Explicating Critical Assumptions in Software Architectures Using AADL

HAMED ORDIBEHESHT

Supervisor: Jörgen Hansson

Master of Software Engineering and Management Thesis

Report No. 2010.
ISSN: 1651-476

University of Gothenburg
Department of Applied Information Technology
Gothenburg, Sweden, June 2010

Abstract

Developers make assumptions constantly at different levels and throughout software development life-cycles. Implicit assumptions made during higher development levels, such as in architecture design, have major impacts which lead to systems failures and poor performances according to research findings. In this thesis we have taken a quantitative approach to develop a modeling method for explication of architectural assumptions by specifying their information in architecture descriptions using the AADL. Our work has resulted in an assumption modeling method that consists of two main parts; assumption specification meta-model and assumption specification approach. Assumption specification meta-model is used for structuring the information about assumptions. The proposed specification approach uses the AADL concepts to specify the meta-models together with architecture descriptions. In this study two approaches are investigated. One of the approaches uses property sets while the other one uses annex in the AADL for specifying meta-models together with architecture description. A case example is also introduced to assist illustrating the methods. We believe that the proposed assumption modeling method can be used by software architects and design decision makers in order to provide assumption awareness and traceability which are two main identified keys in this thesis work towards systematic management of assumptions.

Keywords: assumption management, assumption modeling, assumption meta-model, software architecture, AADL, assumption awareness, traceability, implicit assumption, explicit assumption, architectural assumptions

Acknowledgments

I am grateful to have had the opportunity to work with Prof. Jörgen Hansson on this thesis. He has been a dedicated, passionate, and patient supervisor who showed me the light when things were obscure. His advice has been instrumental to this work. I am thankful to Dr. Dionisio de Niz too for sharing his knowledge with me in the course of this work.

I would like to also thank Dr. Miroslaw Staron for his efforts and devotion towards me throughout my two years of study at IT University of Gothenburg-Chalmers. I was fortunate to have had such a great mentor, colleague, and friend and I appreciate him personally and professionally.

I would like to dedicate this thesis to my family. I am solely indebted to them, especially my parents for their support, encourage, and unlimited love during my years of study at IT University of Gothenburg-Chalmers and before that. They have always been inspiring me towards achieving my goals and realizing my dreams. I admire them for everything they have done, and continue to do, for me.

Hamed Ordibehesht

June 2010

Table of Contents

Chapter 1 Introduction ... 1
1.1 Report Outline ... 3
Chapter 2 Background ... 4
Chapter 3 Problem Formulation ... 6
3.1 Background ... 6
3.2 Problem Analysis ... 7
3.3 Aims and Objectives ... 8
Chapter 4 Assumption Specification ... 10
4.1 Definitions ... 12
Chapter 5 A Case Example ... 13
5.1 AADL Introduction ... 14
5.2 Case Example: Video Streaming System ... 19
5.2.1 Video Encoding Assumption ... 22
Chapter 6 Reusable Property Sets In AADL ... 24
6.1 Background ... 24
6.2 Motivation ... 27
6.3 Proposal ... 30
6.4 Conclusion ... 33
Chapter 7 Modeling Architectural Assumptions ... 35
7.1 Assumption Specification Meta-Model ... 36
7.2 Approach 1: Assumption Specification through Property Sets in AADL ... 39
7.2.1 Extending the Approach using Property Set Reuse Extension ... 43
7.3 Approach 2: Assumption Specification through Annex ... 49
7.4 A Good Solution Criteria ... 53
7.4.1 Legibility ... 53
7.4.2 Practicality ... 53

7.4.3 Extensibility 54

7.4.4 Backward Compatibility 54

7.4.5 Analysis Support 54

7.5 Evaluation 55

Chapter 8 Related Word 58

8.1 Assumption Management Framework 58

8.2 Explicit Assumptions Enrich Architectural Models 59

8.3 Assumption Management System 59

Discussions 60

Chapter 9 Conclusions 62

References 65

Table of Figures

Figure 5-1.The SAE AADL Standard Elements Summery [12] 15
Figure 5-2.Sensor Interface Texual Representation 16
Figure 5-3.Thread Component Pre-Declared Properties 16
Figure 5-4.Summary of AADL elements in graphical representation [12] 17
Figure 5-5.Example Error Model Annex Library Declaration [21] 18
Figure 5-6.Example Error Model Annex Sub-Clause Declaration [21] 18
Figure 5-7.Video Streaming System Component Hierarchy 19
Figure 5-8.Video Streaming System Schematic 20
Figure 5-9.Video Stream System Partial Detail Architecture 21
Figure 5-10.Video Streaming System Specification in AADL text from 22
Figure 6-1.Sample of New Property Set 24
Figure 6-2.Sample of New Property Set Use 25
Figure 6-3.Sample of New Property Type 25
Figure 6-4.Sample of Different Property Types 26
Figure 6-5.Sample of Constant Property 26
Figure 6-6.Sample of Reference Component as a Property Value [12] 27
Figure 6-7.Sample of Lists Property Value 28
Figure 6-8.Property Sets Limitation Example in the SAE AADL Standard 28
Figure 6-9.Common Property Modification Example in the SAE AADL Standard 29
Figure 6-10.Property Set Inheritance Diagram 30
Figure 6-11.Property Set Inheritance Declaration 30
Figure 6-12.Property Set Dependency Declaration 31
Figure 6-13.Property Set Inheritance Example 32
Figure 6-14.Property Sets Dependency Example 33
Figure 7-1.Assumption Meta-Model Dependency Concept Diagram 37
Figure 7-2.Assumption Specification Meta-Model in Property Set Declaration 39
Figure 7-3.Declaration of Property Set for Video Encoding Assumption 41
Figure 7-4.First Step - Declaration of the Assumption Property Set containing Criticality Custom Attribute 41
Figure 7-5.Second Step – Specification of Assumption's Properties in the Architecture Description of the VSS Case Example 42
Figure 7-6.VSS AADL Graphical Representation containing Video Encoding Assumption 43
Figure 7-7.Property Set Reuse for Declaration of Common Assumption Meta-Model 45

Figure 7-8.Assigning Inherited Properties for Video Encoding Assumption....................................... 45
Figure 7-9.Property Set Inheritance for Declaration of Criticality Custom Attributes 46
Figure 7-10.Property Set Reuse for Declaration of Common Dependency Custom Attribute............. 47
Figure 7-11.Assigning Dependency Properties for Video Encoding Assumption 48
Figure 7-12.Assumption Specification Meta-Model Declaration in Annex Library............................. 49
Figure 7-13.Video Encoding Assumption Declaration in Assumption Meta-Model Annex Library ... 50
Figure 7-14.Assignment of Video Encoding Assumption through Annex Sub-Clause 51
Figure 7-15.Example of Common Meta-Model using Extended Sub-Language 52

Chapter 1

Introduction

Software architecture is a set of decisions about the structure of a system and interactions of its components. Several methods and approaches are designed to capture and explain those architecturally significant decisions that are influenced by software requirements. In other words, software requirements are the main reasons for software architecture. Assumptions are other sources of reasons for architectural decisions. Assumptions are implicit facts on which architects rely for making decisions about the organization of the software.

Invalid assumptions are sources of problems. When it comes to critical systems with complex architectures, they lead to system failures or poor performance. This is experienced in several industry and academic projects. The Ariane 5 [3] and ASEOP [10] are the result of invalid assumptions in software architectures. Therefore, it is necessary to find and fix those invalid assumptions.

Architecture description languages (ADLs) are designed to describe software architectures. ADLs do it by providing disciplinary specifications for the decisions made by the architects; however, there is a limited support for describing the reasons behind the decisions. Thus, architectural assumptions become invisible in architecture descriptions. As a result, invalid assumptions will become hard to monitor and recover.

Our approach is to explicate architectural assumptions. We do this by using the SAE AADL standard. The AADL is an architecture description language for modeling and analysis of software architectures [12]. It specially supports model-based analysis and specification of software and hardware components [12]. The AADL aids architects to analyze the software in its interactions with the environment. Environment is an infrastructure that the software works and interacts in. It is mostly made from hardware. In the AADL, analyses of environments can be realized through modeling environmental components together with software components and specifying the bindings and connections among them.

Our modeling method enables specification of architectural assumptions together with the software architecture description. Using the AADL concepts, we define an approach that supports traceability and assumption awareness in software development processes. Traceability of assumptions is important because it makes it possible to track architectural assumptions to implementations and vice

versa. The main benefit of this is that the cost of fixing invalid assumptions decreases significantly. Besides, visibility and transparency is the result of assumption awareness in the development process. Therefore, architectural analyses benefit from an enriched architecture.

In our modeling method, at first, we introduce a meta-model for description of assumptions. The meta-model helps architects to form assumptions into descriptive structures. Secondly, we introduce two approaches by which the assumptions' meta-models can be specified in the description of architectures. These approaches use the SAE AADL standard. Lastly, we discuss our evaluation of the two approaches by identification of their benefits and weaknesses.

1.1 Report Outline

The rest of the report is organized as follow. In Chapter 2, terminology and background knowledge needed for understanding the rest of this report is given. The Chapter 3 contains the formulation of the problem including the background, problem analysis, and aim and objectives. The report continues with Chapter 4 where it describes the domain of this study. It follows with Chapter 5 that explains a brief introduction to the SAE AADL standard together with a case example to test our approach. Chapter 6 describes our proposed extension to the AADL to support reusable property sets. We need this to improve our first approach. Our assumptions modeling method is explained in detail in Chapter 7. This chapter also includes the evaluation of the approaches. The report ends with the discussions and related works in Chapter 8 and a summery in Chapter 9 where conclusions and future work are given.

Chapter 2

Background

We all make assumptions in our daily life. An assumption is a fact to be taken as true but there is no given proof for that. In other words, we make assumptions when we believe in facts without any proof.

Making assumptions is inevitable in software development. In daily tasks, developers frequently make assumptions at different levels. It is also experienced by software industry and researchers that making assumptions by developers is common [5]. It is known that assumptions are important for solving complicated problems and structuring a system design. For example, if it is assumed that a specific controlling component in a system operates in an isolated and secure environment, developing authentication and authorization functionalities for that component does not seem necessary. However, similar to design decisions and implementation, it is important to make sure that the assumptions made by development team members are valid.

From several studies and experience one can identify two main sources of difficulties with regards to assumptions; wrong assumptions and implicit assumptions. Wrong assumptions are the ones that are made incorrect at the first place or are invalidated through fixing or modification processes. Studies of real incidents have shown that wrong assumptions have had major impacts on software, even very harsh in some cases. For instance, the root cause of the Ariane 5 incident was a wrong assumption, which led to the explosion of billions of dollars and years of research and construction [3]. In this disaster, a wrong assumption during migration of the software from Ariane 4 to Ariane 5 led to an overflow of a conversion from 64 bits to 16 bits in the horizontal velocity value component [3]. Tirumala discusses this example as a classic case of software components reuse with invalid assumptions made on the target environment [3]. The assumption made in Ariane 4 that the horizontal velocity value could never overflow 16 bits was invalid for Ariane 5.

Implicit assumptions are the second origin for several system errors and failures. They are implicit because they are not visible in development processes irrespective of whether the assumptions are valid or not. Several studies have identified implicit assumptions as potential source of sever incidents. Unexpected behavior of a car airbag system and consequently tragic death of a baby is a sad example in which a wrong assumption in a car airbag controlling system led to an undeniable error [3]. In this example, the car airbag system comprised primary and backup controllers [3]. The backup controller was designed to take control of the system in the situations when the primary controller was

out of order [3]. This could happen due to high temperature or humidity conditions [3]. The primary controller of the system had a critical assumption of not allowing the airbags to be deployed in the presence of a child-seat in a specific seat. According to experts, the backup controller was mistakenly designed in such a way that the airbag was deployed irrespective of the presence of the child-seat [3]. This happened because of the implicitness of this assumption when the backup controller actually took the action in the system due to some reasons.

Architectural mismatch is another type of impacts of implicit assumptions which is experienced numerously in several projects. Garlan et al. demonstrate the harsh reality experienced in building a family of software design environment from existing parts which was initially estimated to take six months and one person-year to develop [10]. In the design of this software tool Garlan's team used several Commercial-Off-The-Shelf (COTS) components. In reality the project took two years and nearly five person-years for development of the first prototype. Even after the project finished the team confessed that the performance was so poor and it was very hard to maintain the developed system due to the complex code [10]. They identified implicit assumptions as the main reasons for architectural mismatches during component reuse. In their study they motivate that invisible assumptions in COTS components have led to numerous error-prone activities as well as wrong design decisions in software development and component reuse [10].

Various potential reasons are found in each of the studies for wrong or implicit assumptions. Some of the reasons are similar while the others are distinct. However, importance of assumptions in software development can be concluded from almost all the analysis studies. It is clear that invalidity of assumptions has a major impact on the result of software development activities. Therefore, it is needed to make sure that the assumptions made throughout the development activities are all upfront to be able to assure that they are always valid. This is because only by knowing which assumptions are in place one can judge whether they are valid or not.

Chapter 3

Problem Formulation

3.1 Background

The consequences of wrong assumptions have been very harsh. This brought the attention of the software community to find a solution for making sure that the assumptions are valid throughout the software development life-cycle as well as during evolution processes. The efforts in this area brought about different solutions for managing assumptions that are made in different levels and domains of software development.

The efforts that have been carried out by academia and industry for building an assumption management solution cover different domain areas. Some projects focus on implementation practices such as assumption management system [13]. However, the result of these efforts more or less contributes to a combination of the following key tasks in assumption management identified by Ostacchini and Wermelinger [2]:

- Recording assumptions
- Monitoring assumptions (on a regular basis)
- Searching for assumptions
- Recovering techniques for past assumptions by looking through assumption documentation and conduct interviews when necessary

Among these high level tasks, recording assumptions has been the aim of most of the works in this area. The works such as Assumption Management Framework (AMF) [3], Assumption Management System [13], and the assumption modeling meta-model [6] result in developing tools and techniques for explication of the assumptions. The other works ([5]and [10]) target identification of some techniques and practices for recording the assumptions resulted from their professional experiences as well as case studies.

Explication of assumptions is the first step for managing them throughout software development life-cycles. The importance of this fact is investigated deeply by experts [1], [2], [3], [6], [7], [8], and [13]. Lago and van Vliet categorize assumptions into three levels [6]:

- Technical assumptions
- Organizational assumptions

- Managerial assumptions

Among these categories, technical assumptions are the ones that are made during the technical activities within a development process. They are mainly the assumptions that are made by architects, developers, implementers, testers, and maintainers about the on-the-fly technological facts.

Three main sources of making technical assumptions have been identified [5]:

- Design-Time activities
- Implementation-Time activities
- Run-Time activities

Assumptions that are made during any of the above activities are named after the respective activity. This means that design-time assumptions are the assumptions that are made during design of systems. Among the above sources of assumptions, architectural assumptions are part of the design-time assumptions. They are the assumptions that are made by software architects in structuring software and organization of components interactions.

3.2 Problem Analysis

Among software architects it has become common to make assumptions about the environment in which software components work in. Assumptions that are made in this level assist architects in capturing decisions that are essential to solve faced problems. Therefore, it becomes very important to make sure that architectural assumptions are valid. If architectural assumptions stay implicit, the result of the development process becomes greatly affected by the problems raised due to those invalid assumptions.

