (i_B). Symbol '**†**' represents the optionality of the connector, that is, it will be defined if event flows via a connector. Similarly, we insert an asterisk '*' before the sender/receiver component (Ct_s/Ct_r) without specifying the interface name if it is an external component (or system), as explained in Section 4.1.12 and Section 4.2.2.1.

Interactions can occur in different combinations with other interactions. The operators which form these combinations are: AND fork, AND join, OR fork and OR join, each combination is explained categorically in the transaction domain (see Section 4.2.2.1).

In the rest of this section, we elucidate the formal semantics for each interaction scenario using CSP.

AND Fork: Two or more interactions that occur concurrently. Considering different Ct_r and different interface of Ct_s, we have the following definition:

$$AND\ Fork = \begin{matrix} \langle e_1.Cr_A : Ct_s.i_A \rightarrow Ct_{r1}.i_C \rangle \parallel \\ \langle e_2 : Ct_s.i_B \rightarrow Ct_{r2}.i_D \rangle \parallel \ldots \end{matrix} \tag{5.2.2a}$$

Where '||' is the CSP parallel operator which represents concurrent activity. Hence, it is not necessary that AND Fork always has different receiver components (like Ct_{r1} and Ct_{r2} as above), we could have a situation where two or more events flow to one Ct_r via the same or different interfaces as discussed in Section 4.1.12.

Considering the same Ct_r, using the same interface, we can define an AND fork as:

$$AND\ Fork = \begin{matrix} \langle\langle e_1.Cr_A : Ct_s.i_A \parallel e_2 : Ct_s.i_B \parallel \ldots\rangle \\ \rightarrow Ct_r.i_D \rangle \end{matrix} \tag{5.2.2b}$$

In addition to the above expression conditions, we can also define it by considering Ct_s, using the same interface, as:

$$AND\ Fork = \langle\langle e_1.Cr_A \parallel e_2 \parallel \ldots\rangle : Ct_s.i_B \rightarrow Ct_r.i_A \rangle \tag{5.2.2c}$$

AND Join: Two or more interactions that go to the Ct_r concurrently. Considering different Ct_s and different interfaces of Ct_r, it is defined as:

$$AND\ Join = \begin{matrix} \langle\langle e_1.Cr_A : Ct_{s1}.i_A \rightarrow Ct_r.i_B \rangle \wedge \\ \langle e_2 : Ct_{s2}.i_B \rightarrow Ct_r.i_D \rangle \wedge \ldots \rangle \\ \rightarrow \langle WAIT\ \Sigma\ ;\ Ct_r \rangle \end{matrix} \tag{5.2.3a}$$

Where '∧' is the logical AND operator, **WAIT** is a time-based CSP operator [82], 'Σ' is submission (union) of all the events and '**;**' means successfully followed by. Thus, '**WAIT** Σ ; Ct_r' designates: Ct_r will not proceed with other interaction(s) until it receives all the events.

Considering the same interface of Ct_r, we can define AND join as:

$$AND\ Join = \begin{matrix} \langle\langle e_1 : Ct_{s1}.i_B \wedge e_2.Cr_C : Ct_{s2}.i_A \wedge \ldots\rangle \\ \rightarrow Ct_r.i_C \rangle \rightarrow \langle WAIT\ \Sigma\ ;\ Ct_r \rangle \end{matrix} \tag{5.2.3b}$$

Moreover, the definition for the same Ct_s with its different interfaces can be defined in a similar way as above. But if we have the same Ct_s with its same interface and the same interface of Ct_r then we can define it as:

$$AND\ Join = \langle (e_1.Cr_A \wedge e_2 \wedge ...) : Ct_s.i_C \rightarrow Ct_r.i_D \rangle \rightarrow (WAIT\ \Sigma\ ;\ Ct_r) \tag{5.2.3c}$$

OR Fork: Two or more interactions that occur alternatively in accordance to the *condition(s)* and *feature(s)* supported. Considering different $Ct_r$and different interface of Ct_s, we have the following definition:

$$OR\ Fork = (e_1.Cr_A : Ct_s.i_A \rightarrow Ct_{r1}.i_D) \Box (e_2 : Ct_s.i_B \rightarrow Ct_{r2}.i_C) \Box ... \tag{5.2.4a}$$

Where '□' is the CSP deterministic choice operator.

If the same event flows to different Ct_r depending on the *condition(s)* and *feature(s)* supported from the same interface of Ct_s then it can be defined as:

$$OR\ Fork = \langle (e_1.Cr_A : Ct_s.i_A \rightarrow (Ct_{r1}.i_C \Box Ct_{r2}.i_B \Box ...) \rangle \tag{5.2.4b}$$

Another case, when different events flow to same Ct_r to its same interface depending on the *condition(s)* and *feature(s)* supported from the same interface of Ct_s then it can be define as:

$$OR\ Fork = \langle (e_1.Cr_A \Box e_2 \Box ...) : Ct_s.i_A \rightarrow Ct_r.i_D \rangle \tag{5.2.4c}$$

OR Join: Two or more interactions that go to the Ct_r alternatively. Unlike AND join, Ct_rwill proceed with other interaction(s) after receiving the first event from any Ct_s without waiting for all the events to occur. Considering different Ct_s and different interface of Ct_r, it is defined as:

$$OR\ Join = (e_1.Cr_A : Ct_{s1}.i_A \rightarrow Ct_r.i_A) \Box (e_2 : Ct_{s2}.i_A \rightarrow Ct_r.i_C) \Box ... \tag{5.2.5a}$$

Considering the same interface of Ct_r, we can define it as:

$$OR\ Join = \langle (e_1 : Ct_{s1}.i_A \Box e_2.Cr_C : Ct_{s2}.i_A \Box ...) \rightarrow Ct_r.i_B \rangle \tag{5.2.5b}$$

Also, we can define the same Ct_s with its same interface along with the same interface of Ct_ras:

$$OR\ Join = \langle (e_1.Cr_A \Box e_2 \Box ...) : Ct_s.i_A \rightarrow Ct_r.i_D \rangle \tag{5.2.5c}$$

6 Case Studies

In order to illustrate the applicability of the proposed ADL and evaluate its effectiveness, particularly in relation to the identified limitations, we use two case studies. The case studies were selected from two distinct application domains, namely Information Systems (Asset Management System) and embedded systems (Wheel Brake System). A number of selection criteria were applied to decide on the best case studies, including: distinct application domains (to demonstrate cross domain modelling capabilities); existence of inherent variability in the application domain; varying types of connectivity between components; different complexity levels (information overload); varied emphasis on behavioural versus structural descriptions; potential for artefact reusability within the case study; and access to full technical details.

Table 5 demonstrates the comparison between the two case studies against the selection criteria for the case studies.

TABLE 5

Case Studies Criteria

Criteria	Case Studies	
	AMS	**WBS**
Application domain	Information Systems	Embedded Systems
Existence of inherent variability	High	Low
Types of connectivity	With connectors	Direct binding
Level of complexity (overall)	High	Low
Level of complexity (structural)	High	Low
Level of complexity (behavioural)	Low	High
Artefact reusability	Medium	Medium

In the following, a description of each of the case studies is given, followed by example descriptions (with further details provided in Appendix A and Appendix B respectively). The two case studies are then used to assess the extent ALI meets its design principles and addresses the identified limitations.

6.1 Case Study: Asset Management System (AMS)

An Asset Management System (AMS) in banking and finance is used by a fund manager to support making and executing investment decisions for a large-scale investment portfolio. In the following sections, we give a brief overview of AMS and provide the ALI architectural description.

6.1.1 AMS Overview

The primary aim of AMS is to allow a fund manager (or a fund management team) to manage a portfolio of holdings in financial instruments. In this paper, we are only covering the management of equities financial instruments by AMS, under which we focused on company shares and their derivatives. AMS allows fund managers to view the content of their portfolios and trading and market data (in this case, share trades and prices) to make investment decisions. More specifically, it automates the calculation of suggested changes to portfolios on-demand or using a pre-defined schedule. This functionality is performed on a daily basis to calculate the portfolio value at the end of each working day, after the closure of stock market. In an investment bank, portfolio valuation can be performed using two methods, depending on the user's request. The first method is Mark to Market (MTM), where share prices are matched with the current stock market price and individual company share price (companies which are not listed in the stock market). The second method is applied on monthly/quarterly/bi-annual basis by checking the company's financial statement, depending on the company's fiscal period.

6.1.2 AMS Architecture

In this section, we discuss the AMS architecture description using ALI.

