AN ASSUMPTIONS MANAGEMENT FRAMEWORK FOR SYSTEMS SOFTWARE

BY

AJAY SUDARSHAN TIRUMALA

B.E., University of Mysore, 1998
M.S., University of Illinois at Urbana-Champaign, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

ABSTRACT

Critical systems in areas ranging from avionics to consumer car control systems are being built by integrating commercial-off-the-shelf (COTS) components. Software components used in these systems need to satisfy many formally unexpressed, yet necessary conditions, termed as assumptions, for their correct functioning. Invalid assumptions have been determined to be the root cause of failures in many such systems; for example, in the *Ariane 5* rocket failure. In the current software engineering practices, many of these assumptions are not recorded in a machine-checkable format, which makes validating the assumptions a manual and an error-prone task.

This thesis examines this problem in detail and evolves a framework, called the assumptions management framework (AMF), which provides a vocabulary for discussing assumptions, a language for encoding assumptions in a machine-checkable format and facilities to manage the assumptions in terms of composition and setting policies on assumption validation. A relevant subset of assumptions can be validated or flagged as invalid automatically as the system evolves. AMF allows the assumption specification process to blend with the components' source-code and architecture specification. This enables AMF to be applied to existing systems with minor or no modifications in components' implementation and design. Performance and scalability tests show that the AMF implementation is scalable to be applied to large-scale systems.

Case-studies were conducted on representative systems to study the nature and number of defects caused by invalid assumptions. It was found that a significant number of defects in the systems studied had invalid assumptions as the root-cause. It was found that AMF has the ability to encode and validate majority of the assumptions that caused defects in these systems. This can prevent such defects in the future or warn in advance of potential defects when assumptions are invalid. Analyzing and correcting one of the invalid assumptions in Iperf, an end-to-end bandwidth measurement tool, resulted in significantly better bandwidth estimates by Iperf across high-bandwidth networks. In most cases, it also resulted in savings of over 90% in terms of both network traffic generated and bandwidth measurement times.

To Appa, Amma, Ruchi and Anisha.

ACKNOWLEDGMENTS

A dedicated long-term goal, like finishing a Ph.D., requires a lot of things to fall in place, both personally and professionally. I am grateful to have had this opportunity.

My adviser, Prof.Lui Sha, has been a constant source of inspiration. I am particularly grateful to him for introducing me to the system integration problem. He is one of the few advisers who always makes so much time to discuss research with his students. This thesis would not have taken shape without his suggestions and guidance, based on his vast experience, brilliant insight, and his innumerable rounds of feedback. This apart, his personal stories and remarkable anecdotes about every possible situation in life are etched permanently in my memory.

I was fortunate to have a very helpful committee, and I have enjoyed working closely with them. Dr.Peter Feiler was key in shaping the implementation part of the thesis, by giving me an opportunity to work under him and learn AADL and related tools for a summer. Also, I am very thankful that he was able to attend my final defense in person, traveling all the way from Pittsburgh (CMU). Prof.Marco Caccamo provided me an opportunity to work with a real-time networking protocol - RI-EDF. Prof.Jennifer Hou provided an opportunity to study the assisted living project, which has helped me understand the system integration problem better. Prof.Grigore Rosu provided valuable inputs into the language part of the implementation for AMF. Outside of my committee, Dr.Les Cottrell provided me an opportunity and the resources to test Iperf across various nodes across the world. I am grateful to all of them.

My friends and lab-mates ensured that graduate school was not all work and no play. My lab-mates - Qixin, Sumant, Sathish, Hui, Xue, Kihwal and especially Tanya were great comrades and have made my path towards my Ph.D. a joyful one. My association with “Asha for education” during my graduate school re-emphasized the value for education, and that it should not be taken for granted. I also met my loving and charming wife, Ruchi, while I was working for Asha. My room-mates Girish, Arun and Ashvin have shared my food and my whims; I thank them for that.

My family members have played the most important part in this seemingly daunting journey. It was an enjoyable one for me because of them. Appa and Amma placed utmost confidence in my abilities and supported wholeheartedly, my decision to pursue a Ph.D. Amma's outlook on never compromising on education set the seeds for my undertaking this task. This whole endeavor would not have been possible without Amma's and Appa's sacrifices and selflessness, something I immensely owe them for. Ruchi was there with me during the hardest years of my Ph.D., and she was extremely loving and supportive. At the same time, she has performed phenomenally in her research also; a balancing act that I am slowly learning from her. She also opened my mind to various fields and activities outside of college and my research. Anisha was a close friend to me, in addition to being a sister and a responsible daughter; being there with Appa and Amma, while I was away from home. I was lucky to have Radha Chitti's family in town, which made me feel I was not very far away from "desh". Ruchi's parents (Papa and Mummy) placed a lot of confidence in me and I am grateful to them for this.

The Champaign-Urbana community has made me feel at home and I never felt like an outsider even though I had come from 6000 miles away. The College of Engineering and the Department of Computer Science have always set high standards and made this task worth undertaking. I am extremely indebted to NSF and DARPA for funding me throughout my Ph.D.

In spite of all the support, any shortcomings of the thesis are entirely due to the me. I dedicate this thesis to Appa, Amma, Ruchi and Anisha.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Embedded and real-time systems, ranging from cruise control systems in cars to control systems in satellite landing gear are being built using custom and commercial-off-the-shelf software components. The paradigm of complete in-house development of all the hardware and software components for a mission or a project is getting to be obsolete. Even within the same organization, software development is a highly distributed activity and the costs for integrating the software components developed by different teams is non-trivial, with companies like IBM spending over 50% of the product development budgets towards integration and system testing [1]. The problem is magnified further in embedded and real-time systems software development where many environmental assumptions are not recorded in a machine checkable format. Manual testing for validity of these assumptions is time consuming, not cost effective and prone to errors.

Most importantly, there have been catastrophic accidents resulting in loss of revenue in tune of hundreds of millions of dollars [2] and loss of lives [3]; and invalid assumptions were determined to be the cause of failures in these systems.

1.1 Incidents of System Failures Due to Invalid Assumptions

1.1.1 Ariane 5 Disaster

One of the most prominent failures due to an invalid assumption made by a software component is that of the *Ariane 5* rocket. The summary of the expert analysis in [2] is as follows - "In about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 meters, the launcher veered off its flight path, broke up and exploded". *Ariane 5* had reused the same software which was used by *Ariane 4*. There was an operand overflow in the Inertial Reference System(SRI) module, due to a conversion of a 64-bit floating point value to a 16-bit signed integer value. The error occurred because the *Ariane 5* was a much bigger rocket than the *Ariane 4* and had a higher value for the horizontal velocity component, which overflowed a 16-bit variable.

Analysis: Though a simplification, this is a classic case of reuse of a software component with invalid assumptions made on the environment and other components in the system. Though this fact "The horizontal velocity value of Ariane 5 is higher than Ariane 4" was known beforehand, it was not documented in a machine-checkable format. Thus, the assumption made by the old software that the horizontal velocity variable will never overflow 16-bits was left unchecked, it was violated and caused the accident.

1.1.2 Child-seat Airbag Incident

The Ariane 5 disaster was an incident where an invalid assumption was made on the validity of the input-output values for a software component. In many cases, due to efficiency concerns and resource constraints in embedded systems, many implicit assumptions are made by real-time software components, and there is no physical representation of these assumptions in the code. And in some cases, the individual software component's (say component X's) code embed the assumptions and these assumptions are not exposed in the software interface. There is no viable method to validate these assumptions other than manually testing the components under varying operating conditions. Thus, any software component that uses component X may not be aware of the assumptions made by X. The following example is a case where an environmental assumption made by a component was not propagated to other components in the system and the assumption was violated.

There was an fatality in a car due to airbag deployment in the child-seat. In short, the following analysis was given for the accident by experts [3]. The airbag controller for the car was designed not to deploy the airbags in a seat in the presence of a child-seat in the particular seat. However, the airbag control system consisted of primary and backup controllers. The (environmental condition of) the presence of the child-seat was known to the primary airbag controller. In certain extreme temperature and humidity conditions, the primary airbag controller relinquishes control and the simpler backup controller takes charge. The backup airbag controller deployed all airbags on impact causing the fatality.

Analysis: There can be two possibilities. The first being - the backup controller having a simple logic of deploying all the airbags on impact, irrespective of the presence of the child-seat and thus causing the fatality. The second being - the backup controller having the capability of individual control of airbags but the environmental assumption of presence of child seat not being exposed explicitly to the backup controller module, and thus causing the fatality.

1.2 Sources of Assumptions

There are various reasons for the origin of invalid assumptions. First, existing software practices for a particular domain are being applied to other domains without suitably altering the practices for the new domain. Software development has evolved into a highly distributed activity. There is a lot of emphasis placed on reuse of software components. Even critical systems like software in rockets and satellites reuse software components rather than build them from scratch. As mentioned by Booch in [4], “One person’s system is another person’s subsystem”. Not all software is being built with the possibility of it being used as a subsystem of a larger system. In other words, software needs to be built with *composition* in mind. This is elaborated in Sections 1.2.1 and 1.2.2. Next, there are limitations in the existing software interfaces that prevent the encoding of assumptions in a manner that enables efficient validation of these assumptions. This is elaborated in Section 1.2.3.

1.2.1 COTS and Custom Software Components

Commercial sector, space and defense organizations alike are moving towards using COTS and custom software components to build complex systems.

For example, companies like BMW, Daimler-Chrysler, Philips Semiconductors, Freescale, and Bosch, are working together to develop and establish FlexRay, a communication system for advanced automotive control applications as the standard for next generation automobile applications. They plan to replace traditional braking systems with electromechanical braking systems (EMB), and this EMB system is to use the FlexRay as the fault-tolerant communication protocol [5]. In effect, the braking systems in different cars will use FlexRay as a COTS software for their messaging sub-system.

This is also the case with multi-million dollar projects which organizations like NASA and Lockheed Martin are undertaking. For example, a small sub-system of a large-scale project, the ground based command system for the NASA's Hubble space telescope consists of 30 COTS components [6].

1.2.1.1 Benefits and Drawbacks of COTS Adoption

This development of widespread adoption of COTS and concurrent software development has the following benefits.

- *Distributed expertise:* Organizations need not incur the cost of developing expertise in all the technologies they will be using to develop a system.
- *Reusable technologies:* Custom software components can be developed to provide solutions which can be used in various products.
- *Faster turn-around time:* The availability of custom software components saves development time.
- *Uniform integration process:* Exposing functionalities as software interfaces ensures a uniform integration process for organizations that use these COTS components.

But these benefits do come at a cost, especially for integrating embedded and real-time systems. COTS and custom software components give rise to black-box interfaces, they expose a traditional *software interface*, which is used to exchange data and most of the assumptions are embedded in the code or documentation for the component which are generally not machine-checkable. This amounts to increased manual testing efforts to ensure that the software runs in various environments and varying assumptions. This is one of the sources of assumptions. Also, a survey conducted as a part of this thesis, on an open-source operating system for embedded sensor network devices, TinyOS [7], revealed that there are more defects that are related to invalid assumptions than algorithmic defects.

1.2.2 Software Engineering Practices

In addition to making the assumptions machine-checkable, it is realized that we need to synchronize assumptions throughout the software life-cycle.

There has been a substantial body of research in the field of requirements engineering. As early as 1987, F. Brooks stated "The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems". There has been traditionally been acceptable synchronization between low-level design (UML diagrams) and code with mature tools [8] available for object-oriented design. But, though there has been some work in formal requirements engineering [9] [10], synchronization between requirements and code has been a very hard problem.

The study conducted on the simple inverted pendulum control system, consisting of four software components, found that there are over forty implicit assumptions made by the software components that are not represented in the software interface [11]. Most of these assumptions need manual validation during system evolution. While every detail is taken into consideration while developing the requirements and designing the components, it is found that very few assumptions can be discerned from the interface for the components, which in many cases, is the only way to examine a component.

The issue of synchronizing and tracking assumptions made across the software life cycle, especially the ones which are not ultimately physically represented as code is addressed in this research. We term this process of tracking the assumptions across the software life-cycle as *vertical assumption tracking*. This is extremely vital if programming paradigms like extreme programming [12] (where continuous refinements are made to requirements and code in tandem) need to be applied to engineering real-time systems software effectively.

1.2.3 Inadequacy of Current Software Interfaces

The inadequacy of current software interfaces is illustrated using a simple example. An acoustic sensor (or *sensor* for short), like a Berkeley MICA mote, is used to sample the acoustic strength in its vicinity and send the perceived strength to a component, *data collector*, residing

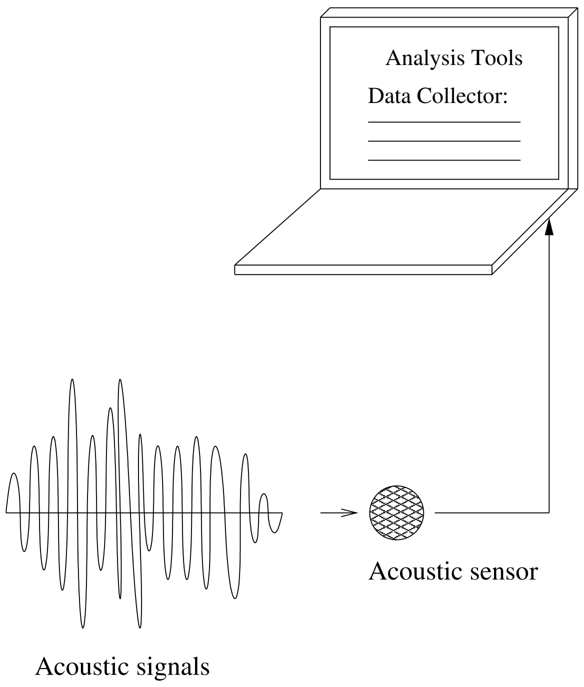


Figure 1.1 A Simple Acoustic Sensor Transmitting Audio Data to a Computer

in a computer it is connected to. The *sensor* can be configured to report values periodically or when the perceived audio signal strength is above a certain threshold value.

The sensor has a simple interface

event dataReady(**uint16_t** dataval)

to report its data values. While we can see that the software interface is very simple, there are a number of environmental and operating assumptions that are not captured by this interface. For example, the following are a few of the assumptions made by the *data collector* on the sensor.

- the units of the acoustic data sent by the sensor
- the maximum sensing delay of the acoustic sensor
- the maximum jitter in sending periodic data values

- the maximum value (say in decibels, if that is the unit of measurement) that the sensor can sense before the A/D converter saturates
- the granularity of the sensor readings.

In addition, there may be complex assumptions which may be related to multiple properties of the *sensor*. For example, the *data collector* may make an assumption that the *sensor*'s saturation value must be less than a limit even when the sensing error is taken into consideration.

We see that the number of assumptions outnumbers the number of parameters in the software interface for this simple example. Also, some assumptions have no representation in the code of the *data collector* or the *sensor*. This warrants a clean vocabulary or a language to represent assumptions and classify them. Cluttering existing software interfaces with assumptions will needlessly burden programmers and may even make programs inefficient.

1.3 Motivation

The motivating factors for this endeavor are (a) the current driving cost in developing embedded and real-time systems (b) the common set of problems faced by commercial, space and defense agencies alike in developing such systems (c) inadequacy of current software interfaces and software engineering practices to efficiently capture and validate assumptions.

The driving cost of developing new products in embedded and real-time systems has shifted from design of efficient algorithms to *integration issues* involved in getting software developed by disparate teams to work together. This is due to custom software development of individual modules by different teams and/or adoption of COTS software. As mentioned in [13], where fifteen different projects were studied, "projects using COTS were obliged to follow a process quite different from traditional projects with more efforts put into requirements, test and integration and less into design and code". Observing the current trend, it is conjectured that majority of the real-time software projects will continue to follow this trend – increased cost and resources towards integration of components developed by different teams or organizations.

Also, there is a common set of problems faced in the integration of embedded and real-time systems, be it a company building products for mass consumption like Ford Motor Company, or an organization like NASA, which builds satellites. This is due to the uniform pattern for building such complex systems by integrating custom components. Sections 1.1.1 and 1.1.2 have given instances of failures faced by automobile industry and international space agencies, which can be traced back to invalid assumptions made by components. They have caused losses in revenue to the tune of hundreds of millions of dollars and loss of lives. After a study of failures of selective set of projects from Risks Digest [14], we have found that the integration of large-scale software has proven to be non-trivial and multi-million dollar software projects have had to be called off. For example, an Australian submarine project was called off with a decision to “completely dump the software and start again, with the new system having less-integrated architecture and utilizing more COTS components” [15].

When embedded system components are shipped as black-boxes with a software interface[1] to access the component's functionality, it makes the task of validating assumptions very hard. Software interfaces in currently widely used programming languages contain syntactic information about the data exchange, while it lacks information about the assumptions made on the data values. It also lacks capabilities for encoding assumptions made on the environment and other components. Encoding environmental and operating assumptions is a fundamental requirement in developing embedded and real-time systems software. This is because real-time systems are very closely tied in with the physical environment they operate in. In many cases, like in sensors, they monitor, collect and/or transmit data about the environment itself. While the actual data collected is critical, they form a small part of the complete software component.

Overall, there is definitely a need for a framework with a well-defined vocabulary to encode assumptions in a machine-checkable format. It should flag the assumptions that are violated in advance and minimize the human effort in validating the assumptions as repeated human-effort is prone to errors. Other related goal is that it should work in tandem with existing software components, and require minimal or no re-engineering effort. As many software components are shipped without revealing the source code, the framework should allow encoding of assumptions without involving modifications to the source code. A need for a framework with similar objectives is independently stressed by the the ‘Mishap Investigation Board’ [16] of the ‘Mars Climate Orbiter’ disaster, and the ‘Inquiry board’ [2] of the ‘Ariane 5’ disaster.

[1]By software interface, we mean the traditional interfaces in languages like C, Java, CORBA, etc., which is used to exchange data between interacting software components.

1.4 Organization of the Report

With this introduction and motivation, the rest of the report is organized as follows.

Chapter 2 provides a high-level overview of the assumptions management framework. It also provides some standard definitions, and lists the challenges in building a framework for managing assumptions.

Chapters 3 – 7 describe the core of the assumptions management framework design and implementation. Chapter 3 describes the classification of assumptions and guarantees and provides a basic vocabulary for assumptions. The basic dimensions of classification of assumptions and guarantees is presented, upon which users can build custom classification schemes. Chapter 4 describes the AMF language design. The language is designed to encode assumptions and provide guarantees in a machine-checkable format. Chapter 5 describes the process of composition of assumptions and guarantees. This process finds the list of unmatched and matched assumptions, which is an important step in building larger systems using smaller sub-systems. Composition is also a pre-requisite for validation of assumptions. Chapter 6 describes how assumption management blends with other aspects of software engineering. AMF integrates assumptions specification with architecture description and the source-code of the components. Chapter 7 describes the implementation of AMF.

Chapters 8 describes the case studies conducted on representative systems. These studies were conducted to check if invalid assumptions caused defects in end-products in these systems. The case-studies also helped analyze the characteristics of assumptions that cause defects and helped refine the design of AMF. Chapter 9 provides the details of analyzing and correcting an invalid assumption of Iperf, a bandwidth measurement tool, which resulted in significant savings in network traffic generated and bandwidth measurement times.

Chapter 10 evaluates the performance and scalability of AMF implementation. The ability of AMF to encode and validate assumptions encountered in the case-study is also evaluated. It also describes the related work and compares it with AMF.

Finally, the conclusions and future applications and extensions of AMF are presented in Chapter 11.

CHAPTER 2

Overview: Definitions, Challenges and the Building Blocks

This chapter provides an overview of the assumptions management framework. Some standard definitions, which are also used in the subsequent chapters, are given in Section 2.1. The challenges involved in designing a framework to manage assumptions is listed in Section 2.2. This is followed by a high-level description of the building blocks of the assumptions management framework (AMF) in Section 2.3.

2.1 Definitions

The example of the acoustic sensor providing sampled acoustic data to a computer connected to it (explained in Section 1.2.3) is used in this section to explain the definitions.

From hereon, a *component* will denote an entity with a well-defined functionality and an interface. The interface allows invocation of the component's functionality and provides access to the output generated by the component. Components can denote software, hardware or composite entities. In the example, *sensor* and *data collector* are components. *data collector* is a software component. *Sensor* is a composite component; sampling acoustic signals at a configured rate via the sensor hardware, and making available the sampled values for export to a software entity, via the sensor software.

The *interface* of a component contains i) a list of parameters with the syntax for each of the parameters needed to invoke the functionality of the component and/or access the output generated by the component and ii) a unique identifier, usually the name of the interface. It is similar to a software interface in programming languages like C or Java. For example, interface in Figure 1.1 is used by the *sensor* to export the acoustic signal samples for the *data collector*. It is specified in nesC, a language for embedded systems software.

An *assumption* made by a component C_1 on a component C_2 is a necessary condition that needs to be satisfied by the C_2 for i) the correct functioning of C_1 and/or the system that contains C_1 or ii) for functioning of C_1 or the system containing C_1 at a desired performance level. For example, for the correct functioning of the *data collector*, the *sensor* needs to provide the acoustic samples in the expected units. Also, the *sensor* needs to provide data at a particular granularity. Providing data at a coarser granularity will not affect the functionality of the *data collector*, but may lead to poor performance.

Other salient points to note about an assumption are that an assumption may not be related to the data-types of the components' interfaces. Also, C_2 can represent the environment in which C_1 resides in. For example, C_1 can represent a networking application and C_2 can represent the operating system on which C_1 runs.

A *guarantee* provides values to evaluate a particular assumption. Based on the values provided, the assumption will be satisfied or not. In a programming language paradigm, if assumptions are analogous to function definitions (returning boolean values) or assertions, guarantees are analogous to calls to the functions.

In systems larger than this example, the components may not directly interact with each other, but may still make assumptions on each other. This will be illustrated in the case study in Section 8.3.3, which deals with a system consisting of four components.

2.2 Challenges in Designing a Framework for Managing Assumptions

Invalid assumptions have caused multi-million dollar failures [2] and in some cases have resulted in loss of human lives [3]. The case studies, described in detail, in Chapter 8 have confirmed that invalid assumptions are indeed a cause of defects in systems whose functionality is tied with the external environment. The three main sources of assumptions mentioned in Section 1.2 are i) Current software engineering practices ii) COTS and custom software components iii) Inadequacy of current software interfaces.

A framework for managing assumptions needs to ensure that the following challenges are met[1].

- Currently, there is a need for a vocabulary for managing assumptions. For example, 'When should an assumption be validated?' In other words, 'what is the time-frame of validity of an assumption?'

- There is a need for a language to encode assumptions in a machine-checkable format that reduces human effort in the long-term.

- If the framework is to be used for the COTS domain, in many cases, there is a requirement that there are minor or no source-code modifications. Source-code modifications may violate certification of critical components. Hence, the framework must be able to work independent of the component's source-code.

- The process of assumptions management should blend into the software-engineering life-cycle rather than requiring a completely new process. Wherever possible, the assumptions should be able to directly reflect the properties exposed by the component's source code and their architecture description. Rather than requiring a new language for the component's source code and architecture description, the framework should ensure that it works together with these existing software engineering aspects.

- As new classes of assumptions are encountered, specific to various domains, the framework should be able to include them seamlessly; *i.e.,* the assumptions management framework itself must be extensible.

- A sample code-base of a medium sized project runs into hundreds of thousands of lines. The framework should be scalable to encode assumptions of this order.

- As complex systems are built using sub-systems, the framework should have the capability of exposing only properties and assumptions that are relevant to a particular context of composition; *i.e.*, the framework should provide support for composing assumptions

[1]The *Ariane 5* 'Inquiry board' mentions the following. "Identify all implicit assumptions made by the code and its justification documents on the values of quantities provided by the equipment. Check these assumptions against the restrictions on use of the equipment." This matches very closely with the objective of the assumptions management framework.

- As with any framework, it should be user-friendly, in terms of providing a graphical user interface and an intuitive language to encode the assumptions.

The following section provides a high-level overview of the design of AMF, which gives the reader an idea of how these challenges are met. Chapters 3 – 6 explain these concepts in further detail. Some of the challenges, like usability, scalability and the ability to encode assumptions, and are related to the implementation of AMF. They are described in Chapters 7 and 10.

2.3 The Building Blocks of AMF

The Assumptions Management Framework (AMF) provides a conceptual foundation for encoding, managing and validating assumptions. AMF builds a vocabulary for assumptions and guarantees with its classification scheme. The language for encoding assumptions helps in (a) allowing assumptions that can be expressed in predicate logic to be recorded in a machine checkable format (b) assisting component developers to provide guarantees for assumptions exposed, since the composition matching algorithm finds matched and unmatched assumptions (and guarantees). AMF allows policies to be set on when to validate assumptions and what assumptions to validate, since the back-end of AMF is designed in a way that it is amenable to database operations. AMF allows assumptions to be specified directly on the source code and the architecture description of the components.

2.3.1 Classification of Assumptions - Building a Vocabulary

In the simple *sensor - data collector* system presented in Section 1.2.3, which consisted only of two components, over a dozen assumptions were encountered. This example is rich enough to explain the basic dimensions of classification that an assumption takes.

The first dimension of classification is the time-frame of validity of an assumption. *Static* assumptions are those, whose validity does not change during the lifetime of the software; for example, the units of acoustic intensity exported by the sensor software. *System configuration* assumptions are those, whose validity changes between executions or they may be specific to the component's environment or hardware properties. For example, the maximum jitter is a characteristic of a particular *sensor* hardware. *Dynamic* assumptions are those, whose validity may change during the execution of the system. For example, there may be a function which detects the aging of the *sensor* hardware during runtime.