The first raised problem is introduction of the software defects which can be caught during the later development phases of the system. This can happen either because of the conflicts faced by a developer during the implementation or unsuccessful test results. If in later phases it is found that the architectural assumptions were initially wrong or invalid, fixing them can be very costly. This is because this process enforces modification of the architecture, design, and implementation as well as changing the test cases and repeating the testing activities. The cost of this in comparison with the cost of a fixing process as a result of wrong implementation assumptions is very high.

The second set of problems is harsh drawbacks as a consequence of wrong architectural assumptions that are not caught during implementation and quality tests. Costly and deadly system failures such as Ariane 5 [3], tragic loss of a baby due to malfunction of an airbag system [3], CERN accelerator [8], sinking of HMS Sheffield [8], London Ambulance System [8] are the real examples of those drawbacks. In these examples the lack of documentation of assumptions is proved to be often the main cause of system failure or poor performance.

The third kind of problem, so called architectural mismatch, that rise due to lack of documentation of architectural assumptions is comprehensively studied by Garlan et al. [10]. In a study they identified the lessons learned from reusing Commercial Off-The-Shelf (COTS) components in a project. The result proved the severe consequence of unrecorded architectural assumptions in reusing COTS.

Last but not least, software evolution is not an exception to this. It is natural that assumptions of a system are subject to change during the evolution of software. In other words, modification activities during the process of software evolution can invalidate one or more of the architectural assumptions [8]. Hence, assumptions that are not documented most likely are not visible throughout the evolution process. This often causes in wrong design decisions, consequently project failures.

3.3 Aims and Objectives

This thesis work focuses on solving the aforementioned problems by introducing an explication method for assumptions in an architecture description language. The aim of this thesis is to

> Develop an assumption modeling method using Architectural Analysis and Description Language (AADL) for specification of architectural assumption.

where

- *modeling method* is the technique to explicate specification of architectural assumptions
- *architectural assumptions* are the assumptions that are made during the architecture of software by architects
- *assumption specification* is a detailed and explicit set of assumption characteristics including
  - *Assumption description*
  - *Assumption properties*
  - *Assumption dependencies*

Explicit assumptions together with their identified dependencies can bring visibility to assumptions [6]. This can be better done through documentation of their specification together with architecture descriptions of software. Visibility of architectural assumptions has two main benefits for software architects. It enables architects [6]

1. To track back the assumptions from architectures to implementations and vice versa (traceability)
2. To externalize the knowledge of architectural assumptions throughout organizations for further analysis purposes. (assumptions awareness)

Dependency of assumptions is the key in this work that is to help software architects to analyze the relationships of assumptions with components in their environment. It also can aid them to analyze

impacts of changes in assumptions by tracking the components and other assumptions that are likely to be affected by those.

The main research question that should be answered in this thesis work is:

"Whether we can model assumptions and their dependencies while designing architectures of software using the AADL?"

For achieving the aim of this project a set of objectives with respective methods are identified. The first goal is to investigate previous work on assumption specification and modeling techniques. This is done by conducting a review of existing work and preparation of a literature analysis report.

The second objective is to define a system that can be used as a test example. The purpose of this is both to explain the modeling method of the study and to evaluate whether the developed method addresses the target problems. In order to define the case example some system architectures in the AADL and some examples given by the literature and previous work are to be reviewed and analyzed.

The third objective is to develop a modeling method for explicating the architectural assumptions in the AADL. For doing so, after conducting the literature review, an assumption specification meta-model is to be defined together with a specification approach.

The last objective is to evaluate the example model by applying the developed modeling method. This is to be done by initially specifying evaluation criteria and analyzing the result of the evaluation of the method against those criteria. The aim of this thesis project will be achieved by reaching this objective.

Chapter 4

Assumption Specification

It is known by specialists that design and architecture specification is vital to a successful implementation. When it comes to designing critical systems attempts to specify the architecture description of software, hardware, and bindings have resulted in several architecture description languages such as Architecture Analysis and Design Language (AADL) [12], Architecture Information Modeling language [17], Embedded Architecture Description Language [18], Ptolemy [19], etc.

For software makers, design decisions are the keys to develop a system that truly satisfies the expected needs. In a software development process, design decisions are important in several aspects. Firstly, design decisions specify the system boundaries. From a developer point of view it is important to know what are the design constraints, dependencies, and assumptions of the system. As long as the decisions made about the overall architecture of the system, design patterns, development methods, data and control flows, component reuse, statics and dynamics of the system, environmental factors such as hardware and resources constraints are not fully and clearly specified developers would be making on the fly decisions during the implementation.

The consequences of lack of specifying the architecture and design decisions are often very harsh. The problems raised due to this lack are seen repeatedly in several critical systems [3], [5], [10], [14], [15], and [22]. Intensive development processes in these examples engaged several teams implementing different components of the systems in parallel. A common reason in the failure of all of these systems was the incomplete description of design decisions.

Secondly, specification of design decisions has other benefits from an architect's point of view. Firstly, it aids software architects to realize knowledge transparency throughout the organization. This enables them to transfer information about the key decisions of the system to the developers and managers. Secondly, it helps them to be able to trace back the problems raised due to wrong decisions made. Hence, it makes the revising of the architecture less costly and much easier. On the other hand, the changes that are made in the design can be traced forward to the implementation. Therefore, the design modifications that occur as a result of design defects, requirement changes, or an evolution process, would not have a destructive impact on the process of development as well as the product itself.

Architectural assumptions are part of the design decisions that are made on the fly during the architecture phase. Architects often make assumptions about the environment, COTS components, controlling components, and data components. An architect most probably makes additional assumptions to support for future development and reuse purposes of the components of a system. The fact that assumptions are important to be specified during the architecture design cannot be neglected.

4.1 Definitions

For us in order to reach our aim, it is important to describe the domain of our study.

As it is mentioned before, an assumption is a fact to be taken as true by software engineers. In this study, we only consider the category of technical assumptions. Among the technical assumptions, the focus of our work is on architectural assumptions.

An architectural assumption is the assumption made during the architecture of a system by a software architect. It is made on the fly to calibrate the architecture and used to impose constraints over the environment, or other components.

From here on, *assumption* denotes architectural assumption throughout this text.

An assumption can be valid or invalid. A valid assumption is the one that it based on a true fact. On the other hand, an invalid or wrong assumption is the one that is not factual. In other words, an invalid assumption is not based on a true fact.

There are two types of invalid assumptions. First, the assumptions which are made based on false facts from the beginning. Second, the assumptions which are invalidated later during the software development process. The second type of assumptions can happen due to a change in parts of the system, the software or the hardware. This change falsifies the fact that the assumptions is based on.

For specification of assumptions, the information about them should be described. In this study, we do this by explaining an approach which uses the SAE AADL standard.

Chapter 5

A Case Example

We explain our studied assumption modeling approaches using a case example. The example that we explain here is an AADL specification of a system. We use the SAE AADL standard textual and graphical representations to illustrate the system specification [20].

Before introducing the example, a brief introduction of the SAE AADL standard is given in the following chapter.

5.1 AADL Introduction

The AADL is an architecture description language for modeling and analysis of system architects [12]. It supports repeated analyses of system architects by specifying both software and hardware components of complex systems. Therefore, it is effectively used for model-based analysis and specification of complex embedded and real-time systems [12].

The SAE AADL standard contains several types of components for specification of embedded system architectures [12]. These components types are categorized into three levels: 1. application software components, 2. execution platform components, 3. composite components.

Application software component are:

- *Thread*
- *Thread group*
- *Process*
- *Data*
- *Subprogram*

Execution platform components (so called hardware components) in the AADL are:

- *Processor*
- *Memory*
- *Device*
- *Bus*

Composite components are the components whose responsibility is to define the composition of other system component. The only composite component in the SAE AADL standard is *system*.

The core language and key specification concepts are summarized in the following Figure 5-1:

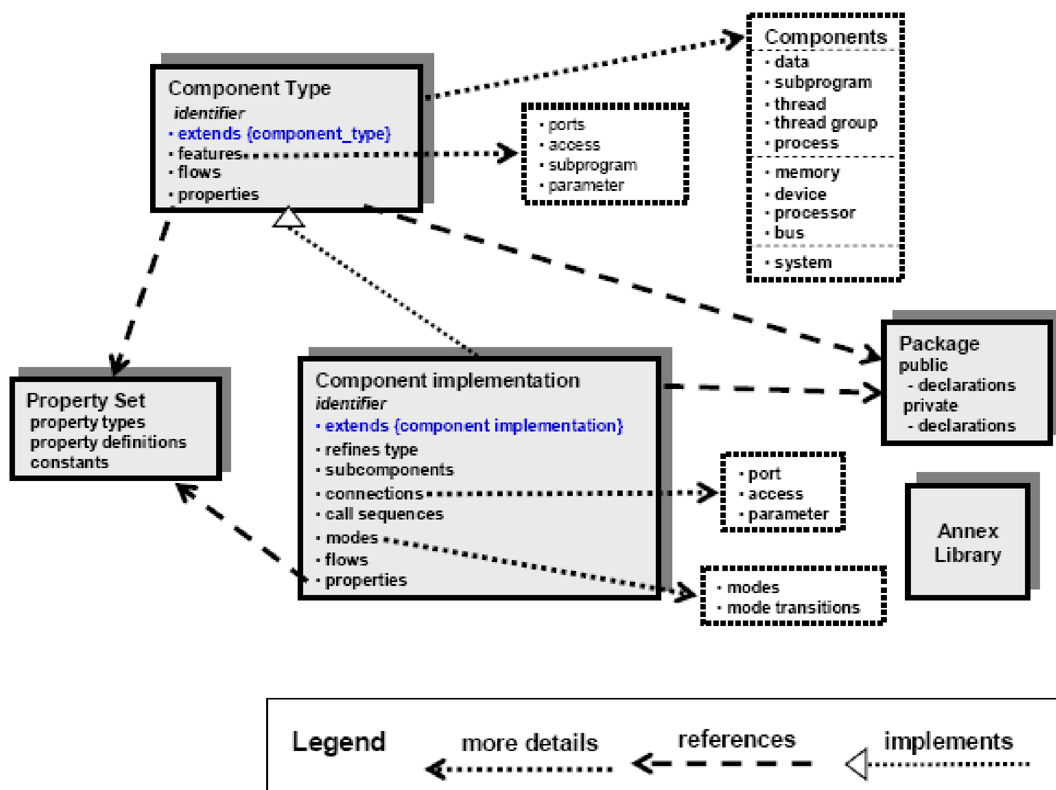

Figure 5-1.The SAE AADL Standard Elements Summery [12]

As it is shown in Figure 5-1, the different abstraction levels of components in the SAE AADL standard are realized through component type and component implementation. Component type declares a definition of a certain component while component implementation implements a certain component type (component types such as thread, processor, system, etc.). The flexibility of the SAE AADL standard enables architects to declare several implementation of the same component type. For example, a *processor Intel* type can have the two distinctive implementations *processor implementation Intel.P64bits* and *processor implementation Intel.P32bits* [12].

In the SAE AADL standard component interfaces with the other components are declared through *feature* section in the component type. Components communication can be specified using different ports such as data and event ports together with port directions. For example, as you see in Figure 5-2 the sensor device defines its interface through data port:

```
device Sensor
    features
        output_data: out data port Sensor_data;
end Sensor;

data Sensor_data
end Sensor_data;
```

Figure 5-2.Sensor Interface Texual Representation

Another key specification concept of AADL is the definition of component properties. For instance, one can specify the period value of a thread component using its *Period* property (see Figure 5-3). A list of standard pre-declared properties are packaged in the *AADL_Properties* that is part of every SAE AADL standard specification [12]. Figure 5-3 shows the use of pre-declared AADL properties for a thread component:

```
thread control
    properties
        -- nominal execution properties
        Compute_Entrypoint => "control_ep";
        Compute_Execution_Time => 5 ms .. 10 ms;
        Compute_Deadline => 20 ms;
        Dispatch_Protocol => Periodic;
        -- initialization execution properties
        Initialize_Entrypoint => "init_control";
        Initialize_Execution_Time => 2 ms .. 5 ms;
        Initialize_Deadline => 10 ms;
        Period => 50 ms;
end control;
```

Figure 5-3.Thread Component Pre-Declared Properties

In addition, the SAE AADL supports introduction of new sets of properties. By doing so, an architect can define new domain specific property sets containing a set of new properties to better specify the system of interest. In our work an extension to this concept that supports property set reuse is proposed in Chapter 6. This concept is used later on in our first approach for assumptions specification (see Chapter 7.2)

Furthermore, the SAE AADL standard provides two representation forms for architecture specification: textual representation and graphical representation. In the textual representation, texts are used to specify, define, and declare the components and the interactions. In the graphical representation, graphical notations are used for describing the system architecture.

For description of the architecture in the textual form, several reserved words are used. The Table 5-1 shows some of the reserved words and their purposes that we have used in our case example in this chapter:

Reserved Word	Usage	Sample
system	Defines a system component type for composing other components.	*system* Controller . . . *end* Controller;
system implementation	Declare a system implementation	*system implementation* Controller.USB . . . *end* Controller.USB;
property set … is	Define a new property set	*property set* NewPropertySet *is* . . . *end* NewPropertySet;

Table 5-1.The SAE AADL Reserved Words

Moreover, the graphical notations for specification of the architecture using the SAE AADL standard are summed up in Figure 5-4 [12]:

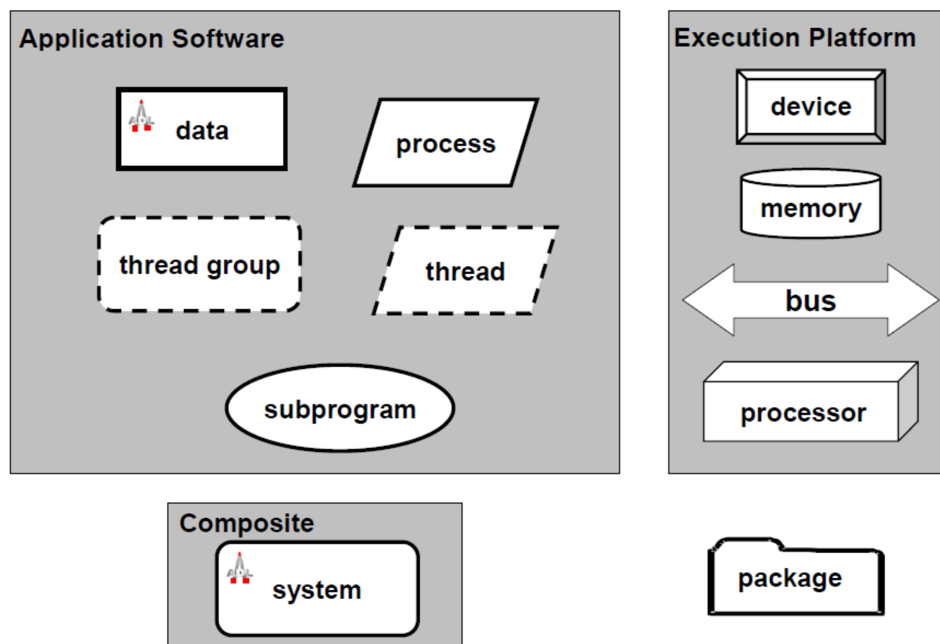

Figure 5-4.Summary of AADL elements in graphical representation [12]

Our assumptions specification approach in this study is based on the concepts of the SAE AADL standard. Therefore, we use both textual and graphical representations to illustrate our case example as well as our approach. For a detailed introduction of the notations and the language standard see [12].