The following are the architectural artefacts of the AMS system for the portfolio value calculation, some of which are used in section 4 as examples of the ALI notation constructs.

a) ***AMS Meta Types***: `Meta_AMSFeature`, `Meta_EquityServer` (defined in Section 4.1.1), `Meta_Processor`, `Meta_DbEquity`, `Meta_ShareValueData`, `Meta_Valuator`, `Meta_Derivative` and `Meta_PortfolioDomain`.

b) ***AMS Features***: `Equity`, `Equity_Share`, `Share_Sector`, `Equity_Derivative`, `MarkToMarket_Method` and `Share_Company_Method` (some of these features are defined in Section 4.1.2).

c) ***AMS Interface Templates***: `MethodInterface` (defined in Section 4.1.3) and `WSDL`.

d) ***AMS Interface Types***: `ArithmeticOperation`, `ValueOperation`, `AverageOperation`, `NumericOperation`, `InvestmentOperation`, `DatabaseOperation`, `DatabaseUpdation`, `DatabaseOrder` and `Deriv-ativeOperation`, that all conform to interface template `MethodInterface` (some of these interface types are defined in Section 4.1.4). Similarly, interface types (such as `PortfolioMessenger` and `ValueData`) that conform to interface template `WSDL` can be defined.

e) ***AMS Connector Types***: `HTTP_AMSUserInterface`, `HTTP_Equity`, `ODBC_EquityPortfolio`, `HTTP_Equity-Valuator`, `Calculator_Equity` (defined in Section 4.1.5), `HTTP_ExternalSystem`, `HTTP_EquityRate` and `Calculator_Derivative`.

f) ***AMS Component Types***: `PortfolioAMS_GUI`, `Portfolio_EquityUIServer`, `Portfolio_EquityValu-ator` (defined in Section 4.1.6), `EquityCalculator`, `Portfolio_Processor`, `AMS_Eq-uityDb`, `PortfolioDb`, `DerivativeValuator` and `InternalEquityData`.

g) ***AMS Products***: `Equity_Share_Derivative` (defined in Section 4.1.7), `Equity_Share_ExchangeTraded` and `Equity_Share_Traded`.

h) ***AMS Events***: `ValuationRequest`, `RequestValuationDetails`, `SendValuationDetails`, etc. See Appendix A for complete list of events.

i) ***AMS Conditions***: `PriceUnchanged`, `PriceChanged`, `ShareTrade`, `Exchange_Traded` and `Illiquid` (defined in Section 4.1.10).

j) ***AMS Scenarios***: `P.RevaluatingPC` (see Section 4.1.11), `P.RevaluatingPC.ST_ET`, `P.RevaluatingPC.ST_IL`, `P.RevaluatingST_ET` and `P.RevaluatingST_IL`.

k) ***AMS Transaction Domain***: `PortfolioValuation` (excerpt of it is described in Section 4.1.12).

l) ***AMS Viewpoint***: `PortofolioInvestment` (defined in Section 4.1.13).

Appendix A contains further details about the above listed architectural constructs of the AMS system. The AMS *system* description can be found in Section 4.1.14.

Fig. 5 shows the graphical structural description of the transaction domain `PortfolioValuation`. The behavioural representation can be found in Fig. 3 and Fig. 4.

6.2 Case Study: Wheel Brake System (WBS)

The case study of the wheel brake system (WBS) of a commercial aircraft is based on the one introduced in the ARP4761 standard [84] published by SAE. In the following, we provide a brief overview of the system, along with ALI architectural descriptions.

6.2.1 WBS Overview

The primary purpose of the WBS in commercial aircraft is to decelerate the wheel on the ground along with the associated functions as explained in [84]. The WBS consists of a digital controller, the Brake System Control Unit (*BSCU*), and the hydraulic pipe assembly that carries the braking pressure to the wheels. Different valves are embedded that receive commands and control the flow of brake pressure.

Based on the safety requirement, the probability of loss of all wheel braking is less than 5x10^{-7} per flight, a design decision was made that each wheel would have a brake assembly operated by two independent sets of hydraulic pistons. One set of pistons is operated from the *green pump* and is used in the *NORMAL* braking mode. The *ALTERNATE* braking system is on standby and is selected

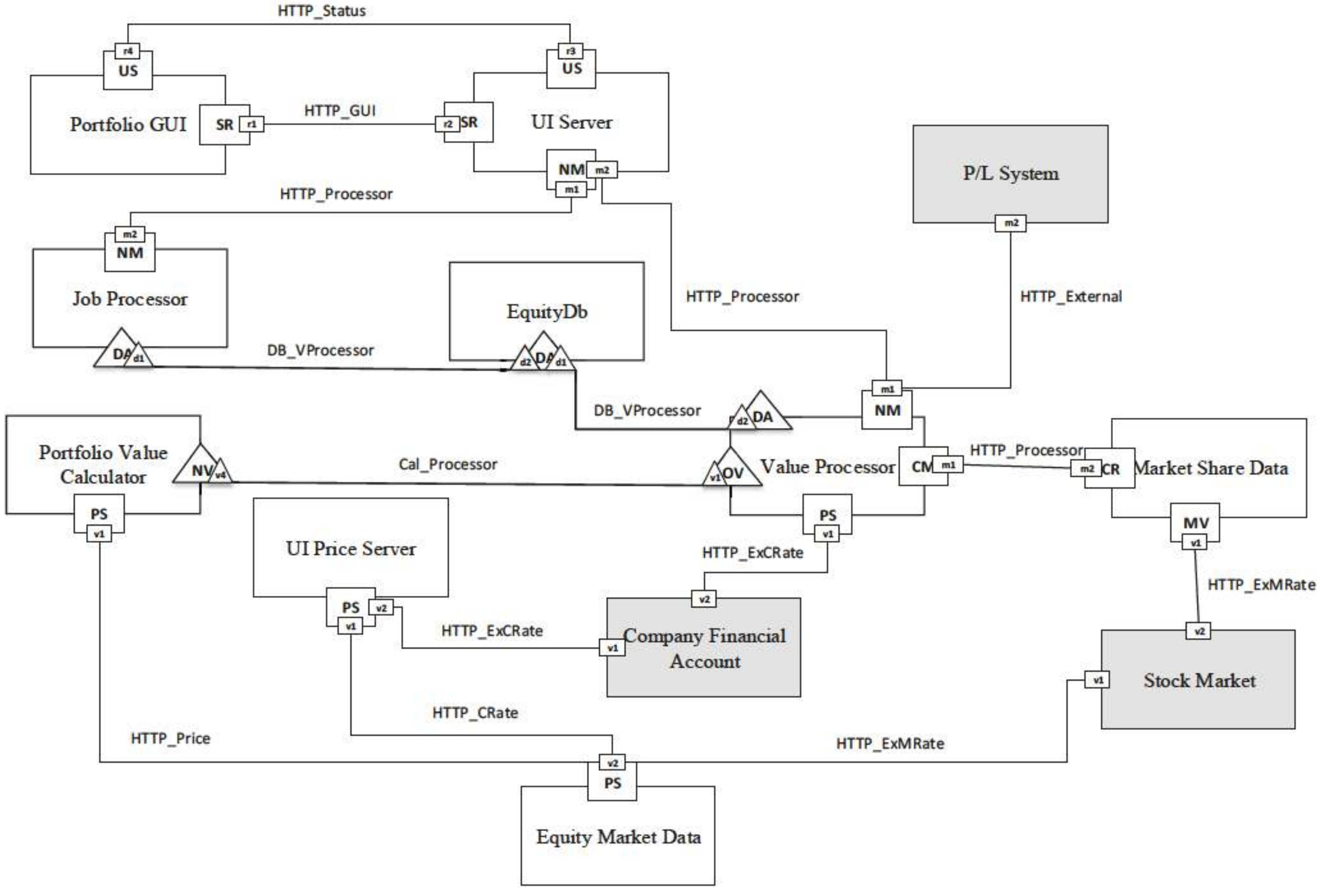


Fig. 5. AMS graphical structural representation of transaction domain `PortfolioValuation`

automatically in case of normal system failure (due to *BSCU* or *green pump* failure). In the *ALTERNATE* mode, the *blue pump* provides the hydraulic pressure to the system and if it fails then the system will be in *EMERGENCY* braking mode. In *EMERGENCY* mode, the wheel will receive the reserve pressure from the *accumulator*. It acts as a parking brake as well. A mechanical pedal is used to apply the brake in both *ALTERNATE* and *EMERGENCY* modes. Switch-over between the hydraulic pistons and the different pumps is automatic under various failure conditions, or can be manually operated.

6.2.2 WBS Architecture

Following are the architectural artefacts of WBS that demonstrate how the wheels of a commercial aircraft can be stopped/decelerated on the ground during landing or take off. Further details can be found in Appendix B.

a) *WBS Meta Types*: `Meta_WheelPedal`, `Meta_Brake`, `Meta_BrakePump`, `Meta_BrakeValve`, `Meta_BrakeCU` and `Meta_DecelerationDomain`.

b) *WBS Features*: `Wheel_Brake`, `Electrical_Brake`, `Electrical_Power`, `Mechanical_Brake`, `Pistion_Pre-ssure` and `Accumulator_Pressure`.

c) *WBS Interface Template*: `MethodInterface` (similar to AMS case study with minor changes in the *constraints* section).

d) *WBS Interface Types*: `DataOperation`, `ElectricOperation`, `CommandOperation`, `PressureOperation`, `ValueOperation` and `Notifier`, that all conforms to interface template `MethodInterface`.

e) *WBS Component Types*: `Aircraft_BrakePedal`, `Aircraft_ElectricPower`, `Brake_ControlUnit`, `Aircraft_WheelControlUnit`, `Aircraft_PressurePu-mp`, `Aircraft_BrakeValve`, `Aircraft_PressureValve`, `Command_Generator`, `Value_Monitor` and `Aircraft_Wh-eel`.

f) *WBS Product*: `CommercialAircraftBrake`.

g) *WBS Events*: `Reserve_Pressure_Request`, `MPedal_Pos-ition_Request`, etc. See Appendix B for complete list of events.

h) *WBS Conditions*: `BSCU_Active`, `BSCU_Failed`, `GreenPressure`, `GreenPressure_Failed`, `BluePressure`, `BluePressure_Failed` and `AccumulatorPump`.

i) *WBS Scenarios*: `NormalOperation`, `AlternateOperation`, `BSCUFailureOperation` and `EmergencyOperation`.

j) *WBS Transaction Domain*: `WheelDecelerationOnGround`.

k) *WBS Viewpoint*: `WheelDeceleration`.