The second dimension of classification relates to the criticality of an assumption. *Critical* assumptions are those, whose violation will cause the system core functionality to be compromised or the system to fail; for example - the units of measurement of acoustic data is critical for the correct functioning of the data collector. *Non-critical* assumptions are those, whose violation will not cause the core system functionality to be compromised, but may cause performance degradation and/or compromise of non-core functionality. For example, the granularity of readings affects the performance of the system in which the data collector is in, but does not affect its functionality.

The third dimension of classification relates to abstracting relevant assumptions as systems get larger. In simple terms, while building a larger component by composing a set of smaller sub-components, assumptions are *public* when they need to be exposed as a part of the larger component or when they need to be satisfied by guarantees from external components. Assumptions are *private* when they are satisfied by internal sub-components and are not exposed as a part of the larger component.

Guarantees are classified based on how the values provided by them are obtained. *Human-entered* guarantees are those, whose values are entered by humans explicitly while encoding the guarantees. For example, the sensor error value obtained from a data-sheet will be explicitly entered by a human. *Machine-generated* guarantees are those, whose values are obtained by executing a routine. For example, the operating system name of the sensor proxy will be obtained by executing a routine during or just before its execution. *Hybrid* guarantees are those, which have both human-entered and machine-generated values. For example, if sensor proxy uses TCP/IP, the bound address will be machine-generated and the port number will be a human-entered value.

End-users can build on top of this basic classification scheme to provide domain specific classification information.

2.3.2 A Language for Managing Assumptions

The AMF language reflects AMF system view. In short, AMF views a system as a set of components. A component in a system has a set of dependent components, that it makes assumptions on or provides guarantees for. An assumption has a name, a set of input (formal) parameters, the body of the assumption - a boolean-valued function, and its classification information. A guarantee provides values to evaluate an assumption. The assumption is valid if the values provided by the guarantee evaluates the assumption body to a boolean value - *TRUE*.

The language is explained in Chapter 4. For example, in the *sensor* - *data collector* system, the components are *sensor* and *data collector*. The *sensor* makes assumptions on the *data collector* and vice-versa. An assumption made by the data collector and the corresponding guarantee is encoded as shown.

```
componentDefinitions name=DataCollector {
    about Sensor {
         assumes min_saturation_value(double error, int saturation_value) {
              (1.0 + error) * 100.0 < saturation_value
         }
         {Criticality=CRITICAL_LEVEL_A}
         {ValidityTimeFrame=SYSTEM_CONFIGURATION};
         // Other assumptions and guarantees about the Sensor
     };
     // Assumption Guarantee Sets for other components
};
componentDefinitions name=Sensor {
    about DataCollector {
        guarantees min_saturation_value {
            double error = 0.1;
            int saturation_value = 128;
        }
        {GuaranteeType=HUMAN_ENTERED};
    };
    // Assumption Guarantee Sets for other components
};
```

The logic to express assumptions encompasses most logical operators in a high-level language like C or Java. The language makes provisions for components that may not know the set of dependent components in advance, like operating systems and middleware components, to express their assumptions. AMF also provides extensions to the language to express complex assumptions and guarantees using method invocations in high-level languages.

2.3.3 Composition of Assumptions and Guarantees

As systems get larger, it is important to match assumptions and guarantees formed by internal sub-components, so that the integrator of the larger component is not required to provide guarantees for such matched assumptions. The algorithm for composition needs to take care of library components, like operating systems and middleware that do not know the set of dependent components in advance. Such components expose a set of library assumptions; they require guarantees to be provided for library assumptions from every dependent component that uses the library. For example, an operating system (say Linux) will require that every component that executes on it is compatible with a particular version of the operating system, though it may not know the list of such components in advance. The composition matching algorithm matches the assumptions exposed by components with guarantees from the respective dependent components.

The list of unmatched assumptions helps component developers to provide guarantees for exposed assumptions.

AMF provides a composition matching algorithm that has a linear running time ($\Theta(n_a+n_g)$, for n_a assumptions and n_g guarantees), when there are no library assumptions and guarantees. In presence of library assumptions and guarantees, it has a running time of $\Theta(n_a+n_g+\widetilde{n_a}.n_{\overline{a_x}})$; the additional factor is on the order of the average number of guarantees to be provided per library assumption ($n_{\overline{a_x}}$) times the number of library assumptions ($\widetilde{n_a}$) .

Composition is also a pre-requisite for validation of assumptions. Only matched assumptions can be validated.

2.3.4 Integration with Programming Language and Architecture Description

AMF language specification allows the body of the assumptions to directly invoke routines on the source code of the components. This allows a tight integration between the properties exposed by the source-code of the components and the assumptions based on these properties. Assumption violations are automatically flagged, if changes in the source-code violate any assumptions. Similarly, AMF maintains compatibility with the names and types of components described in the architecture description with the names of components in assumption definitions. It flags component names used in assumption definitions that do not have a corresponding definition in the architecture description. AMF also flags assumptions that are made between components whose types are not compatible.

For example, if there are two components, *Kernel* and *InitRoutines* developed by different developers, then if the *Kernel* exposes a property in its source code, the *InitRoutine* module can make an assumption directly based on this property. A snippet of the code and the assumption are as shown below.

```
// Source code of Kernel component
public int getInitSleepInterval() {
     return INIT_INTERVAL;
}
// Assumption definitions
componentDefinitions name=Kernel {
    about InitRoutines {
        assumes maxStartupDelay(int maxDelayInMilliSec) {
            maxDelayInMilliSec <= Kernel.getInitSleepInterval();
        }
        // Classification info ...
    };
};
```

Also, in some cases, invoking methods in the source code becomes necessary. The values for certain guarantees cannot be entered while encoding the assumptions and guarantees. The values need to be obtained before or during execution of the component. For example, if there is an assumption that a component must run on Linux, this guarantee cannot be encoded before hand. Before execution, the component provides the name of the operating system by executing a routine to obtain the operating system name. The guarantee is encoded as shown below.

```
componentDefinitions name=Kernel {
    about * {
        guarantees operatingSystemName {
            String osName = Lib.getOsName();
        }
    };
};
```

2.3.5 Policies on Assumption Selection and Validation

AMF is designed to store all the assumption definitions in an XML format, which is amenable to database operations. One of the case studies on an inverted pendulum control system (explained in Appendix A.1), consisting of only four components uncovered over forty assumptions. Hence, setting policies on when to validate assumptions and what assumptions to validate is vital, as systems get larger.

For the *sensor - data collector* system, examples of useful policies are (a) validate all system-configuration changes of the sensor and data-collector, when the sensor hardware changes. (b) validate all the critical assumptions of the sensor and data-collector (when these components are a part of a larger sub-system and validating all assumptions is costly).

Using tools like XPath [17], setting policies on fetching assumptions and validating a relevant subset of assumptions is made possible.

2.4 Summary

This chapter provided the key definitions for the following terms - component, interface, assumption and guarantee. It listed the challenges in designing a framework for managing assumptions. It provided a high-level description of the building blocks of AMF - the classification scheme of assumptions and guarantees, the language, composition of assumptions and guarantees, integration of assumption definitions with software engineering aspects like source code and architecture description and policies on assumption selection and validation.

CHAPTER 3

Classification of Assumptions and Guarantees

The *sensor data-collector* system presented in Section 1.2.3 had one parameter and one return value in its exposed interface, but over a dozen assumptions were uncovered[1]. The simple acoustic sensor system was an actual sub-system of a larger system that was used to classify objects like persons, persons with metallic objects and large vehicles in a field using acoustic, magnetic and infra-red sensors [18]. One can see how the number of assumptions in such a complex system can grow rapidly. The inverted pendulum control system (explained in Appendix A.1) with four defined interfaces for components had over forty assumptions when invalidated could cause the system to fail. This warrants that the assumptions are classified for easier manageability.

This chapter provides a basic classification of assumptions based on the assumption's time-frame of validity, scope and criticality. Also, guarantees are classified as human-entered or machine-generated based on how the values for the guarantees are obtained. Every assumption or guarantee will take on this basic dimensions of classification. Domain experts can optionally build a domain-specific classification on top of this scheme, which also briefly discussed at the end of this chapter.

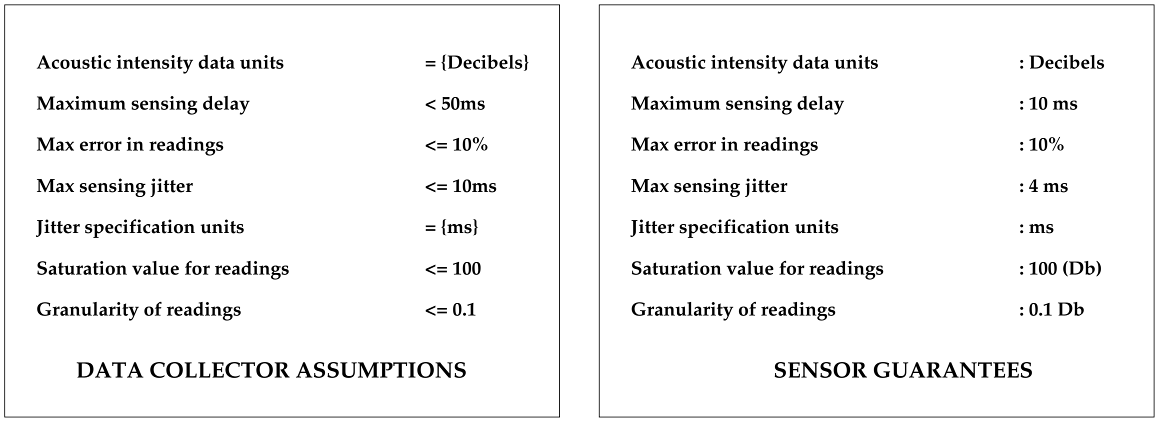
3.1 The Assumptions Interface or the Assumption Set

Recall that, for the acoustic sensor system presented in Section 1.2.3, the *sensor* component has a simple interface to report its data values to the *data collector*. This interface is as shown below.

[1]This trend was also observed in the case studies that were conducted.

event dataReady(**uint16_t** dataval)

There are a number of assumptions made by the *data collector* on the *sensor* for its functioning. The *sensor* needs to provide matching guarantees to the *data collector* for the correct functioning of the *data collector* or the system in which the *data collector* is a part of. The assumptions exposed by the *data collector* for the *sensor* will form the *assumption set* of the *data collector* (for the *sensor*). The formal definition is given below.



Acoustic intensity data units	**= {Decibels}**
Maximum sensing delay	**< 50ms**
Max error in readings	**<= 10%**
Max sensing jitter	**<= 10ms**
Jitter specification units	**= {ms}**
Saturation value for readings	**<= 100**
Granularity of readings	**<= 0.1**

DATA COLLECTOR ASSUMPTIONS

Acoustic intensity data units	**: Decibels**
Maximum sensing delay	**: 10 ms**
Max error in readings	**: 10%**
Max sensing jitter	**: 4 ms**
Jitter specification units	**: ms**
Saturation value for readings	**: 100 (Db)**
Granularity of readings	**: 0.1 Db**

SENSOR GUARANTEES

Figure 3.1 Assumption Set Example

The *assumption set* of a component C_1 for a component C_2 consists of a set of assumptions exposed by C_1 that need to be satisfied by guarantees provided by C_2. For example, the *assumption set* of the *data collector* for the *sensor* is shown in Figure 3.1.

This *assumption set* will be used as an example to explain the various dimensions a particular assumption and guarantee can take.

3.2 Classification of Assumptions

3.2.1 Time-frame of Validity of Assumptions

While designing embedded and real-time systems, not all assumptions need to be validated at all times. The validity of some of the assumptions changes only when the software is changed. The validity of some assumptions never changes per mission or a single execution of the system, but may change between executions. The validity of some assumptions changes during execution, but at a rate much lower than the real-time data flow. Based on this observation, we have the following classification of assumptions.

1. **Static assumptions:** These are assumptions whose validity does not change during the lifetime of the software. For example, the units in which the acoustic intensity is exported to the data collector is a property of the *sensor* software.

2. **System configuration assumptions:** These are assumptions whose validity does not change during a mission or execution of the system. They may change between the executions or they may be specific to the hardware environment. For example, the maximum jitter is a characteristic of a particular *sensor* hardware. Different hardware will guarantee different values for this parameter. This will have to be configured per hardware.

3. **Dynamic assumptions:** These are assumptions whose validity may change during the execution of the system. For example, there may be a function which detects the aging of the *sensor* hardware. The validity of the readings will be based on the result of this function. Since, this may happen during the mission or execution of the system, the related assumption is a dynamic assumption.

This dimension of classification is vital in increasing the efficiency of assumption checks. For example, *static* assumptions check can be triggered by a change in the software components in the system. *System configuration* assumptions checks can be triggered whenever the hardware or the environment of the system changes, or just before the execution begins. *Dynamic* assumptions may need to be validated using run-time monitors. Next, different kinds of tools need to be used to ensure the validity of different classes of assumptions within this dimension. For example, dynamic assumptions require techniques like monitor oriented programming [19], or require changes in the source-code of the components. *Static* and *system configuration* assumptions can be checked offline or just before the component begins its execution.

3.2.2 Criticality of Assumptions

In critical systems, components in the system will have a core functionality and if a component is unable to carry out its core functionality, it will be declared to have failed. In safety-critical systems, not satisfying the core functionality may also affect the safety of the system. We have seen that when some assumptions are violated, the system core functionality is compromised or the system fails. For example, the assumptions on the acoustic intensity data units and saturation value for *sensor* need to be satisfied by the data collector for the correct functioning of a classification system that uses these samples.

There are some assumptions, when violated, do not cause the core functionality of the system to be compromised, but may cause noticeable performance degradation. For example, if the granularity of the sensor readings is different from the assumed value, it causes a performance degradation in the classification system that the *data collector* is a part of, but the core functionality is not compromised.

This observation leads us to the following classification of assumptions. The particular classification system described below is chosen to ensure that the criticality levels can easily be translated to avionics software certification standards like DO-178B [20].

1. **Critical assumption:** These are assumptions, whose violation will cause the system core functionality to be compromised or the system to fail. This assumption will have a rank of A, the highest rank.

2. **Non-critical assumption:** These are assumptions, which when violated will not cause the core system functionality to be compromised, but there may be performance degradation and/or compromise of non-core functionality. Users and system designers can rank the criticality from B through E, with B being the highest rank.

Since the number of assumptions in systems can be very high, it is not always feasible to validate all of them during different phases of development. A prioritization according to criticality permits critical assumptions to be validated first, despite the fact that full certification may require complete validation. This classification can also serve as an input to some dependency management tools that check for the net impact on the system, when a particular assumption is violated. For example, a dependency management tool can check if violation of a non-critical assumption will cascade into a violation of a critical assumption. This classification can also be an input to the tools providing or certifying service gradations.

3.2.3 Compositional Scope of Assumptions

The third basic dimension of classification of an assumption is its *compositional scope.* From hereon, *scope* will be used to denote *compositional scope.* As it is found that the number of assumptions grow rapidly with increasing number of components, it is useful to classify assumptions according to their scope to simplify assumptions management for the system integrators and developers. This will be very useful in the context of composition of assumptions, a concept that will be discussed in Section 5.2.

In simple terms, assumptions are *public* assumptions when they need to be satisfied by external components and *private* assumptions are satisfied by internal sub-components.

Given a system consisting of many components, AMF automatically finds assumptions made by internal components that are matched by guarantees from the dependent components. These assumptions are classified as *private* and the rest of the assumptions that need to be satisfied by external components are classified as *public.* The algorithm for composing assumptions [11] is described in Section 5.2.

3.3 Guarantees: Machine Generated Versus Human Entered

The guarantees provided to evaluate an assumption can be values entered directly by the developer or architect or they can be routines that obtain the values during system-configuration time or runtime.

For example, if there is an assumption that the *data collector* module needs to run on a Linux machine, the assumption can be encoded during system development time. The guarantee needs to be evaluated just before the *data collector* module is started to ensure that it runs on Linux. Hence, the guarantee cannot be a value directly entered by the developer, it will be a routine which will be executed to obtain the operating system type.

On the other hand, there may be properties that cannot be explicitly obtained by a routine and the values need to be entered by the developer or architect. For example, the computations within the *data collector* may assume that the *sensor* data exported is in *Decibels.* This property may be embedded in the sensor software and may need to be explicitly entered by the developer or architect of the sensor software.

Based on these observations, we classify guarantees as:-

- *Human-entered guarantees:* These are guarantees whose values are explicitly entered by the user (architect or developer) of the system.
- *Machine-generated guarantees:* These are guarantees, for which, the developer or the architect encodes the routine to obtain the values. The actual values are obtained by executing the routing during system-configuration, system evolution or runtime.
- *Hybrid guarantees:* These are guarantees where some of the values are explicitly entered by humans (human-entered) and some of the values are encoded as routines and the actual values are obtained by executing the routines (machine-generated).

This classification for guarantees is useful in prompting users to provide guarantees values for human-entered guarantees whenever there are changes in the relevant component. Also, it simplifies and reduces errors in the user-interface design of the assumptions management framework. End-users of the framework are forbidden to provide direct values for machine generated guarantees and the parameters that are machine-generated for hybrid guarantees.

3.4 Revisiting the *Sensor - Data Collector* Example

With the classification scheme in place, Figure 3.2 shows how the assumptions made by the *data collector* on the sensor will be classified. The context of composition is assumed to be the *sensor - data collector* system. Thus, all assumptions are assumed to be public assumptions.

3.5 Domain Specific Classification of Assumptions

The above classification system forms the base for assumptions classification in AMF. AMF allows domain-specific assumptions to be built on top of these basic dimensions of assumptions. For example, domain experts from real-time systems can build a timing and scheduling related assumptions library.

3.6 Summary

This chapter presented the basic classification of assumptions and guarantees. The concept of assumption set is also introduced. The basic dimensions of classification for an assumption

Assumption - Guarantee	**Parameters**	**Expression**	**Classification**
Intensity data units	intensity_units	A: intensity_units.equals("Decibels") G: intensity_units: "Decibels"	A ∈ Static, CriticalLevelA G ∈ Human-entered
Max delay	max_delay	A: max_delay < 50 G: max_delay: 10	A ∈ System Configuration, CriticalLevelA G ∈ Human-entered
Max error	max_error	A: max_error < 0.15 G: max_error: 0.1	A ∈ System Configuration, CriticalLevelA G ∈ Human-entered
Max jitter	max_jitter	A: max_jitter < 10 G: max_jitter: 4	A ∈ System Configuration, NonCriticalLevelB G ∈ Human-entered
Delay (and jitter) units	delay_units	A: delay_units.equals("ms") G: delay_units: ms	A ∈ System Configuration, CriticalLevelA G ∈ Human-entered
Saturation value	error, max_saturation	A: (1+error)*100 < max_saturation G: error: getErr(), max_saturation: 125	A ∈ System configuration, CriticalLevelA G ∈ Hybrid
Granularity	granularity	A: *granularity* <= 0.2 G: granularity: 0.1	A ∈ System configuration, NonCriticalLevelB G ∈ Human-entered
OS	osName	A: osName.contains("Linux") G: osName: getOsName()	A ∈ System configuration, CriticalLevelA G ∈ machine-generated

Figure 3.2 Assumptions and Guarantees Classification Example

are (i) time-frame of validity - static, system configuration or dynamic; (ii) criticality - critical and four levels of non-critical; and (iii) scope - public and private. Guarantees on the other hand can be machine-generated, human-assisted or hybrid. These dimensions of classification help in managing the assumptions better, enables a more efficient validation scheme and are helpful in the context of composition. While every assumption and guarantee takes these basic dimensions, domain specific classification schemes can be built on top of the basic dimensions.

CHAPTER 4

AMF Language Design

One of the primary objectives of AMF is to encode assumptions in a machine-checkable format. This chapter lists the pre-requisites for encoding assumptions for a system of components. The AMF system view is presented, and the AMF language design reflects the AMF system view. After a comprehensive example of the AMF system view, the important language rules of AMF are presented, with examples of assumption definitions based on these rules.

4.1 Pre-requisites for Encoding Assumptions Using AMF

There are some pre-requisites for applying AMF to a system.

- The system (to which AMF is applied) can be partitioned into a well-defined set of components.
- As per the definition of a component in Section 2.1, a component should have a well-defined functionality, and an interface using which we can invoke the functionality and access the results.
- Encoding *static* and *system-configuration* assumptions for a component does not warrant access to the source code of the component; although AMF has provisions to encode assumptions related to properties in the source-code of the component and automatically track them and validate them when these properties change.
- Encoding *dynamic* assumptions in a software component necessitates changes in the execution logic of the component, to insert calls to test the validity of the assumptions in appropriate places in the source-code.

- If AMF is used in conjunction with an architecture description language , with facilities to track changes in the system architecture (like AADL [21]), AMF allows assumptions between hardware and software components to be encoded. Changes is the system configuration can automatically trigger assumption checks.

4.2 AMF Language Capabilities

AMF allows encoding of assumptions in predicate logic. The AMF language is designed to encode assumptions for both static and dynamic architectures. In static architectures, every component in the system has a fixed set of dependent components and assumptions are encoded directly for every dependent component. In dynamic architectures, the components in the system may not be aware of the set of dependent components in advance (while encoding assumptions).

The (basic) dimensions of the assumptions are encoded along with the assumptions itself.

The AMF compiler provides a fairly rich expressive power for encoding the body of the assumption - most logical expression in C or Java, barring expressions involving pointer arithmetic, returning a boolean value is permissible within AMF syntax. This includes function invocations to obtain properties at system-configuration, code compilation or runtime.

For dynamic architectures, or systems where a component may not know in advance the set of other components that may use its services, a component may expose its assumptions and guarantees as a library set of assumptions and guarantees. This allows AMF to be used for applications such as middleware and operating systems.

4.3 Definitions and Structure of Assumption Management Objects

The set of notations used in this section are given in Figure 5.2. As a general mnemonic, c will denote a component, a an assumption and g a guarantee. The sets of components, assumptions and guarantees and will be denoted using the respective upper-case alphabets.

The system as viewed by AMF consists of a set of n components

$$C = \{c_i \mid \quad c_i\ is\ a\ component,\ 1 \leq i \leq n\} \tag{4.1}$$

Notation	**Meaning**
$C = \{c_1, \ldots, c_n\}$	Set of n components. Elements of C are individual components.
$\overline{C_i} = \{c_{i_1}, \ldots, c_{i_m}\}$	The dependent component set for a component c_i. Every $c_{i_j}, 1 \leq j \leq m$, that $\in \overline{C_i}$ also $\in C$.
A_{i_j}	The set of assumptions that the component c_i makes on the dependent component c_{i_j}
G_{i_j}	Guarantee set analogous to A_{i_j}
a_k	an assumption
g_k	a guarantee
n_a, n_g	Total number of assumptions and guarantees in the system respectively

Figure 4.1 Notations in AMF Rules

Every component c_i in the system has a dependent component set with m dependent components

$$\overline{C_i} = \{c_{i_j} \mid \ c_{i_j} \ is\ a\ dependent\ component\ of\ c_i,\ c_{i_j} \in C,\ c_{i_j} \neq c_i,\ 1 \leq j \leq m\} \tag{4.2}$$

The component c_i makes assumptions on or provides guarantees for every dependent component $c_{i_j}, (1 \leq j \leq m)$ in the set $\overline{C_i}$. A component is allowed to have no elements in the dependent component set, $\overline{C_i}$ can have a value of $\emptyset$. Every component in the dependent component set $\overline{C_i}$ is also contained in the set of components of the system C,

$$\overline{C_i} \subset C \tag{4.3}$$

The set A_{i_j}, where each element of the set is an assumption made by c_i on component c_{i_j} referred to as the assumption set from c_i to c_{i_j},

$$A_{i_j} = \{a_k \mid \ a_k \ is\ an\ assumption\ by\ c_i\ on\ c_{i_j}, 1 \leq k \leq p\} \tag{4.4}$$

The set G_{i_j}, where each element of the set is a guarantee provided by c_i for c_{i_j} referred to as the guarantee set from c_i to c_{i_j},

$$G_{i_j} = \{g_k \mid \ g_k \ is\ a\ guarantee\ from\ c_i\ for\ c_{i_j}, 1 \leq k \leq q\} \tag{4.5}$$

There are some restrictions on A_{i_j} and G_{i_j}. For every dependent component, there is at least an assumption or a guarantee,

$$A_{i_j} \cup G_{i_j} \neq \emptyset \tag{4.6}$$

Hence, the sets A_{i_j} and G_{i_j} cannot simultaneously be $\emptyset$.

An assumption consists of

- a name, f
- a list of parameters $(x_1, \ldots, x_r)$ with their type information,
- a predicate or the body of the assumption (a boolean-valued function) $f(X^r) \rightarrow \{TRUE, FALSE\}$,
- classification information, a set of ordered pairs indicating the classification dimension name and a value

A guarantee consists of

- name of the assumption to which this guarantee is for, f
- a list of parameter values (with type information) $(val_1, \ldots, val_r)$ used to evaluate the predicate in the assumption,
- classification information ordered pairs.

If $(val_1, \ldots, val_r)$ are the values provided for $(x_1, \ldots, x_r)$, by a guarantee for an assumption named f, the assumption is said to be valid if $f(val_1, \ldots, val_r) = TRUE$. Otherwise, the assumption is said to be invalid.

An important property of the predicate in the assumption, (which will be useful during implementation of the predicate) is that it is a pure-function. It does not modify any of the parameter-values nor have any side-effects. This will enable the assumptions to be checked during system-configuration or runtime without modifying the properties of the system whose assumptions are being validated.

In summary, AMF language is designed in a way that it views the system to be a set of components, $C = \{c_1, \ldots c_n\}$. Each component c_i has a dependent component set $\overline{C_i} = \{c_{i_1}, \ldots, c_{i_m}\}$, whose elements (components) on which c_i makes assumptions on or provide guarantees for. For each dependent component c_{i_j}, there is an assumption set $A_{i_j} = \{a_1, \ldots, a_p\}$ and a guarantee set $G_{i_j} = \{g_1, \ldots, g_q\}$, at least one of which is not $\emptyset$. Assumptions have a name, list of formal parameters, a boolean-valued predicate along with the classification information.