For extending the language, the SAE AADL standard supports definition of new sublanguages. The standard supports this by a concept called annex. Annex is a place to specify new sub-languages for

specific purposes [12]. They are often used to support custom analyses using special modeling approaches [12].

In the SAE AADL standard, the first step is to define an annex library inside a package declaration. The next step to this concept is to use the defined annex library in the architecture specification by containing the annex sub-clauses in the component type or implementation declarations. The error model annex that can be used for dependability analysis is a good example [21]. Figure 5-5 and Figure 5-6 demonstrate an error model and its annex sub-clause:

```
package My_ErrorModels
public
annex Error_Model {**
error model Example1
…
end Example1;

error model implementation Example1.basic
…
end Example1.basic;

error model Example2
…
end Example2;

error model implementation Example1.basic
…
end Example2.basic;
**};
end My_ErrorModels;
```

Figure 5-5.Example Error Model Annex Library Declaration [21]

```
system computer
end computer;

system implementation computer.personal
subcomponents
  CPU: processor Intel.DualCore;
  RAM: memory SDRAM;
  FSB: bus FrontSideBus;
annex Error_Model {**
      Model => My_ErrorModels::Example1.basic applies to CPU;
   Occurrence => fixed 0.9 applies to error CPU.CorruptedData;
**};
end computer.personal;
```

Figure 5-6.Example Error Model Annex Sub-Clause Declaration [21]

This concept is later on used in our second approach for assumptions specification (see the Chapter 7.3).

5.2 Case Example: Video Streaming System

Video Streaming System (VSS) is our example for explaining and testing our studied approach. It is an embedded real-time system for streaming a video from a sender unit to a receiver unit. The sender unit is responsible for accessing the video data stored on a video file, encoding it to a specific format, packetizing, and sending the packets to the receiver through a bus. On the other side, the receiver unit receives the data packets from the bus and decodes them to create the original video.

The AADL component hierarchy of VSS is illustrated in Figure 5-7:

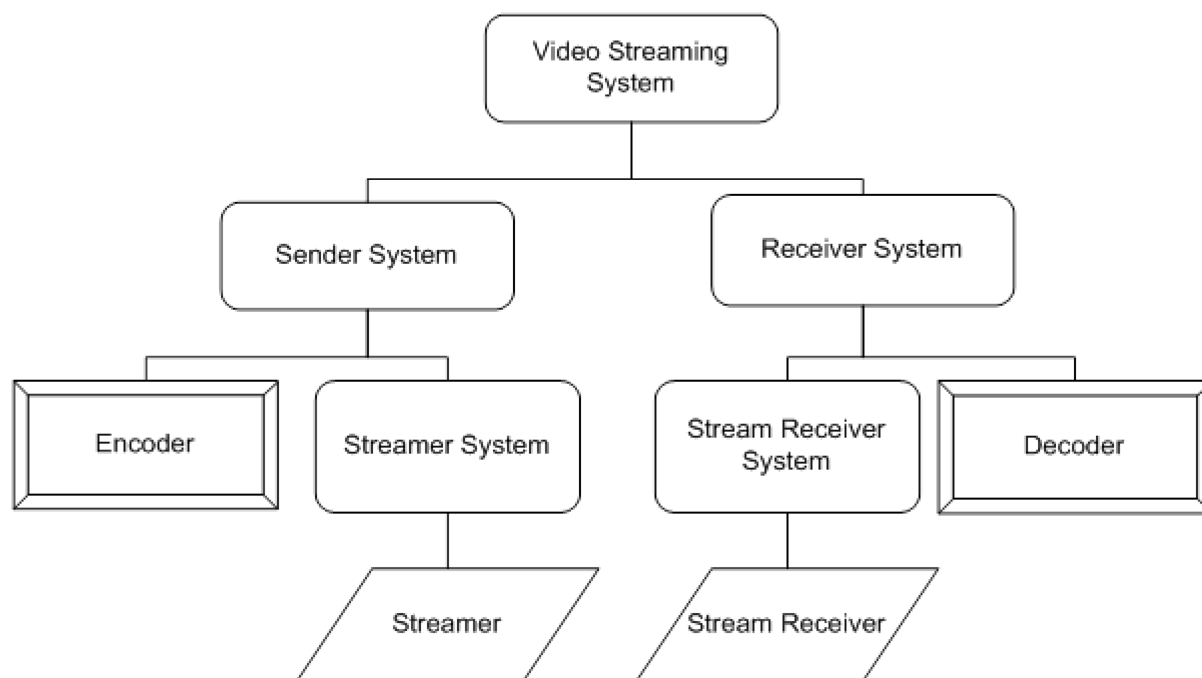

Figure 5-7.Video Streaming System Component Hierarchy

As it is shown in Figure 5-7, the VSS contains two high-level subsystems: sender system and receiver system. The sender system is a software/hardware combination containing an encoder device and a streamer subsystem. The encoder is the COTS device which supports different video encoding protocols. The streamer system also contains the streamer process which interacts with the encoder. In its interaction with the encoder, the streamer process requests the encoder to start encoding the video, and then receives the encoded data, subsequently packetizes the data, and sends the data packets over the bus to the receiver system. Sending of data packets happens upon a request from the receiver system.

The receiver system is also a software/hardware combination containing a de-streamer (stream receiver) subsystem and a decoder device. The de-streamer contains the de-streamer process which collects the data packets from the bus sent from the sender system. This process utilizes event notification for communication between the sender and receiver systems. Later, the de-streamer

system hands in the merged video data to the decoder for decoding. The decoder is a COTS device as well. It supports several video decoding protocols although it does not necessarily support all the encoding protocols that the encoder can perform.

In Figure 5-8 a schematic view of the system is shown:

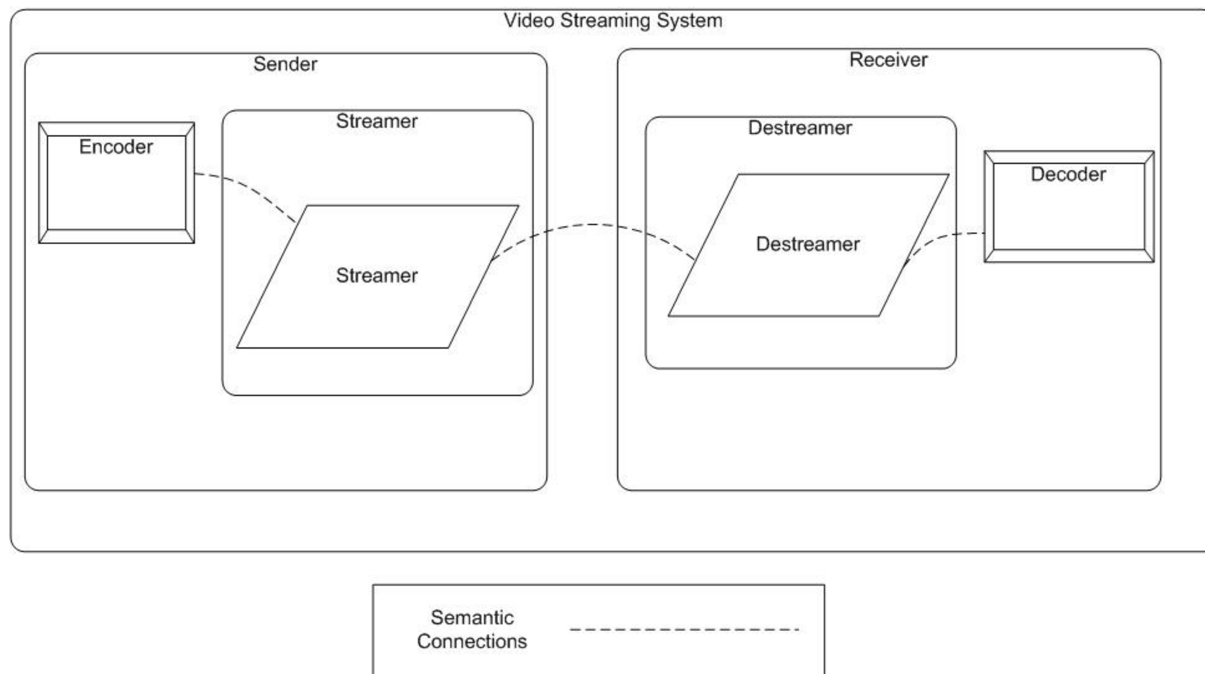

Figure 5-8.Video Streaming System Schematic

Figure 5-8 illustrates the component containment of the VSS. In this schematic it is shown that the streamer process and encoder device have semantic connection. This means that the streamer process and encoder communicate with each other. This communication is later on realized by defining port connections in the components' interfaces. Similarly, the de-streamer process and decoder device have semantic connection with each other which means that they also need to declare ports for their communications. Additionally, the semantic connection between the streamer process and de-streamer process implies that these subsystems need to communicate by providing port connections that they declare in their component specification as well as in their container systems.

Here in Figure 5-9 and Figure 5-10 partial details of the VSS architecture specification in AADL textual and graphical form that suffice for elaboration of our approaches are shown:

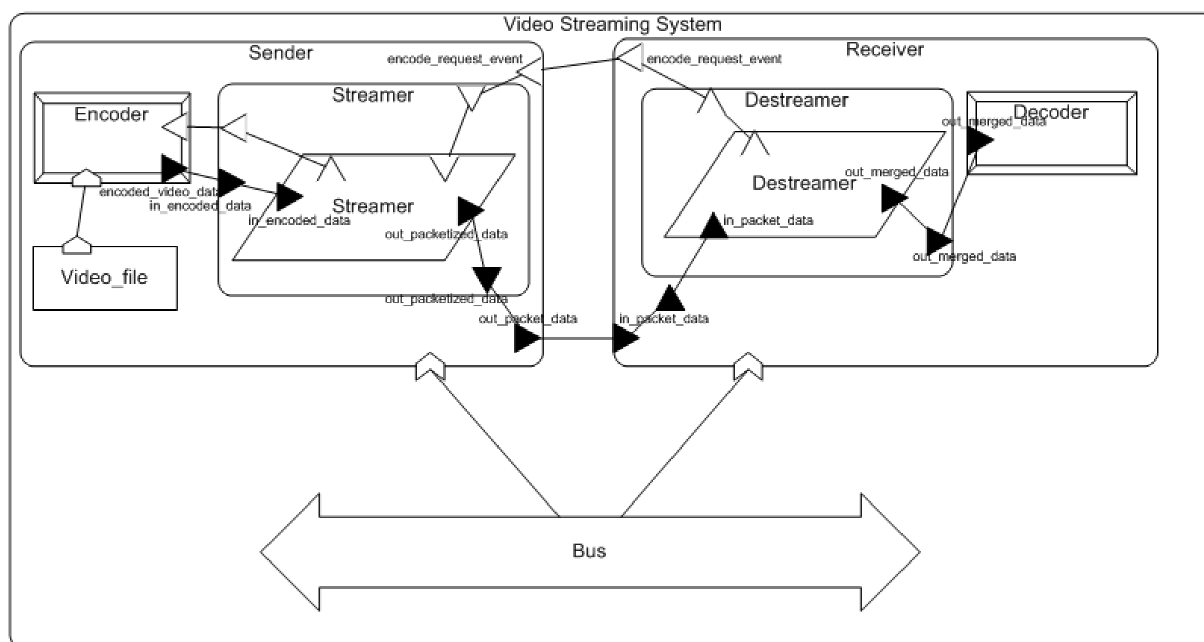

Figure 5-9.Video Stream System Partial Detail Architecture

In Figure 5-9, specification of the system is illustrated using the SAE AADL standard graphical notations. This figure shows the realization of the semantic connections among the VSS components. The components of this system declare specific ports for their communications. In other words, the components define their interfaces using in/out ports and port to port connections for establishing intended communications. Besides, both sender and receiver systems need to access a wired bus for their communication. The data between these two systems is transferred using this bus.

Moreover, the encoder device requires access to the data from a video file in the VSS. This is also shown in Figure 5-9 through a data access connection between the encoder device and the video file data component.

The system specification of our case example explained in this chapter is partial and only suffices for the purpose of describing our approaches. A detailed architecture description of the VSS both in the AADL textual and graphical representations is given in the Appendix 1 for better understanding the components' interactions and behavior of the system.

```
system implementation VideoStreamingSystem.Impl
  subcomponents
    sender_sys: system Sender.Impl;
    receiver_sys: system Receiver.Impl;
    conn_bus: bus WiredBus.Impl;
  ...
end VideoStreamingSystem.Impl;

system Sender
end Sender;

system implementation Sender.Impl
  subcomponents
    streamer_sys: system Streamer.Impl;
    encoder_dev: device Encoder.COTS;
end Sender.Impl;

system Receiver
end Receiver;

system implementation Receiver.Impl
  subcomponents
    destreamer_sys: system Destreamer.Impl;
    decoder_dev: device Decoder.COTS;
end Receiver.Impl;

system Streamer
end Streamer;

system Destreamer
end Destreamer;

device Encoder
end Encoder;
```

Figure 5-10.Video Streaming System Specification in AADL text from

5.2.1 Video Encoding Assumption

One of the architectural assumptions in this case example is about encoding protocol of the video streaming. In the architecture of the system we assume that

The video encoding protocol is MPEG-3.

In the above statement, video encoding protocol is the one by which the video data is encoded to be transferred to the receiver system. Respectively, MPEG-3 is an encoding protocol designed to handle HDTV signals in a high bandwidth to support quality video and audio [23].

It is arguable whether this is an assumption or a requirement. In our case example, we consider this as an assumption. We have two reasons for this. First of all, we have considered that it is not mentioned

as a requirement in the analysis of the VSS. This is a decision that is left to the system architect. Secondly, we need to make a decision in our architecture about the way that we want the video to be streamed over the bus to the receiver. For this purpose, we have considered several on-the-fly facts when we decided to use MPEG-3. It is taken into account that the system should be maintainable, implementable with a low cost, and perform based on a well-known standard. Therefore, we counted on the facts that were rooted from analysis of the system as well as architectural experiences. Hence, we conclude that the decision made about the encoding protocol stands on the on-the-fly facts that have contributed to this assumption.

This assumption imposes some constraints on the system. First of all, the sender system should send its video data encoded in MPEG-3. This means that the COTS encoder should support encoding to this standard. Additionally, this assumption indicates that the receiver system has to be ready to receive video data in MPEG-3 format. This also means that the COTS decoder that is contained in the receiver system must support this file format. These are the invariabilities that are connected to the mentioned assumption. These invariabilities are addressed by the explanation of our assumption modeling method (see Chapter 7).

Chapter 6

Reusable Property Sets In AADL

6.1 Background

Properties are essential in architecture specification using the SAE AADL standard. In the SAE AADL standard it is not possible to describe a system without specifying the properties of its components. For example, if the execution period of a thread component is not specified, the architecture description is not complete. As another example, the event port which its queuing or de-queuing protocol is not specified most likely rises several problems when it comes to its implementation.

The SAE AADL standard includes a list of standard pre-declared properties which are packaged in *AADL_Properties*. These pre-declared properties are part of every specification in the SAE AADL standard. The properties such as thread period, thread deadline, port and connection protocols, etc. are all declared for further usage. Even if these properties are not explicitly assigned in any component declaration, they are automatically assigned to their default values in the architecture specification for each component.