Fig. 6 shows an example of the graphical behavioural representation of the transaction domain `WheelDecelerationOnGround` (the corresponding textual representation is provided in Appendix B). The example demonstrates the *transactions* that occur in `WheelDecelerationOnGround` along with the interactions that take place in the EMERGENCYMODE transaction.

Another behavioural visual perspective of the WBS architecture is the component `Accumulator` interactions in transaction domain `WheelDecelerationOnGround`, as shown in Fig. 7 (defined textually in Appendix B, with interactions demonstrated in Fig. 6). Similarly, other component interactions that are involved in the transaction domain `WheelDecelerationOnGround` can be defined.

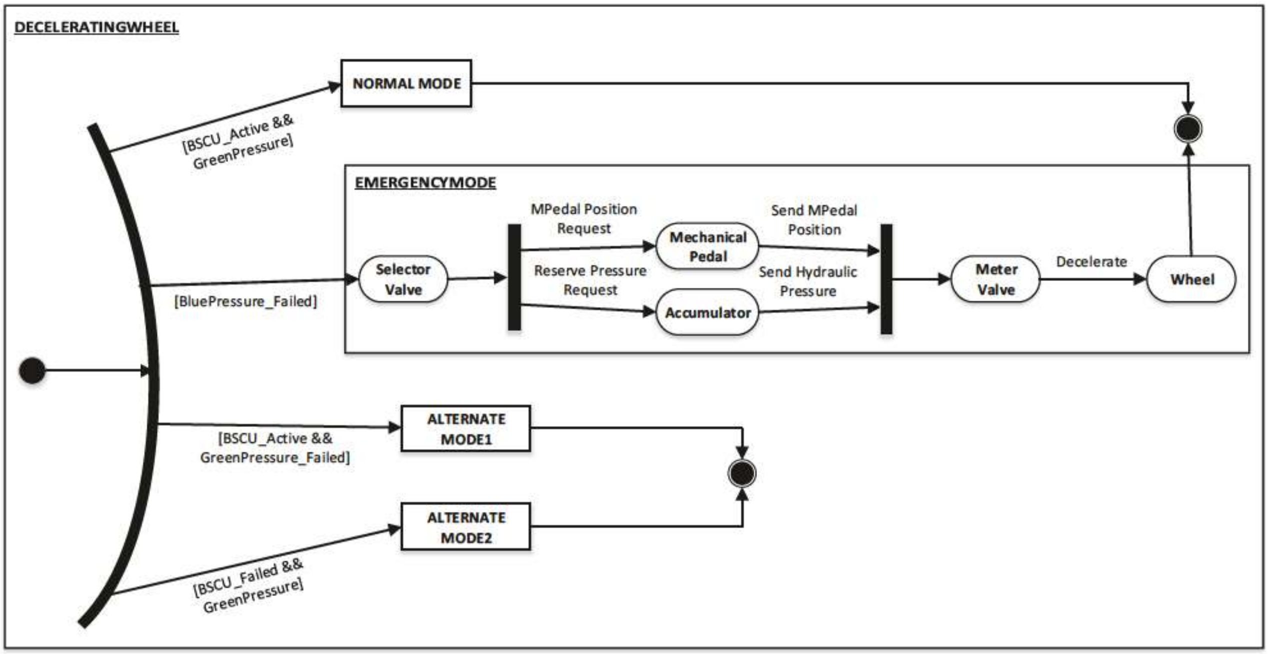


Fig. 6. WBS graphical behavioural representation of transaction domain `WheelDecelerationOnGround`

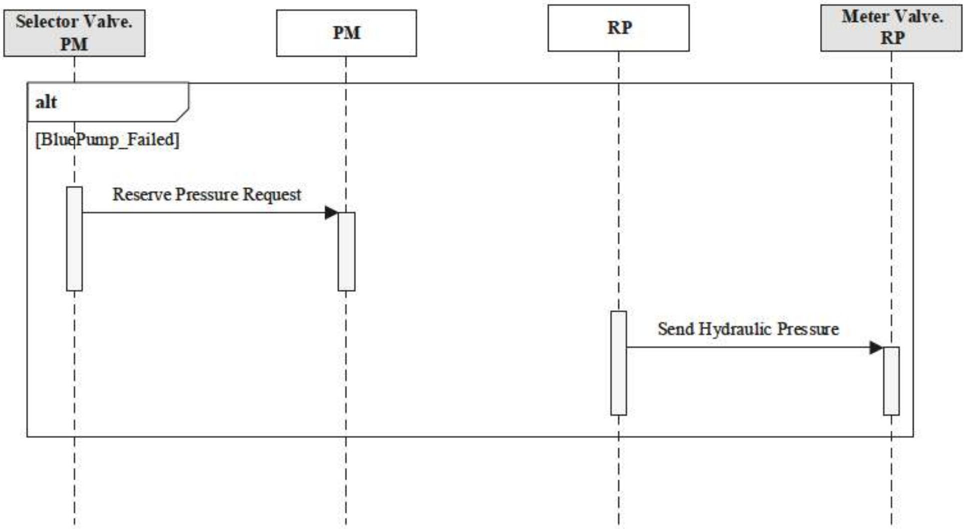


Fig. 7. WBS component `Accumulator` interactions in transaction domain `WheelDecelerationOnGround`

Below is a snippet of the WBS *system* description:

```
system {
  components {
    Selector_Valve<Electrical_Power>:
                                  Aircraft_BrakeValve;
    Wheel<>: Aircraft_Wheel;
    Meter_Valve<Electrical_Brake,
       Mechanical_Brake, Piston_Pressure,
       Accumulator_Pressure, Electrical_Power>:
                               Aircraft_PressureValve;
    …
    if (supported(Mechanical_Brake))
      Mechanical_Pedal<false, true>:
                                  Aircraft_BrakePedal;
    if (supported(Electrical_Brake &&
                  Piston_Pressure)){
      Green_Pump<true, false, true, false>:
                                Aircraft_PressurePump;
    else if (supported(Mechanical_Brake &&
                       Piston_Pressure))
      Blue_Pump<false, true, true, false>:
                                Aircraft_PressurePump;
    else
      Accumulator<false, true, false, true>:
                               Aircraft_PressurePump;}
 } // end of components
 connectors { }
 arrangement {
   …
   if (supported(Mechanical_Brake)){
bind Mechanical_Pedal.MechanicalPosition with
Meter_Valve.MechanicalPosition;
     bind Mechanical_Pedal.MechanicalCommand with
                    Meter_Valve.MechanicalCommand;
  }
  if (supported(Electrical_Brake &&
                Piston_Pressure)){
   {bind Shutoff_Valve.PressureMessage with
                   Selector_Valve.PressureMessage;
    bind Selector_Valve.PressureMessage with
                       Green_Pump.PressureMessage;
    bind Green_Pump.NormalPressure with
                         Meter_Valve.NormalPressure;}
  else if (supported(Mechanical_Brake &&
                     Piston_Pressure))
   {bind Selector_Valve.PressureMessage with
                        Blue_Pump.PressureMessage;
    bind Blue_Pump.AlternatePressure with
                   Meter_Valve.AlternatePressure;}
  else
   {bind Selector_Valve.PressureMessage with
                      Accumulator.PressureMessage;
    bind Accumulator.ReservePressure with
                     Meter_Valve.ReservePressure;}
 } // end of arrangement
viewpoints {
   WheelDeceleration;
 } // end of viewpoints
} // end of WBS
```

Fig. 8 represents the structural description of WBS.

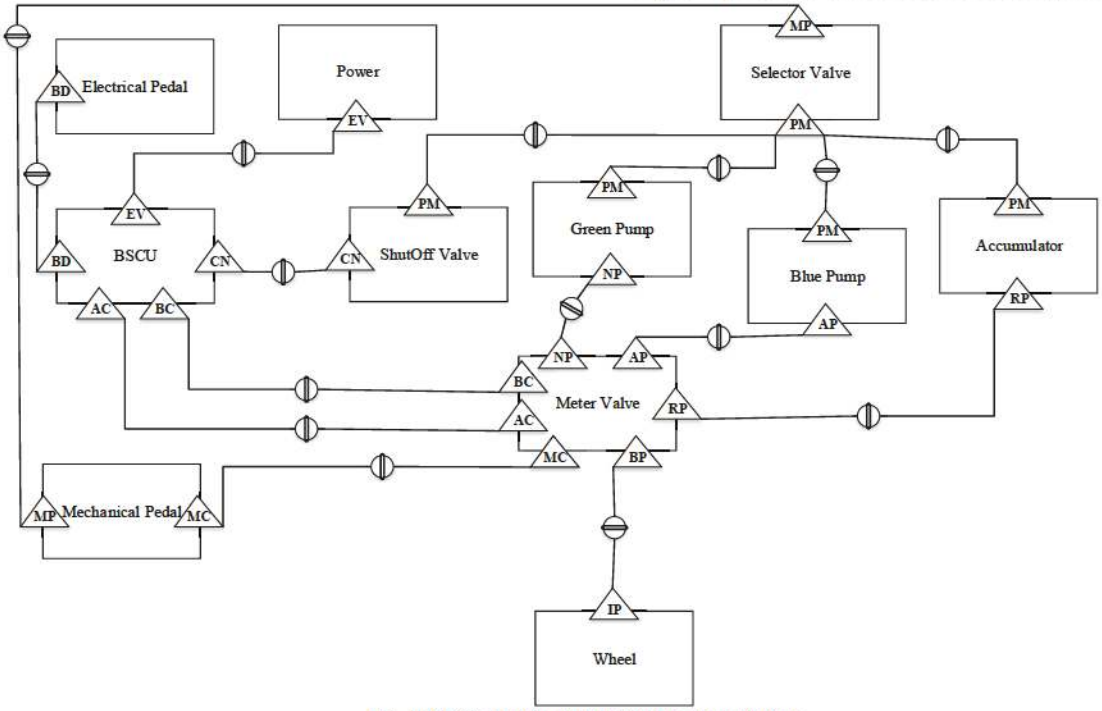


Fig. 8.WBS graphical structural representation

7 Evaluation and Analysis

In this section, the two case studies (AMS and WBS) are used to evaluate the ALI notation. Particularly, we assess the extent to which ALI addresses the limitations identified in Section 2.2 by following the design principles explained in Section 3.1.