Guarantees have the name of the assumption to which they provide guarantees for, a list of formal-parameter instances to evaluate the assumption and the classification information. The assumption holds good if the function is evaluated to true with the values provided by the guarantee.

4.3.1 Assumptions of Library and Services Objects

Notation	**Meaning**
$\widetilde{A_i}$	Set of library (mandatory) assumptions of a component c_i.
$\widetilde{a_k}$	A library (mandatory) assumption
$\widetilde{G_i}$	Set of library (optional) guarantees of a component c_i
$\widetilde{g_k}$	A library (optional) guarantee

Figure 4.2 Extended Notations in AMF for Library and Service Objects

Some library and services components like operating system or middleware services may not know the list of components that utilize their services in advance (at the time of encoding the assumptions). Hence, such a component, c_i, must be able to specify assumptions that must be satisfied by every other component that uses the services of c_i. These assumptions are called as *mandatory assumptions*. Also, c_i must be able to provide guarantees that any component that uses the services of c_i can optionally use. These guarantees are called *optional guarantees*.

Hence a component c_i, that is a library or a service, can specify a set of (p_m) mandatory assumptions,

$$\widetilde{A_i} = \{\widetilde{a_k} \mid \widetilde{a_k}\ is\ a\ mandatory\ assumption\ of\ c_i, 1 \leq k \leq p_m\} \tag{4.7}$$

Similarly, c_i can specify a set of (q_o) optional guarantees,

$$\widetilde{G_i} = \{\widetilde{g_k} \mid \widetilde{g_k}\ is\ an\ optional\ guarantee\ of\ c_i, 1 \leq k \leq q_o\} \tag{4.8}$$

4.4 A System View Example

The *sensor data-collector* example is used in this section to help follow the definitions in Section 4.3.

This is a simple system with two components, the *sensor* and the *data-collector.*

$$c_1 = sensor$$
$$c_2 = data\ collector$$
$$C = \{sensor, data\ collector\}$$
$$n = 2$$

The *data-collector* (c_1) makes assumptions on the *sensor* (c_2). The number of dependent components of the data-collector, $m = 1$.

$$\overline{C_2} = \{c_1\}$$

The *data collector* makes the following assumptions (as shown in the Figure 3.2) on the sensor. We just show the names below. For the body of the assumption, formal-parameters and the classification information, please refer to Figure 3.2.

$a_1 : name = intensity\_data\_units,\ parameters : \quad x_1 = intensity\_units(String)$

$\quad body = intensity\_units.equals(\text{“}Decibels\text{”})$

$\quad classification = \{\{Validity\_time\_frame, \text{“}Static\text{”}\}\{Criticality, \text{“}Critical\_Level\_A\text{”}\}\}$

$a_2 : name = max\_delay, \ldots$

$a_3 : name = max\_error, \ldots$

$a_4 : name = delay\_jitter\_units, \ldots$

$a_5 : name = saturation\_value, \ldots$

$a_6 : name = granularity, \ldots$

$a_7 : name = os, \ldots$

$\mathbf{A_{2_1}} = \{\mathbf{a_1}, \ldots, \mathbf{a_7}\}$

Similarly, there are guarantees provided by the *sensor* to the *data collector.* Again, refer to

Figure 3.2 for the complete list of guarantees, with the classification information.

$$g_1 : name = intensity\_data\_units,\ values : intensity\_units(String) : \text{“}Decibels\text{”}$$
$$classification\{\{Guarantee\_Type, \text{“}Human\_entered\text{”}\}\}$$
$$g_2 : name = max\_delay, \ldots$$
$$\vdots$$
$$g_7 : name = os, \ldots$$
$$\mathbf{G_{1_2}} = \{\mathbf{g_1}, \ldots, \mathbf{g_7}\}$$

There are guarantees provided by the *data collector*, like the software version number and the machine-architecture for which the software is built, which any component that uses the services of the *data collector* can make use of.

$$\widetilde{g_1} : name = version, values : minor(int) : 2, major(int) : 3$$
$$classification\{\{Guarantee\_Type, \text{“}Human\_entered\text{”}\}\}$$
$$\widetilde{g_2} : name = architecture, \ldots$$
$$\widetilde{\mathbf{G_1}} = \{\widetilde{\mathbf{g_1}}, \widetilde{\mathbf{g_2}}\}$$

4.5 The Language and the AMF Compiler

The AMF language syntax allows expressing the structure of assumptions presented in Equations 4.1 – 4.8.

The most important language rules for the AMF compiler is shown below in Bachus-Naur form[1].

- The definitions consists of the component definitions of one or more components. $C = \{c_i \mid c_i\ is\ a\ component,\ 1 \leq i \leq n\}$. Every component is identified by a unique name.

  AMF Compiler Rules:

  ```
  componentDefinitionsSet: (componentDefinitions)+ ;
  componentDefinitions: ‘componentDefinitions’ nameClause componentBody ;
  nameClause: ‘name’ ‘=’ IDENTIFIER ;
  ```

[1]This section does not deal with the implementation of the language. Also, it does not list the comprehensive set of rules. For the complete set of rules in BNF, please refer to Appendix B.

Example:

```
componentDefinitions name=DataCollector
// data-collector definitions body ...
componentDefinitions name=Sensor
// sensor definitions body ...
```

- The dependent component set for a component: $\overline{C_i} = \{c_{i_j} \mid c_{i_j}\ \textit{is a dependent component of}\ c_i,\ c_{i_j} \in C,\ c_{i_j} \neq c_i,\ 1 \leq j \leq m\}$. Each c_{i_j} has an *about* clause in AMF language.

  AMF Compiler Rules:

```
componentBody: ‘{’ (assumptionGuaranteeSet)+ ‘}’ ‘;’ ;
assumptionGuaranteeSet: ‘about’ IDENTIFIER assumptionGuaranteeSetBody
```

Example:

```
componentDefinitions name=DataCollector {
    about Sensor
    // body of the assumptionGuaranteeSet for the sensor
}
...
```

- Equations 4.4, 4.5 and 4.6 state that there is an assumption-set and a guarantee-set for a component c_i on its dependent component c_{i_j}, at least one set of which is not empty.

  AMF Compiler Rules:

```
assumptionGuaranteeSetBody: ‘{’ (assumption | guarantee)+ ‘}’ ‘;’ ;
assumption: ‘assumes’ IDENTIFIER ‘(’ formalParamList ‘)’
             assumptionBody
             classificationInfo ;
guarantee: ‘guarantees’ IDENTIFIER guaranteeBody classificationInfo ;
```

Example:

```
componentDefinitions name=DataCollector {
    about Sensor {
        assumes intensity_data_units( /* formalParamList */ ) {
                  // assumption body
        }
        // classification info for intensity_data_units
        // other assumptions and guarantees
    }
} ...
```

- The basic data types of the formal-parameters supported by AMF are *byte, short, int, long, float, double, char, boolean* and *String*. These types have the usual semantics of the respective types in the Java programming language. Provisions are made in the AMF compiler to build complex data types based on these basic types.

- The precedence relationship (and the appropriate operands) for the operators in AMF are similar to those in a high-level programming language like Java or C. *In addition to these operators, AMF also supports method invocation as an operator; this will be dealt in detail in subsequent chapters.* The basic operations supported by AMF in increasing order of precedence are as follows.

  1. Logical OR '||'
  2. Logical AND '&&'
  3. Bit-wise OR '|',
  4. Exclusive OR '̂',
  5. Bitwise AND '&',
  6. Equality '=', Inequality '! ='
  7. Less-than '<', Greater-than '>', Less-than-or-equal-to '<=', Greater-than-or-equal-to '>='
  8. Left-shift '<<', Signed right-shift '>>', Unsigned right-shift '>>>'
  9. Addition '+', Subtraction '-',
  10. Multiplication '*', Division '/', Modulo '%',
  11. Unary minus '-', Unary plus '+',
  12. Bit-wise compliment '~' , Logical NOT '!',

The important compiler rules for parameters, types and the body of an assumption and guarantee is given below. A *logicalORExpression* is a recursive definition that includes all operations listed here. Please see the Appendix B for the complete set of rules.

AMF Compiler Rules:

```
formalParamList : ‘(’ paramDecl (‘,’ paramDecl)* ‘)’ ;
paramDecl : paramType IDENTIFIER ;
paramType : ‘byte’ | ‘short’ | ‘int’ | ‘long’ | ‘float’ | ‘double’
            | ‘char’ | ‘boolean’ | ‘String’ ;
assumptionBody : ‘{’ logicalOrExpression ‘}’ ;
guaranteeBody : ‘{’ (paramType IDENTIFIER ‘=’ paramValue ‘;’)+ ‘}’ ‘;’ ;
classificationInfo : (criticalityClause)? (validityTimeFrameClause)?
                     (scopeClause)? ;
validityTimeFrameClause : ‘{’ ‘ValidityTimeFrame’ ‘=’ timeFrameConsts ‘}’ ;
timeFrameConsts : ‘STATIC’ | ‘SYSTEM_CONFIGURATION’ | ‘DYNAMIC’ ;
// Other classificationInfo clauses.
```

Example:

```
componentDefinitions name=DataCollector {
    about Sensor {
         assumes min_saturation_value(double error, int saturation_value) {
              (1.0 + error) * 100.0 < saturation_value
         }
         {Criticality=CRITICAL_LEVEL_A}
         {ValidityTimeFrame=SYSTEM_CONFIGURATION};
         // Other assumptions and guarantees about the Sensor
     };
     // Assumption Guarantee Sets for other components
};
componentDefinitions name=Sensor {
    about DataCollector {
        guarantees min_saturation_value {
            double error = 0.1;
            int saturation_value = 128;
        }
        {GuaranteeType=HUMAN_ENTERED};
    };
    // Assumption Guarantee Sets for other components
};
```

- AMF allows specification of library (mandatory) assumptions and library (optional) guarantees as in Equations 4.7 and 4.8 for components that may not know the set of dependent components in advance. The dependent component set rule is modified as follows which allows this. The example states that every component that uses the services of the data-collector must guarantee that it supports 32-bit data structures. Also, any component can optionally use the guarantee that the data-collector holds a buffer of 1024 values.

AMF Compiler Rules:

```
assumptionGuaranteeSet: ‘about’ (IDENTIFIER|‘*’) assumptionGuaranteeSetBody
```

Example:

```
componentDefinitions name=DataCollector {
    about * {
        assumes dataImportExportArch(int widthInBits) {
            widthInBits >= 32;
        }
        {Criticality=CRITICAL}
        {ValidityTimeFrame=SYSTEM_CONFIGURATION};
        guarantees dataBufferSize {
            int size = 1024;
        }
        {GuaranteeType=HUMAN_ENTERED};
    }
} ...
```

4.6 Language Extensions to Support Method Invocation

In the *sensor - data collector* example, the *data collector* needs to run on Linux. The *data collector* is a part of a larger classification system. There is an environmental assumption made by the system that the operating system is Linux. This guarantee cannot be provided at the time of encoding the assumptions, it needs to be checked during the system configuration time.

Next, if the error of the sensor and the saturation value is provided in decibels, the data-collector may need to normalize the error value calculate the Gaussian error. These type of mathematical functions are available as libraries in high-level programming languages like Java. It is appropriate to use them instead of providing an implementation in AMF that replicates this function.

For these reasons, AMF provides method invocation as a language extension. AMF also provides optional declarations to enable declaration of a language and importing libraries. Using the language extension, values for guarantees can be obtained by invoking a high-level programming language method at system-configuration or runtime. Complex assumptions can make use of standard or custom library functions in high-level programming languages.

Please see Appendix B for the rules for method invocation. An example of an assumption and a guarantee is given below.

```
componentDefinitions name=DataCollector {
    { language=JAVA;
      import edu.uiuc.cs.rtsl.amf.AMFLibrary;
    }
    about * {
        guarantees operatingSystem {
             String osName = AMFLibrary.getOsName();
        }
    }
    about Sensor {
        assumes maxGaussianError(double errInDb, double saturationInDb) {
            AMFLibrary.normalize(errInDb,saturationInDb) < 0.15;
        }
    }
}
```

4.7 Summary

This chapter explained the AMF language definitions. AMF views the system as a set of components. Every component has a set of dependent components that it makes assumptions on or provides guarantees for. An assumption has a uniquely identifiable name, a set of input parameters (each with a pre-defined type and a name), a boolean-valued function and classification information. A guarantee has a name of the assumption for which it provides values; a type, name and value for each corresponding parameter of the assumption.

The logic to express the assumptions encompasses most logical operators in a high-level language like C or Java. AMF also allows components that may not know the set of dependent components in advance, like operating systems and middleware components, to express their assumptions. These assumptions must be satisfied by every dependent component. Also, such components may provide guarantees that dependent components may optionally use.

AMF has language extensions that makes provisions for complex assumptions and guarantees to be expressed as method invocations in high-level languages.

CHAPTER 5

Composition of Assumptions and Guarantees

One of the prime objectives of AMF is to introduce the concept of manageability of assumptions. The inverted pendulum control system (IPCS) had over forty assumptions with just four components. The system exposed three software interfaces (sensor, controller and the actuator). The *sensor - data collector* example had over a dozen assumptions with a single software interface between the *sensor* and the *data collector*. One can see how the number of assumptions will explode as systems get larger. The next generation of software must be capable of handling system of systems. A recent study [22] states that the software challenge of the future is to manage systems of systems whose size may run into a billion lines of code. The system's architect must be abstracted from low-level and unwanted details of sub-systems. At the same time, one must be able to validate the functionality of each of the sub-systems, preferably automatically.

Automatically matching assumptions with guarantees and validating a relevant subset of assumptions is vital from an assumptions management perspective. This will warrant being able to set policies on the set of assumptions. For example, we must be able to automatically validate all the *static* assumptions of a component, when the code for the component changes.

In short, managing assumptions consists of (a) abstracting internal assumptions of subcomponents in the larger context of integration (b) being able to validate a relevant subset of assumptions in an efficient manner (c) allow for the evolution of assumptions themselves [11].

5.1 Composition Overview

Composing components involves combining a group of related components into a new entity. The new entity will have a well-defined functionality that uses the services of its sub-components and has a well-defined interface to invoke this new functionality. It is similar to the faćade pattern [23] in the software-engineering paradigm. A few properties of the new entity can be directly obtained from its sub-components. Other properties need to be generated based on the properties of the sub-components. This whole process is termed as *composition of components.*

From an assumptions management perspective, composition involves the following.

- Assumptions are matched with their respective guarantees and all assumptions that have matching guarantees that can be validated are tagged as *private.* The integrator of the larger sub-system, of which the new entity is a part of, is abstracted from these *private* assumptions.

- All assumptions made by the component on external components and those not having matching guarantees are tagged as *public* assumptions. The integrator of the larger sub-system needs to be aware of these *public* assumptions.

- In presence of library components, the matching process for assumptions and guarantees needs to follow the algorithm in Section 5.2.

- New assumptions and guarantees are generated to reflect the properties of the composed entity.

For example, as shown in Figure 5.1, the sensor, data collector along with the data analyzer and validator are composed to form the acoustic sub-system. This sub-system is a part of a larger classification system, that classifies objects based on readings from acoustic, magnetic and infra-red sensors. In the acoustic sub-system, most of the assumptions (system-configuration and static) made by the *data collector* on the *sensor* can be matched and validated within the sub-system itself. If required, new assumptions and guarantees based on the assumptions and guarantees of the set of components being composed are generated.

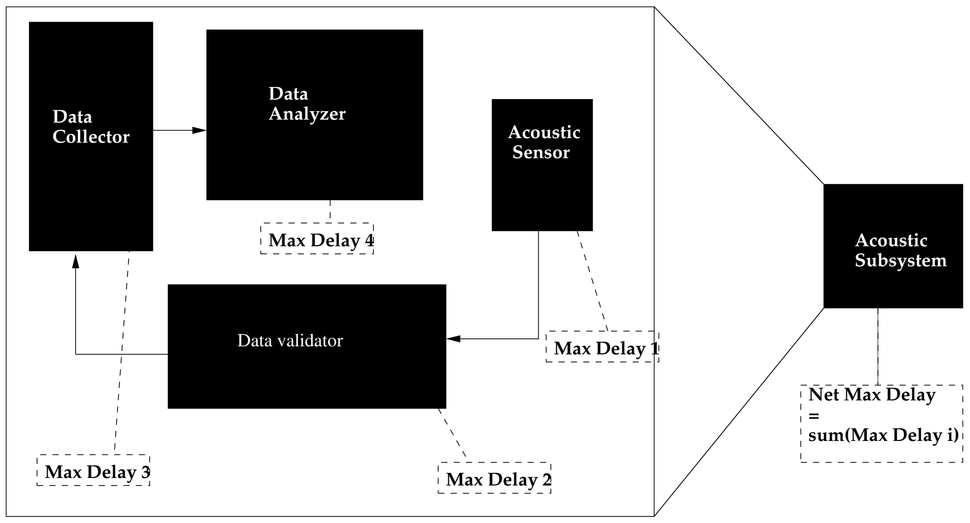


Figure 5.1 Composition Example: Acoustic Sub-system

5.2 Composition in Presence of Library Services

For explaining the composition rules, a convenience operator $match(a_x, g_y)$ is introduced, which means that g_y is a matching guarantee for a_x. Also, $tag(a_x, M)$ tags an assumption a_x as *matched*. $tag(a_x, U)$ will tag an assumption a_x as *unmatched*. Note that finding a matching guarantee for an assumption is not enough to tag the assumption as matched. For a library assumption, every component that uses this library must provide a matching guarantee to tag the assumption as matched. These are combined rules below take care of library assumptions and guarantees.

5.2.1 Composition Rules

1. For a library assumption $\widetilde{a}_x$ we use the following rule to tag it as *matched*.

$$((\widetilde{a_x} \in \widetilde{A}_i) \wedge tag(\widetilde{a_x}, M)) \leftrightarrow (\forall(c_j)((c_i \in \overline{C_j}) \rightarrow$$
$$(\exists(g_y)(g_y \in G_{j_i} \wedge match(\widetilde{a}_x, g_y))) \vee (\exists(\widetilde{g_y})(\widetilde{g_y} \in \widetilde{G_j} \wedge match(\widetilde{a}_x, \widetilde{g_y})))) \quad (5.1)$$

The explanation for this rule is as follows:-

A library assumption $\widetilde{a_x}$ of a component c_i is tagged as *matched*, *i.e.,* $((\widetilde{a_x} \in \widetilde{A}_i) \wedge tag(\widetilde{a_x}, M))$, if and only if, for every component that has c_i as a dependent

Notation	Meaning
$n_a, n_g, \widetilde{n}_a, \widetilde{n}_g$	Total number of assumptions, guarantees, library assumptions and library guarantees in the system respectively
$n_{\widetilde{a}_i}$	The number of components that provide guarantees to the library assumption $\widetilde{a}_i$
$tag(\_\_, M)$	$\_\_$ can be any assumption or guarantee. Tag $\_\_$ as matched
$tag(\_\_, U)$	$\_\_$ can be any assumption or guarantee. Tag $\_\_$ as unmatched
$match(a_x, g_y)$	Guarantee g_y matches assumption a_x

Figure 5.2 Notations in AMF Rules

component, *i.e.,* $\forall(c_j)(c_i \in \overline{C_j})$, a matching guarantee must be provided for $\widetilde{a_x}$; this matching guarantee can be a guarantee explicitly for the component c_i or it can be a library guarantee *i.e.,* $(\exists(g_y)(g_y \in G_{j_i} \wedge match(\widetilde{a}_x, g_y))) \vee (\exists(\widetilde{g_y})(\widetilde{g_y} \in \widetilde{G_j} \wedge match(\widetilde{a}_x, \widetilde{g_y})))$.

2. For an assumption that is not a library assumption, we use the following rule to tag it as matched.

$$\begin{aligned} &((a_x \in A_{i_j}) \wedge tag(a_x, M)) \leftrightarrow \\ (\exists(g_y)((g_y \in G_{j_i}) \wedge match(a_x, g_y))) &\vee (\exists(\widetilde{g_y})((\widetilde{g_y} \in \widetilde{G_j}) \wedge match(a_x, \widetilde{g_y}))) \end{aligned} \tag{5.2}$$

The expression in (5.2) states that an assumption a_x made by a component c_i on a dependent component c_j is declared as matched, *i.e.,* $((a_x \in A_{i_j}) \wedge tag(a_x, M))$, if and only if, it is matched by a guarantee in the set G_{j_i} or by a library guarantee in $\widetilde{G}_j$, *i.e.,* $(\exists(g_y)(g_y \in G_{j_i} \wedge match(a_x, g_y))) \vee (\exists(\widetilde{g_y})(\widetilde{g_y} \in \widetilde{G_j} \wedge match(a_x, \widetilde{g_y})))$.

3. For a guarantee explicitly made for a component (not a library guarantee), we use the following rule to tag it as matched.

$$\begin{aligned} &((g_y \in G_{i_j}) \wedge tag(g_y, M)) \leftrightarrow \\ (\exists(a_x)((a_x \in A_{j_i}) \wedge match(a_x, g_y))) &\vee (\exists(\widetilde{a_x})((\widetilde{a_x} \in \widetilde{A_j}) \wedge match(\widetilde{a_x}, g_y))) \end{aligned} \tag{5.3}$$

The expression in (5.3) states that a guarantee g_x provided by a component c_i for a dependent component c_j is declared as matched, *i.e.,* $(g_y \in G_{i_j} \wedge tag(g_y, M))$, if and only if, it is matched either by an explicit assumption a_y belonging to A_{j_i} or by a library assumption in $\widetilde{A}_j$, *i.e.,* $(\exists(a_x)(a_x \in A_{j_i} \wedge match(a_x, g_y))) \vee (\exists(\widetilde{a_x})(\widetilde{a_x} \in \widetilde{A_j} \wedge match(\widetilde{a_x}, g_y)))$

4. All library guarantees are tagged as *matched* by default. This is because of the definition of a library guarantee, where any dependent component can *optionally* make an assumption on it.

$$\forall(c_i)((\widetilde{g_y} \in \widetilde{C_i}) \leftrightarrow (tag(\widetilde{g_y}, M))) \tag{5.4}$$

5. All other assumptions and guarantees are declared as *unmatched*.

5.2.2 Algorithm to Compose Assumptions and Guarantees

```
procedure matchAssumptionsAndGuarantees ($C$) //$C$ - set of components
$MA \leftarrow \emptyset$, $MG \leftarrow \emptyset$ //matched assumptions set and matched guarantees set   (1)
$UA \leftarrow \emptyset$, $UG \leftarrow \emptyset$ //unmatched assumptions set and unmatched guarantees set   (2)
Add every assumption to the set $MA$ //tag all as matched   (3)
Add every library guarantee $\widetilde{g_y}$ to the set $MG$ and other guarantees to the set $UG$   (4)
for each component $c_i \in C$ do   (5)
    for each component $c_{i_j} \in \overline{C}_i$ do// every dependent component of $c_i$   (6)
        //Every $c_{i_j}$ has a set $A_{i_j}$, $A_{i_j}$ can be $\emptyset$
        for each assumption $a_x \in A_{i_j}$ do // assumptions specific to $c_j$   (7)
            call assignToCorrectSet($a_x$,$c_i$,$c_{i_j}$)   (8)
    if $\widetilde{C_i}! = \emptyset$ // there are library assumptions   (9)
        for each component $c_j$ that has $c_i$ in its $\overline{C}_j$ do   (10)
            for each assumption $\widetilde{a}_x$ in $\widetilde{C_i}$ do// every library assumption   (11)
                call assignToCorrectSet($\widetilde{a}_x$,$c_i$,$c_j$)   (12)


procedure assignToCorrectSet($\overline{a}$,$c_i$,$c_j$ )
//$\overline{a}$ - assumption made by $c_i$ (can specifically be for $c_j$ or can be a library assumption),
//$c_j$ - dependent component.
    //Every $c_j$ has sets $G_{j_i}$ and $\widetilde{G}_j$, both of which can be $\emptyset$
    if $\overline{a}$ matches with a guarantee ($g_y \in G_{j_i}$) or ($\widetilde{g_y} \in \widetilde{G}_j$)   (13)
        if $\overline{a}$ matched with $g_y \in u\_g\_set$   (14)
            Remove $g_y$ from $UG$ and add it to $MG$ // Tag $g_y$ as matched   (15)
    else   (16)
        if ($\overline{a} \in MA$) Remove $\overline{a}$ from $MA$ and add it to $UA$// Tag $\overline{a}$ as unmatched   (17)
```

Figure 5.3 Algorithm to Find Matched Assumptions and Guarantees

In this section, the algorithm for implementing the rules 1 - 5 is given. The algorithm takes into consideration the the structure of assumptions management objects as specified by the AMF language in Chapter 4. The end result of the algorithm is the following:-

- All unmatched assumptions are in the set UA
- All matched assumptions are in the set MA
- All unmatched guarantees are in the set UG
- All matched guarantees are in the set MG

The explanation of the algorithm is as follows. Lines (1) and (2) initialize all the four sets to an empty-set. In line (3), every assumption is tagged as matched or added to MA. In line (4), every library guarantee is tagged as matched or is added to the set MG, and every other guarantee is tagged as unmatched, added to the set UG. Lines (5) through (12) loops for every component c_i. Lines (6) through (8) handle assumptions by a component c_i specifically for each of its dependent components c_{i_j}; lines (9) through (12) handle library assumptions of component c_i, or the set of assumptions in $\widetilde{C_i}$.

Lines (13) through (17) describe a procedure that moves assumptions and guarantees to the correct matched or unmatched set. If an assumption does not match either with a guarantee from its dependent component set or a library guarantee in its dependent component set, it is tagged as unmatched and moved to UA. If the assumption matches a guarantee, which is in the unmatched guarantees set, UG, the guarantee is moved to the matched guarantees set, MG.