The SAE AADL standard provides additional support for declaration of new properties. This is done by defining a new property set and declaring new properties, property types, and constants in that new property set. Figure 6-1 shows the definition of a new property set using the AADL in its textual representation:

```
property set SubSystem_PropertySet is
    MetricsSystem: enumeration (InternationalSystem, ImperialSystem)
    applies to (all);
    .
    .
    .
end SubSystem_PropertySet;
```

Figure 6-1.Sample of New Property Set

Figure 6-1 shows the declaration of a new property set called *SubSystem_PropertySet* which is intended to define new properties in an architecture specification using AADL. As it is, a new property named *MetricsSystem* is declared that can be applied to all components in the system. The type of this property is an enumeration of two values: *InternationalSystem* and *ImperialSystem.* It can be seen in

Figure 6-2 that by assigning this property in a system to one of the enumeration values a completely new property is added to a component definition for a specific purpose:

```
property set SubSystem_PropertySet is
    MetricsSystem: enumeration (InternationalSystem, ImperialSystem)
    applies to (all);
    .
    .
    .
end SubSystem_PropertySet;


system Calculator
end Calculator;

system implementation Calculator.Impl
    subcomponents
        calc_unit: processor PIC;
        .
        .
        .

    properties
        SubSystem_PropertySet::MetricsSystem => InternationalSystem;
end Calculator.Impl;
```

Figure 6-2.Sample of New Property Set Use

Another important aspect of defining new property sets in the SAE AADL standard is the definition of new property types. The AADL supports the definition of new property types by associating an identifier with it and establishing a set of legal values for a property of that type. Figure 6-3 extends the previous example to include declaration and use of a new property type:

```
property set SubSystem_PropertySet is
    MetricsSystemType: type enumeration (InternationalSystem, ImperialSystem);
    MetricsSystem: MetricsSystemType
    applies to (all);
    .
    .
    .
end SubSystem_PropertySet;
```

Figure 6-3.Sample of New Property Type

The declaration shown in Figure 6-3 illustrates the definition of a new property type named *MetricsSystemType* which later specifies the type of a property named *MetricsSystem* that can only be assigned to two metric system values.

According to the SAE AADL standard, a property type can be an AADL built-in type[1], a new property type explicitly defined within the declaration, or a reference to a previously defined property type. Figure 6-4 shows the definitions for these three different property types:

```
property set ReferencePropertySet is
    SystemSize: type aadlinteger;
end ReferencePropertySet;

property set SubSystem_PropertySet is
    RevisionNo: aadlinteger applies to(all);
    MaxSystemSize: ReferencePropertySet::SystemSize applies to(system);
    MetricsSystemType: type enumeration
                         (InternationalSystem, ImperialSystem);
    MetricsSystem: MetricsSystemType
                         applies to (all);
    .
    .
    .
end SubSystem_PropertySet;
```

Figure 6-4.Sample of Different Property Types

It is shown in Figure 6-4 that the type of *RevisionNo* property is *aadlinteger* which is an AADL built-in type. It is also shown that the type of *MetricsSystem* property is *MetricsSystemType* which is an explicitly declared type. Finally, the figure shows the reference to a property type. It is done by defining the *SystemSize* property type in the first property set and referring to that type by the declaration of the *MaxSystemSize* property in the second property set.

The SAE AADL standard supports the definition of constants as well. A constant is a property that its value is initially specified and cannot be assigned or changed in any component specification. Figure 6-5 shows the declaration of a constant property:

```
property set SubSystem_PropertySet is
    MaxSystemSize: constant aadlinteger => 500;
    .
    .
    .
end SubSystem_PropertySet;
```

Figure 6-5.Sample of Constant Property

For more information about the standard specification see [12] and [20].

[1] The SAE AADL standard contains built-in property types such as *aadlstring*, *aadlboolean*, *aadlinteger*, etc. [20]

6.2 Motivation

The SAE AADL standard provides a simple support for complex properties. This is done through three different property declarations:

- Reference value
- List of values
- Value range

Reference value is the method for assigning a property value by a reference to a component in AADL. Consider partial graphical representation of an avionics system shown in Figure 6-6 [12]:

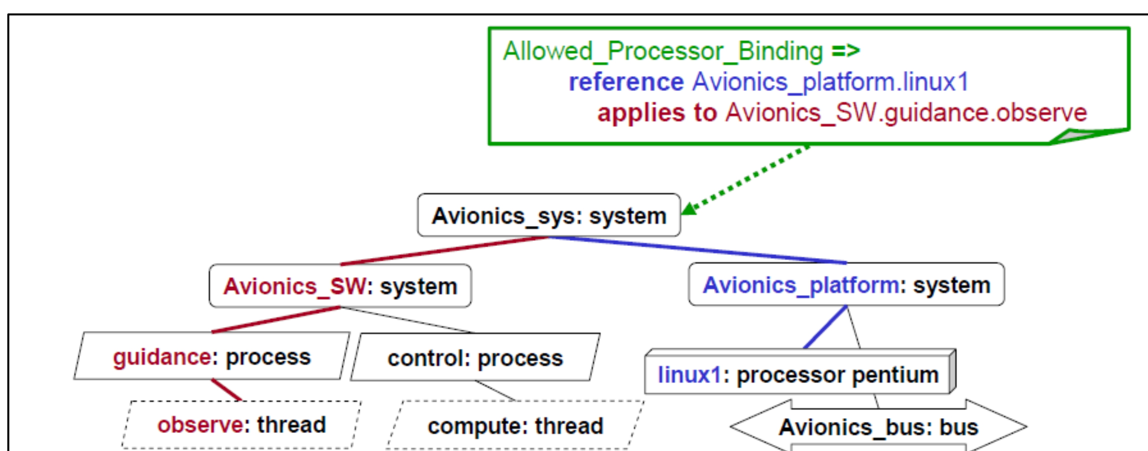

Figure 6-6.Sample of Reference Component as a Property Value [12]

In the adopted Figure 6-6 [12], it is expressed in the specification of an avionics system that the *observe* thread can be bound to the processor *linux1*. This is done by assigning the contained property *Allowed_Processor_Binding* to the reference value of the *linux1* processor which is applied to the *thread* process. Similar properties can be defined in a new property set which can be assigned to a reference value [12].

The second property declaration method is a list of values. Using this declaration, a property can be assigned with more than one value of the same property type. This means that a property can hold a list of values of the same property type. Figure 5-7 is an example of this method:

```
property set Subsystem_PropertySet is
    Revisioners: list of aadlstring applies to (all);
end SubSystem_PropertySet;

system Calculator
    .
    .
    .
    properties
        Subsystem_PropertySet::Revisioners => ("Kin", "Bill", "Kate");
end;
```

Figure 6-7.Sample of Lists Property Value

The third way to create complex properties in the SAE AADL standard is to use range values. This method gives the ability to set a range of values to a property. It also enables the creation of new properties that can accept a range of values, i.e. time range.

However, the support for complex properties in AADL is very limited. The main limitation is on reusing explicitly declared property sets. The SAE AADL standard limits the use of property sets to components specification whilst it does not support reuse of property sets in the definition of other property sets. In the situations where one requires reusing an explicitly defined property set in the declaration of a new property set, the only way to do so is to re-specify the properties and property types of the previously declared property set in the new property set.

For a simple example two property sets declarations are defined in Figure 6-8:

```
property set RevisionPropsSet is
    RevisionNumber: aadlstring applies to (system);
    Revisioner: aadlstring applies to (system);
end RevisionPropsSet;

property set DoubleRevisionPropsSet is
    RevisionNumber: aadlstring applies to (system);
    Revisioner: aadlstring applies to (system);
    ReRevisioner: aadlstring applies to (system);
end DoubleRevisionPropsSet;
```

Figure 6-8.Property Sets Limitation Example in the SAE AADL Standard

The declarations in Figure 6-8 show two new property sets. The first one is *RevisionPropsSet* which can be used in the system components specification. The properties of this set can be assigned with the revision number of the component (*RevisionNumber*) and the person name who lately revisioned the component specification (*Revisioner*). Both of these properties accept values of the type string (*aadlstring*).

The second property set declaration shown in Figure 6-8 is *DoubleRevisionPropsSet*. The properties of this set can be assigned with the revision number of a system component as well as the name of the person who revisioned the system specification lately just like in the first property set. Additionally, this property set defines a new property for the name of the second person who revisioned the component specification (*ReRevisioner*).

There are a number of problems that can rise due to this limitation. The main problem appears when one needs to modify a common property or property type. A common property is the property declaration which is repeated in a number of property sets for the same specific purpose. Similarly, common property type is the property type that its declaration is repeated in several property sets for common specific purposes. In these situations when a common property or property type should be modified, the one and only way to do so in the SAE AADL standard is to modify all the places that the commonality exist.

To be more specific, in our example illustrated in Figure 6-8 if we decide to change the type of the common property *RevisionNumber* to *aadlinteger*, there is no other way than change the type in both places, inside the *RevisionPropsSet* and *DoubleRevisionPropsSet*. Besides, if we decide to add a common property type, let's say *RevisionRangeType* (shown in the following figure) and change the type of *RevisionNumber* property to that, we need to repeatedly modify several places. The changes in the declaration are shown in Figure 6-9:

```
property set RevisionPropsSet is
    RevisionRangeType: type range of aadlreal 0.0 ..100.0;
    RevisionNumber: RevisionRangeType applies to (system);
    Revisioner: aadlstring applies to (system);
end RevisionPropsSet;

property set DoubleRevisionPropsSet is
    RevisionRangeType: type range of aadlreal 0.0 ..100.0;
    RevisionNumber: RevisionRangeType applies to (system);
    Revisioner: aadlstring applies to (system);
    ReRevisioner: aadlstring applies to (system);
end DoubleRevisionPropsSet;
```

Figure 6-9.Common Property Modification Example in the SAE AADL Standard

Last but not least, another problem that rises is that when the architecture specification expands and the number of components as well as the number of property sets increase, it becomes much harder to maintain the commonalities. It becomes an error-prone task to re-declare and modify all the common properties and property types with every modification. The drawbacks of the modifications are mistaken property names, regressive modifications, etc.

6.3 Proposal

Our proposal is to break the stated limitation by extending the SAE AADL standard to support reuse of property sets. In this proposal we suggest to improve the language concepts by the following approach.

Our approach is to support reusability of property sets. In our approach this is enabled by adding two new concepts to the AADL which allows the property sets to be inherited or used as property types. The first concept for enabling inheritance in property sets is illustrated in the hierarchy diagram in Figure 6-10 and AADL textual declarations in Figure 6-11:

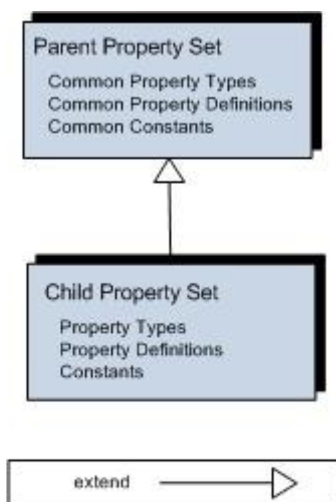

Figure 6-10.Property Set Inheritance Diagram

```
-- Parent property set declaration
property set Parent is
    common_property: property_type applies to
                        (component_level);
    .
    .
    .
end Parent;

-- Child property set declaration
property set Child extends Parent is
    property_name: property_type applies to (component_level);
    .
    .
    .
end Child;

-- Property use in component specifications
Child::property_name => …
Child::common_property => …
```

Figure 6-11.Property Set Inheritance Declaration

The declaration in Figure 6-11 shows the *parent* property set and *child* property set. The child property set inherits from the parent. In our declaration we used *extends* keyword which is part of our proposal to enable the inheritance declaration. Using this concept one can bundle the common properties, property types, and constants in a parent property set. Therefore, a child property set using *extends* keyword can inherit all the definitions from the parent property set. Consequently, any changes of the definitions in the parent property set are automatically inherited by all of the child property sets.

The second concept in our proposal is to support reuse of a property set through enabling definition of a property set as a property type, similar to the dependency concept in object-oriented programming paradigms. Figure 6-12 shows the proposal declaration of this concept:

```
-- SetA property set declaration
property set SetA is
    common_property: property_type applies to (component_level);
    .
    .
    .
end SetA;

-- SetB property set declaration
property set SetB is
    propertytype_of_propertyset: type Parent applies to (component_level);
    property_of_propertyset: Parent applies to (component_level);
    listproperty_of_propertyset: list of Parent applies to (component_level);

    .
    .
    .
end SetB;

-- Property set use
SetB::property_of_propertyset::common_property => ...;
SetB::listproperty_of_propertyset => ( {common_property => ..., ... } ,
                                        {...},
                                         ...);
or
SetB::listproperty_of_propertyset +=>  {common_property => ..., };
```

Figure 6-12.Property Set Dependency Declaration

Two property sets are declared in the declaration shown in Figure 6-12. The first property set is *SetA* which contains the definition of the reusable properties or constants. The second property set is *SetB* which contains its own properties as well as a dependent property (*property_of_propertyset*) that enables accessing the properties of *SetA*. This declaration shows the possibility of creation and use of

complex property sets. The use of these complex property sets is also demonstrated in Figure 6-12. The syntax to use the properties is similar to Object-Oriented languages.

By combination of these two concepts one can create complex property sets in the same time keep it simple to reuse and modify explicitly declared sets. Now by applying these concepts to the example in Figure 6-9, the following complex property sets can be declared:

```
-- Parent property set declaration
property set RevisionPropsSet is
    RevisionRangeType: type range of aadlinteger;
    RevisionNumber: RevisionRangeType applies to (system);
    Revisioner: aadlstring applies to (system);
end RevisionPropsSet;

-- Child property set declaration
property set DoubleRevisionPropsSet extends RevisionPropsSet is
    ReRevisioner: aadlstring applies to (system);
end DoubleRevisionPropsSet;

-- Property sets use
system Calculator
    properties
        DoubleRevisionPropsSet::RevisionNumber => 7;
        DoubleRevisionPropsSet::Revisioner => "Mike";
        DoubleRevisionPropsSet::ReRevisioner => "Kate";
end Calculator;
```

Figure 6-13.Property Set Inheritance Example

```
-- Property set
property set RevisionPropsSet is
    RevisionRangeType: type range of aadlinteger;
    RevisionNumber: RevisionRangeType applies to (system);
    Revisioner: aadlstring applies to (system);
end RevisionPropsSet;

-- Dependent property set
property set DoubleRevisionPropsSet is
    RevisionProps: RevisionPropsSet applies to (systems);
    ReRevisioner: aadlstring applies to (system);
end DoubleRevisionPropsSet;

-- Property sets use
system Calculator
    properties
        DoubleRevisionPropsSet::RevisionProps::RevisionNumber => 7;
        DoubleRevisionPropsSet::RevisionProps::Revisioner => "Mike";
        DoubleRevisionPropsSet::ReRevisioner => "Kate";
end Calculator;
```

Figure 6-14.Property Sets Dependency Example

The example in Figure 6-13 shows the use of property set inheritance concept. It demonstrates how one property set can be inherited by another property set using the *exnteds* keyword. This way the *DoubleRevisionPropsSet* inherits the properties, property types, and constants of its parent property set (*RevisionPropsSet*).