7.1 AMS Analysis

Based on the AMS case study discussed in the previous section, Table 6 lists the limitations and how they are addressed in the case study, along with the relevant principles.

7.2 WBS Analysis

Similarly, based on the WBS case study, Table 7 below lists the limitations and how they are addressed in the case study, along with the relevant principles.

TABLE 6

Evaluation of the AMS Case Study

Limitations Addressed	CASE STUDY: Asset Management System (AMS)	ALI Principles Used
L1	AMS demonstrates support for capturing variability in the architecture at various points. For example, one variability in portfolio valuation is the trigger for recalculation due to change in the share price and/or share trading. This is captured using the following features: `Equity_Share`, `MarkToMarket_Method` and `Share_Company_Method`, and the conditions defined in Section 4.1.10.	P1
L2	The capability to trace requirements into architecture is demonstrated using the features `Equity_Share`, `MarkToMarket_Method` and `Share_Company_Method` which represent the actual requirements and link them to architectural artefacts in the `system` description (Section 4.1.14). Traceability is also applicable in the behavioural description, where the conditions `PriceUnchanged`, `PriceChanged`, `ShareTrade`, `ExchangeTraded` and `Illiquid` can occur during the equity portfolio valuation depending on external requirements. These conditions are represented in the transaction domain `PortfolioValuation`.	P2
L3	This is discussed in section 7.3.	P3
L4	The AMS architecture description demonstrates the notation flexibility in various constructs. For example, component type `EquityCalculator` (defined in Appendix A) used pre-defined interfaces (`OperationalValue` of type `ArithmeticOperation`, `AverageValue` of type `AverageOperation`, `AverageMessage` and `DerivativeRequest` of type `PortfolioMessenger` and `DerivativeValue` of type `DerivativeOperation`) instead of defining them within its *definition* section using the interface templates `MethodInterface` and `WSDL`. The component type `EquityCalculator` has the flexibility to define an interface similarly to the method used in the *interface types* section in its *definition* section (as explained in Section 4.1.6). Similarly, the event `Inform` supports the interface templates `MethodInterface` and `WSDL` in the transaction domain `PortfolioValuation`, but it has the flexibility to support only the interface template `MethodInterface` in another transaction domain.	P4
L5	AMS utilises the reusability aspects of ALI extensively. For example, Interface template `MethodInterface` (i.e., an interface that supports method invocation) and the interfaces of type `MethodInterface` (defined in Section 4.1.3 and 6.1.2, respectively) can be reused in any type of system architecture wherever an interface of this type is required (e.g. other AMS systems for different financial instruments). Same thing applies to *connector types* and *component types* (defined in Section 6.1.2). Within the new system, artefacts can be adopted by mapping their feature set to the target system which automatically reconfigures the artefact to produce the desired supported functionality. For example, connector type `Calculator_Equity` (defined in Section 4.1.5) and component type `Portfolio_EquityValuator` (defined in Section 4.1.6) have features `Weight_Average_Method` and `Weighted_Average_Value_Method`, respectively. This feature is one of the methods used to calculate the equity portfolio and is not adopted by an investment bank nowadays where they must manage large-scale equity portfolios. Therefore, it is not considered in the `system` description (see Section 4.1.14). But the artefact description of the connector type `Calculator_Equity` and component type `Portfolio_EquityValuator`	P5

Limitations Addressed	CASE STUDY: Asset Management System (AMS)	ALI Principles Used
	are defined in such a way that it may be used in another system where they support the weighted average value method to calculate their equity portfolio value due to the support of its relevant features.	
L6	Information overload in AMS is addressed by having multiple views, each focusing on different aspects. For example, in order to calculate the equity portfolio value, the transaction domain `PortfolioValuation` (defined textually in Section 4.1.12 and presented graphically using event traces in Fig. 3) illustrates its behavioural description. In addition to the behavioural view, the sequential interaction of all the components involved in the transaction domain `PortfolioValuation` (such as component `UIServer` in Fig. 4) are presented. While the `system` description (defined textually in Section 4.1.14 and presented graphically in Fig. 5) illustrates its structural description. The AMS architecture provides multiple views of a particular function (as a *transaction domain*) with a clear separation between structural and behavioural descriptions while maintaining consistency.	P6
L7	Behaviour in AMS has a dedicated section. For example, events such as `ValuationRequest`, `RequestValuationDetails`, etc., are defined clearly, with their source and destination *interface templates* (specified in Appendix A). From a visualisation perspective, an example is the transaction domain `PortfolioValuation` which is presented in the form of event traces that demonstrate the ways an event can occur to calculate the equity portfolio, as demonstrated in Fig. 3. `RequestPriceList/HTTP_Processor` and `RequestPrice/HTTP_ExCRate` are the events that depend on the conditions `Exchange_Traded` and `Illiquid`, respectively. This is represented using the OR Fork notation (as defined in Table 4), meaning that they can only occur serially to do equity portfolio valuation. Moreover, the interactions of the component `UIServer` in the transaction domain `PortfolioValuation` are explicitly presented using a UML sequence diagram in Fig. 4.	P6

TABLE 7

EVALUATION OF THE WBS CASE STUDY

Limitations Addressed	CASE STUDY: Wheel Brake System (WBS)	ALI Principles Used
L1	Variability representation is widely used in WBS. For example, the case study has various variation points to represent the different braking modes. This is captured using the following features: `Electrical_Brake`, `Mechanical_Brake`, `Electrical_Power`, `Piston_Pressure` and `Accumulator_Pressure`, as well as the conditions defined in Appendix B.	P1
L2	The capability to trace requirements into the architecture is demonstrated using the features `Electrical_Brake`, `Electrical_Power` and `Piston_Pressure` when the brake is applied in the *NORMAL* mode. This represents actual requirements and links them to architectural artefacts in the `system` description (Section 6.2.2). When the brake is applied in the *ALTERNATE* modes, the features `Mechanical_Brake`, `Piston_Pressure` and/or `Electrical_Power` are enabled. While, if it is applied in an *EMERGENCY* mode, the features `Mechanical_Brake` and `Accumulator_Pressure` are enabled. Traceability is also demonstrated in behavioural description, where the conditions `BSCU_Active` and `GreenPressure` demonstrates the *NORMAL* braking mode while `BSCU_Failed` and/or `GreenPressure_Failed` demonstrates the *ALTERNATE* braking modes. An *EMERGENCY* braking mode is demonstrated by the condition `BluePressure_Failed`. These conditions are represented in the transaction domain `WheelDecelerationOnGround`. Such conditions allow us to trace related requirements into various aspects of the architecture.	P2
L3	This is discussed in section 7.3.	P3

Limitations Addressed	CASE STUDY: Wheel Brake System (WBS)	ALI Principles Used
L4	Support for flexibility in the WBS architecture description can be demonstrated in various constructs. For example, component type Aircraft_BrakePedal (defined in Appendix B) used pre-defined interfaces (BrakeData, MechanicalPosition and MechanicalCommand of type DataOperation, ValueOperation and CommandOperation, respectively) rather than defining them locally within its *definition* section. On the other hand, component type Aircraft_BrakePedal can defines its interfaces locally (as explained in Section 4.1.6).	P4
L5	WBS utilises the reusability aspects of ALI extensively. Interface template MethodInterface and the interfaces of type MethodInterface (defined in Appendix B) can be reused in any other system wherever an interface of this type is required (e.g. other WBS systems for different types of aircraft). Same thing applies to *component types*. Within the new system, artefacts can be adapted by mapping their feature set to the target system. For example, component type Aircraft_BrakePedal has features Electronic_Brake and Mechanic_Brake. These features are one of the options used to apply the brake. The artefact description of the component type Aircraft_BrakePedal is defined in such a way that it can be reused in another system where electrical braking is not supported by simply selecting the feature Mechanic_Brake.	P5
L6	In the WBS architecture, information overload is addressed by having multiple views, each focusing on different aspects. For example, in order to stop/decelerate, the transaction domain WheelDecelerationOnGround (defined textually in Appendix B and presented graphically using event traces in Fig. 6) illustrates its behavioural description. In addition to the behavioural view, the sequential interaction of all the components involved in the transaction domain WheelDecelerationOnGround (such as component Accumulator in Fig. 7) are presented. While the system description (defined textually in Section 6.2.2 and presented graphically in Fig. 8) illustrates its structural description. The WBS architecture provides multiple views of a particular function (as a *transaction domain*) with a clear separation between structural and behavioural descriptions while maintaining consistency between them.	P6
L7	Behaviour in WBS has a dedicated section. For example, events such as HydraulicPressureRequest, AntiSkid, etc., have been defined clearly, with their source and destination *interface templates* (specified in Appendix B). From a visualisation perspective, the transaction domain WheelDecelerationOnGround which is presented in the form of event traces that demonstrate the ways an event can occur to stop/decelerate the aircraft, as demonstrated in Fig. 6. For example, in the transaction domain WheelDecelerationOnGround, Reserve_Pressure_Request and MPedal_Position_Request are the events that depend on the condition BluePressure_Failed. This is represented using the AND Fork notation (as defined in Table 4) meaning that it can occur concurrently in the *EMERGENCY* braking mode. Moreover, the interactions of the component Accumulator in the transaction domain WheelDecelerationOnGround are explicitly presented using a UML sequence diagram in Fig. 7.	P6