For the following, n_a and n_g are the total number of assumptions in the system that are made specifically for a dependent component. $\widetilde{n_a}$ and $\widetilde{n_g}$ are the number of library assumptions and guarantees in the system. Constants are denoted by $k_1, k_2, \ldots$.

Lemma 5.2.1. *In a system with a total of n_a assumptions and n_g guarantees, with no library assumptions or guarantees, the worst-case running time of composition algorithm is $\Theta(n_a + n_g)$. It is linear in the number of assumptions and guarantees in the system.*

Proof. Lines (1) and (2) execute once, their total execution time is a constant, say k_1. Line (3) executes once for each assumption; since there are no library assumptions, the total execution time is $k_2.n_a$, where k_2 is the constant execution time of line (3). Similarly, line (4) executes once for each guarantee, and since there are no library guarantees, its total execution time is $k_3.n_g$. Line (8) executes once for each assumption explicitly made by a component c_i on a dependent component c_{i_j}. Since there are n_a such assumptions, it executes n_a times. Hence, the $assignToCorrectSet$ procedure is called n_a times.

If there is a limit on the length of the names of assumptions and guarantees, a universal hash function can be designed that can check for the presence of a string in a hashtable in constant amortized time. This is equivalent to set membership. Line (13) executes 2 set membership functions, with a constant execution time of k_4. Similarly line (14) executes for a constant time k_5. Lines (15) and (17) also have constant execution times, say k_6 and k_7, respectively.

The worst-case running time of the algorithm in absence of library assumptions and guarantees

$$\begin{aligned} &= k_1 \ + \ k_2.n_a \ + \ k_3.n_g \ + \ (k_4 + k_5 + k_6 + k7).n_a \\ &= k_1 + k_3.n_g + k_8.n_a, \qquad \textit{where} \ \ k_8 = k_2 + k_4 + k_5 + k_6 + k_7, \\ &= \Theta(n_a + n_g) \end{aligned}$$

□

Before stating the next lemma, a few notations are recalled. $\widetilde{n_a}$ will represent the total number of library assumptions present in the system. If all the library assumptions in the system are labeled $\widetilde{a_1}, \widetilde{a_2}, \ldots$, then $n_{\widetilde{a_i}}, (1 < i < \widetilde{n_a})$ will represent the total number of components that should provide a guarantee for the assumption $\widetilde{a_i}$, for $\widetilde{a_i}$ to be tagged as matched. A notation $n_{\overline{\widetilde{a_x}}}$ is introduced for convenience to indicate the *average* of $n_{\widetilde{a_i}}$ for all $\widetilde{n_a}$ library assumptions.

Lemma 5.2.2. *In a system with a total of n_a assumptions and n_g guarantees made specifically to other components, $\widetilde{n_a}$ library assumptions, the worst-case running time for the composition algorithm is $\Theta(n_a + n_g + \widetilde{n_a}.n_{\overline{\widetilde{a_x}}})$, where $n_{\overline{\widetilde{a_x}}}$ is the average number of dependent components for a component having a library assumption.*

Proof. The presence of library assumptions will make the algorithm execute the additional lines (10) through (12). If the $\widetilde{n_a}$ library assumptions are labeled $\widetilde{a_1}, \widetilde{a_2}, \ldots$, then line (12) is executed $\sum_{i=0}^{\widetilde{n_a}} n_{\widetilde{a_i}}$ times. As shown in Lemma 5.2.1, the execution time for the procedure in lines (13) through (15) is a constant, say k_9.

The worst-case execution time of the composition algorithm in presence of library assumptions and guarantees

$$
\begin{aligned}
&= \Theta(n_a + n_g) + \sum_{i=0}^{\widetilde{n_a}} k_9.n_{\widetilde{a_i}} \\
&= \Theta(n_a + n_g) + k_9.\sum_{i=0}^{\widetilde{n_a}} n_{\widetilde{a_i}} \\
&= \Theta(n_a + n_g) + k_9.\widetilde{n_a}.n_{\overline{a_x}} \\
&= \Theta(n_a + n_g + \widetilde{n_a}.n_{\overline{a_x}})
\end{aligned}
$$

□

5.2.3 A Note on Lemma 5.2.2

In absence of library assumptions, the composition matching algorithm has a worst case running time as linear in the order of number of assumptions and guarantees. Library assumptions mandate guarantees from every dependent component using the library. Hence, component developers need to pay attention as to what constitutes a library assumption. Identifying a set of dependent components in the system, and having assumptions specific to dependent components is an important step. This step also reduces the human effort required to maintain the assumptions in the long-term, since fewer unmatched assumptions will be flagged.

5.3 Preserving Composability

Matching assumptions and guarantees in presence of library assumptions forms the core of the composition.

For assumptions management to be scalable and applicable for a larger systems in a uniform fashion, it is important that composition preserves the concept of a component - providing a functionality accessible using a well-defined interface. This will enable recursive application of composition in a uniform manner.

For example, the acoustic sensor, data collector, data validator and the data analyzer are composed to form the acoustic subsystem as shown in Figure 5.1. The acoustic subsystem needs to have a well-defined functionality and an interface to invoke the functionality and access the results. The integrator or architect of the classification system uses the acoustic subsystem on the whole. He(She) should not deal with the internal assumptions of the acoustic subsystem that are matched and validated.

At the same time, an internal system-configuration change (say, the sensor hardware) should trigger an assumptions validation of the system-configuration assumptions of the sensor and its dependent components. If they are still valid, the propagation of the system-configuration changes stop here. Otherwise, the system configuration changes need to be propagated to the higher level component, the acoustic subsystem, and so on recursively.

5.4 Summary

This chapter introduced the concept of composition of assumptions and guarantees. This concept is tied with the concept of composition of smaller sub-components to form larger components, which is the process followed to build most systems using COTS components. Composition of assumptions and guarantees involves matching assumptions and guarantees that are satisfied within the set of sub-components themselves, so that the higher level integrator is abstracted from these details. At the same time, assumption violations in the sub-components are reported. Composition rules and the algorithm for matching assumptions and guarantees presented in this chapter handle the presence of library assumptions and guarantees also. The worst case running time of the algorithm is linear in the number of non-library assumptions and guarantees in presence of no library assumptions, $(\Theta(n_a + n_g))$. Library assumptions add to the running time in the order of the average number of guarantees per library assumption times the number of library assumptions, $(\Theta(n_a + n_g + \widetilde{n_a}.n_{\overline{a_x}}))$. In addition to finding the unmatched assumptions and guarantees, new assumptions and guarantees can be introduced for the component formed by the composition of related sub-components. In order to apply composition in a uniform fashion for larger systems formed by composition of smaller sub-systems, the concept of component, with a well-defined functionality and a well-defined interface to invoke the functionality is to be preserved.

CHAPTER 6

Vertical Assumptions Management

Composition of assumptions, which is explained in Chapter 5 helps manage assumptions across different component development teams. It also helps in managing assumptions in the process of building larger components using a set of related sub-components.

The other source of assumptions is the software engineering practice within a team itself. For example, a software component, in addition to the actual code, may have requirements documents, design documents, test cases, an architecture description document that describes the interactions within a component, to name a few. Enabling a machine-checkable framework to track these assumptions across the software development life-cycle is one of the objectives of AMF. The whole process of managing assumptions that arise out of software-engineering process is termed as *vertical assumptions management*. A wide-array of research falls into this category, but in terms of making assumptions machine-checkable, AMF tries to tackle the following.

- Tracking assumptions between the code developed by different developers within the same component, *i.e.,* tying assumptions with the actual code. This is the primary objective.
- Tracking assumptions between the high-level architecture description and the actual interactions within a component.
- Enabling AMF language extensions to allow examination of system-wide impact of assumption failures, using dependency management tools.

6.1 Interfacing AMF with Programming Languages

It is very common in software development that changing one component's code affects other components in an unexpected fashion. Sometimes, components get so monolithic that it is very difficult to track assumptions made by different modules within the component itself. For example, organizations have provided patches for their patched patches [24]. In some instances, it is not possible to modify source code; and in some instances, the source code becomes monolithic and unreadable if assumptions are encoded along with the core algorithm implementation. These reasons force developers to document assumptions in a natural language. While AMF is not intended to replace natural language documentation, wherever possible, making the assumptions machine-checkable will result in significant savings in repeated manual testing efforts, which tends to be error-prone.

Ensuring correct behavior of the individual modules falls under the domain of model checking and unit testing. While dealing with the source code, AMF is aimed at making assumptions made by software components on other software components and the environment machine checkable.

6.1.1 An Example

An example from Etherware [25], a middleware for networked control systems, implemented in Java, is presented here. Etherware requires that three basic services (the scheduler, network messenger, and the time service) are started before starting other services. Etherware assumes that the basic services start within two seconds, which is a reasonable assumption that works for most systems. When Etherware is run in a resource-constrained environment, this assumption may be violated. While running on an embedded computer, the delay for initializing the services was over two seconds on multiple occasions.

In this case, the assumption for maximum allowed delay to start the basic services can directly reference the source code of the component, where that parameter is declared. This ensures that when changes are made to this value, if it violates the assumption for maximum permissible delay, an assumption violation will show up.

The start-up routine in the *Kernel* is as shown below.

```
// Start the basic services
setupBasicConfiguration();
// Sleep so that service initialization over the network is completed
try {
    Thread.sleep(INIT_INTERVAL);
} catch (InterruptedException e) {
    e.printStackTrace();
}
// Start other services
```

The *Kernel* is called by the main routine, which we name *InitRoutine*. Depending on the type of system that Etherware runs on, *InitRoutine* may require different values for the maximum initialization sleep interval.

The following is required of AMF when the *InitRoutine* and the *Kernel* modules are developed by different developers. *Kernel* makes an assumption that the maximum permissible delay for startup is less than the value indicated by the variable $INIT\_INTERVAL$. *InitRoutine* provides a guarantee for this value based on the type of system that it runs on. The developer of the *Kernel* must encode the assumption in a way that it directly gives the value set for the $INIT\_INTERVAL$ variable. This ensures that changes in the source code will trigger an assumptions check and if there is an assumption violation, it will show up.

6.1.2 Pre-Requisites for Encoding Assumptions on the Source-Code

There are some pre-requisites for the assumptions made by the software to be directly checked in AMF, as and when there are changes in the source code.

- The software should export the properties that need to be validated as the return value of a function, (public function in case of an object oriented language). This enables the standard AMF language extension to be used to obtain this property. AMF language extensions make it possible to encode method invocations as a part of the body of the assumptions.
- The function used to obtain the property that needs to be validated must be a pure function. In other words, it should not (i) modify any global property of the component

or the system containing the component, nor (ii) should the function modify any of its input parameters. These rules ensure that AMF never accidentally alters the properties of the system, whose assumptions are being validated.

Validation of assumptions does consume CPU and memory; hence, AMF may alter the schedulability and resource related properties of the system, in which assumptions are being validated. This does not matter for validation of system-configuration or static assumptions. While validating dynamic assumptions, if the system being validated is a real-time system, one needs to ensure that the resource allocation takes care of the resources required for validation of assumptions also.

The rule of non-modifiable input parameters for a function can be automated for most programming languages - like Java (by declaring parameters for the functions as final) and C (where parameters are passed by value by default).

6.1.3 Procedure for Encoding Assumptions on the Source Code

The following procedure is used to encode assumptions and guarantees that directly reflect the properties of variables in the source code. The procedure given here is for Java.

- Find the list of properties that are present in the source code that need to be monitored by assumptions or provided by guarantees. For example, the $Kernel$ module needs to monitor $INIT\_INTERVAL$ in an assumption.

- Export all properties that need to be directly monitored in assumptions and provided in guarantees as the return value of a *public* function. For example, the $Kernel$ exports the $INIT\_INTERVAL$ variable using a *public* function.

```
public int getInitSleepInterval() {
     return INIT_INTERVAL;
}
```

- Encode the assumption or the guarantee using the public function. (i) In the optional declaration part of AMF, indicate the language in which the method invocation is encoded. (ii) Indicate the set of classes imported by the component to specify the assumption (iii)

Encode the body of the assumption making use of the *public* functions from the imported classes.

```
componentDefinitions name=Kernel {
    {language=JAVA;
     import ether.etherware.kernel.Kernel;
    }
    about InitRoutines {
        assumes maxStartupDelay(int maxDelayInMilliSec) {
            maxDelayInMilliSec <= Kernel.getInitSleepInterval();
        }
        // Classification info ...
    };
}
```

The developer of *InitRoutine* provides a guarantee based on the type of system. A test routine is used that checks for various parameters of the system and estimates the maximum delay that will be encountered to start the basic services in the system.

```
componentDefinitions name=InitRoutines {
    {language=JAVA;
     import ether.etherware.diagnosis.TestMachineParams;
    }
    about Kernel {
        guarantees maxStartupDelay {
            int maxStartupDelay = TestMachineParams.getMaxStartDelayEstimate();
        }
        // Classification info ...
    };
}
```

The above procedure ensures that wherever possible, AMF can validate assumptions based on the properties of the source code itself. This becomes important as the size of components get bigger and there are multiple developers involved in developing a component. AMF ensures that developers can make assumptions about source code developed by other developers without modifying their sources. While AMF cannot completely replace natural language documentation, wherever possible, enabling assumptions on the source code to be encoded in a machine checkable format saves human testing effort.

6.2 Integration with AADL

Embedded system components are generally composite components containing both hardware and software components. Most hardware components export their properties in the form of data sheets, which precludes any machine-checkable assumptions to be encoded on them. To enable machine checkable assumptions, the properties of the hardware sub-systems must be exported by a software proxy. A unified framework to encode properties of software and hardware components with a well-defined syntax and semantics is necessary. Such a framework in conjunction with AMF can be used to trigger assumption checks when properties of hardware components change. The Architecture Analysis and Description Language (AADL) is chosen for this.

```
system ActousticClassifier
end ActousticClassifier;

process DataCollector
    features
        Output: out data port AppData;
        Input: in data port AppData;
    properties
        Compute_Execution_Time => 0 ms .. 10ms
            in binding (powerpc.speed_350Mhz);
        —- Other properties
end DataCollector;

—- Similar definitions for
—- sensor device and sense process
system implementation AcousticClassifier.Basic
    subcomponents
        dc_ : process DataCollector;
        sensor_ : device Sensor;
        —- other parts of the system
    connections
        data port sensor_.Output -> dc_.Input
            in modes (stable);
        —- other connections
    properties
        —- standard properties
end AcousticClassifier.Basic;
```

Figure 6.1 Sample of AADL Specification of the Acoustic Sensing System

6.2.1 The Architecture Analysis and Description Language (AADL)

AADL provides a framework for specifying the architecture of real-time and embedded systems incorporating both the hardware and software components' properties and behavior or functionality. Specifically, the standard has precise definitions of key hardware components like *Processor, Bus, Memory, Device* and software components like *Process, Thread, Thread Group, Subprogram, Data* and composite components like *System.* The AADL specification [21] contains a well-defined syntax for specification of the system architecture. Importantly, for AMF, it contains the semantics, standard properties and processing requirements for each of the components mentioned above. Every component has features, which are attributes or properties of the components that are exposed to other components. It is equivalent to a *public interface* of the component. Components can specify attributes which are non-standard using the *Properties* construct. The *Properties* construct is a set of name-value pairs per component or component instance.

A part of the specification of the *data collector* component in AADL is given in Figure 6.1. The *data collector* component specifies its standard input and output as a part of its features. Properties like execution time for a software component on a particular processor can also be specified. The sub-components and connections (interactions) for a component are specified along with its implementation specification.

AADL is an extensible language; it allows other analysis tools to be plugged-in as annexes. AADL provides a well-defined procedure for annexes to access the components and their properties defined in the AADL name-space. Annexes can use this information for domain specific analysis. The AMF language is made available as an annex plug-in for AADL. AADL has a set of approved plug-ins that can be used for tasks like ensuring schedulability and ensuring resource constraints are met while designing real-time systems. The AMF plug-in can be used to ensure that assumptions on the operating environment and assumptions made by software on hardware and vice versa are satisfied. An example of the assumptions made by the *data collector* on the *sensor* is shown in Figure 6.2. It shows how the assumptions specification is combined with the AADL specification of the system in Figure 6.1.

With the integration with AADL, AMF utilizes the name-space of AADL to get the component types. This ensures that the names used within AMF are consistent with those in the

```
process DataCollector
  features
    Output: out data port AppData;
    Input: in data port AppData;
  properties
    Compute_Execution_Time => 0 ms .. 10ms
      in binding (powerpc.speed_350Mhz);
    --- Other properties
  annex assumptions {**
    // assumptions specified in AMF specification language
    componentassumptions name=DataCollector {
      // Dependent component Sensor
      about Sensor {
        assumes saturationValueLimit
          (float maxSensorError, float saturationValue) {
          (1 + maxSensorError)*50 < saturationValue
        }
        {Criticality= CRITICAL_LEVEL_A}
        {ValidityTimeFrame= SYSTEM_CONFIGURATION};

        guarantees dataImportControl {
          int importInterface = PS2_BIDIRECTIONAL;
        }

        // Other assumptions and guarantees
    **}
  end DataCollector;
```

Figure 6.2 Specifying Assumptions Within AADL

architecture definitions. Also, AADL specification lists the component types that can interact with each other. Using this information, AMF ensures that meaningful component types make assumptions on each other. For example, it is not meaningful for a component of type *Memory* to make assumptions on a component of type *Thread Group*.

6.3 Mapping Assumptions to Component Failures

It is logical to record the immediate impact of an assumption violation along with the assumption itself. There are specialized dependency management tools like DMF [26] to analyze the system-wide impact of an error given the error propagation rules between immediately interacting components. By using a standard vocabulary to record the immediate impact on an assumption violation, assumption violations can serve as inputs to these dependency management tools. Hence, the system-wide impact of an assumption violation can be discerned.

To enable this, a language extension is needed to allow the encoding of immediate impact on assumption violation. This is as shown below. The *impactConsts* clause encodes all the standard failures present in the DMF.

```
classificationInfo : (criticalityClause)? (validityTimeFrameClause)?
                     (scopeClause)? (violationImpactClause)? ;
validityTimeFrameClause : ‘{’ ‘ViolationImpact’ ‘=’ (impactConsts| IDENTIFIER) ‘}’ ;
impactConsts : ‘VALUE_ERROR’ | ‘CRASH’ | ‘LOCKUP’ | ‘SUSPEND’ | ‘OMISSION’
    ‘STATE_TRANSITION_ERROR’ | ‘BYZANTINE’ | ‘DEADLINE_MISS’ | ‘BUDGET_OVERRUN’
     | ‘RESOURCE_SHARING_ERROR’ ;
```

Based on this extension, immediate impact on assumption violation can be recorded as follows.

```
componentDefinitions name=DataCollector {
    about Sensor {
         assumes min_saturation_value(double error, int saturation_value) {
              (1.0 + error) * 100.0 < saturation_value
         }
         {ViolationImpact=VALUE_ERROR}
         // Other info
     };
     // assumption Guarantee Sets for other components
};
```

The error propagation rules will be encoded in DMF as shown below. The first line of the rules in DMF syntax states that the value error in the *data collector* will cause a value error in the *data analyzer* component. The complete list of similar error propagation rules for a system will be encoded in DMF.

```
DataCollector.VALUE_ERROR -> DataAnalyzer = VALUE_ERROR
DataAnalyzer.VALUE_ERROR -> // Other error propagation rules...
```

Given this information, system-wide impact of the assumption *min_saturation_value* being violated can be found using DMF. Thus, language extensions in AMF enable it to work with useful tools that allow studying the system-wide impact of assumption violations.

6.4 Summary

From a software-engineering perspective, assumptions arise during system design (like architecture description), system analysis, coding, integration and testing phases. This warrants that the AMF language makes it amenable to easily specify assumptions during these phases. The whole process of managing these assumptions that arise out of the software-engineering process is termed as *vertical assumptions management.*

AMF language extensions allow specification of assumptions directly on the source-code of the components. This allows assumption violations to be detected when properties in the source code changes. This becomes important as the components get bigger and there are multiple developers involved in developing a component. AMF ensures that developers can make assumptions about source code developed by other developers without modifying their sources.

AMF language can be an annex sub-language of AADL. Importantly, for AMF, AADL enables properties of hardware components and software components to be exported in a machine-checkable format. AADL also provides a well-defined grammar and semantics for specifying system architecture using hardware and software components. Integration with an architecture description language like AADL helps end-users specify assumptions between hardware and software components, along with the system architecture itself.

AMF provides language extensions that enable the users to encode the immediate impact of assumption violations using standard vocabulary used by dependency management tools.

These language extensions of AMF allows assumptions to be specified directly on the source code, with minor or no source code modifications required; assumptions to be specified along with the system architecture description and allows analysis tools to detect impact of assumption violations. This easy integration with various aspects of software engineering makes assumptions specification blend into the software engineering life cycle.

CHAPTER 7

Implementation of the Assumptions Management Framework

The assumptions management framework (AMF) is implemented in Java using the Eclipse Modeling Framework (EMF) [27]. The grammar for the AMF Language is specified using ANTLR [28]. The implementation provides facilities to specify assumptions using a text-based interface and a model-based graphical user interface. It provides automatic composition and validation facilities. The AMF objects can be saved in a well-defined XML format that makes it amenable for database operations. Users can set policies on when to validate assumptions and what assumptions need to be validated. The framework also provides source-code integration for Java projects to directly specify assumptions on the source code, language extensions to specify complex assumptions and guarantees as methods and integration with the Architecture Analysis and Description Language (AADL) framework's toolkit (called OSATE). AMF provides user-friendly error-reporting that enables the user to directly view the source of the error, like AMF errors in AADL annex definitions, errors in Java sources generated by AMF, or assumption definitions themselves.

7.1 Structure of the Implementation Objects

EMF was chosen for the implementing AMF for the following features that AMF makes extensive use of.

- EMF provides a code generation framework that takes care of generating appropriate accessor method stubs for the fields in the objects. Directives can be provided to EMF to specify whether the field is *changeable* - this will make EMF only provide a *get*() method and not a *set*() method; whether the field is *volatile* - indicating that the value of this field needs to be calculated on the fly and no storage is allocated to it; whether a field is *transient* - indicating that the field should not be serialized when the object is serialized; whether the field is *unsettable* - indicating that the user should be able to query whether any value was explicitly set to this field; whether the field the *contained* - indicating that the actual object representing the field and not a reference to the field is stored during serialization.

For example, *formal parameters* are a part of the *assumption* object and hence, *containment* will be set to true. An *assumption* contains at least one *formal parameter.* This is specified as below. EMF will take care of serialization; it will make sure that the there is at least one formal-parameter present for an assumption.

```
/**
 * @model type="FormalParameter" containment="true" lower="1"
 */
EList getFormalParameters();
```

The printable name for an assumption is a qualified name of the assumption containing the component name, the dependent component name and the assumption name, with a separator between these names. These values can be obtained on the fly and there needs to be no storage associated with this function. Also, this field need not be serialized and it cannot be changed manually. This is specified as follows. EMF will handle serialization. It will create a stub for a method in its implementation class which can be filled to create this functionality.

```
/**
 * @model changeable="false" volatile="true" transient="true"
 */
String getPrintableName();
```

- It provides type-safe enumerations with default values that can be set to fields. For example, by default all assumptions have the composition scope as $PUBLIC$. The scope is also a required field and one can query if the value for this field was explicitly set. This is specified as follows

```
/**
 * @model required="true" default="Public" unsettable="true"
 */
Scope getScope();
```

- EMF automatically provides a model-based GUI reflecting the overall design. This is shown in Section 7.2.2.

- EMF separates the object implementation from the interface of the object. It provides default factory methods to create the implementation objects that hides the implementation details from the end-users.

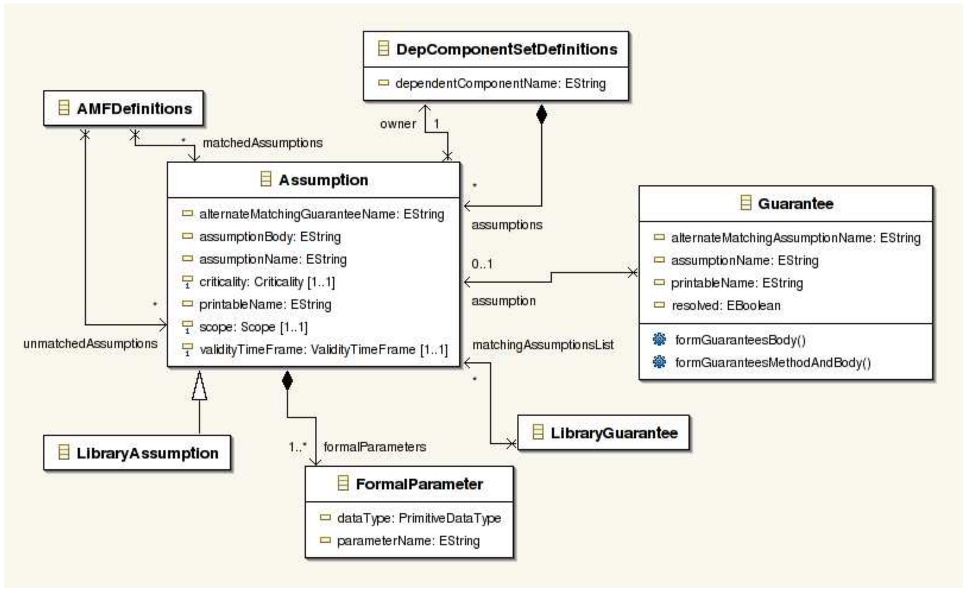


Figure 7.1 Class Diagram for the Assumption Object Implementation using EMF

EMF objects are created to accurately reflect the structure of the assumption management objects described in Chapter 4. For example, an assumption has a name, the body of the assumption and a set of formal parameters to invoke the assumption, in addition to its classification and set containment information. The EMF objects may contain other implementation specific parameters for an object. For example, every assumption will have a printable name and an alternate matching guarantees name. The part of the design of the assumptions object using EMF is as shown in Figure 7.1. It only includes the fields to and from the assumption object. Please refer to Appendix B for the complete design.