The second example in Figure 6-14 shows the declaration of property sets dependency concept of our proposal. The declaration in this figure illustrates how one property set can have access to the properties and constants of another property set. This way *DoubleRevisionPropsSet* reuses the other property set *RevisionPropsSet*.

6.4 Conclusion

We proposed an extension to the SAE AADL standard to enable reusability of property sets. We did this by identifying the problems that can rise due to the limitations existing in the current version of the AADL.

Our proposal included two methods for reusing property sets. First, property set inheritance through using *extends* keyword. Second, definition of a property set as a property type. We gave examples on these methods to explain and show the benefits of this extension.

Last but not least, our extension supports backward compatibility. This means that a complete specification of an architecture using AADL without the support for property set reuse is still valid in our extension. This is because in our proposal we use the same concepts of the SAE AADL standard

and only extend the approach by supporting reusability of property sets. There appears to be no conflict between our proposal and the current concepts of the AADL.

Chapter 7

Modeling Architectural Assumptions

For specification of assumptions, firstly it is necessary to structure the information about the assumptions. Secondly, a method is needed to suitably explicate the specification of the assumption information. In this thesis work, we studied different approaches for this purpose.

This chapter gives a description of our approach for modeling architectural assumptions. At first, the assumption specification meta-model is introduced concluded from a literature review. Second, we introduce two approaches to integrate this meta-model in the context of an AADL architecture specification. We do it together with using a case example. Later, we evaluate each of the approaches against the important criteria, which were initially identified in our study. At the end, we discuss around our selection and give our motivational arguments.

7.1 Assumption Specification Meta-Model

From a literature analysis, it is seen that there are several kinds and categories of assumptions identified by different researchers [3], [5], [10], and [13]. Also different studies have introduced different important information about the assumptions [2] and [6]. However, in our approach we introduce a meta-model that covers a wide range of assumption category. This meta-model enables us to adopt the approach to different assumption specification purposes.

The building blocks of the meta-model in our approach are as follow:

- Assumption name
- Assumption description
- Assumption custom attributes
- Assumption dependency
    - Impact components
    - Realize components
    - Dependency custom attributes

This meta-model contains four main concepts. The first concept is about enforcing the notion of name and description specification of assumptions. The name of assumption is a unique string that identifies an assumption. A description of an assumption is a verbal statement of the nature of the assumption. The specification of these two attributes creates the basic for assumption awareness in software architectures.

The second important concept is the possibility of adding custom attributes to the assumption specification meta-model. Custom attributes are specific and additional information of an assumption that their specification increases the assumption awareness in software architecture. This enables an architect to specify additional information together with an assumption. An architect can benefit from this to customize the assumption specification meta-model for their specific needs.

There are several examples of the benefits of utilizing this concept. For instance, the assumption type is the additional information that can enrich software architectures [13]. Type of assumptions is also important when it comes to architectural mismatches [10]. Hence, this can be specified as a custom attribute in the meta-model for assumption. For this purpose, we can add a custom attribute to our meta-model and call it *type*. The following list is the type categories that specify the assumption type in our model [13]:

- Control
- Environment
- Data

- Usage
- Convention

The third and most important concept in the meta-model is assumption dependency. Dependency of assumption is the key in our assumption specification meta-model. Thus, the meta-model contains a set of dependency attributes. These attributes specify both the components that are impacted by the assumption and the components that realize the assumption. The benefit of the dependency is the ability it gives to trace the assumptions and change impacts. Figure 7-1 illustrates this concept:

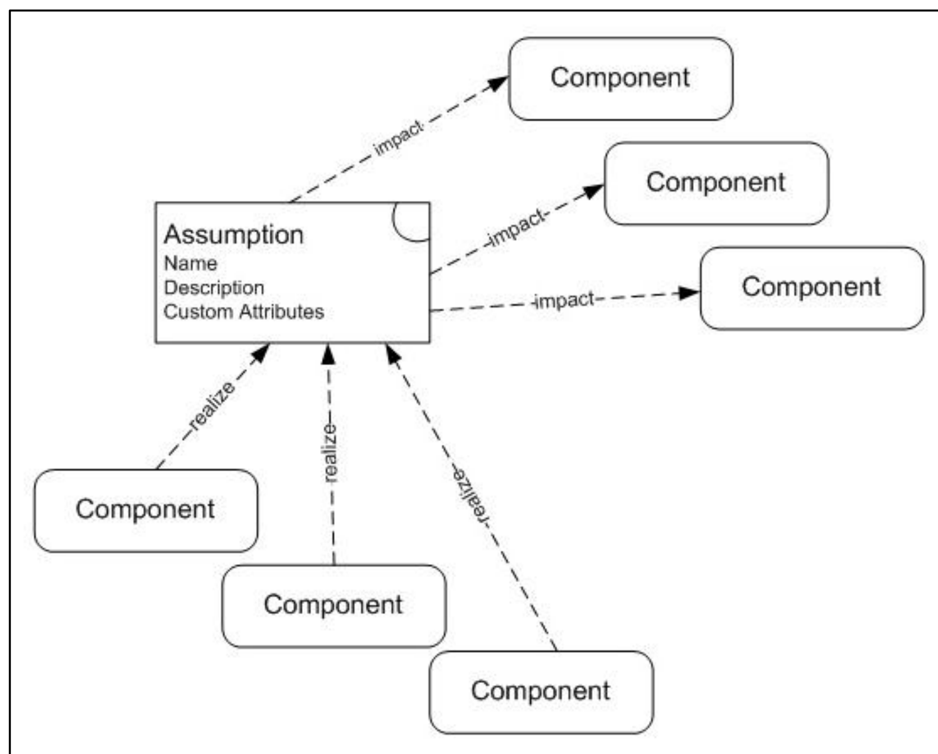

Figure 7-1.Assumption Meta-Model Dependency Concept Diagram

To support assumption dependency, the meta-model enforces the identification of *impact* dependencies as well as *realize* dependencies. The relations named *realize* in Figure 7-1 show the realize dependencies among the assumption and components while the relations named *impact* show the impact dependencies among them. In the meta-model impact dependencies identify the components that are likely to be impacted by any change in the assumption. On the other hand, the realize dependencies identify the components that are the reasons for the assumption; a change in these components is likely to be realized in the assumption. In our approach we assume that an assumption has to have at least one impact component.

Lastly, the meta-model supports specification of additional custom attributes for assumption dependency. Dependency custom attributes support the further development of our meta-model as

well as future works for dependency analysis methods. As an example, impact level of an assumption dependency is additional information that can be specified for assumption dependencies [6].

For specification of the meta-model in the context of a system architecture using AADL we have studied two approaches. The first approach uses property sets to specify an assumption specification into software architecture description. The second approach uses annexes for the same purpose.

7.2 Approach 1: Assumption Specification through Property Sets in AADL

In our first approach we introduce a new method for specification of the meta-model in the context of an AADL architecture description. In this approach we use the concept of property set and declaration of new property sets in the SAE AADL standard.

As it is explained in Chapter 5.1 properties are used to specify the characteristics of the components in AADL. Additionally, AADL permits the definition of new properties for domain specific purposes. This is supported through declaration of new property sets that contains the new properties, property types, and constants.

For specification of assumptions in this approach two steps are taken:

1. Defining assumptions specification meta-models by declaring a new property set for each meta-model
2. Specifying the defined assumption meta-model by assigning its attributes in the architecture description.

In the first step, declaration of a new property set is done to define an assumption specification meta-model. In this step the meta-mode is mapped to property set declaration. Figure 7-2 shows the declaration of assumption's property set:

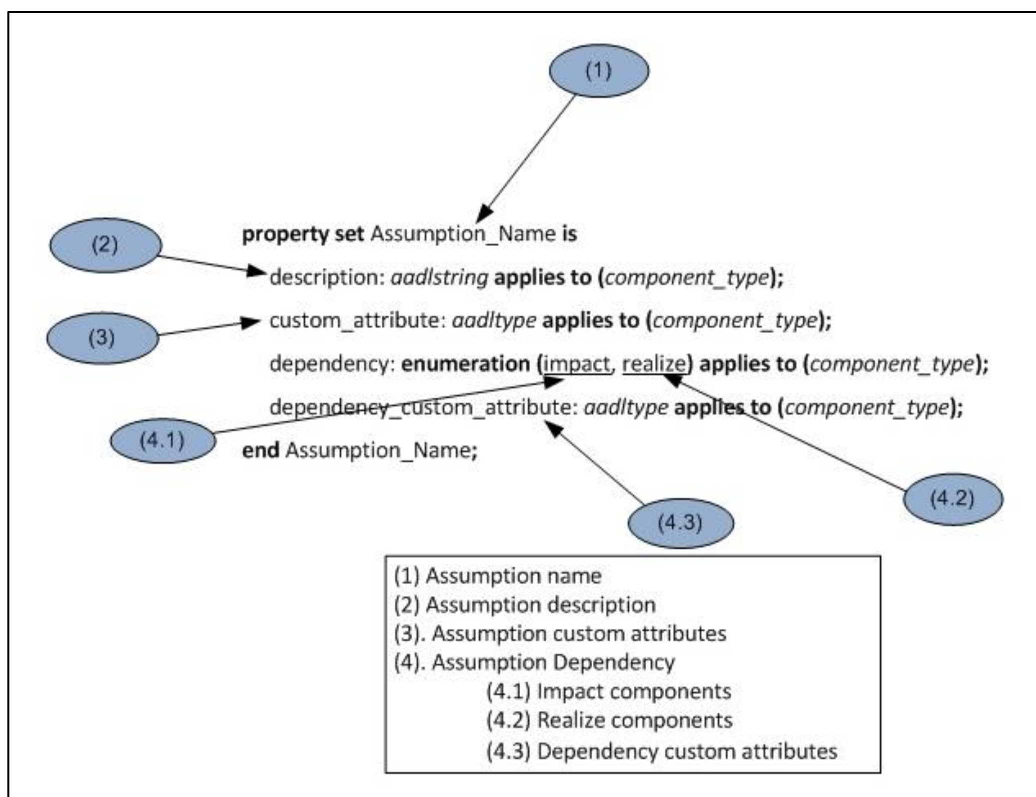

Figure 7-2.Assumption Specification Meta-Model in Property Set Declaration

Figure 7-2 illustrates the mapping of the building blocks of the assumption specification meta-model to the declaration of the assumption meta-model property set. The syntax used in this figure is according to the SAE AADL textual representation. The bold texts are reserved words, the italic texts are AADL built-in type or component classification and the normal texts are the identifiers.

The key part of the declaration represented in Figure 7-2 is the mapping of assumption dependencies. In this approach enumeration value property is used for defining the dependency of the type impact or realize. An example of this concept is given later in this chapter.

Moreover, all the meta-model property sets has to be included in a package called *Assumptions_Specification*. Packaging the property sets in the AADL enables us to centralize the notion of property meta-models which later on can be used for analysis of the architecture.

In the second step of this approach, the assumptions of the components in the system are specified by assigning the properties of the pre-declared assumption property set. This is done by simply using the declared property sets for assumptions and their properties, then assigning them with the assumption information according to the SAE AADL syntax.

For clarifying this approach, the video encoding assumption of the VSS case example is used (see Chapter 5.2.1). Let's recall the assumption:

Assumption: The video encoding protocol is MPEG-3.

In order to specify this assumption we need to elaborate its ingredients. The encoder device should support encoding to MPEG-3 format for streaming the video data from the sender to the receiver system (see Chapter 5.2.1). On the other hand, the decoder should support decoding MPEG-3 formats to create the original video data (see Chapter 5.2.1). Therefore, we can say that the encoder component fulfills the assumption by encoding to the assumed protocol. Consequently, the assumed fact has impact over the decoder to make sure it supports the assumed protocol. This appears to make sense because the encoder is the component that can invalidate the assumption. For example, if the encoder uses MPEG-4 to encode the video data, the stated assumption is invalidated. On the other hand, if the encoder still performs encoding to MPEG-3, the assumption is still valid; however, if the decoder performs decoding from MPEG-4 formats, the assumption is violated.

As the first step to specify this assumption using our approach we need to define a meta-model property set for it according to the syntax demonstrated in Figure 7-2. Figure 7-3 shows the declaration with respect to the aforementioned syntax:

```
property set VideoEncodingAssumption is
  description: aadlstring  applies to (all);
  dependency: enumeration (impact, realize)
              applies to (all);
end VideoEncodingAssumption;
```

Figure 7-3.Declaration of Property Set for Video Encoding Assumption

The declaration in Figure 7-3 shows the definition of the example video encoding assumption. In this declaration the meta-model of our example assumption, which contains its name, description, and dependency, is defined.

If we need to specify additional information for our example assumption we can make use of custom attributes in the meta-model. Suppose that one needs to specify the criticality of this assumption. This can be done by defining a property named *criticality* in the declaration of *VideoEncodingAssumption* property set. Figure 7-4 shows the modified property set for this assumption:

```
property set VideoEncodingAssumption is
  description: aadlstring  applies to (device);
  criticality: enumeration (Critical, Non_Critical)
               applies to (device);
  dependency: enumeration (impact, realize)
              applies to (device);
end VideoEncodingAssumption;
```

Figure 7-4.First Step - Declaration of the Assumption Property Set containing Criticality Custom Attribute

In the declaration shown in Figure 7-4 the criticality attribute is added to the property set for our assumption meta-model. The possible values for this attribute are *critical* or *Non-Critical* enumeration values which can be assigned in the second step of assumption specification. This example shows the way that system architects can customize the assumption meta-model according to their domain constraints and organizational needs.

The second step in our specification approach is to assign the meta-model's attributes in the architecture description. In our example assumption, we explained that the decoder device is impacted by the assumption while the encoder device realizes the assumption. Figure 7-5 shows a partial specification of the system components of our case example after assigning the assumption properties:

```
system implementation VideoStreamingSystem.Impl
  subcomponents
    sender_sys: system Sender.Impl;
    receiver_sys: system Receiver.Impl;
    conn_bus: bus WiredBus.Impl;
  properties
    VideoEncodingAssumption::description => "The video encoding protocol is MPEG-3";
    VideoEncodingAssumption::criticality => Critical;
    VideoEncodingAssumption::dependency =>  impact
                                            applies to receiver_sys.decoder_dev;
    VideoEncodingAssumption::dependency =>  realize
                                            applies to sender_sys.encoder_dev;
end VideoStreamingSystem.Impl;
```

Figure 7-5.Second Step – Specification of Assumption's Properties in the Architecture Description of the VSS Case Example

The second step of our approach is expressed in Figure 7-5. In this figure we show the method to assign the values of the assumption meta-model property set. For this purpose, the description property of the assumption is set. Additionally, for specification of the assumption dependency two property assignments are done. The first assigns the dependency by the value *impact* and applies it to the decoder device in the receiver subsystem. The second assigns the dependency by the value *realize* and applies it to the encoder device in the sender subsystem. By doing so, the video encoding assumption of our example is specified in the AADL architecture description of our case example. The criticality custom attribute is also assigned with *critical* value to specify that the assumption is critical to the system.