7.3 Discussion

The previous two subsections demonstrated how each case study addressed the limitations identified (Section 2.1) using the ALI principles (Section 3.1). However, some cross-cutting aspects such as cross application domain modelling are discussed in this section since two different domains are needed to fully illustrate cross-cutting issues.

As a product line comprising various back office portfolio management applications for financial instruments such as equity, commodity and currency, AMS corresponds to the Information Systems application domain. On the other hand, WBS is a single product system with multiple variants that specify different modes of how brakes can be applied to wheels of commercial aircraft and corresponds to the Embedded Systems application domain. Accordingly, ALI is the first ADL to be used successfully to model both Embedded Systems and Information Systems [71]. This demonstrates ALI's ability of modelling multiple application domains (principle P3), which addresses limitation L3.

The structural design of the AMS architecture uses connectors to join components. This can be visualised using the AMS graphical structural notation in Fig. 5. On the other hand, connections made between components in the WBS architecture are done via direct binding without the use of connectors (see Fig. 8). This demonstrates the flexibility of ALI in supporting architecture descriptions (P4) whether or not they require connectors to be first class citizens (L4), giving the architect the flexibility to choose one style or another.

As for varying complexity, in the WBS case study, it was enough to capture the complete structural information in a single view (i.e. an overall system architecture) as shown in Fig. 8. However, this would have been substantially more difficult to do with the AMS case study given the complexity of the system and the amount of information that needed to be captured. Accordingly, various structural views were produced, each capturing the information pertaining to a single major architectural artefact such as a transaction (see Fig. 2) or a transaction domain (see Fig. 5), depending on stakeholder concerns. Providing such multiple views allowed the notation to scale (P6), while still capturing all the required information at the appropriate level of abstraction (L6).

The AMS case study had comparatively a more complex structural description (components and connectors). Yet, it contained fewer and simpler interactions in the transactions leading to a relatively simple behavioural architecture. On the other hand, the WBS case study had a simpler structural architecture, yet far more sophisticated behavioural architecture. For instance, in WBS, interactions between components take place often with multiple events flowing concurrently. In both cases, ALI demonstrated its ability to provide the right mechanisms to capture the structural complexity (e.g. using structural notation, Fig. 5) and the behavioural complexity (e.g. using event traces, Fig. 6) as needed (addressing limitation L7).

Finally, architectural elements (such as components, connectors and interfaces) defined in both case studies can be reused with minimal changes across the two application domains (P5). For example, an interface template `MethodInterface` in AMS can be reused in the WBS by making minor changes in its *constraints*. Although this could be achieved with most ADLs that provide formal artefact descriptions, the reuse granularity in ALI is much larger (L5). This is due to ALI's support for capturing variability in artefact descriptions and linking that variability to external features. This allows the creation of highly configurable artefacts that can be adapted to various application domains by making minimal or no changes to their internal description (which is achieved by only varying the artefact feature set, an inherent characteristic of Software Product Line Engineering).

8 Study Limitations

This section discusses the limitations of our work. First, two large industry-scale case studies were developed, and evaluation work was performed to show how ALI addresses many of the limitations exhibited by existing ADLs (e.g. variability management, see Sections 2 and 7). However, the architecting work in the case studies was done by the authors of this study, in collaboration with domain experts (who provided the requirements for the two systems). From [71], we learned that some of the barriers to adopting ADLs is beyond just the ADL itself and include human aspects. This can only be evaluated when ALI is used by practitioners, and observed by the researchers without external support (to ensure unbiased feedback).

Second, given the large number of ADLs available, it was not possible to make a comparison between ALI and every other ADL, as this would have required the two case studies to be developed using every ADL. Accordingly, a different approach was used to show ALI's unique capabilities. We started by critically analysing existing works (Section 2.1), identified their limitations (Section 2.2), then conducted two case studies that showed ALI did not exhibit these limitations (Sections 6 and 7).

Finally, there is an inherent limitation in generalising the findings from a limited number of case studies. In reality, the majority of similar existing work reported on one or two case studies (and in some cases only toy running examples) due to the substantial resource requirements needed to develop such case studies. However, in analytic generalisation, there are two key criteria for judging the causal inference from the experimental design. The first is the statistical significance of the number of case studies (to allow extrapolation of results). And the second is the degree to which the selected case studies represent the whole population, in this case, embedded systems and Information Systems representing systems design in general (hence our choice of two completely different application domains).

9 Conclusion and Future Work

In this paper, the ALI ADL is presented including its various constructs and capabilities. ALI was designed to overcome a number of limitations (Section 2.2) by adopting a set of design principles (Section 3.1). Case studies were used to demonstrate ALI's abilities against the set of design principles and identified limitations.

The ALI notation demonstrated various unique capabilities. This includes the notation's support for large-scale reuse of architectural artefacts (components, connectors and interfaces); the ability to provide multiple architectural views as a mechanism to address information overloading via the separation of concerns; and the balance between formalism and flexibility.

The formal specification of ALI was also discussed by defining the structural and behavioural semantics explicitly. Rules were defined for the structural aspect of ALI using mathematical set theory. Behavioural semantics were defined using the CSP notation.

Finally, two case studies illustrated ALI. The case studies were chosen from different application domains with varying structural and behavioural complexities.

Firstly, an Asset Management System (AMS) case study was used to demonstrate the applicability of ALI in the Information Systems domain using a product line engineering process. The AMS architecture describes the portfolio management of financial instruments. In this case study, a portfolio valuation system, which considers equities (particularly shares) as the financial instrument, was designed. The second case study was a Wheel Brake System (WBS) that demonstrated the applicability of ALI in the Embedded Systems domain. The design encompassed having a single product with multiple variants (rather than a product line). The WBS architecture describes the braking system of a commercial aircraft.

Our future work will be focused on three fronts. First, the immediate intention is to continue to use ALI to model systems in more application domains and using other case studies to refine the notation and evaluate further aspects of the language (e.g. the usage of pattern templates as defined in Section 4.1.7). Second, the medium term aim is to create better tool support to facilitate the creation and analysis of ALI architectural descriptions. The ALI toolset will be developed in collaboration with industrial partners to help integrate it within existing tool chains used in architecture practices. Finally, and once the tool support has matured, we aim to recruit practitioners to run independent trials in operational environment which will allow us to evaluate the human aspect.

APPENDIX A: AMS ARCHITECTURE

AMS Component Types:

`PortfolioAMS_GUI` provides the asset managers using the system with the ability to view, analyse and value portfolios, to request (and monitor the progress of) long running system operations (such as order generation) and to check, enter, dispatch and monitor orders that go for execution in trading systems.

```
component type PortfolioAMS_GUI
  {
   meta: { }
   features: {
     /* no optional/alternative and parameterized
        features */
    }

   interfaces: {
    definition: {
       // no need to define any interface/s
     }
    implements: {
       ServiceRequest, UpdationStatus:
                               PortfolioMessenger;
     }
    }
   sub-system: {
    components { }
    connectors { }
    arrangement { }
   } // end of sub-system
  } // end of component type
```