7.2 Specifying Assumptions

Assumptions can be specified either using a model-based GUI, which is generated by EMF or in a textual format.

7.2.1 The AMF Text Based Input

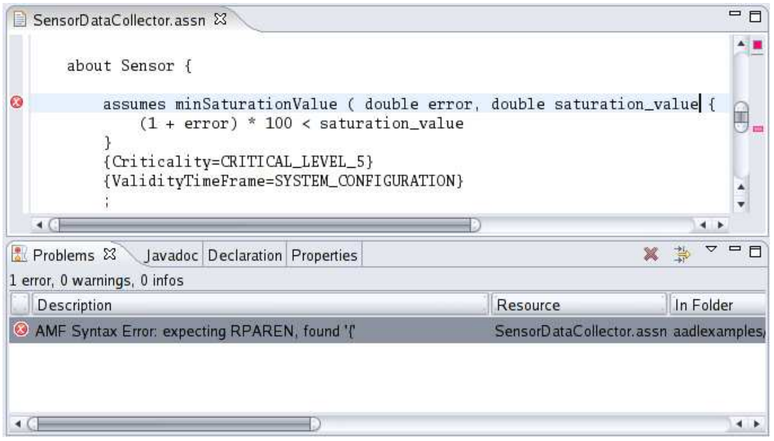
The parser for the textual format is constructed using the ANTLR Parser generator [28]. The input for the parser is the AMF definitions that adhere to the grammar specified in Appendix B and the output of the parser is the Java (EMF) object representing the AMF definitions.

In addition to adhering to the grammar, the parser ensures the uniqueness of component names, uniqueness of dependent component names within a component, unique names for assumptions and guarantees, and unique formal parameter names. The parser creates appropriate EMF objects representing the assumption definitions as and adds it to the root of the tree representing AMF definitions. On successfully parsing the entire textual input, it returns the root object. Otherwise, it reports the errors and returns a *null* Java object.

ANTLR allows actions to be taken when particular clauses in the grammar are successfully parsed. Actions for a clause are specified within braces after the clause is recognized. An example of the action taken when a formal parameter in an assumptions is recognized is given below. A formal parameter contains a type, which is parsed by the grammar clause *primitiveDataType* and a name represented by the token $IDENT$, which is unique for an assumption. After the respective clauses are parsed successfully, actions to be taken can be specified in Java. As shown in the example below, the actions on successful parsing of a clause are between the braces ('{' and '}'). In the example, $currentFPTable$ represents a hashtable that stores the names of the formal parameters for the current assumption; af is the factory object to create the EMF implementation objects; $currentAssumption$ represents the assumption being parsed. If a parameter with the same name as $IDENT$ already exists, an error is reported by the parser. Otherwise, it is added to the hashtable of existing parameters.

```
param:  primitiveDataType {
            FormalParameter fp = af.createFormalParameter();
            fp.setDataType(currentType);
            currentFP = fp;
        }
        i:IDENT^ {
            if (!currentFPTable.containsKey(i.getText())) {
                currentFP.setParameterName(i.getText());
                currentAssumption.getFormalParameters().add(currentFP);
                currentFPTable.put(i.getText(),"");
            }
            else {
                reportError("Duplicate parameter: " + i.getText(), i.getLine());
                error =  true;
            }
        }
;
```

The parser recursively constructs the assumptions definitions tree represented by the textual input in this fashion. The Eclipse based implementation has error reporting features that takes the user directly to the line where there is a syntax error as shown in Figure 7.2. In the figure, the user has forgotten a closing parenthesis ")".



```
about Sensor {

    assumes minSaturationValue ( double error, double saturation_value {
        (1 + error) * 100 < saturation_value
    }
    {Criticality=CRITICAL_LEVEL_5}
    {ValidityTimeFrame=SYSTEM_CONFIGURATION}
    ;
```

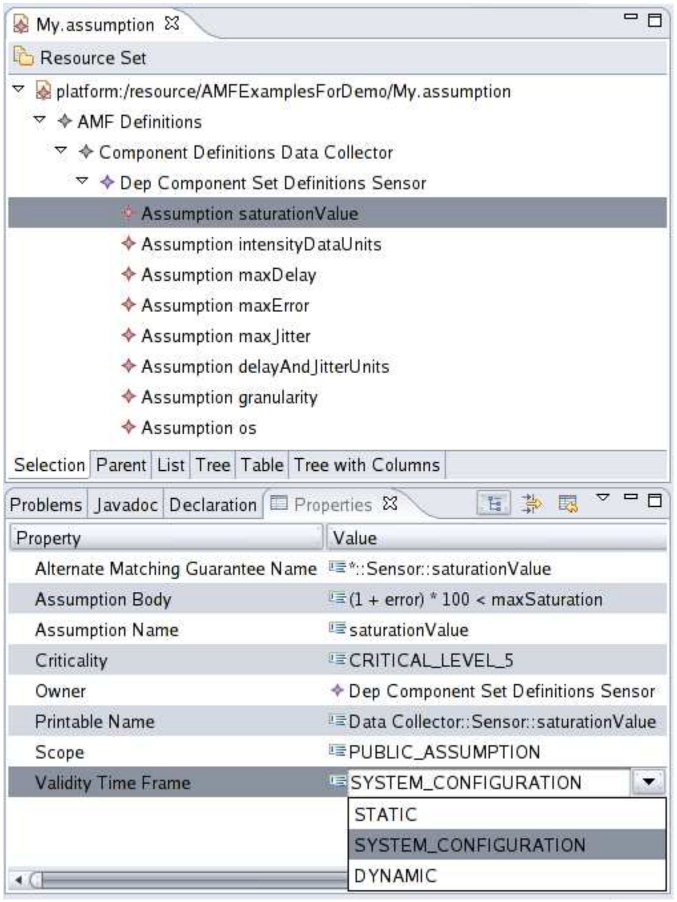
1 error, 0 warnings, 0 infos

Description	Resource	In Folder
AMF Syntax Error: expecting RPAREN, found '{'	SensorDataCollector.assn	aadlexamples

Figure 7.2 Syntax Errors in Textual AMF Specification

AMF provides an *unparser* implementation to allow the user to extract from the Java (EMF) object, the textual AMF definitions that adhere to the standard syntax.

7.2.2 The AMF GUI Based Input


My.assumption

Resource Set

platform:/resource/AMFExamplesForDemo/My.assumption
- AMF Definitions
  - Component Definitions Data Collector
    - Dep Component Set Definitions Sensor
      - Assumption saturationValue
      - Assumption intensityDataUnits
      - Assumption maxDelay
      - Assumption maxError
      - Assumption maxJitter
      - Assumption delayAndJitterUnits
      - Assumption granularity
      - Assumption os

Property	Value
Alternate Matching Guarantee Name	*::Sensor::saturationValue
Assumption Body	(1 + error) * 100 < maxSaturation
Assumption Name	saturationValue
Criticality	CRITICAL_LEVEL_5
Owner	Dep Component Set Definitions Sensor
Printable Name	Data Collector::Sensor::saturationValue
Scope	PUBLIC_ASSUMPTION
Validity Time Frame	SYSTEM_CONFIGURATION

Figure 7.3 Model Based GUI to Specify the Assumptions

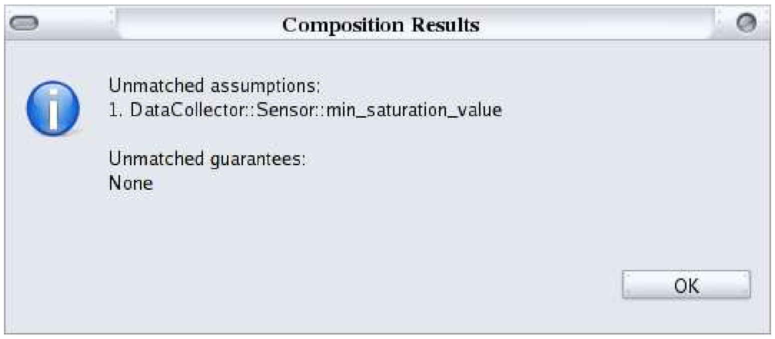
AMF implementation also provides a model-based GUI, which is automatically generated by EMF, based on the model definitions of the assumptions management objects and the implementation methods provided. This helps the user to specify assumptions without learning the AMF language syntax. The model-based GUI automatically fills in the default values for the fields, and only allows valid values for enumerations, which simplifies assumptions specification. The user can save the definitions in XML format and load the XML file to create the Java (EMF) object representing the assumptions definitions. An example of creating assumption definitions for the sensor - data collector system is shown in Figure 7.3.

The AMF unparser can be used to convert the Java (EMF) object into textual definitions.

7.3 Composing Assumptions

AMF implements the composition algorithm is presented in Section 5.2. The algorithm makes extensive use of Java *Hashtable* class to test for the presence of an assumption or a guarantee in their respective matched and unmatched sets. After the composition routine is executed, AMF internally stores the list of matched assumptions and guarantees, and the list of unmatched assumptions and guarantees. It provides functions to the users to access the list of matched and unmatched assumptions and guarantees. It also provides query functions to check if an assumption or a guarantee is matched or unmatched, given its fully qualified name (component name, dependent component name and assumption/guarantee name).

End-users or system integrators can make use of the list of unmatched assumptions to decide which assumptions should be tagged as *PUBLIC* assumptions. Composition is also a pre-requisite for validation of assumptions. Only matched assumptions can proceed to the validation stage.



```
Unmatched assumptions:
1. DataCollector::Sensor::min_saturation_value

Unmatched guarantees:
None
```

Figure 7.4 Sample Results of Composition

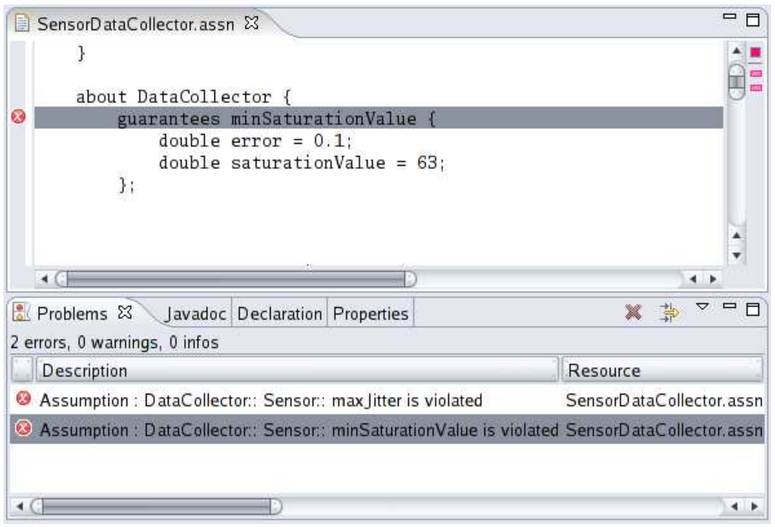
AMF displays the list of unmatched assumptions and guarantees as shown in Figure 7.4.

7.4 Validation of Assumptions

After the composition, all matched assumptions and guarantees can proceed to the validation stage. AMF generates Java code to represent the body of the assumption. Since there is a one-to-one correspondence between the basic data-types supported by AMF and the basic data-types present in Java, the body of the assumption directly translates into a Java method. All the types have native support in any Java implementation and no imports are required.

AMF generates one Java class per component, with all the assumptions for its dependent components as *public static* functions that can be invoked. This implementation detail is abstracted from the end-user and the implementation can change in the future without affecting the user's perception of AMF functionality. For example, AMF can directly generate Java byte-code wherever possible, or AMF can generate executables if it is only being used for a particular platform. This particular implementation choice was made to enable AMF to maximize the portability without modifying the definitions or the code generated.

AMF reports all the validation errors (invalid assumptions) as problem markers in Eclipse. Users can click on each of the markers, and AMF will display the guarantee which caused the assumption to be violated. For example, Figure 7.5 represents a screen showing that two assumptions are violated. When the user clicks on the problem markers, AMF displays the guarantee that caused the assumption to be violated.



```
    }

    about DataCollector {
        guarantees minSaturationValue {
            double error = 0.1;
            double saturationValue = 63;
        };
```

```
2 errors, 0 warnings, 0 infos
Description                                                      Resource
Assumption : DataCollector:: Sensor:: maxJitter is violated          SensorDataCollector.assn
Assumption : DataCollector:: Sensor:: minSaturationValue is violated SensorDataCollector.assn
```

Figure 7.5 Screen Indicating the List of Violated Assumptions

7.5 Services for Dynamic Validation of Assumptions

The entire AMF implementation can be hosted in a server outside Eclipse using a command line interface or direct Java function calls to invoke all the features mentioned below. In particular, the AMF definitions object exposes the following classes of services.

- APIs to compose, compile the definitions and generate and compile Java code: These APIs are used whenever the system architecture changes or the composition context changes.
- APIs to obtain the list of matched and unmatched assumptions (or guarantees).
- APIs to query for a particular assumption (or a guarantee) and check whether it is matched or unmatched, given the fully-qualified name. These APIs are more efficient than obtaining the list and searching for the assumption or the guarantee; they use the internal hash-table.
- APIs to save the entire definitions in XML format; this will save the classification information also, which will make enforcing policies on validating assumptions easier.
- APIs to unparse the definitions into an output stream. This enables obtaining the definitions in textual format that conforms to the AMF language syntax from the Java(EMF) object.
- Validate a particular assumption given its fully qualified name. If the assumption is a library assumption, qualified names for each matching guarantee can be individually called.

In particular, the service – validation of a particular assumption given its name – is the key for dynamic assumptions validation. After composition, compilation and the code-generation phase, if an assumption that needs to be validated is tagged as matched, users can use this service to validate the assumption. AMF can invoke routines that obtain the values for the guarantees and check if the assumption is valid.

While the implementation of the framework takes efficiency into concern (14,000 assumptions and guarantees were composed in under in 900ms, parsing and code-generation time of the same set, with over 113,000 lines as input, was under 14 seconds), the current AMF implementation does not provide hard-guarantees on the upper-bound for validation time. The implementation is quite efficient for validating assumptions in soft real-time systems. The class load-time for a guarantees class consisting of 800 guarantees was about 15 milli-seconds, and the average validation time for an assumption consisting of 3 formal parameters and 3 boolean operations was about 250 micro-seconds. The majority of this time was spent in (i) translating the assumption name into the method name to be invoked and (ii) finding the method within the class to be invoked. The actual method execution time was inconsequential[1].

7.6 Integration with Java Source Code

Section 6.1 explained in detail, the procedure to be followed to make assumptions directly on the Java source code. In brief, the AMF implementation enables assumptions and the source code for a component (currently Java source-code) to be within the same integrated development environment and they can cross-reference each other. AMF language extensions allow method invocations on the Java sources as a part of the body of the assumptions. For soft-real time and non-real-time systems, users can make calls to validate assumptions from the Java sources.

AMF provides facilities for assumptions to be validated from the source code. The assumption validation can be triggered by calling the *validateAssumptions* library method provided by AMF using the qualified assumption name as the input parameter. AMF will obtain the parameters for the validation of the assumptions from the guarantees definitions; if needed, methods are invoked to obtain the guarantee values.

For example, AMF implementation was used to encode and validate the assumptions in the wireless-networked control system controlling a set of experimental toy cars. The whole system was controlled by a middleware - Etherware [25]. Etherware makes an assumption that there is no other instance of the middleware running on a machine and that the ports required to start the basic services are available. Every other service that runs on Etherware depends on these services. This assumption is encoded as follows.

[1]These numbers were obtained using a machine with an Intel T2300 processor running at 1.66 GHz, 1 GB of memory, Windows XP operating system and JDK 1.5.

```
assumes canStartNetworkServicesOnPorts(String protocol, String address,
        int datagramPort, int broadcastPort, int streamPort) {
         ( (protocol != null) && (protocol.equals("TCP/IP"))
           && (EtherwareAssumptionsLib.datagramPortAvailable(address,datagramPort))
           && (EtherwareAssumptionsLib.datagramPortAvailable(address,broadcastPort))
           && (EtherwareAssumptionsLib.streamPortAvailable(address,streamPort)) )
}
{Criticality=CRITICAL_LEVEL_5}
{ValidityTimeFrame=DYNAMIC}
{ViolationImpact=CRASH};
```

A library *EtherwareAssumptionsLib* was created that has methods *datagramPortAvailable*() and *streamPortAvailable*() to check if a datagram or a stream port is available. AMF implementation can directly reference these methods since the body of the AMF code is in Java.

Etherware can also trigger an assumption validation routine directly from its Java source code; Etherware can invoke the following routine each time it needs to start up on a new machine.

$$validateAssumption(\text{“}Kernel :: Environment :: canStartNetworkServices\text{”})$$

Assumptions can also be manually validated on demand by the user whenever there are changes in the sources. Assumption violations will show up as shown in Figure 7.5.

AMF provides an integrated development environment for Java sources and assumptions specification as shown in Figure 7.6. All the Java sources are under the source folder *javasrc*, and the assumptions are specified under the folder assumptions. The code-base for the Java code generated is also *javasrc* which enables assumptions to invoke the Java methods on the sources and vice-versa.

7.7 Integration with AADL

Section 6.2 explained the process of integration of AMF with AADL. In brief, AMF implementation is integrated with AADL's open-source toolkit environment (OSATE) to unify architecture and assumptions specification. AMF imports all the components declared in the AADL namespace and matches it with the components declared in the AMF namespace. Any components in AMF that are not declared in AADL namespace are flagged. Also, AMF obtains the type information for each of the components from AADL and it ensures that components of appropriate types can make assumptions on each other. For example, if a component-type is *Thread Group* in AADL, it is not meaningful for it to make assumptions on a component of type *Memory*. The assumptions may need to be restructured to ensure appropriate types of components make assumptions on each other. AMF behaves as an annex sub-language of AADL, which enables AMF to extract and use the namespace of AADL for component mapping.

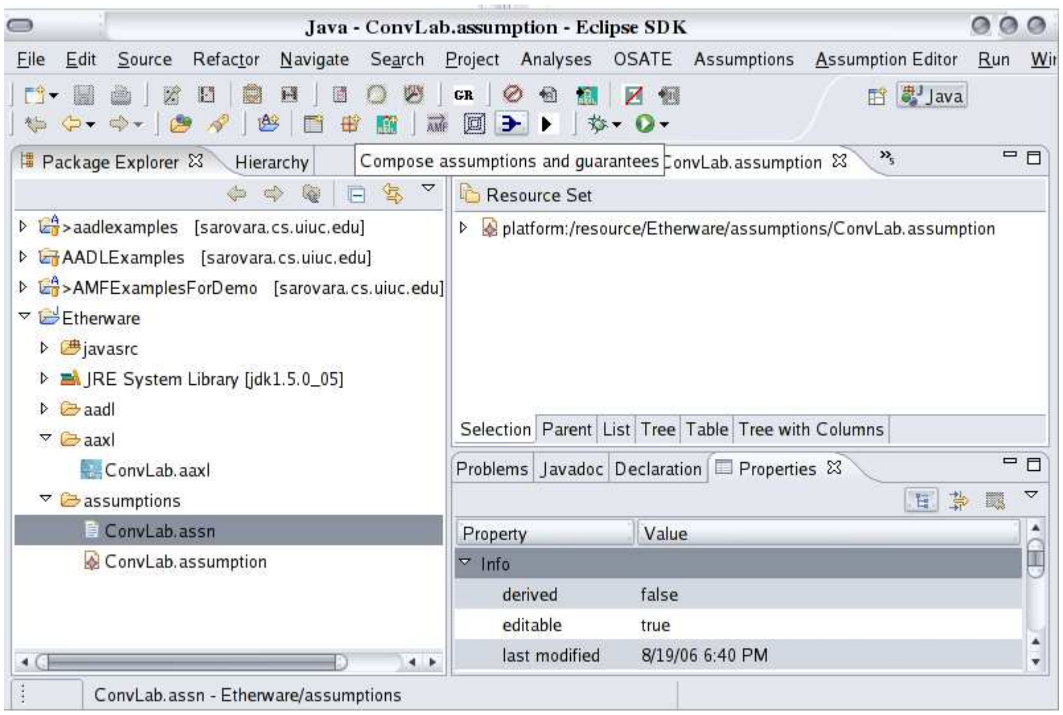
AMF also provides methods to extract the assumptions from AADL as an XML object or in the textual format that conforms to the AMF language syntax. In Figure 7.6, the folder *aadl* has the textual AADL definitions and the folder *aaxl* has the XML representation of AADL definitions.

The error-reporting of AMF translates syntax-errors and assumption violations directly into errors in the respective AADL objects, when assumptions are defined as an annex to AADL objects. Thus, there is uniform error reporting for architecture, assumptions and the source code.

7.8 Policies on Assumption Selection and Validation Times

AMF allows any Java (EMF) object to be saved as an XML file representing the entire definitions, using a feature provided by EMF. Also, AMF provides a library method to invoke assumption validation given its fully-qualified name as input. These two features makes the framework very amenable for setting policies on when to validate assumptions and what assumptions to validate.

For example, consider the partial definitions of the $sensor - datacollector$ system. The AMF definitions are as shown below.


Java - ConvLab.assumption - Eclipse SDK
File Edit Source Refactor Navigate Search Project Analyses OSATE Assumptions Assumption Editor Run Wi
Java
Package Explorer
Hierarchy
Compose assumptions and guarantees
ConvLab.assumption
Resource Set
Selection Parent List Tree Table Tree with Columns
Problems Javadoc Declaration Properties

```
>aadlexamples [sarovara.cs.uiuc.edu]
AADLExamples [sarovara.cs.uiuc.edu]
>AMFExamplesForDemo [sarovara.cs.uiuc.edu]
Etherware
  javasrc
  JRE System Library [jdk1.5.0_05]
  aadl
  aaxl
    ConvLab.aaxl
  assumptions
    ConvLab.assn
    ConvLab.assumption
```

platform:/resource/Etherware/assumptions/ConvLab.assumption

Property	Value
Info	
derived	false
editable	true
last modified	8/19/06 6:40 PM

ConvLab.assn - Etherware/assumptions

Figure 7.6 Integrated Environment for Assumptions, Architecture and Source-Code

```
componentDefinition name=DataCollector{
    about Sensor {
    assumes minSaturationValue ( double error, double saturation_value) {
            (1 + error) * 100 < saturation_value
        }
        {Criticality=CRITICAL_LEVEL_5}
        {ValidityTimeFrame=SYSTEM_CONFIGURATION};
        assumes intensityUnits(String intensityUnits) {
            ‘‘Decibels’’.equals(intensityUnits)
        }
        {Criticality=CRITICAL_LEVEL_5}
        {ValidityTimeFrame=STATIC};
    };
};
```

When the Java object is saved in XML format, the contents are as shown below. The XML representation preserves the structure of the definitions but it makes it amenable to database query operations. XPath is used to select the required nodes within an XML document. In very simple terms, XPath is a query language for XML documents. The input to this language is a constraint based on XML node names, element contents, attribute names and values and the output is a set of nodes within the document that match these constraints.

```
<componentDefinitions xmi:version="2.0">
  <!-- other header declarations -->
  <componentDefinition componentName="DataCollector">
    <depComponentDefinitions owner="#//@assnGuarSetsList.0"
      dependentComponentName="Sensor">
       <assumption criticality="CRITICAL_LEVEL_5"
         assumptionBody="(1 + error) * 100 &lt; saturation_value"
         assumptionName="minSaturationValue"
         owner="#//@assnGuarSetsList.0/@assnGuarSetList.0"
         validityTimeFrame="SYSTEM_CONFIGURATION">
               <formalParameters dataType="double" parameterName="error"/>
               <formalParameters dataType="double" parameterName="saturation_value"/>
      </assumption>
      <!-- other assumptions -->
    </depComponentDefinitions>
  </componentDefinition>
</componentDefinitions>
```

A sample of the useful queries for AMF are shown below.

- The query for selecting all the system-configuration assumptions is as follows.

  $$//assumption/[@validityTimeFrame = \text{“}SYSTEM\_CONFIGURATION\text{”}]$$

  In XPath, ‘//’ is a global selector, it selects all nodes with the given constraint. ‘/’ is a local selector. Hence, within the selected nodes with the where node name is *assumption*, the part of the query after ‘/’ is executed. The query part $@validityTimeFrame = $ “$SYSTEM\_CONFIGURATION$” selects XML nodes that have an attribute $validityTimeFrame$ with a value $SYSTEM\_CONFIGURATION$. If this query is run on the definitions described in this section, it will return the assumption $minSaturationValue$.

- Similarly, the following query is used to select all the critical assumptions in the system.

  $$//assumption/[@criticality = \text{“}CRITICAL\_LEVEL\_5\text{”}]$$

- The following query selects all assumptions made by the *datacollector* on the *sensor*.

$$//assumption/[::parent@dependentComponentName = \text{“}Sensor\text{”}/$$
$$::parent@componentName = \text{“}DataCollector\text{”}]$$

XPath constraints can be in the form of a count of the number of XML elements or the position of XML elements, or a logical expression containing a combination of constraints. It provides selection of ancestors, descendants, siblings, immediate parents and children of XML elements. This is sufficiently powerful to set policies to validate just the relevant set of assumptions, which increases the efficiency of validation process. The JDK version 1.5 includes an implementation of the XPath specification. For further reference, the user is directed to the XML [29] and XPath [17] specifications.

7.9 Summary

AMF provides a model-based implementation for managing assumptions using the Eclipse Modeling Framework (EMF).