The elaboration of the example studied for this approach helps in understanding the benefit of our approach. If one looks at the architecture of the case example, it is now explicit that there is a relation between the encoder and the decoder. This relation could not be made in the SAE AADL before. Using our approach a conceptual relationship is made among the assumption, the decoder which is impacted by the assumption, and the encoder which realizes it. In Figure 7-6 the graphical representation of our case example is demonstrated:

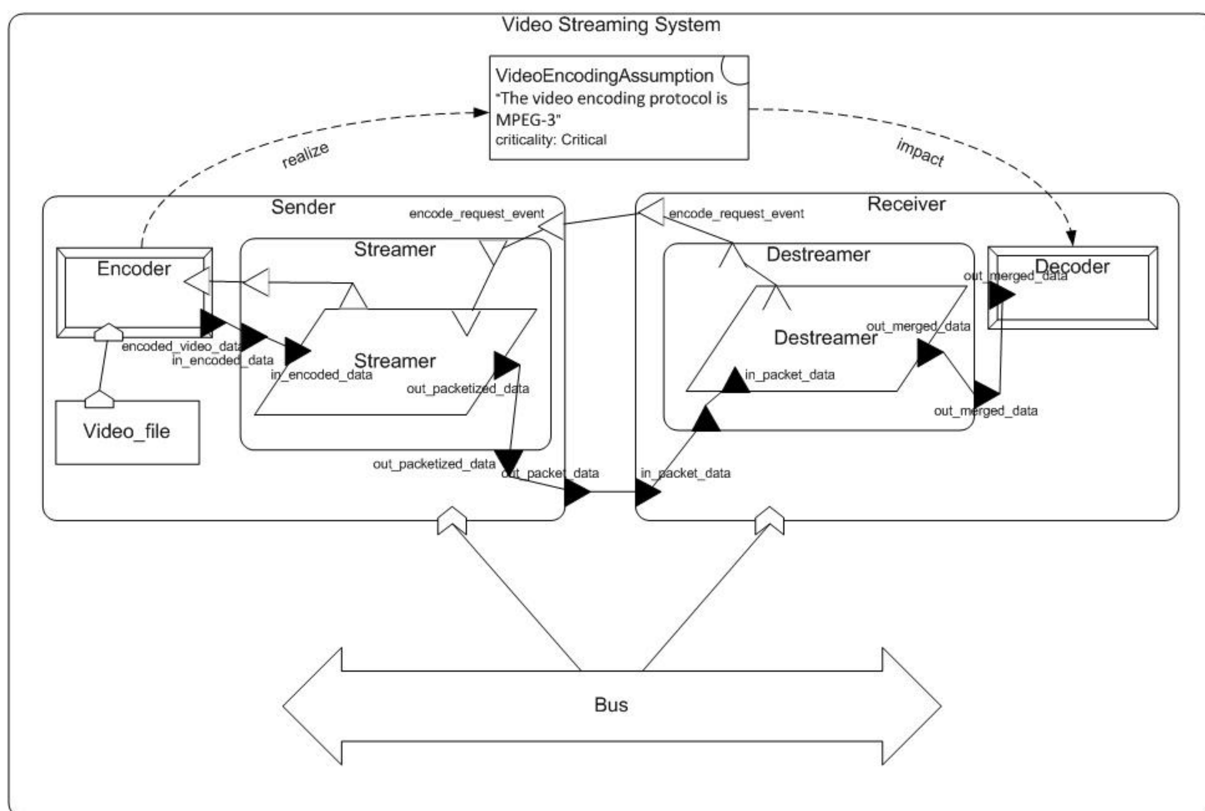

Figure 7-6.VSS AADL Graphical Representation containing Video Encoding Assumption

The proposed approach utilizes the concept of property set in the SAE AADL standard. Together with our proposed extension to this standard, which is explained in Chapter 6, we extend our approach to maximize its benefits.

7.2.1 Extending the Approach using Property Set Reuse Extension

The approach explained previously in this chapter uses property set concept in the SAE AADL standard for specification of the assumptions. Although the approach makes it possible to define the assumption specification meta-model and the custom attributes, there are some concerns in defining complex meta-models. These concerns can be addressed by utilization of reusable property sets in the standard, which we proposed in the Chapter 6, in order to create complex and coherent assumptions specifications.

The first concern is the repetitive task of declaration of the property sets. Our meta-model enforces declaration of the assumption specification meta-model including at least the two properties of description and dependency. For an architect this means that one needs to declare one property set for each assumption and repeating at least the two common properties, description and dependency, in all of them. When the system architecture gets complex, this becomes an exhaustive task. Thereby, reuse of property sets becomes vital to this point.

The second concern, which brings attention to the need for property set reuse in the SAE AADL standard, is about adding a common custom attribute to all the assumptions. In a situation where an architect decides to include an additional common set of assumption information in all the specification of the assumptions one must repeatedly declare a common set of custom attributes in all the declarations of the property sets for meta-models. Besides, later on when it is needed to change the common attributes, i.e., this can happen due to design changes, one must again change all the meta-models' property sets for assumptions. These repetitive tasks obviously take a lot of time and more importantly are very error-prone activities.

Moreover, the definition of custom attributes for dependencies can better be realized by the concept of reusability in property sets. Reusing a property set enables an architect to package common custom attributes for dependencies together with the definition of dependencies in one place and reuse it in the property sets of meta-models. This way, specifying the assumptions makes more sense. Also a change in dependency custom attributes does not cast the error-prone task of modification of all the meta-models' property sets over the project.

Last but not least, the result of the specification can be more readable and easy to understand when assumptions specification meta-models are declared by the concept of property set reuse. By reusability of property sets the meta-model declarations have organized structure and assignments of the meta-model properties in the AADL code are simpler.

Now we explain our approach to utilize our property set reuse concept by extending the previous video encoding assumption example to show the solution for on the previously mentioned concerns. The assumption was:

Assumption: The video encoding protocol is MPEG-3.

It is shown in Figure 7-3 that all the property sets of the assumption meta-models need to specify the assumption description and dependency. Said previously, this becomes an exhaustive task when one needs to declare a large number of assumptions specification meta-models. Thus, we employ property set reuse concepts (see Chapter 6) to create a common meta-model property set which defines the description and dependency properties. It is shown in Figure 7-7 and later reused by inheriting it in the declaration of other meta-models' property sets:

```
-- Declaration of the common assumption's meta-model
property set CommonAssumptionMetaModel is
  description: aadlstring  applies to (all);
  dependency: enumeration (impact, realize)
              applies to (all);
end CommonAssumptionMetaModel;

-- Inheriting the common property set for assumption's meta-model
property set VideoEncodingAssumption extends CommonAssumptionMetaModel is
end VideoEncodingAssumption;
```

Figure 7-7.Property Set Reuse for Declaration of Common Assumption Meta-Model

Figure 7-7 illustrates how to declare the common properties of the assumption specification meta-model in a property set and reuse its property set. The *CommonAssumptionMetaModel* is declared to define the description and dependency properties. The *VideoEncodingAssumption* uses *extends* keyword to inherit the common property set properties. From now on, for specification of the video encoding assumption in the AADL code one can simply use the inherited properties as it is shown in Figure 7-8:

```
system implementation VideoStreamingSystem.Impl
  subcomponents
    sender_sys: system Sender.Impl;
    receiver_sys: system Receiver.Impl;
    conn_bus: bus WiredBus.Impl;
  properties
    -- assignment of the inherited description property
    VideoEncodingAssumption::description => "The video encoding protocol is MPEG-3";
    -- assignment of the inherited dependency property
    VideoEncodingAssumption::dependency =>  impact
                                            applies to receiver_sys.decoder_dev;
    -- assignment of the inherited dependency property
    VideoEncodingAssumption::dependency =>  realize
                                            applies to sender_sys.encoder_dev;
end VideoStreamingSystem.Impl;
```

Figure 7-8.Assigning Inherited Properties for Video Encoding Assumption

Similarly, for adding a set of common custom attributes to all or a subset of assumptions specification meta-models, we use the reusability concept to declare them all in one place and inherit them by the meta-models' property sets that should include them. Figure 7-9 illustrates this approach on the video encoding assumption example:

```
-- Declaration of the common assumption's meta-model
property set CommonAssumptionMetaModel is
  description: aadlstring  applies to (all);
  dependency: enumeration (impact, realize)
              applies to (all);
end CommonAssumptionMetaModel;

-- Declaration of the common criticality custom attributes
-- Inheriting the common property set for assumption's meta-model plus adding
-- ciriticality custom attribute
property set CriticalityCustomAttributes extends CommonAssumptionMetaModle is
      criticality: enumeration (Critical, Non_Critical)
               applies to (device);
end CriticalityCustomAttributes;

-- Delaraction of the video encoding assumption
-- Inheriting the criticality common custom attributes
property set VideoEncodingAssumption extends CriticalityCustomAttributes is
end VideoEncodingAssumption;
```

Figure 7-9.Property Set Inheritance for Declaration of Criticality Custom Attributes

As Figure 7-9 shows, this approach is mixed with the previous approach of defining the common meta-model properties. This shows that the *CommonAssumptionMetaModel* is inherited by the declaration of *CriticalityCustomAttributes* property set. In this way, when the *VideoEncodingAssumption* property set inherits the common custom attributes property set, it automatically inherits *criticality* property as well as the description and dependency properties.

On the other hand, if all the common custom attributes must be added to all the assumptions specification meta-models, this can be done by simply defining them in the *CommonAssumptionMetaModel*(see Figure 7-7).

In a similar fashion we can reuse a property set for wrapping the dependencies with their custom attributes. For our video encoding assumption, we need to specify the level of engagement of the component in the assumption impact [6]. For doing so, we have to add a custom attribute to our meta-model for each dependency and call it *association_type* which can be assigned by *complete* or *partial* values. Because this custom attribute is going to be used by several assumptions specification meta-models, we can benefit from reusing the property set by declaring it as a common dependency custom attribute together with the dependency property in a common dependency property set (see Figure 7-10).

```
-- Declaration of the common assumption's meta-model
property set CommonDependency is
  -- the association between the assumption and the component
  -- it is applied to
  association: enumeration (impact, realize)
              applies to (all);
  -- The association type which can be complete or partial
  association_type: enumeration (complete, partial)
              applies to (all);
end CommonDependency;

-- Declaration of the common assumption's meta-model
property set CommonAssumptionMetaModel is
  description: aadlstring  applies to (all);
  dependency: CommonDependency applies to (all);
end CommonAssumptionMetaModel;

-- Delaraction of the video encoding assumption
property set VideoEncodingAssumption extends CommonAssumptionMetaModel is
end VideoEncodingAssumption;
```

Figure 7-10.Property Set Reuse for Declaration of Common Dependency Custom Attribute

The declaration of the common dependency custom attribute in the *CommonDependency* property set is shown in Figure 7-10. The *association* property together with the *association_type* custom attribute are defined in the declaration of *CommonDependency* which is used later to define the type of *dependency* property in our assumption specification meta-model. This approach creates a structure for our video encoding assumption meta-model which has put together the dependency association with its type. Figure 7-11 shows the specification of video encoding assumption along with its dependency and association type attribute in the AADL code:

```
system implementation VideoStreamingSystem.Impl
  subcomponents
    sender_sys: system Sender.Impl;
    receiver_sys: system Receiver.Impl;
    conn_bus: bus WiredBus.Impl;
  properties
    -- assignment of the inherited description property
    VideoEncodingAssumption::description => "The video encoding protocol is MPEG-3";
    -- assignment of the association property of the inherited dependency
    VideoEncodingAssumption::dependency::association =>  impact
                                            applies to receiver_sys.decoder_dev;
    -- assignment of the association type property of the inherited dependency
    VideoEncodingAssumption::dependency::association_type =>  complete
                                            applies to receiver_sys.decoder_dev;
    -- assignment of the association property of the inherited dependency
    VideoEncodingAssumption::dependency::association =>  realize
                                            applies to sender_sys.encoder_dev;
    -- assignment of the association type property of the inherited dependency
    VideoEncodingAssumption::dependency::association_type =>  complete
                                            applies to sender_sys.encoder_dev;
end VideoStreamingSystem.Impl;
```

Figure 7-11.Assigning Dependency Properties for Video Encoding Assumption

The code for assumption specification shown in Figure 7-11 is very clear and understandable (see Chapter 6. The associations and association types that are applied to each component are tied together. The complete association type for the encoder device makes it explicit that the device is completely impacted by a change in the video encoding assumption.

The approach explained in this chapter benefits from reusable property sets. The examples given here are additional motivations for the need to reusable property sets that we have proposed. In order for us to be able to utilize it, the SAE AADL should support our proposed extension.

7.3 Approach 2: Assumption Specification through Annex

Our second approach for specification of assumptions in architecture descriptions is inspired from Assumption Management Framework by Tirumala [3]. In our approach we use annex concept to define a new sub-language for assumption specification.

This approach has two steps.

1. Defining assumptions specification meta-models in an annex library
2. Specifying assumptions in architecture descriptions by declaring annex sub-clauses in the components specifications

In the first step, we use a new sub-language in our annex library. This assumption specification sub-language supports definition of the assumption meta-models in an annex. We have chosen syntax similar to the SAE AADL standard. The meta-model is defined using assumption specification sub-language in an annex library. Figure 7-12 shows a syntactic declaration of an assumption using assumption specification sub-language:

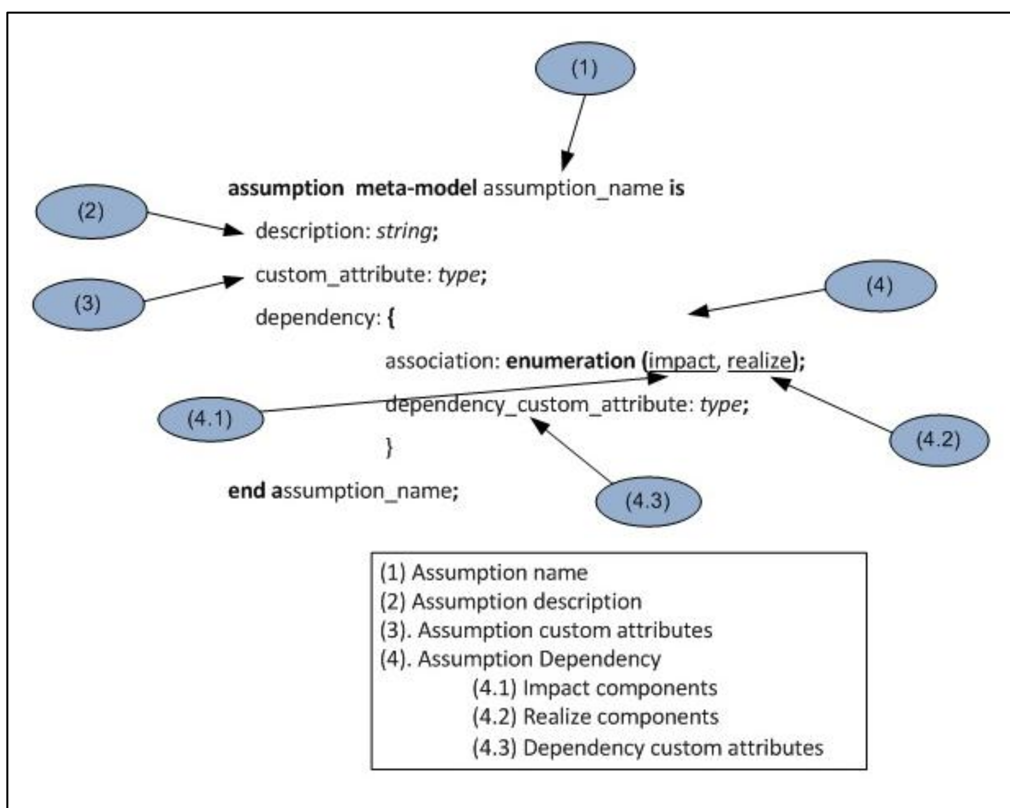

Figure 7-12.Assumption Specification Meta-Model Declaration in Annex Library

Figure 7-12 illustrates mapping of the building blocks of our assumption specification meta-model to its declaration in the definition of our assumption annex library. The syntax used in this figure is according to our assumption specification sub-language. Words in bold show reserved words; types

are shown in italics and the normal texts are the identifiers. The main difference between this declaration and a property set declaration in the Figure 7-2 is the declaration of dependencies. In this approach annex brackets are used to declare a dependency which ties an association with its custom attributes of any kind. An example of its use is given later in the chapter.