`EquityCalculator` performs the mathematical operation based on the value and the method or message it received and then outputs the calculated portfolio value. It also calculates the derivative value of equities, if requested.

```
component type EquityCalculator
  {
   meta: Meta_Valuator {
     acceptance_value: “any numerical
value”;
     value_approximation: “2 significant
                           figures”;
     last_request: 10-02-2016;
    }
   features: {
      MTM_Method: “Share prices matched with
                  market price”'
      SCompany_Method: “Unlisted share price
                        of an individual
                        company”,
      WAV_Method: “Portfolio Valuation is
done
                  on the basis of average
                  share price”,
      E_Derivative: “Used as a security for
                     the equity asset”;
    }
   interfaces: {
    definition: { //no need to define any
                interface/s }
    implements:{
       if (supported(MTM_Method ||
                     SCompany_Method))
          OperationalValue:

ArithmeticOperation;
       if (supported(WAV_Method)){
          AverageMessage:
PortfolioMessenger;
          AverageValue: AverageOperation;}
       if (supported(E_Derivative)){
         DerivativeRequest:

PortfolioMessenger;
         DerivativeValue;

DerivativeOperation;}
        }
      } //end of interfaces
    sub-system: {
      components {
        if (supported(E_Derivative))
           PShareDerivative< >:

    DerivativeValuator;
        if (supported(WAV_Method))
           WAVData <false, false, true>:

    Internal_EquityData;
       }
      connectors {
        if (supported(E_Derivative)){
           Cal_Derivative< >:

    Calculator_Derivative;
           HTTP_DValue<true, true>:
    HTTP_Equity;}
        if (supported(WAV_Method)){
           HTTP_CalWAV<false, false, true>:

    HTTP_EquityCalculator;
           Cal_WAV<false, false, true>:

    Calculator_Equity;}
      }
arrangement {
     if (supported(E_Derivative)){
         connect
    PShareDerivative.DerivativeRequest with

    HTTP_DValue.messageport2;
         connect my.DerivativeRequest with

    HTTP_DValue.messageport1;
         connect
    PShareDerivative.DerivativeValue
                    with
    Cal_Derivative.valueport1;
         connect my.DerivativeValue with

    Cal_Derivative.valueport2;}
      if (supported(WAV_Method)){
         connect WAVData.ValuationRequest with

    HTTP_CalWAV.messageport2;
         connect my.AverageMessage with

    HTTP_CalWAV.messageport1;
         connect WAVData.AverageValue with

    Cal_WAV.valueport3;
         connect my.AverageValue with

    Cal_WAV.valueport4;}
      } // end of arrangement
    } // end of sub-system
  } // end of component type
```

AMS Products:

```
product configurations {
      … // defined in Section 4.1.7
     Equity_Share_ExchangeTraded: {
            Equity {Equity_Type = (long, short)};
            Equity_Share = true;
            MarkToMarket_Method = true;
            Share_Company_Method = false;
      }

     Equity_Share_Traded: {
            Equity {Equity_Type = long};
            Equity_Share = true;
            MarkToMarket_Method = false;
            Share_Company_Method = true;
      }
  } // end of product configurations
```

AMS Scenarios:

```
scenarios {
   … // defined in Section 4.1.11
  P.RevaluatingPC.ST_ET: {
    Description: "Revaluating portfolio due to
                  change in share price and ex
                  change trading both";
    Parameterisation: {
                         PriceChanged = true;
                         PriceUnchanged = false;
                         ShareTrade = true;
                         Exchange_Traded = true;
                         Illiquid = false;
                        }
    }
   P.RevaluatingPC.ST_IL: {
     Description: "Revaluating portfolio due to
                   change in share price and il-
                   liquid shares trading both";
     Parameterisation: {
                          PriceChanged = true;
                          PriceUnchanged = false;
                          ShareTrade = true;
                          Exchange_Traded = true;
                          Illiquid = true;
                         }
     }
  P.RevaluatingST_ET: {
    Description: “Revaluating portfolio due to
                 Exchange trading”;
    Parameterisation: {
                       PriceChanged = false;
                       PriceUnchanged = true;
                       ShareTrade = true;
                       Exchange_Traded = true;
                       Illiquid = false;
                      }
  }

  P.RevaluatingST_IL: {
    Description: “Revaluating portfolio due to
                 Illiquid shares trading”;
    Parameterisation: {
                       PriceChanged = false;
                       PriceUnchanged = true;
                       ShareTrade = true;
                       Exchange_Traded = false;
                       Illiquid = true;
                      }
  }
 } // end of scenarios
```

AMS Events:

```
events {
  ValuationRequest: <WSDL, WSDL>;
  RequestValuationDetails: <MethodInterface,
                               MethodInterface>;
  SendValuationDetails: <MethodInterface,
                               MethodInterface>;
  RequestPrice: <WSDL, WSDL>;
  CurrentStatus: <WSDL, WSDL>;
  RequestPriceList: <WSDL, WSDL>;
  CurrentPrice: <WSDL, WSDL>;
  UpdatedPriceList: <WSDL, WSDL>;
  SendValuation: <MethodInterface,
                               MethodInterface>;
  UpdateValue: <MethodInterface, MethodInterface>;
  Update: <MethodInterface, MethodInterface>;
  Notify: <MethodInterface, MethodInterface>;
  Inform: <(MethodInterface, WSDL),
                       (MethodInterface, WSDL)>;
  Access: <MethodInterface, MethodInterface>;
} // end of events
```

APPENDIX B: WBS ARCHITECTURE

WBS Meta Types:

```
meta type Meta_Brake {
   tag monitored_by, application: text;
   tag battery_charged_on*: date;
 }

meta type Meta_BrakePump {
   tag responsible_technician, failure_rate: text;
   tag threshold_value: number;
 }
```

WBS Features:

```
features {
   Electrical_Brake: {
      alternative names: {
        Designer.AF1, Developer.EB, Evaluator.F12;
       }
      parameters: {
        {Pedal_Value = number};
       }
    }

   Mechanical_Brake: {
      alternative names: {
        Designer.AF3, Developer.MP, Evaluator.F14;
       }
      parameters: {
        {Max_Pedal_Force = string};
       }
    }
        …
 } // end of features
```

WBS Interface Types:

```
interface type {
    DataOperation: MethodInterface {
        Provider: {
         function InsertBrakeData
                       {
                         impLanguage: Java;
                         invocation: insert;
                         parameterlist: (string);
                         return_type: void;

                       }
         }
       Consumer: {
           Call: insert (string);
        }
    }
   …
  } // end of interface types
```

WBS Component Types:

`Aircraft_BrakePedal` provides electrical braking data to the braking system as an input to the control unit. In the case of mechanical braking, it provides the pedal force and its position values to the metering valve.

```
component type Aircraft_BrakePedal {
   meta: Meta_WheelPedal {
     intention: "To apply the brake";
     consequences: "Aircraft will not stop";
     cost: 5000;
     last_checked: 28-04-2016;
   }
  features: {
    Electronic_Brake: "Electrical pedal used to
                       stop the aircraft wheel",
    Mechanic_Brake: "Mechanical pedal applied to
                     stop the aircraft wheel";
  }
  interfaces: {
    definition: {
      // no need to define any interface/s
     }
    implements:{
      if (supported(Electronic_Brake))
       BrakeData: DataOperation;
      if (supported(Mechanic_Brake)){
       MechanicalPosition: ValueOperation;
       MechanicalCommand: CommandOperation;}
    }
  } //end of interfaces
  sub-system: {
    components { }
    connectors { }
    arrangement { }
  } // end of sub-system
} // end of component type
```

WBS Product:

```
product configurations {
   CommercialAircraftBrake: {
     Electrical_Brake {Pedal_Value = 850KN};
     Electrical_Power {Voltage = 240V AC};
     Mechanical_Brake {Max_Pedal_Force = 980KN};
     Piston_Pressure {Maximum = 10.75 Pa,
                      Minimum = 5.25 Pa};
     Accumulator_Pressure {
                     Pressure_Supplied = 9.5 Pa};
    }
 } // end of product configurations
```

WBS Events:

```
events {
  Send_EPedal_Position1: <MethodInterface,
                               MethodInterface>;
  Send_EPedal_Position1: <MethodInterface,
                               MethodInterface>;
  Send_Power_Signal1: <MethodInterface,
                               MethodInterface>;
  Send_Power_Signal2: <MethodInterface,
                               MethodInterface>;
  Inform: <MethodInterface, MethodInterface>;
  Notify: <MethodInterface, MethodInterface>;
  CMD: <MethodInterface, MethodInterface>;
  AntiSkid: <MethodInterface, MethodInterface>;
  Hydraulic_Pressure_Request: <MethodInterface,
                               MethodInterface>;
  Send_Hydraulic_Pressure: <MethodInterface,
                               MethodInterface>;
  No_Hydraulic_Pressure_Supply: <MethodInterface,
                               MethodInterface>;
  MPedal_Position_Request: <MethodInterface,
                               MethodInterface>;
  Send_MPedal_Position: <MethodInterface,
                               MethodInterface>;
  Reserve_Pressure_Request: <MethodInterface,
                               MethodInterface>;
  Decelerate: <MethodInterface, MethodInterface>;
} // end of events
```

WBS Conditions:

```
conditions {
   BSCU: "BSCU working properly";
   BSCU_Failed: "Unable to provide brake command";
   GreenPressure: "Provide hydraulic pressure in a
                   normal mode";
   GreenPressure_Failed: "No hydraulic pressure
                  supply or below threshold value";
   BluePressure: "Provide hydraulic pressure in an
                  alternate mode";
   BluePressure_Failed: "No hydraulic pressure
                  supply or below threshold value
                  in an alternate mode";
   AccumulatorPump: "Provide hydraulic pressure in
                    an emergency mode";
} // end of conditions
```

WBS Scenarios:

```
scenarios {
   NormalOperation {
      Description: "WBS in a normal mode";
      Parameterisation {
                         BSCU_Active = true;
                         GreenPressure = true;
                         BluePressure = false;
                         AccumulatorPump = false;
                       }
     }

   AlternateOperation {
      Description: "WBS is in alternate mode with
                Antiskid command";
      Parameterisation {
                         BSCU_Active = true;
                         GreenPressure_Failed =
                                          true;
                         BluePressure = true;
                         AccumulatorPump = false;
                       }
     }

   BSCUFailureOperation {
      Description: "WBS is in alternate mode with
                out Antiskid command";
      Parameterisation {
                         BSCU_Failed = true;
                         GreenPressure = true;
                         BluePressure = true;
                         AccumulatorPump = false;
                       }
     }

   EmergencyOperation {
      Description: "WBS is in emergency mode";
      Parameterisation {
                         BSCU_Failed = true;
                         GreenPressure_Failed =
                                          true;
                         BluePressure_Failed =
                                          true;
                         AccumulatorPump = true;
                       }
     }
} // end of scenarios
```