Assumptions can be specified in a textual format that confirms to the AMF language syntax specified in Appendix B. AMF provides a parser using the ANTLR Parser Generator to parse the textual input and create the Java (EMF) objects. Assumptions can also be specified using the model-based GUI, which is automatically generated by EMF.

AMF provides an implementation of the composition algorithm that helps the user find the list of matched and unmatched assumptions (and guarantees). It also provides amortized constant-time query operations after composition to check if an assumption (or guarantee) is matched.

AMF generates Java code to represent the body of the assumptions; guarantees provide values to invoke assumption validation. AMF provides functions to validate an assumption, given its name. This, along with a set of functions that can be accessed natively using Java or from command-line, helps in enabling dynamic assumptions validation. The implementation is suitable for validating dynamic assumptions in non real-time or soft real-time systems.

Integration with Java source code allows users to invoke assumptions from the source-code or for assumptions and guarantees to invoke methods on the source code. Integration with OSATE helps in synchronizing the names of components in the architecture specification with the names used in AMF. AMF ensures that assumptions exist between components of compatible types.

The facility of EMF to store the definitions in XML format makes it amenable to set policies on when to validate assumptions and what assumptions to validate. This increases the efficiency of validation.

AMF provides user-friendly error-reporting that enables the user to directly view the source of the error, like AMF errors in AADL annex definitions, errors in Java sources generated by AMF, or assumption definitions themselves. The error types handled include syntax errors in assumptions specification, assumption violations, errors due to assumptions between components of incompatible types.

CHAPTER 8

Case studies of Representative Projects

A set of representative projects were studied to determine if invalid assumptions caused defects in end-products. In addition, these studies are invaluable in examining the nature of assumptions that cause these defects. In this chapter, two hypotheses are stated relating to the necessity and feasibility of assumption management frameworks. Defects in products (software or hardware) that end-users encounter, is one of the acceptable benchmarks to gauge the drawbacks of the products. For determining the necessity of an assumptions management framework, the metric used was the number of defects in end-products that could be traced back to invalid assumptions. Pilot projects were selected and case studies were performed to accept or reject the hypotheses. The case studies were performed according to the case study guidelines described by Kitchenham, Pickard and Pfleeger [30].

8.1 Case-Study Hypotheses

Since defects in products (logged by end-users) are used as a metric for the case-studies, below is an explanation of what exactly constitutes as a *defect due an invalid assumption.*

The term *defect due to an invalid assumption* is defined as follows. If the root cause of a defect in a component is due to (i) invalid assumption(s) made by a component on the environment or other components in the system, or (ii) invalid assumption(s) made by the environment or other components in the system on this component, it is classified as a *defect due to an invalid assumption.*

The term *algorithmic defects* of a component is used to broadly classify defects due to an incorrect algorithm or an incorrect implementation of an algorithm. They have the characteristic that they are not related to any other component or the environment in which the component functions in. Examples of algorithmic defects are the classic loop-off-by-one error and memory related errors like memory leaks and dangling pointers.

The areas of model-checking and unit-testing deal with reducing algorithmic defects. The scope of this research is to reduce defects due to invalid assumptions.

The statements of hypotheses for the case-studies are as follows:-

1. *In systems whose functionality is closely tied with their environment, there is a higher proportion of defects due to invalid assumptions than those related to algorithmic defects, i.e., among the set of algorithmic defects and defects due to invalid assumptions, over 50% are due to invalid assumptions*

2. *For majority of the defects due to invalid assumptions (> 50%), we can encode these assumptions in a machine checkable format and validate them or flag them as invalid (which can prevent such defects or warn in advance when the assumptions are violated)*

Hypothesis 1 is related to necessity of such a tool and hypothesis 2 is related to the feasibility in reducing defects using an assumption management framework. This chapter deals with proving hypothesis 1. Chapter 10 deals with hypothesis 2. The case-studies help in understanding the classes of errors in real-world systems. The observations made in the case-studies were used to tailor the AMF design and implementation, explained in the following Chapters 3 – 7.

8.2 Pilot Projects Selection and Plan for the Case-Study

8.2.1 Pilot Projects Selection

Three pilot projects were chosen for the case study. The following criteria were used in the pilot project selection

- The project functionality is closely tied in with the operating environment, which is the characteristic of most embedded and real-time systems.

- The list of defects or the design of the project is publicly accessible for any end-user to log and analyze the defects or study the design.

- Domain expertise (of the author) to analyze the cause of defects and study the underlying assumptions, if any.

Based on the above criteria, the following three projects were selected.

1. Iperf: An open source network bandwidth measurement tool.
2. TinyOS: An open source operating system for embedded devices.
3. Inverted Pendulum Control System (IPCS): A real-time feedback control application that balances an inverted pendulum.

Iperf and TinyOS have facilities, where end-users of the software can log defects and the database of these defects is publicly accessible.

IPCS, on the other hand, is not a downloadable system and it does not have a public repository of defects. It consists of both hardware and software components. It is analogous to how practical systems are built using COTS software and hardware components; and hence was chosen for the case-study.

Hypothesis 1 is tested for Iperf and TinyOS.

8.2.2 Testing the Hypothesis

The following methodology was used for testing the hypothesis 1. For Iperf and TinyOS, defects were carefully chosen, to make the study as exhaustive as possible. Over 100 defects were analyzed for each of them. The following actions were taken for each defect. If the defect were clearly due to (an) invalid assumption(s), a score of 10 was given for the defect. If the defect were clearly an algorithmic defect, and not related to invalid assumptions, a score of 0 was given. If the defect was related to, but not entirely due to invalid assumption(s), a score of 5 was given for the defect. Defects that were related to installation, compilation problems and version control, were considered as defect with confounding factors and they were disregarded.

In summary, for each defect D_i that was selected for analysis, a random variable X_i indicating the score for the defect was assigned as follows.

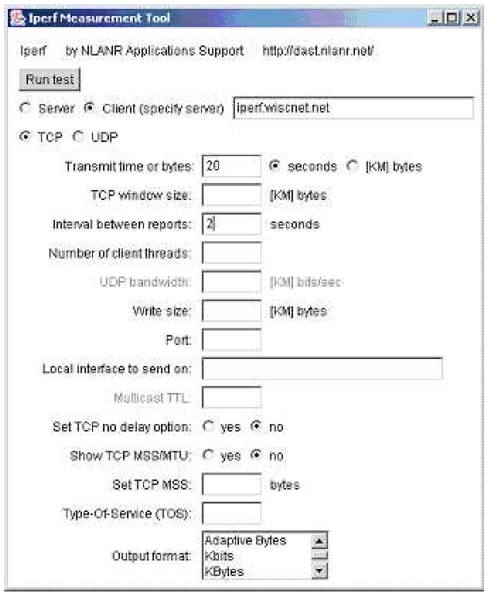
$$X_i = \begin{cases} 0, & \text{if } D_i \text{ is clearly not related to invalid assumptions;} \\ 5, & \text{if } D_i \text{ was related to, but not entirely due to invalid assumptions;} \\ 10, & \text{if } D_i \text{ was clearly due to invalid assumptions} \end{cases} \tag{8.1}$$

Assuming a normal distribution of defects, the mean and standard deviation for the X_i's were calculated. As in standard hypothesis testing, the null hypothesis 1 was accepted if the hypothesis was valid for the 95% confidence interval, and rejected if it was not valid for the 95% confidence interval. In case of rejection of the null hypothesis, the probability of incorrectly rejecting the hypothesis is given.

8.3 Description of the Project Functionality

A background of the projects being evaluated is given in this section to ensure that the applications have close interactions with the environment and are suitable for the case studies.

8.3.1 Iperf


Iperf Measurement Tool

Iperf by NLANR Applications Support http://dast.nlanr.net/

Run test

Server [✓ Client (specify server)] iperf.wiscnet.net

[✓ TCP] UDP

Transmit time or bytes:	20	[✓ seconds] [KM] bytes
TCP window size:		[KM] bytes
Interval between reports:	2	seconds
Number of client threads:		
UDP bandwidth:		[KM] bits/sec
Write size:		[KM] bytes
Port:		
Local interface to send on:		
Multicast TTL:		
Set TCP no delay option:	yes [✓ no]	
Show TCP MSS/MTU:	yes [✓ no]	
Set TCP MSS:		bytes
Type-Of-Service (TOS):		
Output format:	Adaptive Bytes Kbits KBytes	

Figure 8.1 Iperf Bandwidth Measurement Tool

Iperf [31], [32] is a bandwidth measurement tool, which is used to measure the end-to-end achievable bandwidth, using TCP streams, allowing variations in parameters like TCP window size, number of parallel streams, Maximum Segment Size (MSS), read block sizes and TCP no-delay option. End-to-end achievable bandwidth is the bandwidth at which an application in one end-host can send data to an application in the other end-host. Iperf approximates the cumulative bandwidth (the total data transferred between the end-hosts over the total transfer period) to the end-to-end achievable bandwidth. Iperf is also used to measure the packet loss and jitter for datagram packets.

Iperf belongs to the category of active bandwidth measurement tools, *i.e.,* it actually sends data for short periods of time to determine the available bandwidth. There has been some research in the area of passive bandwidth measurement tools also, where the time separation between reception times of packets of known length is used to determine the end-to-end achievable bandwidth. This methodology does not work well in practice for high bandwidth networks due to coalescing of multiple packets on reception and delivering a bunch of packets to the application. For example, around 830,000 packets each of length 1500 bytes will be delivered to the application every second if the achievable end-to-end bandwidth is 10Gbps. But practical values of CPU interrupts are around 10ms in Linux, which means over 800 packets will be delivered on every interrupt. This renders passive measurement techniques not usable in practice for high bandwidth networks.

Iperf works very closely with its operating environment. For correct operation, it makes a number of assumptions. Most importantly, it assumes the operating system can guarantee enough buffer size, that is greater than the bandwidth-delay product for the end-to-end application. There are also assumptions made on the MTU, TCP MSS, the link layer compression function in cases of dial-up networks, and the system bus bandwidth while measuring bandwidth in high bandwidth networks. In addition, Iperf was originally designed for SunOS and Linux, but, due to widespread use of Iperf, it has been ported to various platforms which have resulted in platform specific assumptions.

Since Iperf is shipped as a complete software module with configurable parameters, entire Iperf module is viewed as a single software component.

The list of Iperf assumptions that have resulted in defects is cataloged in Appendix A.2.

8.3.2 TinyOS

TinyOS [7] is an open-source operating system designed for wireless embedded sensor network devices. It has an event driven architecture, and all the software functions that need to be executed on the embedded sensors need to be loaded as a single application. TinyOS has a single shared stack and there is no user/kernel space differentiation. The OS is designed for low overhead, very small memory footprint and low power consumption. The memory for the applications are assigned at compile time and there is no dynamic memory allocation and no pointer arithmetic.

There are two basic constructs for execution in TinyOS. a) Tasks and b) Events. There are two threads in TinyOS. One thread executes tasks with bounded number of pending tasks in the queue. These tasks are executed in FIFO order. The other thread handles events; where hardware interrupts trigger the lowest level events. Events can signal other events, post tasks and/or call other commands. Software events propagate from lower level to upper level through function calls.

TinyOS also makes quite a few assumptions on the hardware, operating environment and the types of applications that run on it. Most of the data exported to applications are through the sensors attached to the device. Hence, there are quite a few implicit assumptions made on interpreting the data and commands to request the data. The list of defects due to invalid assumptions is presented in Appendix A.3.

As in the case of Iperf, TinyOS can be viewed as a single software component for the user.

8.3.3 Inverted Pendulum Control System

Inverted Pendulum Control System (IPCS), is a simple real-time control system. Unlike Iperf and TinyOS, there is no public repository of defects. On the other hand, the system is built using integration of COTS components, and thus, this application will serve as a valuable case-study for implicit assumptions made between the components. The system consists of 4 main software components.

- Physical plant : The inverted pendulum
- Sensor: Detects the angle and track position of the pendulum

- Controller: Calculates the new position of the pendulum, depending on the current position in the track and the angle.
- Actuator: The component that sends the electrical signals to the DC motor of the cart that carries the pendulum.

The components are shown in Figure 8.2. The plant (hardware) consists of a pendulum which sits on a two dimensional linear track. The pendulum can be moved left or right on this track, the pendulum sits on a cart which can be moved along the track. The movement of the cart is controlled by a DC motor which gets its input from the actuator.

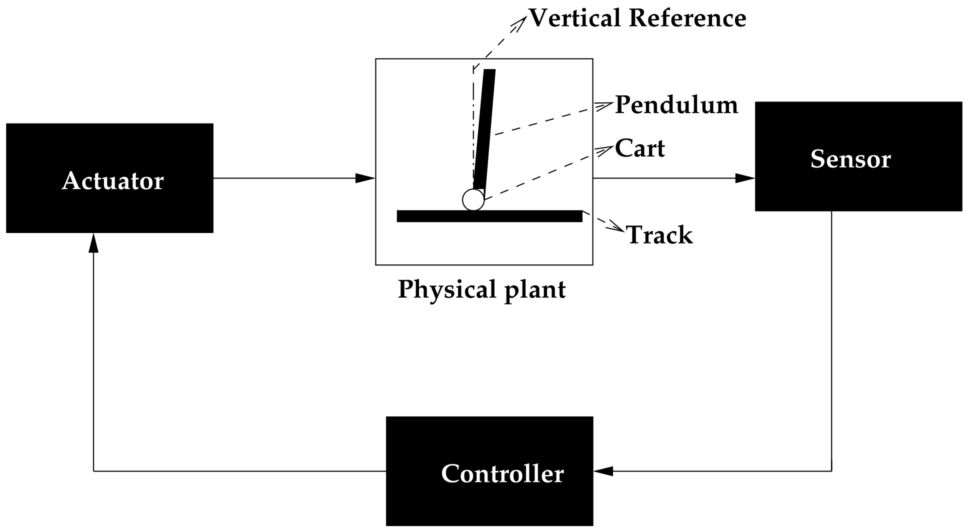


Figure 8.2 The Inverted Pendulum Controller

As the inverted pendulum plant is a hardware which needs to be controlled, we abstract it using a software component called the *plant proxy* which will export the important properties of the inverted pendulum plant, and will interface with the hardware plant to expose these properties. Thus the software components that we need to deal with are simplified and shown in Figure 8.3. There are four software components - Plant proxy(PP), Actuator(A), Controller(C) and Sensor(S).

Also, this system will highlight the interactions between the different sub-components that are used to build the system. The list of assumptions made by the inverted pendulum is given in Appendix A.1.

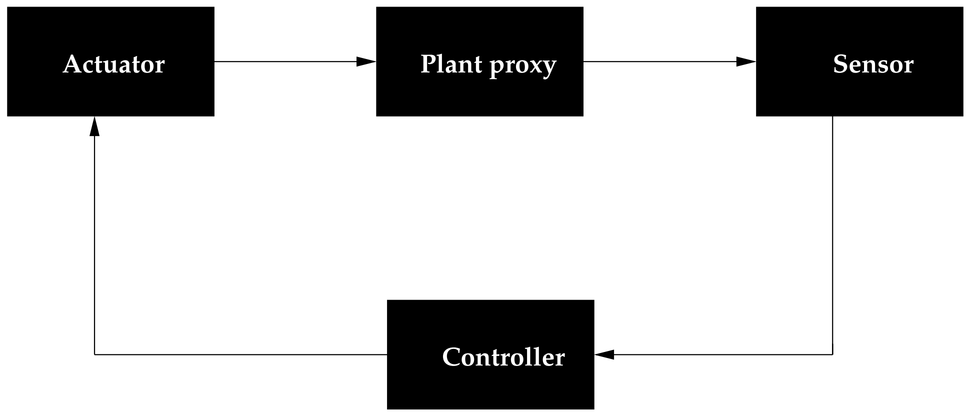


Figure 8.3 System View with Plant Proxy

8.4 Case Study Results

8.4.1 Testing Hypothesis 1 for TinyOS

The SourceForge repository for TinyOS has around 160 defects which have been assigned the status *closed*. Only *closed* defects were considered since we can have a higher confidence level for the cause of the defect. Out of these 160 defects, many of the compile problems and installation problems were discarded since we do not consider a product complete with these problems [1]. There were some problems which showed up in the end-product which were related to incorrect assumptions made by the software on the environment during compilation and install time. We do analyze these defects. After careful analysis, around 47 defects were chosen for further consideration.

We encountered the following class of invalid assumptions among the 47 defects chosen for analysis.

- *Mode in which the software runs (simulation or actual run).* Since the code written for the actual run can be run in a simulator, the simulator makes incorrect assumptions on the running environment. For example, in defect # 1084879, TinyOS makes an incorrect assumption on the type of hardware the application will run on. In defect # 947169, TOSSIM (TinyOS simulator) replies with an acknowledgment for a packet even when the

[1] Installation and compiler related problems are definitely not algorithmic defects. If they are considered as assumption management issues, the hypothesis 1 will trivially hold good

node is switched off. This cannot happen in actual run. Most of the platform related errors also belong to this class.

- *Implicit assumptions on length of data structures.* In defect # 1055439, the *Bcast* module transmits only maximum length broadcast packets.
- *Hardware limitations and resource related assumptions.* Under certain overload conditions, TinyOS timer cannot function well if granularity is less than 10ms. In defect # 913123, there is an undocumented TinyOS assumption which allows only 7 tasks to be in the tasks queue at any point in time.
- *Use of deprecated data structures.* For example, in defect # 1033732, Java communication tools make incorrect assumptions on message size.
- *Value range errors.* In defect # 979119, node ID is assumed to be 8-bits while 16 bit node IDs are allowed. In defect # 900058, an *int* is used for measuring time, while the value can be in the range of *long*. In defect # 833450, an incorrect answer is given with negative arguments.
- *Data interpretation errors.* In defect # 891749, a module dumps low probability values in exponential format, which cannot be read by the importing module. In defect # 960088, a *send* command with length 0 can break the module.

At least 8 of the defects were related to assumptions made on parameters which are not present in the software component.

All defects related to initialization, classical loop off-by-one cases, bit shifts, certain mutex related defects were classified as algorithmic defects (related to model checking and/or unit testing). Out of the 47 defects selected, 23 of them were consisting of confounding factors, *i.e.,* the root cause could also be traced to incorrect installation, compiler issues and/or version control. They were filtered out. Out of the remaining defects studied, the following were the values assigned to the score X_i for each defect.

- 15 defects were related to invalid assumptions. Out of these 11 of them clearly had invalid assumptions as the root cause, and for every defect D_i in this category, X_i was assigned a score of 10. For every defect D_i which was related to, but not entirely due to invalid assumptions, X_i was assigned a score of 5.

- 9 defects were clearly not related to assumptions but were algorithmic defects and for every defect D_i in this category, X_i was assigned a score of 0.

For testing a hypothesis, the following procedure is used. If the obtained mean, μ, of the X_is $\geq$ a minimum value μ_0 for a normal distribution with n samples at a confidence level of $(1-\alpha) * 100\%$, then

- We normalize the distribution to $N(0,1)$ for n samples with mean $\overline{X}$ and standard deviation σ and minimum expected value μ_0 using $u = \frac{\sqrt{n}(\overline{X}-\mu_0)}{\sigma}$
- For a confidence level of $(1-\alpha) * 100\%$, the rejection test is $u \leq -K_\alpha$, where

  $$\int_{-K_\alpha}^{\infty} \frac{1}{\sqrt{2\pi}} e^{\frac{-v^2}{2}} dv = 1 - \alpha$$

  The value of K_α is found from a reverse lookup in the standard Z-table which gives $P[Z < x]$ for different values of x, x being a $N(0,1)$ random variable.

For our case study, with values of $\alpha = 0.05$ (95% confidence level), $\mu_x = 5.4167, \sigma = 4.6431, \mu_0 = 5$, the value of $u = 0.4397$ and $-K_{0.05} = -1.65$. Since $u > -K_{0.05}$, **the hypothesis 1 is accepted at 95% confidence level** $\square$

8.4.2 Testing Hypothesis 1 for Iperf

Classification of defects for Iperf was harder than TinyOS since Iperf has a mailing list for issues and not a bug repository as in TinyOS. A concrete definition of the defect was formed after analyzing a set of mails related to a defect. This step is laborious, but it gives a clearer description of the cause of the defect. After this step, all defects that had a resolution and were fixed, were treated similar to the *closed* status defects in TinyOS. After analyzing the list of defects, 32 *closed* defects were selected. The analysis for each defect to assign scores was similar to that of TinyOS.

These were the main classes of defects related to invalid assumptions.

- *Overflow due to use of 32 bit data structures.* Defect *#feb03/msg00036* was due to incorrect reports in operating systems which did not provide an equivalent of *int_64*. If such systems are connected to high bandwidth networks, when the total amount of data transferred is > 4GB, Iperf will have incorrect throughput summaries.

- *TCP/UDP Window size assumptions.* Defect # *apr04/msg00013* was due to Iperf statically setting the minimum window size to 2048 bytes to prevent poor performance. This will preclude testing in extremely low-end devices, like IP enabled sensors. Many other operating systems have a minimum window size, example SunOS 8 has a minimum of 4.5KB and Linux has a minimum of 256 bytes. As mentioned in defect # *jul03/msg00007*, Windows 95 and 98 do not support RFC1213 support for large window sizes. This will preclude setting a window size of greater than 64KB[2]. Even in operating systems that allow large window sizes ($> 64K$), the administrator(root) can set limits for the maximum buffer size. A very important assumption is that TCP slow-start period is less than 2-3 seconds in most networks which enables Iperf to report fairly accurate bandwidth with default options (single threaded and 10 second test).

- *OS optimizations related assumptions.* The window size requested by the user may not be the exact size returned by the operating system, even if the requested value is less than the maximum allowed window size, since operating systems optimize on various factors like page size, etc. Defect # */jul03/msg00029* is related to this issue. Defect # */aug03/msg00006* is caused because of the compression feature of systems connected to dial-up networks. In such systems, the link layer compresses the data before transmission. This can lead to higher bandwidth reported, if uncompressed data is used for bandwidth measurement.

- *OS non-conformance to specified standards.* There were quite a few defects in this category. For example, FreeBSD has a IPv6 network stack, but it has compatibility issues with IPv6 and IPv4 working together. Windows NT discards ICMP error messages on UDP errors. This will have the Iperf client waiting for a UDP FIN. Win2K does not support setting of ToS (Type of Service) bits. Win32 has the UDP as a single thread per port, which precludes server side UDP tests for multiple clients.

[2]As a general rule, the TCP window size must be at least equal to end-to-end bandwidth times the delay, for TCP to hold the unacknowledged packets in the window for retransmission

- *Feature interaction related assumptions.* For example, in defect # */feb04/msg00008*, when users specify both number of packets and time to run in UDP mode, one of the option is quietly ignored. When MTU option is set, it has to be less than TCP window size, otherwise UDP FIN will not be received.

- *Resource related assumptions (other than TCP/UDP windows).* Iperf assumes system bus bandwidth is greater than the end-to-end network bandwidth. This may not be true in very high bandwidth networks (Iperf has been used to measure bandwidths greater than 10Gbps). Defect # /apr04/msg00023 was due to the user trying to launch 250 threads in an Iperf client. Defect # /apr03/msg00008 occurs because Iperf uses a tight-loop timer to measure the UDP jitter accurately at a specified bandwidth.

- *Assumptions on default actions in presence of multiple alternatives.* The defect # */feb03/msg00016* is due to the OS binding to the first available Network Interface Card (NIC) in presence of multiple NICs in a machine. In defect # */mar04/msg00047*, Win32 returns the local address, not the bound multicast address for the $getSockName()$ API.

Similar scores were assigned to the random variables X_i's and the following was the summary of the Iperf defect analysis.

- 26 defects were related to invalid assumptions. Out of these 15 of them were only due to invalid assumptions and for every defect D_i in this category, X_i was assigned a score of 10. For every defect D_i which was related to, but not entirely due to invalid assumptions, X_i was assigned a score of 5; there were 11 defects in this category.

- Some defects were clearly not related to assumptions but were algorithmic errors and for every defect D_i in this category, X_i was assigned a score of 0. There were 6 defects in this category.

For Iperf's case study, with values of $\alpha = 0.05$ (95% confidence level), $\mu = 6.40625, \sigma = 3.8592, \mu_0 = 5$, the value of u is 2.0612 and $-K_{0.05} = -1.65$. The hypothesis is trivially satisfied as $u > -K_{0.05}$ $\square$

	TinyOS	Iperf
Total # of Defects Considered	24	32
# of Defects with $X_i = 10$	11	15
# of Defects with $X_i = 5$	4	11
# of Defects with $X_i = 0$	9	6
μ_x	5.417	6.4063
σ_x	4.643	3.8592
u	0.440	2.0612
$-K_{0.05}$	-1.65	-1.65
Accept/Reject Hyp 1	**Accept**	**Accept**

Figure 8.4 Summary of Testing Hypothesis 1 for Iperf and TinyOS

8.5 Results Summary and Key Observations

For both Iperf and TinyOS, it is proven that there are more defects in the released software due to invalid assumptions than defects that are classified as algorithmic defects. As described in Appendix A, in the IPCS application also, the software components make a large number of assumptions on the operating environment and other software components. Invalidating any of these assumptions during system evolution (like replacing the pendulum plant, software) will cause the system to fail.

Many of the assumptions are related to parameters which are not ultimately represented in the physical code.

The classes of defects studied in this case study suggests that we need to deal with assumptions in various dimensions.

First, the time of encoding and checking the assumptions. Assumptions in the class of *length of data structures, the rules for interpreting the data* do not change as long as the software does not change. Assumptions like *OS specific optimizations* can only be checked when the software is deployed/compiled for a particular platform. Assumptions like *mode of running* can only be checked only during the actual run of the software.

Second, some of the invalid assumptions cause performance degradation while others cause the system to fail. For example, Iperf UDP mode will hog the CPU at low bandwidth transmissions, while OS's inability to grant adequate window size will result in Iperf reporting incorrect results. Thus, criticality versus performance (various gradations) is another dimension for classifying the assumptions.