The second step is to specify the assumptions in architecture descriptions. To do this, annex sub-clauses can be declared anywhere in the main system component implementation. In the declaration inside an annex sub-clause the pre-defined assumption meta-models from the first step are specified. In the specifications within annex sub-clauses our assumption specification sub-language is again used to assign meta-models' attributes.

For better understanding this approach, in the continuing of this chapter we apply it to the VSS case example for specification of the video encoding assumption. Let's recall the assumption:

Assumption: The video encoding protocol is MPEG-3.

For explanation of this example assumption see Chapter 7.2.

In order to specify this assumption, its meta-model is first defined in an assumption specification annex library. Figure 7-13 demonstrates the defined annex library for this assumption:

```
package Assumptions_Specification_AnnexLibrary

--declaration of the public assumption meta model annex library
public
annex Assumption_MetaModel{**
    assumption meta-model VideoEncodingAssumption is
        description: aadlstring;
        criticality: enumeration (critical, non_critical);
        dependency: {
            association: enumeration (impact, realize);
            association_type: enumeration (complete, partial);
        }
    end VideoEncodingAssumption;
**};

end Assumptions_Specification_AnnexLibrary;
```

Figure 7-13.Video Encoding Assumption Declaration in Assumption Meta-Model Annex Library

Figure 7-13 shows the definition of assumption specification meta-model in an annex library called *Assumption_MetaModel*. For organizing the meta-models, we use a package named *Assumptions_specification_AnnexLibrary* in our declaration. This package aids bundling our annex library of assumption meta-models in one accessible unit. As it is shown in the figure, the syntax of the assumption specification sub-language is very similar to the SAE AADL standard.

Once the video encoding assumption is defined in the *Assumption_MetaModel* annex library, it can be used in the next step of our approach to specify the assumption and its dependencies in the architecture description of the VSS. In order to realize it, as it is mentioned earlier, annex sub-clauses are used in the specification of the main system component in the AADL code of the case example. Figure 7-14 illustrates the result of this step:

```
system implementation VideoStreamingSystem.Impl
      subcomponents
        sender_sys: system Sender.Impl;
        receiver_sys: system Receiver.Impl;
        conn_bus: bus WiredBus.Impl;
      annex Assumption_MetaModel {**
        VideoEncodingAssumption::description => "The video encoding protocol is MPEG-3";
        VideoEncodingAssumption::criticality => critical;
        VideoEncodingAssumption::dependency::association =>  impact
                                    applies to receiver_sys.decoder_dev;
        VideoEncodingAssumption::dependency::association_type =>  complete
                                         applies to receiver_sys.decoder_dev;
        VideoEncodingAssumption::dependency::association =>  realize
                                    applies to sender_sys.encoder_dev;
        VideoEncodingAssumption::dependency::association_type =>  complete
                                         applies to sender_sys.encoder_dev;

      **};
    end VideoStreamingSystem.Impl;
```

Figure 7-14.Assignment of Video Encoding Assumption through Annex Sub-Clause

The declaration in Figure 7-14 shows the step of specifying our example assumption meta-model information. It is illustrated that our video encoding assumption is critical. This is done by assigning criticality attribute in the annex sub-clauses. It is also specified that this assumption impacts on the decoder device completely. Association and association type are the attributes used for this purpose. Additionally, it is declared that the encoder device is responsible for realization of this assumption to which its contribution is also complete.

The assumption specification sub-language in this approach can be improved to support definition of complex meta-models. By doing so, we can achieve inheritance and reuse of meta-models as well in this approach (see Chapter 7.2.1). For example, using an extended assumption specification sub-language one can create a meta-model that defines all the common attributes of an assumption specification. By inheriting this common meta-model in other assumptions meta-models there is no need to repeat the declaration of all the common attributes. Figure 7-15 shows the definition of a common meta-model using the extended sub-language:

```
package Assumptions_Specification_AnnexLibrary

    --declaration of the public assumption meta model annex library
    public
    annex Assumption_MetaModel{**
        --declaration of a common meta-model
        assumption meta-model Common_MetaModel is
            description: aadlstring;
            dependency: {
                association: enumeration (impact, realize);
            dependency: {
                association: enumeration (impact, realize);
                association_type: enumeration (complete, partial);
            }
        end Common_MetaModel;

        --declaration of an extended meta-model
        assumption meta-model VideoEncodingAssumption
                          extends Common_MetaModel is
            criticality: enumeration (critical, non_critical);
        end VideoEncodingAssumption;
    **};

end Assumptions_Specification_AnnexLibrary;
```

Figure 7-15.Example of Common Meta-Model using Extended Sub-Language

Figure 7-15 shows the definition of a common meta-model called *Common_MetaModel*. This defines the common attributes of all the meta-models in our approach. Attributes such as description and dependency are of this kind. Later, this common meta-model is inherited in the definition of another meta-model for the video encoding assumption. Therefore, the video encoding assumption meta-model inherits all those common attributes as well as adding a new attribute in its body called *criticality*. The specification of this extended annex is the same as we mentioned before. One can use annex sub-clauses to specify the video encoding assumption in an architecture description (see Figure 7-14).

7.4 A Good Solution Criteria

Selection of a qualified approach for modeling architectural assumptions depends on various criteria. One can think of several criteria depending on the domain of interest. In this chapter we discuss the criteria that we found interesting from different perspective. The criteria that we have identified are generally from an architect's point of view as well as the technology.

7.4.1 Legibility

One of the main communication medium in software development processes is document. Depending on the development methodology, documents are created during each phase. Documents can be the main deliverables is some phases such as analysis. The SRS as the main deliverable of this phase is a well written and formatted document. Sometimes documents are not the main deliverables, but are important to complement the activity. For example, documentation of coding is necessary to complement source codes.

In development projects it is very important that the documents are easy to read. Because documents are used as the main communication medium within a team, among the teams in the organization, and throughout the project, the legibility is one of the main factors. Therefore, it is absolutely essential that the approaches taken for documentation in any activity result in readable and easy to understand documents.

Furthermore, modeling methods that result in textual forms complemented with visual or graphical forms have a great value for increasing understandability and distribution of the knowledge throughout the organization. It also aids to increase the transparency in the internal as well as external communications of project teams.

7.4.2 Practicality

A good modeling method is easy to implement. This enables software architects to easily adopt it for doing their tasks. An easy to implement method is important from different aspects.

First of all, an approach should not be very complex. A complex approach takes a lot of energy to adopt. Instead, it should ease the architect's task. Architects' job is already complicated enough, so that it is important that modeling methods do not put additional unnecessary burden on their shoulders.

Furthermore, a good method should not interfere in the routines of other approaches that are currently being used by architects. Rather, the new method should effectively complement other approaches by adding value to their results or decrease the effort needed for doing conventional tasks. In other words, a qualified modeling method must be a best practice approach. This means that it should deliver a better outcome in comparison with the traditional approaches.

7.4.3 Extensibility

Extensibility is the support for further development. An extensible modeling method is an approach that supports further improvements. This is an important aspect. From an architect's point of view, a modeling approach should have enough flexibility to be extended. Although an architectural modeling method imposes certain disciplines, in the same time it should keep architects' hands open to be able to make their own changes in the method to improve it according to their needs and interests.

7.4.4 Backward Compatibility

Backward compatibility is the ability to support previous versions of models with new methods. A modeling method needs to have this ability to be the first choice of architects.. It is important from different aspects.

First of all, architectural modeling methods that enforce core changes in the original model are not embraced. The original model is the one that the modeling method is going to add value to. A modeling method has a great value if it extends the original model instead of imposing fundamental changes to it. This is because in software architecture, frequent analyses of models lead to several revisions. Therefore, it appears to be important that new modeling methods should be able to support previous versions.

Secondly, backward compatibility is a core value of any extensible method. Extended modeling methods should have the ability to support the models using the previous versions of the method. Therefore, extensibility only makes sense when it is backward compatible. This is a critical factor in architecture evolutions.

7.4.5 Analysis Support

Software architectures are continually analyzed. Architecture analyses are performed to evaluate the quality system, find the errors, etc. Simulation is also a method to analyze the impact of design decisions and so on. Thus, architecture analyses are directly or indirectly focused by architectural modeling methods.

Software architects adopt approaches that support their purposed analysis. For this reason, modeling approaches are designed to support different analyses. For example, error model annex in the AADL is designed to model errors in components and connections of an AADL specification [21]. This method provides support for components dependability analysis. Therefore, any method for modeling architectural decisions has to take analysis into account in one way or another. This fact enforces critical rules and decisions in the design of modeling approaches.

7.5 Evaluation

In our study we have defined two distinguished approaches for explicating architectural assumptions using the SAE AADL standard. In our first approach, we used property set concept to specify assumptions specification meta-models (see Chapter 7.2). We also have proposed an extension to this concept in the AADL to better support declaration of complex property sets (see Chapter 7.2.1). Our second approach uses annex concept of the SAE AADL standard (see Chapter 7.3). This concept enabled us to add our assumption specification sub-language to the AADL for specification of architectural assumptions.

While both approaches result in a similar assumptions specification, there are distinguished differences between them. In this chapter we will discuss their differences in order to evaluate and compare their values.

For selecting a best approach, we use our evaluation criteria for a good modeling method. We have introduced these criteria in Chapter 7.4. Using these criteria we test whether the approaches are:

- Legible - Easy to read
- Practical - Easy to implement
- Extensible
- Backward compatible
- Analysis supportable

For us to be able to explain the evaluation Table 7-1 is given:

Criteria	Approach 1	Approach 2
Legibility	Yes	Yes
Practicality	Yes	Yes
Extensibility	Yes	Yes
Backward Compatibility	Yes	Yes
Analysis Support	Yes	No

Table 7-1.Comparison Table

Table 7-1 shows the results of our evaluation of the two approaches. As it was said before the first approach uses property set concept while the second approach uses annex. The first column of the table is the evaluation criteria by which both approaches were tested against. Under each approach's column, *yes* or *no* identifies whether the approach satisfies the respective criterion.

In order to explain our evaluation, we start with the legibility criterion. The importance of model legibility of a modeling method is mentioned in Chapter 7.4.1. An example of assumptions specification using the first approach was given in Figure 7-11. Also assumption specification of the same example using the second approach was given in Figure 7-14. According to the legibility

criterion, we can easily realize that both of the outputs appear to be clear and simple to read in both cases.

Second criterion is used to assess the practicality of the approaches. An easy to implement modeling method should neither be complex nor interfere with other methods (see Chapter 7.4.2). We can see from the examples given in chapters 7.2 and 7.3 that none of the approaches are complex. The steps explained show that one can simply adopt them for modeling assumptions. Besides, none of the approaches seems to interfere with other modeling methods. Instead, they assist architects to enrich architectural models by explicating the assumptions [6].

Furthermore, extensibility is naturally supported by both of the approaches. This is because in the AADL either of property set and annex concepts are extensible by nature. We showed the ways to extend the approaches using custom attributes (see chapters 7.2 and 7.3). For example, by modifying our assumption specification sub-language in the second approach, one can extend the assumptions annex library for improving the modeling method to support hierarchy of assumptions meta-models (see Chapter 7.3). Besides, the property set reuse extension that we applied to our first approach improves our modeling method to support creation of complex meta-models (see Chapter 7.2.17.2). These examples show that both of the studied approaches are extensible.

Backward compatibility, which concerns about supporting previous versions of the model, is achieved in the first and second approach. In the first approach, assumption specifications are added to the original model by declaration and specification of new property set, therefore original model still works in other analysis tools. It is the same in the second approach. The assumption specification annex that is defined and used in architecture descriptions does not reject backward compatibility.

Finally, support for analysis is the main advantage of the first approach in comparison to the second approach. As annex concept supports extending the AADL with sub-languages, it is mainly used for performing architecture analyses. Unfortunately, the SAE AADL standard does not provide any way for an annex to access another annex's content. This is because the AADL parser cannot guarantee which annex will be parsed first. Therefore, the analyses which can be done by various annexes cannot be performed on the assumptions specified using the second approach. Thus, the only ways to do analysis on them is to merge the sub-language of analysis methods into the sub-language of assumption specification annex or vice versa. This has several disadvantages. One main disadvantage is the coupling of analysis method and assumptions specification. It becomes crucial later when one needs to perform a completely different analysis on the assumptions. On the other hand, because in our first approach assumptions specification is applied to AADL models by assigning properties to components, performing analyses that use annexes is supported without any need for modification of the approach.

To sum up the evaluation, both of the approaches have similar benefit and values except support for future analyses. The result of our evaluation is that our first approach has all the selected criteria for a good solution.

Chapter 8

Related Work

We have identified three studies that contribute to explication of assumptions. Here we present related work within the area of assumptions management.

8.1 Assumption Management Framework

The Assumption Management Framework (AMF) has been introduced by Tirumala [3]. The aim of this framework is a well-defined vocabulary to encode assumptions in a machine-checkable format [3]. Additionally, he introduces a systematic process that performs automatic validations for machine-checkable assumptions. The work focuses on assumptions of two levels: architecture and coding. For validation of architectural assumptions this framework uses a CASE tool built on OSATE. For coding assumptions, a software tool for invoking the validation routines in java codes was developed. At the end of validation processes, a set of invalid assumptions are flagged which implies violation of assumption rules in the components [3].

This framework addresses the problems made from inadequacy of component interfaces. The motivation behind this is that current software engineering practices do not efficiently capture and validate assumptions at the interface level of components [3]. The introduced framework minimizes human efforts in validation activities and allows encoding assumptions without imposing modifications to source code.

The AMF only supports machine-checkable assumptions in component interfaces. Some assumptions, such as the video encoding protocol example, are hardly possible to be specified in machine-checkable formats. In this approach, we address a wider range of architectural assumptions by providing an explication method that includes informal description for explanation of non-machine-checkable assumptions. The proposed approach supports additional categories of assumptions as well that are beyond component interfaces. For example, our video encoding assumption encompasses the entire decoding device including controlling flows.

Furthermore, the proposed approach supports specification of an assumption with several components engaged in its realization. It also provides a way to explicate assumptions by which several components are impacted. This increases the practicality of the proposed method in comparison with the AMF.

Finally, the proposed modeling method appears to have achieved supportability for future analysis purposes. Unlike the AMF it does not utilize an annex for modeling assumptions. The limitation in the AMF is that no other annex can use this framework in the AADL. Therefore, other analysis methods that are implemented in the other annexes cannot be applied to the models designed using the AMF. On the other hand, analyses that use annexes can still be performed on the assumption models resulted from the proposed method. This is another advantage of the proposed approach in comparison with the AMF.