WBS Transaction Domain:

```
transaction domain WheelDecelerationOnGround
  {
   meta: Meta_DecelerationDomain
     {
      purpose: "To stop the commercial aircraft on
                ground";
      minimum_wheels_active: 4;
     }
   contents:
     {
      /*provides the list of components involved
        in this transaction domain*/
      Components: {Electrical_Pedal,
                Mechanical_Pedal, Power, BSCU,
                ShutOff_Valve, Selector_Valve,
                Green_Pump, Blue_Pump, Accumulator,
                Meter_Valve, Wheel}
       //No connectors -direct binding
    }
   transactions:
   {
    NORMALMODE: {…}

    EMERGENCYMODE: {
     events: {MPedalPositionRequest, ReservePres-
              sureRequest, SendHydraulicPressure,
              SendMPedalPosition, Decelerate}
     interactions: {
      [Selector_Valve.MechanicalPosition sends
       MPedalPositionRequest to
       Mechanical_Pedal.MechanicalPosition,
       Selector_Valve.PressureMessage sends
       ReservePressureRequest to
       Accumulator.PressureMessage];
      [Meter_Valve.MechanicalCommand receives
       SendMPedalPosition from
       Mechanical_Pedal.MechanicalCommand,
      Meter_Valve.ReservePressure receives
      SendHydraulicPressure from
      Accumulator.ReservePressure];
     Meter_Valve.BrakePressure sends Decelerate
     to Wheel.InputPressure;
    }
   }
   ALTERNATEMODE1: {…}

   ALTERNATEMODE2: {…}

   DECELERATINGWHEEL: {
     /* No events in this transaction therefore,
        there is no event section */
    interactions: {
      if (supported(Electircal_Brake &&
                   Electrical_Power) &&
          (BSCU_Active && GreenPressure))
         {NORMALMODE;}
      else if (unsupported(Electrical_Power &&
                         Piston_Pressure) &&
             (BluePressure_Failed))
         {EMERGENCYMODE;}
      else if (supported(Electircal_Power) &&
               unsupported (Accumulator_Pressue)
               &&(BSCU_Active && GreenPressure-
                  _Failed))
         {ALTERNATEMODE1;}
      else
         {ALTERNATEMODE2;}
     } // end of interaction
   } // end of transaction
  } // end of transactions section
} // end of transaction domain
```

WBS Viewpoint:

```
viewpoints {
   WheelDeceleration: {
      Description: “Decelerating the aircraft
                    wheel”;
      Transaction Domain: {
                         WheelDecelerationOnGround,
                               WheelDecelerationOnGear;}
   }
} // end of viewpoints
```

REFERENCES

[1] P. Kruchten, H. Obbink, and J. Stafford, "The Past, Present, and Future for Software Architecture," *IEEE Software*, vol. 23, no. 2, pp. 22-30, 2006.

[2] M. Shaw and P. Clements, "The golden age of software architecture," *IEEE Software*, vol. 23, no. 2, pp. 31-39, 2006.

[3] M. Shaw, "The coming-of-age of software architecture research," in *Proceedings of the 23rd International Conference on Software Engineering, ICSE*, 2001, pp. 657-664a.

[4] D. Garlan, "Software architecture: a travelogue," in *Proceedings of the Future of Software Engineering*, Hyderabad, India, 2014, pp. 29-39: ACM.

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[6] P. Lago, I. Malavolta, H. Muccini, P. Pelliccione, and A. Tang, "The Road Ahead for Architectural Languages," *IEEE Software*, vol. 32, no. 1, pp. 98-105, 2015.

[7] *Architectural Languages Today*. Available: http://www.di.univaq.it/malavolta/al/ [Accessed: 4-May-2018]

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

[9] P. C. Clements, "A Survey of Architecture Description Languages," in *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996, pp. 16-25: IEEE Computer Society.

[10] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The Architecture Analysis & Design Language AADL: An Introduction," *Software Engineering Institute, Carnegie Mellon University*, Pittsburgh, USA.2006.

[11] P. Cuenot *et al.*, "The EAST-ADL Architecture Description Language for Automotive Embedded Software," in *Proceedings of the International Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, 2010, vol. 6100, pp. 297-307: Springer Berlin Heidelberg.

[12] R. v. Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," *IEEE Computer*, vol. 33, pp. 78-85, 2000.

[13] R. Bashroush, T. J. Brown, I. Spence, and P. Kilpatrick, "ADLARS: An Architecture Description Language for Software Product Lines," in *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop*, 2005, pp. 163-173.

[14] R. Bashroush, I. Spence, P. Kilpatrick, and J. Brown, "Towards more flexible architecture description languages for industrial applications," in *Proceedings of the Third European conference on Software Architecture*, Nantes, France, 2006, pp. 212-219, 2081985: Springer-Verlag.

[15] E. Woods and R. Hilliard, "Architecture Description Languages in Practice Session Report," in *Proceedings of the 5th Working IEEE/IFIP Conference onSoftware Architecture, WICSA*, 2005, pp. 243-246.

[16] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What Industry Needs from Architectural Languages: A

Survey," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 869-891, 2013.

[17] M. Galster, T. Männistö, D. Weyns, and P. Avgeriou, "Variability in software architecture: the road ahead," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 4, pp. 33-34, 2014.

[18] U. Haider, "Representing Variability in Software Architecture," PhD Thesis, School of Architecture, Computing & Engineering, University of East London, London, UK, 2016.

[19] F. Oquendo, "π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures," *ACM SIGSOFT Softw. Eng. Notes*, vol. 29, no. 3, pp. 1-14, 2004.

[20] C. Yao, L. Xiaoqing, Y. Lingyun, L. Dayong, T. Liu, and Y. Hongli, "A ten-year survey of software architecture," in *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences (ICSESS)*, 2010, pp. 729-733.

[21] P. Binns, M. Englehart, M. Jackson, and S. Vestal, "Domain-Specific Software Architectures for Guidance, Navigation and Control.," *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 2, pp. 201-227, 1996.

[22] J. Magee and J. Kramer, "Dynamic structure in software architectures," in *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, San Francisco, California, USA, 1996, pp. 3-14: ACM.

[23] T. N. Qureshi, D. Chen, H. Lönn, and M. Törngren, "From EAST-ADL to AUTOSAR software architecture: a mapping scheme," presented at the Proceedings of the 5th European conference on Software architecture, Essen, Germany, 2011.

[24] D. Garlan, R. Monroe, and D. Wile, "Acme: an architecture description interchange language," in *Proceedings of the Centre for Advanced Studies on Collaborative research, CASCON'*, 1997, pp. 169-183.

[25] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Transactions on Software Engineering,*, vol. 21, no. 4, pp. 314-335, 1995.

[26] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "A comprehensive approach for the development of modular software architecture description languages," *ACM Transactions on Software Engineering Methodology*, vol. 14, pp. 199-245, 2005.

[27] N. Medvidovic, R. N. Taylor, and E. J. J. Whithead, "Formal modeling of software architectures at multiple levels of abstraction," in *Proceedings of the California Software Symposium*, California, USA, 1996, pp. 28-40.

[28] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using Rapide," *IEEE Transactions on Software Engineering*, vol. 21, pp. 336-355, 1995.

[29] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 3, pp. 213-249, 1997.

[30] C. A. R. Hoare, *Communicating sequential processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[31] R. Allen, R. Douence, and D. Garlan, "Specifying and Analyzing Dynamic Software Architectures," in *Proceedings of the First International Conference on Fundamental Approaches to Software Engineering, FASE*, 1998, vol. 1382, pp. 21-37: Springer Berlin Heidelberg.

[32] "ISO/IEC/IEEE Systems and software engineering -- Architecture description," *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pp. 1-46, 2011.

[33] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting style in architectural design environments," *SIGSOFT Softw. Eng. Notes*, vol. 19, no. 5, pp. 175-188, 1994.

[34] A. Garcia *et al.*, "On the Modular Representation of Architectural Aspects," in *Proceedings of the Third European Workshop on Software Architecture, EWSA*, 2006, vol. 4344, pp. 82-97: Springer Berlin Heidelberg.

[35] A. Haber, H. Rendel, B. Rumpe, I. Schaefer, and F. van der Linden, "Hierarchical Variability Modeling for Software Architectures," in *Proceedings of the 15th International Software Product Line Conference (SPLC)*, 2011, pp. 150-159.

[36] M. M. Gorlick and R. R. Razouk, "Using weaves for software construction and analysis," in *Proceedings of the 13th International Conference on Software Engineering, ICSE*, 1991, pp. 23-34.

[37] H. Mei, F. Chen, A. Q. Wang, and A. Y.-D. Feng, "ABC/ADL: An ADL Supporting Component Composition," presented at the Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, London, UK, 2002.

[38] Z. Wang *et al.*, "An Architecture Description Language Based on Dynamic Description Logics," in *Proceedings of the 7th TC 12 International Conference on Intelligent Information Processing VI*, 2012, vol. 385, pp. 157-166: Springer Berlin Heidelberg.