Next, the number of assumptions uncovered suggests that when larger systems are built using these systems as sub-components, managing the assumptions is important. Abstracting assumptions that are not needed for the larger composition context is necessary.

These set of observations refined the design of our framework described in the Chapters 3 – 7.

8.6 Summary

In this chapter, the hypotheses pertaining to the necessity and feasibility of an assumptions management framework were stated. The necessity of an assumptions management framework, by studying small but complete systems, was confirmed. The necessity of a framework with similar objectives has been independently stressed by the 'Mishap Investigation Board' of the 'Mars Climate Orbiter' [16] disaster, and the 'Inquiry board' of the 'Ariane 5' [2] disaster. Iperf and TinyOS had a significant number of defects that users encountered, with invalid assumptions as the root cause.

CHAPTER 9

Analyzing and Correcting the Key Invalid Assumption of Iperf

In this chapter, an in depth look is taken at one of the assumptions made by Iperf that lead to erroneous reports in some cases and poor performance in other cases.

To recap, Iperf is a bandwidth measurement tool which is used to measure the end-to-end achievable bandwidth, using TCP streams, allowing variations in parameters like TCP window size and number of parallel streams. End-to-end achievable bandwidth is the bandwidth at which an application in one end-host can send data to an application in the other end-host. Iperf approximates the cumulative bandwidth (the total data transferred between the end-hosts over the total transfer period) to the end-to-end achievable bandwidth.

9.1 Key Assumptions of Iperf and the Errors They Cause

The default mode of Iperf makes the following critical assumptions:

Slow-start duration assumption: TCP slow-start duration will be less than 2 seconds for most networks.

End of slow-start assumption: Majority of the data transferred is after the end of slow-start phase.

Based on these assumptions, Iperf runs for a default time of 10 seconds while running bandwidth tests. The time to run is a configurable parameter, but the user must discern that Iperf was not out of slow-start phase for over 2 seconds to re-run it longer than 10 seconds. This requires manual inspection of the periodic bandwidth reports.

When the assumption on the TCP slow-start duration holds good, it will ensure that the cumulative data transferred by Iperf in 10 seconds a fairly accurate representation of the end-

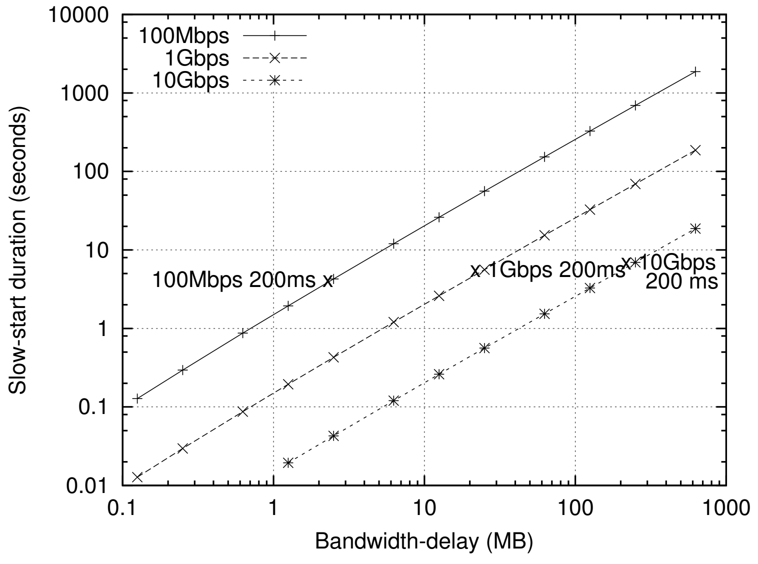


Figure 9.1 Estimated Slow-start Times for Various Bandwidth-delay Products

to-end achievable bandwidth. The duration of slow-start is roughly $\lceil log_2(ideal\ window\ size\ in\ MSS) \rceil * RTT$, where MSS is the TCP maximum segment size and RTT is the round-trip-time(delay) between the nodes. In practice, it is a little less than twice this value because of delayed acknowledgments. Estimated slow-start times for various bandwidth-delay products is shown in Figure 9.1. Since Iperf runs at the application layer in the network stack, it has no means of finding out the duration of slow-start. TCP is implemented in the operating system kernel and it abstracts this information. This can cause the following problems.

Case when assumption violation proves critical: Consider an end-to-end transfer across a 1 Gbps network with a round trip delay (RTT) of 200 ms. The bandwidth-delay product for 1 Gbps network with an RTT of 200 ms is equivalent to 16667 1500-byte segments. The slow-start duration, when a single TCP stream is used, will be approximately $\lceil log_2(16667) \rceil * 2 * 0.2$, which is 5.6 seconds. This will cause a lower than achievable bandwidth to be reported by Iperf when it is run for the default (10s) duration. This is a *critical* error.

Case when assumption violation leads to poor performance: Consider a case where the machines are connected in the same network, with a round-trip delay of 5 ms, and the achievable bandwidth between the machines is 10 Mbps. Iperf can report accurate bandwidth by running for well under a second. Running for 10 seconds wastes a lot of bandwidth. It causes a *non-critical* error, since bandwidth reported is correct (core functionality is not compromised), but it sends over 10 times the amount of data required to measure the achievable bandwidth. This leads to very poor performance.

9.2 Modified Algorithm of Iperf

The summary of the modified algorithm that takes care of the TCP slow-start duration assumption is as follows.

Step I: *Ensure that the end of slow-start assumption is satisfied, i.e.,* automatically detect when the TCP connection is out of the slow-start phase for a transfer.

Step II: *Transfer data for a small period of time, say 1 second, using the same connection.* Since TCP will be in steady-state, measuring the amount of data transferred for a short period of time will suffice.

To explain the modified algorithm, a brief background of TCP congestion windows is required.

9.2.1 TCP Congestion Windows

TCP Reno, the default implemented version in most operating systems, doubles the congestion window every RTT until it reaches the threshold congestion window size or it experiences a retransmission timeout. After the slow-start period, Reno sets the threshold window to half the congestion window where (if) it experienced a loss. Later, in the steady state, it increases its congestion window at a rate of one segment per RTT.

TCP Vegas doubles the congestion window only every other RTT so that a valid comparison of the actual and expected rates can be made [33]. It calculates the actual sending rate (ASR) and expected sending rate (ESR). It tries to keep $Diff = ASR - ESR$ between α and β. It increases the congestion window linearly if $Diff < \alpha$ and decreases it linearly if $Diff > \beta$. It tries to keep $Diff$ in the range $\alpha < Diff < \beta$. The most common values of α and β are 1 and 3 MSS respectively. Though TCP Vegas is fairer than TCP Reno, TCP Vegas clients may not receive a fair share of bandwidth while competing with TCP Reno clients [34] and hence TCP Vegas is not implemented in many practical systems.

9.2.2 Algorithm to Validate the End of Slow-Start Assumption

The objective of the algorithm is to determine the end of slow-start as fast as possible to minimize the amount of network traffic generated. The algorithm is designed for TCP Reno and it works for TCP Vegas without any modifications. Iperf uses Web 100 [35], which is a tool that exports key TCP-related variables to the application, in real-time, using the *proc* file system in Linux.

The algorithm is as shown in Figure 9.2. The procedure is called immediately after a connection is setup. It exits when the connection is out of slow-start. Initially, a flag is set indicating that the connection is in slow-start; as in Line (1) in the algorithm. The maximum segment size or the MSS value for the connection is initialized from Web 100. The congestion window value is initialized to zero; as in Line (3); and is continuously updated every iteration in the loop. The difference in the congestion window sizes between successive iterations determine whether the connection is out of slow start.

Inside the loop, Iperf sets the value for the RTT obtained from Web 100, as in Line (5). Sometimes, when no acknowledgments are obtained, RTT may be invalid. When the RTT is invalid, the process (or thread) sleeps for 20 milli-seconds; otherwise the thread sleeps for twice the duration of the RTT; as in Lines (6)-(9). The congestion window value is compared with the previously obtained congestion window value; as in Lines (10)-(12).

If the RTT is valid, then if the connection is out of slow-start, the difference in congestion windows is less than three times the value of MSS. In extremely low-bandwidth networks, the congestion window will never change and will always be below four times the RTT. If either of these conditions are satisfied; as in Lines (13)-(15), the connection is determined to be out of slow-start.

9.3 Implementation and Results

9.3.1 Implementation

Iperf runs in a client-server mode between the machines that the achievable bandwidth is to be measured. All the code changes in Iperf are confined to the client and only the host running the client needs a Web100-enabled kernel. The remote hosts, running Iperf servers, can

```
procedure DetectEndOfSlowStartForAConnection
//Entry - start of connection, Exit - end of slow-start
set slow_start_in_progress ← true//Connection initially in slow-start                                  (1)
set mss ← mss value from web100                                                                        (2)
set new_congestion_window ← 0//Used for comparison of old and new congestion window values             (3)
while (slow_start_in_progress)                                                                         (4)
    set rtt ← smoothed rtt value from web100                                                           (5)
    if (rtt is valid)                                                                                  (6)
        sleep for a duration of (2*rtt)                                                                (7)
    else                                                                                               (8)
        sleep for a duration of 20 ms                                                                  (9)
    set old_cwnd ← new_cwnd                                                                            (10)
    set new_cwnd ← congestion window value from web100                                                 (11)
    set diff_cwnd ← |new_cwnd - old_cwnd|                                                              (12)

    if (rtt is valid)                                                                                  (13)
        if (((old_cwnd = 4*mss) and (diff_cwnd < 3*mss)) or                                            (14)
            ((old_cwnd < 4*mss) and (diff_cwnd = 0)))                                                  (15)
                set slow_start_in_progress ← false                                                     (16)
```

Figure 9.2 Algorithm to Detect the End of TCP Slow-start Using Web 100

run the standard Iperf available for various operating systems. Using the Web100 user-library (userland), we can read the variables out of the Linux proc file system. The library routine to validate the end of slow-start assumption was around 150 lines of code.

9.3.2 Testing environment

For the testing environment, Iperf servers were running as a part of the IEPM-BW framework [36] on various nodes across the world. Bandwidth was measured from Stanford Linear Accelerator Center (SLAC) to twenty high performance sites across the world. These sites included various nodes spread across US, Asia (Japan) and Europe (UK, Switzerland, Italy, France). The operating systems in the remote hosts were either Solaris or Linux. The bandwidths to these nodes varied from 1 Mbps to greater than 400 Mbps. A local host at SLAC which ran Linux 2.4.16 (Web100-enabled kernel) with a dual 1130 MHz Intel Pentium 3 processor was used for the client.

9.3.3 Experimental Results

Achievable bandwidth was measured by running Iperf for 20 seconds, since the 10 second measurement, which is the default running time, would not suffice for a few high bandwidth-delay networks from SLAC; for example from SLAC to Japan. The 20 second measurements were compared with the bandwidths obtained by running Iperf, after end-of-slow-start assumption was satisfied. Bandwidth estimates obtained after the end-of-slow-start assumption was satisfied were differing by less than 10% when compared to the 20 second Iperf tests as shown in Figure 9.3.

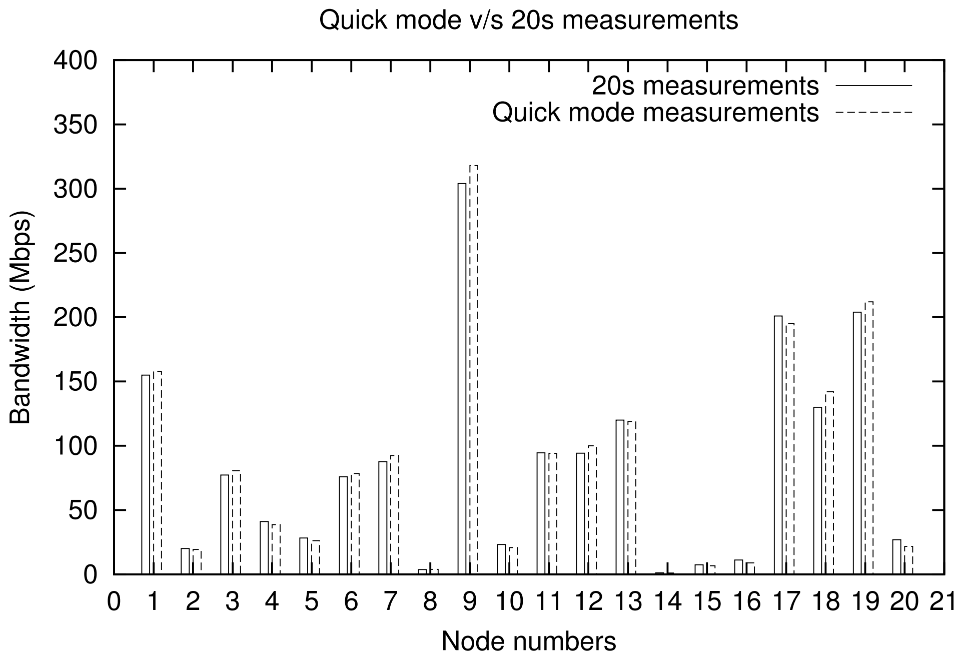


Figure 9.3 Slow-start Validated Quick Mode v/s 20 Second Measurements

The instantaneous bandwidth (*InstBW* in Figure 9.4) was calculated every 20ms averaged over the last 500 ms, and found that it increases and decreases with the congestion window as expected. The transfer from SLAC to Caltech shown in Figure 9.4 is the behavior which is expected in most transfers. The maximum TCP window size at both ends was 16 MB, which was greater than the bandwidth-delay product of about 800KB. The RTT is about 24ms (but Linux TCP stack rounds off the RTTs in 10s of milli-seconds).

In the SLAC $\rightarrow$ Japan transfer, across a high latency network, where the RTT was 140 ms, the bandwidth-delay product was about 6MB and the maximum TCP window size was 16MB.

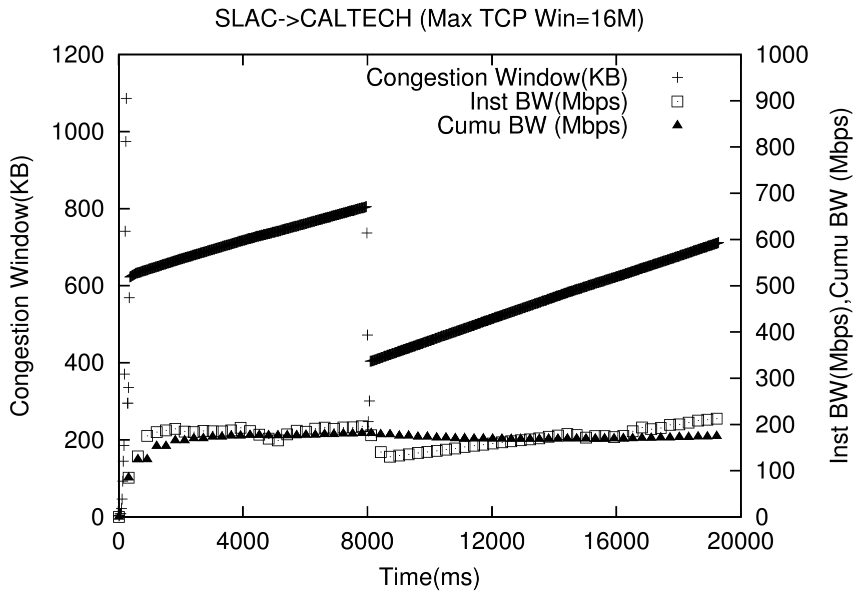


Figure 9.4 Throughput and Congestion Window (SLAC → Caltech)

In this case, the slow-start validated quick mode measurement gives a almost the exact value of the bandwidth, and results in savings of well over 90% of data transferred. The graphs of all the transfers similar to the one in Figure 9.4 are made available online [37].

9.4 Summary

This chapter explained the key assumption of Iperf, on the duration of slow-start. Iperf assumes that slow-start duration is less than 2 seconds and that most of the data transferred is after the end of slow-start. This is true for most networks, but the assumption is invalid for high-bandwidth networks. It requires manual inspection of periodic bandwidth reports for the user to discern the slow-start duration. This results in incorrect bandwidth reports, when Iperf is used to measure bandwidths across high bandwidth networks (like Gigabit networks) or even across high latency networks (like a transfer from US to Japan). Also, in many local area network bandwidth measurements, the slow-start duration is much less than the 2 seconds, and a measurement of about 1 second would suffice, instead of the default 10 seconds. Again, this requires manual inspection of bandwidth reports.

The modified algorithm ensures that the slow-start assumption of Iperf is first validated, *i.e.,* Iperf can detect when it is out of slow-start, and later the bandwidth is measured for a short period, (1 second). This modified algorithm, results in (i) correct bandwidth reports for low-bandwidth and high-bandwidth networks alike, and (ii) savings of about 90% in terms of both network traffic generated and measurement times.

CHAPTER 10

Evaluation of AMF and Comparison with Related Work

The first part of this chapter provides an evaluation of AMF in terms of performance, scalability and its ability to encode and validate assumptions. The second part of the chapter is devoted to related work and it compares AMF with the related work.

10.1 Evaluation of AMF - Performance and Scalability

This section describes the performance tests performed on AMF implementation. All the performance tests were conducted on a machine with an Intel T2300 processor, 1 GB of RAM running Windows XP. The Java version used was J2SE version 1.5. AMF implementation was invoked from a command-line interface (not within Eclipse). To test the scalability of AMF, the size of input in terms of number of assumptions and guarantees were increased from 1400 to 14000 in steps of 1400. The number of lines in the input was increased from 11300 to 113000 in steps of 11300. The tests recorded the times for composition, parsing, code-generation and Java compilation. Tests were also conducted for validation of assumptions without calls to library functions. These validation tests exerted the AMF translation times from fully-qualified names to method names and the method lookup within the guarantees class, in addition to executing the body of the assumption.

10.1.1 Performance of the Composition Matching Algorithm

Section 5.2 gives the matching algorithm for the composition of components, that finds the list of matched and unmatched assumptions (and guarantees). For a system with no library assumptions, the time complexity was $\Theta(n_a+n_g)$, where n_a and n_g are the number of assumptions and guarantees in the system respectively.

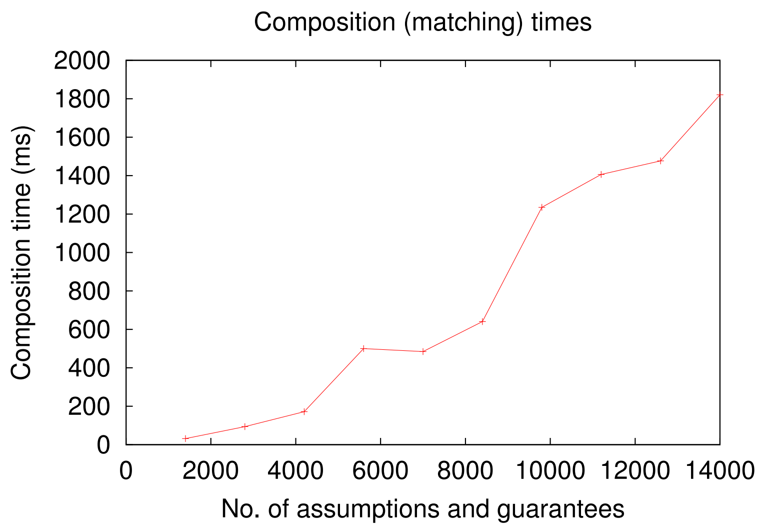


Figure 10.1 Performance of the Composition Matching Algorithm Implementation

The running time in presence of library assumptions and guarantees is $\Theta(n_a + n_g + \widetilde{n_a}.n_{\overline{a_x}})$, where $\widetilde{n_a}$ is the number of library assumptions and $n_{\overline{a_x}}$ is the average number of components that need to provide guarantees for each library assumption. This aspect is entirely dependent on the architecture specification. If all library assumptions are matched, then there are a total of $n_a + n_g + \widetilde{n_a}.n_{\overline{a_x}}$ assumptions and guarantees in the system and the composition time is linear in the order of number of assumptions and guarantees in the system. On the other hand, if most of the library assumptions have no matching guarantees from all the components that should provide matching guarantees, then on an average for each library assumption, there are $n_{\overline{a_x}}$ matching checks in dependent components to tag the assumption as unmatched.

Assumptions (non-library) were created in increasing numbers from 1400 assumptions and guarantees through 14000 assumptions and guarantees, in steps of 1400. Each result was averaged over 3 executions. As expected, the composition times grew almost linearly on the whole. While inspecting composition times in shorter time-frames, there were sharp increments in composition times, followed by relatively slower increments in composition times. It is due to the following reason. Java's implementation of Hashtable tries to keep the load factor below

0.75. When the number of elements in the Hashtable increases beyond this factor, it creates a new Hashtable with a little over double the current capacity and copies the contents of the old Hashtable to the new one. Even with this factor taken into account for the running time, the amortized worst-case complexity of the algorithm is linear. The results of the performance test for the composition matching algorithm is shown in Figure 10.1

10.1.2 Performance of Parsing, Code-Generation and Java Compilation

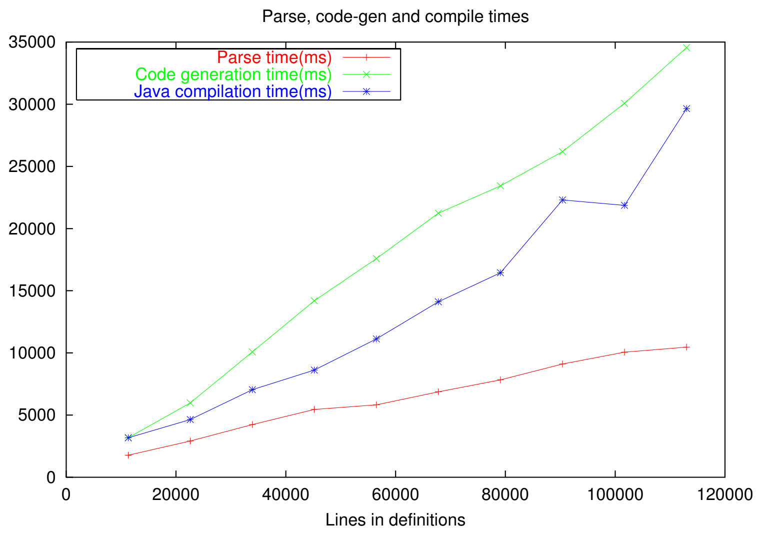


Figure 10.2 Parsing, Code-Generation and Compilation Performance

To test the scalability of AMF, the size of the textual input was increased from 11300 lines to 113000 lines of code in steps of 11300 lines. Also, correspondingly the number of components increased from 200 to 2000. AMF generates one Java class per component. The main steps in initializing the framework for an input specification is to (i) parse the input; if no syntax errors are found (ii) run the composition matching algorithm (iii) generate Java code for the matched assumptions and guarantees (iv) compile the Java code.

In the particular set of inputs, all assumptions and guarantees were matched. The results of this scalability test is presented in Figure 10.2. The total time for parsing, composition, code-generation and compilation is 80 seconds for 113000 lines of code, with 2000 components. It is also noticed that the code-generation time for the Java classes is a constant factor more that the Java compilation time. This is an acceptable benchmark since the Java compiler is a native Windows executable and the code generation is through Java code that has additional overhead.

10.1.3 Evaluating the Time to Validate Assumptions

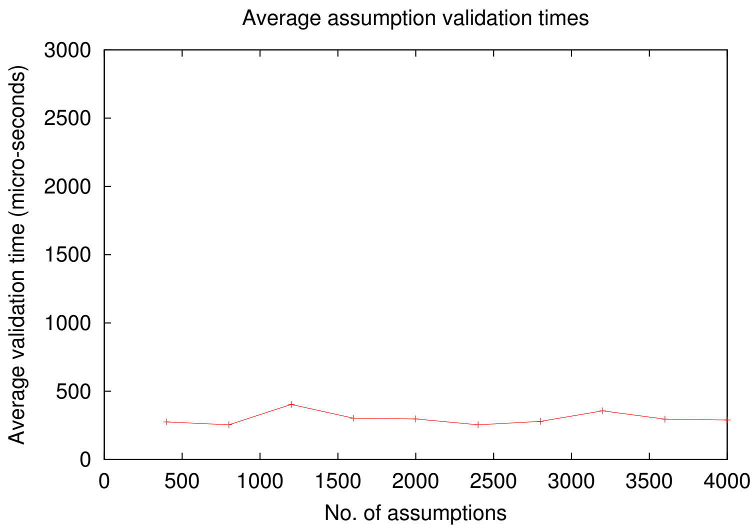


Figure 10.3 Average Assumption Validation Times

Since AMF provides facilities to validate assumptions given the name of the assumption, the time to validate an assumption was tested. For this, assumptions were generated with simple logical and arithmetic operations - average of 5 logical and arithmetic operators per assumption, and on an average 3 input formal parameters per assumption. There were no external method calls from the assumptions. This test was designed to evaluate the translation and method lookup time in addition to the execution of the body of the assumption. Assumptions were generated (with matching guarantees) from 400 through 4000 in steps of 400 and the average time to validate an assumption given its name was evaluated. For a given set of assumptions, all the assumptions were validated once by calling the *validateAssumption*() method given the assumption name. The average assumption validation time was almost a constant, as expected. The results of this test are shown in Figure 10.3.

10.2 Evaluation of AMF in Encoding and Validating Assumptions

There are two primary factors that evaluate the effectiveness AMF for managing assumptions. AMF must be able to encode the assumption in a machine checkable format and be able to set a policy on when to validate the assumption. The next factor is the ability of AMF to validate the assumption.

In terms of encoding the assumption in a machine-checkable format, it is possible to encode all the assumptions in the defect list (those entirely due to invalid assumptions) in a machine-checkable format.