8.2 Explicit Assumptions Enrich Architectural Models

Lago and van Vliet have introduced a meta-model with required formal basis of explicating assumptions and their relationships with architectural assets [6]. They conjecture that it is not only useful to explicitly model variability in software architecture, but that it is also useful to explicitly model invariabilities [6]. In their approach they do this by driving a meta-model from an experiment. In that experiment they explicitly modeled some assumptions in the architectural views of an existing product family, namely the product feature model view and product component model view.

The meta-model introduced by Lago and van Vliet supports explication of assumptions that impact the features of a product family or components of software. Their proposed meta-model does this by defining three categories of associations in its model: *f-impact*, *s-impact*, and *realize* [6]. An *f-impact* association is used to model the dependency between an assumption and a generic feature in the product feature model. This type of relationship identifies the direct features that are influenced by an associated assumption. An *s-impact* association is similarly used to model a dependency between a component and an assumption in product component models. Additionally, a *realize* association is used to model the dependencies between assumptions and the features that fulfill those assumptions.

In our modeling method, we use similar associations to model dependencies between assumptions and components in architecture descriptions. However, the model introduced by Lago and van Vliet does not include description of assumptions. The proposed approach on the other hand achieves this by its integration in the SAE AADL. Therefore, explication of assumptions is realized in this approach which leads to enriched architecture descriptions.

8.3 Assumption Management System

The aim of the Assumption Management System is to develop a method that assists developers to record and monitor assumptions during implementation of source codes [13]. The focus of this method is to store and restore assumptions together with the code that later team members can monitor for tracing the validations, etc. [13]. Using this method one can explicate assumptions, made during implementation or higher levels, by integrating them in java source code.

The result of this approach is an XML structure that is used to record assumptions in the source code. Later on, a developed software tool parses the code and extracts the assumptions XML structures to store them in a database repository. The tool also contains functionalities which help team members, who possess enough knowledge for assumption validation, to search, assert, and declare each assumption's validation state.

In this approach we consider the importance of explicating architectural assumptions in the architecture. Based on the observation, we believe that invalid assumptions at architectural levels may lead to system failure, poor performance, or high fixing cost. Therefore, it is important to make them explicit at the first place where they are made in initially. This benefits software architects to identify invalid assumptions and their impact sooner in development processes.

Furthermore, our approach supports a disciplinary extension to the meta-model of assumptions specification. This aids architects to define domain-specific assumption meta-models that can be used to specify additional information together with architectural assumptions in architectures.

Discussions

The proposed assumption modeling method has several advantages for software architects. As we discussed earlier, the method appears to be easy to apply. Adoption of this method results in specification of architectural assumptions that are easy to understand. Assumptions of the environment in which the software works in can be modeled as well using this method. Because of using the SAE AADL standard for model description, the method supports specification of a variety of assumptions and their dependencies with the supported system components in AADL. For example, if an assumption impacts only on a certain data port the model can include specification of an assumption impact dependency between the assumption and the respective port declaration.

In the design of the modeling method, we have identified an extensible assumption specification meta-model. The structure of the meta-model together with reusable property sets brings extensibility to the modeling method. By adopting this method, architects can customize the meta-model for assumptions to support specification of additional information in which they are interested. For example, they can create categories of assumptions by adding custom attributes such as type and criticality to assumptions, identify various attributes for assumption dependencies, etc. Therefore, organizations can increasingly benefit from customized specification of architectural assumptions depending on their needs and domains of interest.

The proposed modeling method assists software architects to make modifications to assumption models. By adopting reusable property sets one can create and reuse common property sets. The modification of common property sets is automatically and with the least burden casted over the whole model in this way. Consider a situation when one needs to add a new attribute to all of the assumptions in the model that are already specified. Property set reuse plays an important role here for simplifying the modification. One can easily add the new attribute to all the pre-declared assumptions by adding its declaration to the common property set that is inherited by other property sets of the meta-models.

Explication of assumptions is the first step of a systematic assumption management process. Further on, specification of assumptions can be used to analyze the architecture for different purposes. This appears to be another important benefit of the proposed method. The models created by applying our method to architectures can be further used in different analyses because assumptions information is specified as properties of the components in the architecture description. For example, an analysis annex can access the assumption model to extract the information of assumptions from the properties of components.

Chapter 9

Conclusions

In recent years several attempts have been done to identify a systematic assumptions management. The first step towards that is to explicate assumptions. Among assumptions made throughout software development processes, architectural assumptions are important for several reasons. The main reason is that they have a great impact over the architecturally influential design decisions that are made for developing software. If architectural assumptions are incorrect or the decisions made are not according to the assumptions, harsh drawbacks are expected.

In this study we have aimed to develop a modeling method for explicating assumptions. Our work has resulted in a modeling method for specification of architectural assumptions. At first we have identified a meta-model for identification of assumption information. Using the meta-model, architects can define assumptions by identifying the attributes that are important for them to specify in architectures. Secondly, we have investigated two approaches for specification of assumptions meta-models in the context of architecture description. In our approaches we have used the SAE AADL standard as the architecture and modeling description language to show the applicability of our approach. The first specification approach uses property sets concept of the AADL to specify assumptions' meta-models as the properties of components in architecture descriptions. The second investigated approach is developed using annex concept as a new sub-language to extend the AADL descriptions for modeling assumptions. An evaluation of both of the approaches against some preliminary criteria is given at the end. The result of the evaluation has shown that the first proposed approach that uses property sets concept is more suitable for modeling assumptions of complex systems.

The proposed method has several advantages for software architects. It enables explication of architectural assumptions through a systematic approach. It supports identification of assumptions and their dependencies with the system components. Therefore, it provides enriched architectures. This brings assumption awareness and traceability to software development processes.

Traceability is the key in this method which is realized through assumption dependency. By identifying the relationships among assumptions and components in the model one can track assumption from architecture design to implementation and vice versa. This can contribute to find invalid assumption as well as components affected by those incorrect assumptions.

Furthermore, in addition to the introduced modeling method, we have proposed an extension to the SAE AADL standard to support reusability of property sets in its language. We have reasoned that it is beneficial that the AADL allows reusing property sets through inheritance. In our modeling method we have utilized reusable property sets to define complex meta-models.

Finally, we have identified some possible research extensions to our method that are listed below:

Assumption modeling CASE (Computer Aided Software Engineering) tool

Functionalities for our proposed assumption modeling method can be developed on top of Eclipse that can be integrated with OSATE and TOPCASED[1].

Scalability case-studies

Scalability of our method is important in modeling assumptions of complex systems. This can be evaluated through carrying out some case-studies on real systems such as Volvo cars.

Rich assumption specification meta-model

A good solution can be identification of domain specific assumption specification meta-models. These meta-models can assist in concise and rapid modeling of assumptions in a specific domain.

[1] OSATE and TOPCASED are a combination of AADL functionalities collection, architecture analysis, and graphical user interface as plug-ins to the open-source Eclipse environment (see [24]).

References

[1] Wiegers, Karl E.. "Chapter 10 - Documenting the Requirements". *Software Requirements, Second Edition*.Microsoft Press. © 2003. Books24x7. <http://common.books24x7.com/book/id_6818/book.asp> (accessed February 5, 2010)

[2] Ostacchini, I. and Wermelinger, M. 2009. Managing assumptions during agile development. In *Proceedings of the 2009 ICSE Workshop on Sharing and Reusing Architectural Knowledge* (May 16 - 16, 2009). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 9-16. DOI= http://dx.doi.org/10.1109/SHARK.2009.5069110.

[3] Tirumala, A. S. 2006 *An Assumptions Management Framework for Systems Software*. Doctoral Thesis. UMI Order Number: AAI3243011., University of Illinois at Urbana-Champaign.

[4] Mansouri-Samani, M. & Sloman, M. GEM: A Generalized Event Monitoring Language for Distributed Systems. Imperial College of London,Research Report No. DOC 95/8, August 1995.

[5] Steingruebl, A. and Peterson, G. 2009. Software Assumptions Lead to Preventable Errors. *IEEE Security and Privacy* 7, 4 (Jul. 2009), 84-87. DOI= http://dx.doi.org/10.1109/MSP.2009.107.

[6] Lago, P. and van Vliet, H. 2005. Explicit assumptions enrich architectural models. In *Proceedings of the 27th international Conference on Software Engineering* (St. Louis, MO, USA, May 15 - 21, 2005). ICSE '05. ACM, New York, NY, 206-214. DOI= http://doi.acm.org/10.1145/1062455.1062503.

[7] Miranskyya, A., Madhavji, N., Davison, M., and Reesor, M. 2005. Modelling Assumptions and Requirements in the Context of Project Risk. In *Proceedings of the 13th IEEE international Conference on Requirements Engineering* (August 29 - September 02, 2005). RE. IEEE Computer Society, Washington, DC, 471-472. DOI= http://dx.doi.org/10.1109/RE.2005.44.

[8] Lehman, M. M. 2005. The Role and Impact of Assumptions in Software Development, Maintenance and Evolution. In*Proceedings of the IEEE international Workshop on Software Evolvability* (September 26 - 26, 2005). SOFTWARE-EVOLVABILITY. IEEE Computer Society, Washington, DC, 3-16. DOI= http://dx.doi.org/10.1109/IWSE.2005.14

[9] King, R. D. and Turnitsa, C. D. 2008. The landscape of assumptions. In *Proceedings of the 2008 Spring Simulation Multiconference* (Ottawa, Canada, April 14 - 17, 2008). Spring Simulation Multiconference. Society for Computer Simulation International, San Diego, CA, 81-88.

[10] Garlan, D., Allen, R., and Ockerbloom, J. 1995. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Softw.* 12, 6 (Nov. 1995), 17-26. DOI= http://dx.doi.org/10.1109/52.469757.

[11] Sublanguage Example, https://wiki.sei.cmu.edu/aadl/index.php/Sublanguage_Example

[12] Feiler, P. H., Gluch, D. P., and Hudak, J. J. 2006. The Architecture Analysis & Design Language (AADL): An Introduction. Tech. rep. CMU/SEI-2006-TN-011.

[13] Lewis, A. G., Mahatham, T., and Wrage, L. 2004. Assumptions Management in Software Development. Tech. rep. CMU/SEI-2004-TN-021.

[14] Giese, H. 2000. Contract-Based Component System Design. In *Proceedings of the 33rd Hawaii international Conference on System Sciences-Volume 8 - Volume 8* (January 04 - 07, 2000). HICSS. IEEE Computer Society, Washington, DC, 8051.

[15] Meyer, B. 1992. Applying "Design by Contract". *Computer* 25, 10 (Oct. 1992), 40-51. DOI= http://dx.doi.org/10.1109/2.161279.

[16] Beugnard, A., Jézéquel, J., Plouzeau, N., and Watkins, D. 1999. Making Components Contract Aware. *Computer* 32, 7 (Jul. 1999), 38-45. DOI= http://dx.doi.org/10.1109/2.774917.

[17] Radjenovic, A. and Paige, R. 2006. Architecture Description Languages for High-Integrity Real-Time Systems. *IEEE Softw.*23, 2 (Mar. 2006), 71-79. DOI= http://dx.doi.org/10.1109/MS.2006.36.

[18] Li, J., Pilkington, N. T., Xie, F., and Liu, Q. 2010. Embedded architecture description language. *J. Syst. Softw.* 83, 2 (Feb. 2010), 235-252. DOI= http://dx.doi.org/10.1016/j.jss.2009.09.043.

[19] E. A. Lee. 2003. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, UC Berkeley.

[20] Society of Automotive Engineers. SAE Standards: Architecture Analysis & Design Language (AADL), AS5506, November 2004. http://www.sae.org/servlets/productDetail?PROD_TYP=STD&PROD_CD=AS5506 (2006)

[21] Feiler, P. H. and Rugina, A. 2007. Dependability Modeling with the Architecture Analysis & Design Language (AADL). Tech. rep. CMU/SEI-2007-TN-043.

[22] Murdoch, J. S., Drimer, S., Anderson, R., Bond, M. 2010. Chip and Pin is Broken. To appear at the *IEEE Symposium on Security and Privacy*(2010).

[23] MPEG-3, Retrieved 30/5/2010, from Wikipedia website: http://en.wikipedia.org/wiki/MPEG-3

[24] AADL Toolset, Retrieved 11/5/2010, from AADL information website: http://www.aadl.info/aadl/currentsite/tool/toolsets.html

Appendix 1: Video Streaming System AADL Specification

```
system VideoStreamingSystem
end VideoStreamingSystem;

system implementation VideoStreamingSystem.Impl
  subcomponents
    sender_sys: system Sender.Impl;
    receiver_sys: system Receiver.Impl;
    conn_bus: bus WiredBus.Impl;
  connections
    DataConnection1: data port sender_sys.out_packet_data ->
receiver_sys.in_packet_data;
    EventConnection1: event port receiver_sys.encode_request_event ->
sender_sys.encode_request_event;
    bus access conn_bus -> sender_sys.sender_bus_access;
    bus access conn_bus -> receiver_sys.receiver_bus_access;

end VideoStreamingSystem.Impl;

system Sender
  features
    out_packet_data: out data port;
    encode_request_event: in event port;
    sender_bus_access: requires bus access WiredBus.Impl;
end Sender;

system implementation Sender.Impl
  subcomponents
    streamer_sys: system Streamer.Impl;
    encoder_dev: device Encoder.COTS;
end Sender.Impl;

system Receiver
  features
    in_packet_data: in data port;
    encode_request_event: out event port;
    receiver_bus_access: requires bus access WiredBus.Impl;
end Receiver;

system implementation Receiver.Impl
  subcomponents
    destreamer_sys: system Destreamer.Impl;
    decoder_dev: device Decoder.COTS;
end Receiver.Impl;

system Streamer
  features
    in_encoded_data: in data port;
    out_packetized_data: out data port;
end Streamer;

system implementation Streamer.Impl
  subcomponents
    streamer_proc: process Streamer_Proc.Impl;
  connections
    DataConnection1: data port in_encoded_data ->
streamer_proc.in_encoded_data;
    DataConnection2: data port streamer_proc.out_packetized_data ->
out_packetized_data;
end Streamer.Impl;

system Destreamer
  features
    in_packet_data: in data port;
    out_merged_data: out data port;
```

```
        end Destreamer;

        system implementation Destreamer.Impl
          subcomponents
            destreamer_proc: process Destreamer_Proc.Impl;
          connections
            DataConnection1: data port in_packet_data ->
destreamer_proc.in_packet_data;
            DataConnection2: data port destreamer_proc.out_merged_data ->
out_merged_data;
        end Destreamer.Impl;

        process Streamer_Proc
          features
            in_encoded_data: in data port;
            out_packetized_data: out data port;
        end Streamer_Proc;

        process implementation Streamer_Proc.Impl
        end Streamer_Proc.Impl;

        process Destreamer_Proc
          features
            in_packet_data: in data port;
            out_merged_data: out data port;
        end Destreamer_Proc;

        process implementation Destreamer_Proc.Impl
        end Destreamer_Proc.Impl;

        bus WiredBus
        end WiredBus;

        bus implementation WiredBus.Impl
        end WiredBus.Impl;

        processor PC_processor
          features
            bus_access: requires bus access WiredBus.Impl;
        end PC_processor;

        device Encoder
          features
            in_video_data: in data port;
            encoded_video_data: out data port;
        end Encoder;

        device implementation Encoder.COTS
        end Encoder.COTS;

        device Decoder
          features
            in_merged_data: in data port;
            out_decoded_data: out data port;
        end Decoder;

        device implementation Decoder.COTS
        end Decoder.COTS;
```