[39] A. Rademaker, C. Braga, and A. Sztajnberg, "A Rewriting Semantics for a Software Architecture Description Language," *Electronic Notes in Theoretical Computer Science*, vol. 130, no. 0, pp. 345-377, 5/12/ 2005.

[40] C. Canal, E. Pimentel, and J. M. Troya, "Specification and Refinement of Dynamic Software Architectures," in *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA)*, San Antonio, Texas USA, 1999, pp. 107-126: Kluwer, B.V.

[41] B. Magableh and S. Barrett, "Primitive component architecture description language," in *Proceedings of the 7th International Conference on Informatics and Systems (INFOS)*, 2010, pp. 1-7.

[42] J. Perez, I. Ramos, J. Jaen, P. Letelier, and E. Navarro, "PRISMA: towards quality, aspect oriented and dynamic software architectures," in *Proceedings of the Third International Conference on Quality Software*, 2003, pp. 59-66.

[43] J. Xiangyang, Y. Shi, C. Honghua, and D. Xie, "A New Architecture Description Language for Service-Oriented Architecture," in *Proceedings of the Sixth International Conference on Grid and Cooperative Computing, GCC*, 2007, pp. 96-103.

[44] G.-q. Zhang, H.-j. Shi, and M. Rong, "Mismatch Detection of Asynchronous Web Services with Timed Constraints," in *Proceedings of the IEEE Asia-Pacific Services Computing Conference (APSCC)*, 2011, pp. 251-258.

[45] T. Zhang, D. Xiang, and H. Wang, "vADL: A Variability-Supported Architecture Description Language for Specifying Product Line Architectures," in *Proceedings of the Second International Software Product Lines Young Researchers Workshop (SPLYR) in conjunction with the 9th International Software Product Line conference (SPLC)*, Rennes, France, 2005, pp. 31-37.

[46] I. Alloui and F. Oquendo, "Supporting Decentralised Software-Intensive Processes Using ZETA Component-Based Architecture Description Language.," in *Enterprise Information Systems III*, vol. 3, J. Filipe, B. Sharp, and P. Miranda, Eds., 2002, pp. 97-106.

[47] C. Chaudet and F. Oquendo, "pi-SPACE: a formal architecture

description language based on process algebra for evolving software systems," in *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, ASE*, 2000, pp. 245-248.

[48] M. Auguston, "Monterey Phoenix, or how to make software architecture executable," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, Orlando, Florida, USA, 2009, pp. 1031-1040: ACM.

[49] M. Pinto, L. Fuentes, and J. M. Troya, "DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development," in *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering, GPCE*, 2003, vol. 2830, pp. 118-137: Springer Berlin Heidelberg.

[50] P. Poizat and J.-C. Royer, "A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic," *Journal of Universal Computer Science*, vol. 12, no. 12, pp. 1741-1782, 2006.

[51] C. K. Chang and K. Seongwoon, "I<sup>3</sup>: a Petri-net based specification method for architectural components," in *Proceedings of the Twenty-Third Annual International Computer Software and Applications Conference, 1999. COMPSAC '99.*, 1999, pp. 396-402.

[52] P. Klien, "The Architecture Description Language MoDeL," in *Graph Transformations and Model-Driven Engineering*, vol. 5765, G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, Eds. (Lecture Notes in Computer Science: Springer Berlin Heidelberg, 2010, pp. 249-273.

[53] S. Faulkner and M. Kolp, "Towards An Agent Architectural Description Language For Information Systems," in *Proceedings of the 5th Int. Conf. on Enterprise Information Systems ICEIS*, France, 2003, pp. 59-66: Press.

[54] D. Cassou, B. Bertran, N. Loriant, and C. Consel, "A generative programming approach to developing pervasive computing systems," *ACM SIGPLAN Notices*, vol. 45, no. 2, pp. 137-146, 2009.

[55] N. Ubayashi, J. Nomura, and T. Tamai, "Archface: a contract place where architectural design and code meet together," in *Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering, ICSE*, 2010, vol. 1, pp. 75-84.

[56] D. Su, B. De Fraine, and W. Vanderperren, "FuseJ: An architectural description language for unifying aspects and components," in *Proceedings of the AOSD 2005 Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT '05)*. 2005, pp. 1-8.

[57] A. Navasa, M. A. Pérez-Toledano, and J. M. Murillo, "An ADL dealing with aspects at software architecture stage," *Information and Software Technology*, vol. 51, no. 2, pp. 306-324, 2// 2009.

[58] M. Svahnberg and J. Bosch, "Issues Concerning Variability in Software Product Lines," in *Proceedings of the International Workshop on Software Architectures for Product Families*, 2000, pp. 146-157.

[59] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou, "Variability in Software Systems - A Systematic Literature Review.," *IEEE Transaction Software Engineering*, vol. 40, no. 3, pp. 282-306, 2014.

[60] E. A. Barbosa, T. Batista, A. Garcia, and E. Silva, "PL-AspectualACME: an aspect-oriented architectural description language for software product lines," presented at the Proceedings of the 5th European conference on Software architecture (ECSA), Essen, Germany, 2011.

[61] Y. Oh, D. H. Lee, S. Kang, and J. H. Lee, "Extended Architecture Analysis Description Language for Software Product Line Approach in Embedded Systems," in *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE*, Nice, France, 2007, pp. 87-88: IEEE.

[62] E. Silva, A. L. Medeiros, E. Cavalcante, and T. V. Batista, "A Lightweight Language for Software Product Lines Architecture Description," in *Proceedings of the 7th European Conference on Software Architecture, ECSA*, Montpellier, France, 2013, vol. 7957, pp. 114-121: Springer.

[63] S. Adjoyan and A. Seriai, "An Architecture Description Language for Dynamic Service-Oriented Product Lines," in *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Pittsburgh, USA, 2015.

[64] A. Haber *et al.*, "Engineering Delta Modelling Languages," presented at the Proceedings of the 17th International Software Product Line Conference (SPLC), Tokyo, Japan, 2013.

[65] E. Cavalcante, A. L. Medeiros, and T. Batista, "Describing Cloud Applications Architectures," in *Proceedings of the 7th European conference on Software Architecture, ECSA*, Montpellier, France, 2013, pp. 320-323: Springer-Verlag.

[66] A. Haber, J. O. Ringert, and B. Rumpe, "MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems," RWTH Aachen University, Germany, 2012.

[67] J. O. Ringert, B. Rumpe, and A. Wortmann, "MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems," in *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany., 2013.

[68] D. Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio, "ByADL: An MDE Framework for Building Extensible Architecture Description Languages," in *Proceedings of the 4th European Conference on Software Architecture, ECSA*, 2010, vol. 6285, pp. 527-531: Springer Berlin Heidelberg.

[69] T. J. Brown, R. Gawley, R. Bashroush, I. Spence, P. Kilpatrick, and C. Gillan, "Weaving behavior into feature models for embedded system families," in *Proceedings of the 10th International Software Product Line Conference (SPLC)*, 2006, pp. 52-61.

[70] R. Bashroush, I. Spence, P. Kilpatrick, T. J. Brown, W. Gilani, and M. Fritzsche, "ALI: An Extensible Architecture Description Language for Industrial Applications," in *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, ECBS*, 2008, pp. 297-304, 1396050: IEEE Computer Society.

[71] E. Woods and R. Bashroush, "Modelling large-scale information systems using ADLs – An industrial experience report," *Journal of Systems and Software*, vol. 99, pp. 97-108, 1// 2015.

[72] M. Galster and P. Avgeriou, "Handling Variability in Software Architecture: Problems and Implications," in *Proceedings of the Ninth Working IEEE/IFIP Conference on Software Architecture*, 2011, pp. 171-180, 2015583: IEEE Computer Society.

[73] J. Johnson and A. Henderson, "Conceptual models: begin by designing what to design," *Interactions*, vol. 9, no. 1, pp. 25-32, 2002.

[74] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide, 2nd Edition (Addison-Wesley Object*

Technology Series). Addison-Wesley Professional, 2005.

[75] T. Halpin, "Object-Role Modeling: Principles and Benefits," *International Journal of Information System Modeling and Design*, vol. 1, no. 1, pp. 33-57, 2010.

[76] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and L. William, *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

[77] T. Halpin and T. Morgan, *Information Modeling and Relational Databases*. Morgan Kaufmann Publishers Inc., 2008.

[78] *Java Compiler Compiler tm (JavaCC tm) - The Java Parser GeneratorJava Compiler Compiler tm (JavaCC tm) - The Java Parser Generator*. Available: https://javacc.java.net/

[79] R. S. Scowen, "Extended BNF - A Generic Base Standard," in *Proceedings of the Software Engineering Standards Symposium (SESS)*, Brighton, United Kingdom, 1993.

[80] D. L. Moody, "The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756 - 779, 2009.

[81] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541-580, 1989.

[82] D. Harel and B. Rumpe, "Meaningful modeling: what's the semantics of "semantics"?," *Computer*, vol. 37, no. 10, pp. 64-72, 2004.

[83] P. Armstrong, G. Lowe, J. Ouaknine, and A. W. Roscoe, "Model checking Timed CSP," in *Proceedings of the HOWARD-60. A Festschrift on the Occasion of Howard Barringer's 60th Birthday*, 2012, pp. 13-33: EasyChair.

[84] *ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, December 1996.