In terms of validation of the assumptions, all *static* and *system-configuration* assumptions can be validated before execution of the component begins. For this, access to the binaries and in some cases the source code of the component, and the documentation of the APIs exposed by the components are required.

Dynamic assumptions require modification of the source-code of the components. Also, a translation between the native language in which the component is developed to Java is required for invoking assumptions validation from the source-code of the component. In some cases, like assumption validation from Iperf source-code, this is not a problem if a middleware service like CORBA [38] is used, which allows components developed using different languages to communicate in a standard format. For TinyOS, this poses a problem, since the resources in sensors are constrained and it is currently not feasible and not practical to allow such costly calls to be made from the source-code of the component. This can be addressed by developing translations between AMF and other languages that components use, when these components cannot invoke Java or middleware routines. The method body of the assumptions and the guarantees need to be translated to the native language of the component.

Here are the list of assumptions that are classified as *dynamic* assumptions.

- Iperf: Assumption that the network is the bottleneck will be invalid during high bandwidth transfers. Host configuration can be the bottleneck for the bandwidth reported in high bandwidth networks. Based on the bandwidth measured, Iperf needs to report a warning when bandwidth nears or exceeds a limit dependent on the host configuration. Need to modify the Iperf source to check this assumption before final bandwidth report.

- Iperf: User application assumes both requests from (-t) and (-f) options are satisfied, when used together. When user application specifies the time-to-run (-t) option and the file-transfer (-f) option, depending on the network bandwidth and the time for which Iperf is run, one of the requests may not be satisfied. Need to modify Iperf source to check this assumption before final bandwidth report.

- Iperf: Assumption made by bidirectional tests about the maximum data transferred. Iperf source needs to be modified to check if this assumption on the maximum data transferred holds good before final bandwidth reports.

- Iperf: User application's assumption on Iperf's CPU utilization. Iperf uses tight-loop timers to precisely measure delay and jitter for UDP transfers. This may spike CPU utilization and cause problems for the user application. This is not exactly a defect, but a warning needs to be printed in such cases.

- TinyOS: Assumption on packet-sizes. TinyOS source needs to be modified to check if the assumption on maximum length packets holds good.

- TinyOS: Assumption due to mode of running: During simulation, TOSSIM [39] assumes all nodes are ON by default. This assumption needs to be validated during runtime in TOSSIM, which will prevent TOSSIM from delivering packets to nodes that are turned off. This can never happen in an actual run.

- TinyOS: Assumption due to mode of running: During simulation (in TOSSIM), invalid assumption about the address type causes all listeners to reply to unicast packets. Need to validate the assumption that listener address is multicast during simulation. This can never happen in an actual run.

- TinyOS: An assumption (undocumented) that allows only 7 tasks to be in the queue. When tasks are posted dynamically (during execution), this assumption must be validated or an error must be reported.

Iperf source can be easily modified to incorporate these assumption validation routines from its source code. Iperf is implemented in C/C++, which is supported by middleware frameworks like CORBA. On the other hand, TinyOS runtime assumptions, like maximum number of tasks in the queue and assumption on packet sizes require a translation from AMF to nesC. Hence, AMF currently cannot be used to validate these dynamic assumptions (4 of them) of TinyOS.

The part of setting policies on when to validate assumptions is possible through saving the assumption definitions in the back-end XML format and fetching appropriate assumptions using XPath queries. This is explained in detail in Section 7.8.

Overall, AMF can be directly used to encode and validate all the assumptions, but for the four mentioned above. Validation of dynamic assumptions requires source code modifications in the logic (like the 4 assumptions of Iperf). 33 assumptions required no modifications of the source code, but for exporting the properties to enable a clean interface to validate the assumptions. This exporting of properties from the source code does not alter the behavior of the components.

10.3 Related Work and Comparison with AMF

10.3.1 Design by Contract

```
put-child (new: NODE) is
– Add new to the children of current node
    require
        new /= Void
    do
        ... Insertion algorithm ...
    ensure
        new.parent = Current;
        child-count = old child-count + 1
end – put-child
```

Figure 10.4 Contracts Example

Figure 10.4 is an example taken from [40]. It explains the key principles of 'Design by Contract'. The main concepts of contracts are the notion of pre-conditions and post-conditions. For a routine or a function, *pre-conditions* are assertions that need to be satisfied by the caller (or the client) and *post-conditions* are assertions that are guaranteed by the supplier (the called routine). In addition to pre-conditions and post-conditions, there are *class-invariants* which need to be satisfied by all the instances of the class, and every routine must ensure that the class-invariants are preserved on exit, if they are satisfied on entry.

Hence, contracts is very useful in exposing restrictions of variables which are ultimately represented as code. It is strongly tied up with the source code. A variety of cases can be handled by contracts. For example in [41], it is explained how the Ariane 5 disaster could have been averted by using ‘Design by Contract’. In fact, this principle is highly suitable for specifying constraints on variables and objects that are physically represented as code. But, contracts cannot handle assumptions made by the software components that do not have a physical representation in the source code. In the example presented in Section 1.2.3, the *maximum sensing delay*, *maximum jitter* are not represented in the code and we cannot use contracts to ensure that the clients (callers of the routine) make incorrect assumptions about these parameters. It is typical in real-time systems software that all the assumptions are not ultimately represented as code.

Also, with the current COTS and custom software methodology, certain software modules are shipped with certification for properties like safety. The certification is subject to not modifying the source code. In such cases, it is difficult to apply the ‘Design by Contract’ principle. We will need a mechanism which can encode the assumptions made by the certified module explicitly, but without modifying the source-code.

In summary, ‘Design by Contract’ is an extremely useful principle and can be used to check for assumptions made on the variables present in the code. But, it cannot be effectively applied to encode assumptions made on parameters not present in the code, and in cases where the source code cannot be modified.

10.3.2 Real-time CORBA

CORBA [38] was developed as a middleware solution for integrating diverse applications within distributed heterogeneous environments. The main focus of CORBA was to provide a platform that allows for seamless integration of applications abstracting the communication layers or the network between the interacting applications. However, CORBA did not have provisions for specifying end-to-end QoS requirements / real-time constraints. To this end, Real-time CORBA developed by Schmidt *et.al.* [42] concentrated on providing specification mechanisms and architectural support to deliver end-to-end QoS to applications.

Notably, RT-CORBA provides standard interfaces that allow applications to configure and control processor, memory and communication resources through standard (ORB) operations. They also have an implementation of RT-CORBA, the details of which are available in [43].

RT-CORBA has been effective in making explicit the resources an application requires out of a distributed system and providing architectural support to guarantee these real-time requirements [44]. It is also effective in masking the complexity of mapping intermediate (network and processor) priorities and managing resources in these intermediate components (networks and processors). This in turn enables a software component to make explicit some of the assumptions a component makes w.r.t the environment, mainly the resources. It still cannot handle negotiation of semantic assumptions which real-time systems components make on the environment and other components.

For example, if there are two components in the system *a velocity sensor 'X'* and *a data collector 'Y'* using RT-CORBA to communicate with each other, RT-CORBA can be used to specify and validate the *maximum acceptable delay* between the components, since this concerns the resources being used in the system. On the other hand, RT-CORBA cannot be used (and was not meant) to specify semantic assumptions the software components make like *units of velocity, granularity of readings, minimum and maximum readings which can be considered valid* etc.

In summary, RT-CORBA is effective in making explicit the resource assumptions that the software components make on the environment, but it was not meant to handle other environmental and semantic assumptions that the real-time system software components typically make.

10.3.3 VEST

The Virginia Embedded Systems Toolkit (VEST) [45] is an tool for composing real-time systems. It is built using the Generic Modeling Environment [46]. VEST has a prescriptive aspects library, using which language independent advice can be applied to a design. It provides for dependency checks on aspects. It also provides a composition environment (GUI) to compose embedded systems, perform dependency checks, and invoke prescriptive aspects of a design. The prescriptive aspects can be used to compose non-functional properties. However, it is remote from the actual running code where the software interfaces are actually used.

For example, for the acoustic sensor example in Section 1.2.3, VEST can be used to check if the sensor meets the data collector's requirements varying various real-time parameters like maximum delay, etc. Also, VEST can allow analysis of different aspects like redundancy and/or persistence of data. However, VEST cannot pinpoint that the implemented software interface of the module makes semantic assumptions on the units of acoustic data, sensing delay etc., while there is only one 16 bit value returned by the *dataReady* event.

We believe that our research will compliment modeling tools like VEST, since our research makes the assumptions on the *implemented / shipped* software interfaces explicit and machine checkable, and tools like VEST perform dependency analysis (like whether timing requirements are met) on inter-connected real-time components.

10.3.4 SpecTRM

SpecTRM (Specification Toolkit and Requirements Methodology) is designed to assist in the development of software-intensive safety critical systems [47] [48]. The whole of SpecTRM methodology is based on a human-centric safety driven process of requirements methodology. The methodology is supported by an artifact called intent specification [49] [50]. This is an easy to read software model which can be executed and analyzed by formal methods also. Intent Specification starts from a human-readable specification for each requirement and this passes through various stages – system purpose, system design principles, black-box behavior, physical representation and system operations.

SpecTRM allows for forward and backward references during intent specification at various stages. For example, a clause in the *system design principles* stage may refer to a clause at the *system purpose* stage and vice-versa. The only part which is not formally tracked is assumptions which are just mentioned in human-readable format.

The following example taken from [49] will make it clearer. "R1: Provide collision avoidance protection for any two aircraft closing horizontally at any rate up to 1200 knots ... . *Assumption:* This requirement is derived from the assumption that commercial aircraft can operate up to 600 knots and ... "

In SpecTRM, the requirement *R1* can be formally tracked across various stages of the software life-cycle. But, the assumption on the requirement is only stated informally and is not tracked across the software life-cycle. Also, SpecTRM translates the requirements into an FSM

```
component PendulumController is
  in setpoint: SETPOINT;
  in status: STATUS;
  out actuator: VOLTAGE;
end PendulumController;
in setpoint: SETPOINT (cm) is
  range - (Track.Length/2 - Stability_Range) to Track.Length/2 - Stability_Range;
  delta - Max_Step_Size to Max_Step_Size;
  every 200 ms;
end setpoint;
```

Figure 10.5 Example of Impact Analysis Framework

with all the transition rules [51]. Tools like these are useful in designing safety critical systems from the scratch.

But, while building systems out of COTS systems, it is firstly very difficult to analyze the internal transition rules of the shipped component, we need to work on the software interface and the assumptions made by the software component. In the example in Section 1.2.3, if the acoustic sensor is shipped by vendor A and the data collector by vendor B, we will only have the software interface and the assumptions made by the software component (usually as data-sheets) to work with.

In summary, SpecTRM is useful in building safety-critical systems from scratch, but while working with COTS systems (and black-box interface components) which only expose a software interface and a set of assumptions, it is difficult to apply this tool.

10.3.5 Impact Analysis in Real-Time Systems

In [52], an approach for modeling architecture for real-time systems and recording time-sensitive properties is discussed. Their work identifies semantic inconsistencies and system impact on the result of a change in the system. Their approach also deals with encoding undocumented assumptions. In this aspect, this work is closely related to what we are trying to achieve.

It makes use of the architecture description language (ADL) to specify component input and output ports. Also, it can make use of the ADL to describe the connections between components. For a Pendulum controller part of the component specification is shown in Figure 10.5, this example is taken from [52].

AMF, on the other hand, works with architecture specification (AADL), dependency management tools like DMF, and provides native integration with Java source code. Also, the classification of assumptions introduced in AMF makes it very easy to manage assumptions as the number of assumptions grows.

10.3.6 Industry Solutions for Specific Problems

There have been industry solutions which are specific to the kind of products developed by individual industries. Notable solution in the areas of industrial automation, consumer electronic devices, timing property specification and cell-phones/PDAs have been mentioned below.

In the area of industrial automation, IEC standard for the development and use of componentized software for Programmable Logic Controllers (PLCs) was developed by PLCopen [53], a consortium of PLC users and producers.

For consumer electronics devices, Philips has developed and has been using an architectural description language to build products like TVs, VCRs, recorders and combinations using a component model called Kaola [54], [55], [56].

There have been some tools developed by Articus Systems for operation with the use of Rubus [57] for syntactic support of system data and support for specifying timing properties.

PECOS is a similar project [58] is aimed at embedded devices like cell-phones and PDAs. These models though very useful for specific domains, do not address the problem of integrating a large scale system from COTS components which may contain both real-time and non-real-time systems and from varied vendors.

10.3.7 Source-Code Related Assumptions Testing and Model-Checking

The concept of having monitors and proving the correctness of programs has its beginnings in the 1960s and 1970s [59], [60], but it has remained an elusive problem in terms of a complete solution. This is due to the inherent complexity of software. But, there have been many attractive solutions in specific areas within this field. Recently, there has been some work in specifying assumptions on the source-code and documenting assumptions on the architecture.

Explicitly modeling assumptions in product families which provides a basis for formally documenting assumptions and their relations with architectural artifacts is dealt in [61]. This work helps in understanding the software engineering and design decisions. Model checking tools like [62] have been very effective in guaranteeing the assumptions of the source code with the assumptions made at the design level. There are model-checking tools that aid programming by providing monitors for assertions made in temporal logic [63]. In JavaMOP [19], after the monitor (assertion) is setup, it provides violation handlers that are executed when the assertion is violated and validation handlers that are executed when the assertions are validated. There have been a large body of work that concentrates on the assume-guarantee methodology on the source code [64], [65], [66] and the design of the software. Interesting and one of the first applications of formal methods to software is described in [67].

To the best of the author's knowledge, the work in this thesis is the first to introduce the concept of classification and manageability of assumptions. Also, the work in this thesis targets environmental assumptions that are outside the software and are the root cause of many defects as encountered in the case studies.

10.4 Summary

This chapter provided an evaluation of AMF, in terms of its scalability, performance and its ability to encode and validate the assumptions encountered in the case studies.

The performance of the composition matching algorithm was as expected. It grew linearly with a running time of 1.8 seconds to compose 14,000 assumptions and guarantees. There were temporary surges in the running times due to the implementation of Hashtable in Java, which is used by the algorithm implementation to check for set membership. In terms of parsing, code-generation and Java code compilation, AMF provided an acceptable performance with performing all three tasks under 80 seconds when the input definitions was 113000 lines with 2000 components and 14000 assumptions. The assumption validation time was found to be a constant (independent of the size of the input), as expected. The maximum validation time was never over 500 micro-seconds, with simple assumptions consisting of 3 input parameters, 5 arithmetic and logical operations. Most of the time was spent in looking up the assumption method body, given its name, and translating the input qualified assumption body to its method name.

In terms of effectively encoding and validating assumptions, AMF was able to encode and validate 33 out of the 41 assumptions that caused defects in Iperf and TinyOS, with minor or no modifications to the source code. Also, with a middleware framework, AMF has the capability to encode *dynamic* assumptions for systems like Iperf. For systems like TinyOS, AMF needs to generate code in *nesC* language for the body of the assumption. Hence, 4 of the *dynamic* assumptions in TinyOS cannot be currently handled by AMF.

This chapter also provides comprehensive set of related work and compares their work with AMF.

CHAPTER 11

Conclusions

Invalid assumptions have been the root cause of failures in projects like the Ariane 5 and the Mars Climate Orbiter. The inquiry boards of these projects have independently stressed the development of a framework with objectives similar to that of AMF [2], [16].

This thesis examined this problem in detail and evolved a framework, which provides a vocabulary for discussing assumptions, a language for encoding assumptions in a machine-checkable format and facilities to manage the assumptions in terms of composition and setting policies on assumption validation. AMF allows assumption specification process to blend with source-code and architecture specification. This enables AMF to be applied to existing systems with minor or no modifications.

AMF provides a basic classification system for assumptions and guarantees, for easier manageability. Users can build domain specific classification schemes on top of this. This is explained in Chapter 3. The basic dimensions of classification for an assumption are (i) time-frame of validity (ii) criticality and (iii) scope. Guarantees are classified based on whether the values are provided explicitly by humans or they are obtained by executing a routine.

AMF provides a user-friendly language, as explained in Chapter 4, with minimal number of keywords for encoding assumptions and guarantees in a system. The logic to express the assumptions encompasses most logical operators in a high-level language like C or Java. AMF language makes provisions for components that may not know the set of dependent components in advance, like operating systems and middleware components, to express their assumptions. AMF also provides extensions to the language to express complex assumptions and guarantees using method invocations in high-level languages.

Composition of assumptions and guarantees allows AMF to be applied to larger systems in a scalable format, as explained in Chapter 5. AMF has a composition matching algorithm that runs in linear time ($\Theta(n_a + n_g)$, for n_a assumptions and n_g guarantees), when there are no library assumptions and guarantees. In presence of library assumptions and guarantees, it has a running time of $\Theta(n_a + n_g + \widetilde{n_a}.n_{\overline{a_x}})$; the additional factor is on the order of the average number of guarantees to be provided per library assumption times the number of library assumptions. All matched and validated assumptions can be tagged as private, and can be abstracted from the higher level system architecture. This greatly increases scalability.

AMF language specification allows assumptions to be directly invoked on the source code of the components. It is also integrated with AADL to allow a uniform view of components in terms of system architecture specification and assumptions specification. AADL integration also allows properties of hardware components to be exposed in a machine-checkable format. AMF provides language extensions that enable users to encode the immediate impact of assumption violations using standard vocabulary used by dependency management tools. This process to enabling AMF to encode assumptions across various aspects of software-engineering is termed as vertical assumptions management, which is explained in Chapter 6.

AMF provides a model-based implementation for managing assumptions using the Eclipse Modeling Framework (EMF), as explained in Chapter 7. Assumptions can be specified in a textual format or using a model-based GUI, which is automatically generated by EMF. AMF provides an implementation of the composition algorithm and amortized constant-time query operations after composition to check if an assumption (or guarantee) is matched. AMF provides functions to validate assumptions given its name. This, along with a set of functions that can be accessed natively using Java or from command-line, helps in enabling dynamic assumptions validation. The implementation is suitable for validating dynamic assumptions in non real-time or soft real-time systems. The facility of EMF to store the definitions in XML format makes it amenable to set policies on when to validate assumptions and what assumptions to validate.

Case-studies were conducted on representative projects, as explained in Chapter 8, to study the defects caused due to invalid assumptions and the nature of these assumptions. It was found that in two of the systems studied, there were a significant proportion of defects due to invalid assumptions than algorithmic defects. This independently provided a basis for developing AMF. The case-studies were also useful in studying the nature of the assumptions that cause defects.

The case studies helped rectify one of the key invalid assumptions of Iperf. The modified algorithm that first validates the assumption and then performs bandwidth tests results in correct bandwidth reports for low-bandwidth and high-bandwidth networks alike, and savings of about 90% in terms of both network traffic generated and measurement times, which is explained in Chapter 9.

The evaluation of AMF, which is explained in Chapter 10, in terms of scalability and performance indicated that AMF is scalable to be applied for large-scale systems. Also, AMF had capability to encode and validate majority of the assumptions encountered in the case-studies that lead to defects.

11.1 Further Applications and Extensions

11.1.1 Domain Specific Assumption Libraries

In addition to the basic classification scheme provided in AMF, one can develop domain specific assumptions and guarantees for components of the domain. These libraries can assist in rapid encoding of assumptions and increasing the applicability of AMF in specific domains.

For example, one can develop deadline and resource specific assumptions for real-time systems. As shown, in one study of e-simplex [68], the following critical assumption made by e-simplex about the system was revealed. *E-simplex is the only (real-time) process running in the processor.* Any process with a lower-priority than that of the complex controller which runs in the same processor may miss its deadline. Such assumptions are common in most real-time systems, and a library of such assumptions with a useful vocabulary will increase applicability of AMF. Similar libraries can be developed for real-time networking domain. For example, for correct operation, the real-time networking protocol, FAI-EDF [69] makes many assumptions like a) precise MAC layer time-stamping is available b) nodes do not send packets greater than their alloted budget c) all nodes can perform carrier-sensing function correctly.

11.1.2 Applying AMF to Large-Scale Sensor Networks

Sensor networks is a very interesting field of application for AMF, for it encompasses various domains such as networking, computational systems, sensor and control systems. Sensor networks are meant to monitor and sometimes control their environment. Based on the case studies, a large number of assumptions that can be encoded in a machine-checkable format is foreseen for such systems. The model-based GUI for AMF allows people with relatively less programming language experience to encode assumptions. The default values are filled in automatically and only valid values are allowed for enumerations (like classification information), which makes the task of encoding assumptions a lot simpler. It will be interesting to study the applicability of AMF for such large-scale systems.

APPENDIX A

Assumptions in Iperf, TinyOS and IPCS

The assumptions made by Iperf and TinyOS that lead to defects are discussed here. This was used as the basis for the case studies. Also, a description of the system architecture and the assumptions of the Inverted Pendulum Control System (IPCS) is given.

A.1 Description of the Inverted Pendulum Control System

Software interfaces in the system

There are four directly interacting software interfaces in the Inverted Pendulum Control System (IPCS). These software interfaces that are used to exchange real-time data during the operation of the system.

- Plant proxy → sensor
- Sensor → Controller
- Controller → Actuator
- Actuator → Plant proxy

A study was conducted on the source code of the IPCS system. It was found that the software interfaces were inadequate to represent the assumptions made on the environment and on other components in the system. Some assumptions were embedded in the source code as a part of the implementation. Some did not have a representation even in the source code.

For example, the software interface for the controller was as follows:

float calc_command_cx(**float** angle, **float** track);

The controller obtains the angle and track information from the sensor and it calculates the new value of the voltage that is to be given by the actuator to the DC motor controlling the cart. The only inference from the software interface is that the controller receives two 32-bit floating point parameters which represent the current angle and track positions respectively, and it returns an 32-bit floating point value which represents a voltage to be given by the actuator to the plant.

This simplistic interface is sufficient only when the whole system is developed by a single team where all the implicit assumptions are known, but in the IPCS, the physical plant (with a plant proxy) is supplied by a one vendor, the controller and the sensor are obtained from different vendors.

Assumptions made by the software components

It was found that over 40 assumptions were made by the software components. A few critical ones are listed below.

For a particular set of components interacting with each other through a software interface, it was observed that assumptions were made by both the provider and the caller of the software interface. Assumptions were also discovered between components that did *not* interact directly.

The assumptions between the set of components are as below.

1. Plant proxy $\leftrightarrow$ Sensor
    - (a) Track position units (meters in the source code)
    - (b) Angle units
    - (c) Maximum permissible velocity of the cart
    - (d) Valid positions of the cart on the track
    - (e) Reference line to measure the inclination of the pendulum (not found in source code, but assumed to be vertical)
2. Sensor $\leftrightarrow$ Controller
    - (a) Maximum permissible sensing delay

   (b) Maximum permissible jitter in sensing delay
   (c) Units of track position
   (d) Units of angle of inclination
   (e) Reference point of track position (end of track or center of track)
   (f) Reference values for angles (horizontal/vertical)
   (g) Maximum permissible error in sensing track position
   (h) Maximum permissible error in sensing angle position

3. Controller $\leftrightarrow$ Actuator
   (a) Voltage units
   (b) (Control) Computational delay
   (c) Valid values for voltage
   (d) Actuation delay
   (e) Minimum delay needed between new control commands
   (f) Maximum permissible delay between new control commands

4. Actuator $\leftrightarrow$ Plant proxy
   (a) Unit displacement/acceleration per unit volt
   (b) Maximum velocity of displacement
   (c) Frictional constants of the cart w.r.t the track

5. **Controller $\leftrightarrow$ Plant Proxy**
   (a) Mass of the cart
   (b) Mass of the pendulum
   (c) Frictional constants of the pendulum w.r.t the cart
   (d) Length of the pendulum
   (e) (Various) electrical constants of the DC motor
   (f) Wear of the motor gear (determines if/when the cart needs to be replaced and the displacement per unit voltage)

6. Global environmental system assumptions like control loop delay, etc.

Overall, over 40 assumptions made by the components that were not reflected in the software interface [1]. Any violation of the assumptions listed above will cause the **inverted pendulum control system to fail**. For example, if the pendulum is replaced and the new pendulum has a different mass (it will violate the controller assumption) and the system will fail.

In addition, one notices that though the controller and the plant proxy **do not interact directly** using software interfaces, assumptions exist between the controller and the plant proxy.

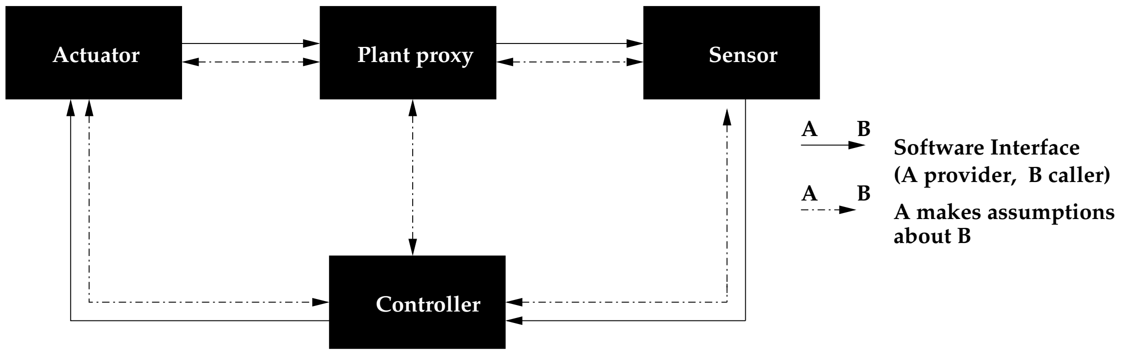


Figure A.1 Assumptions Made by the Software Components

Figure A.1 illustrates the system architecture alongside the set of components that have assumptions between them. It shows that (i) assumptions are made both by the provider and caller of the software interface (ii) assumptions exist even between components that do not have a direct interface in the system architecture.

Summary of observations

- The system consisted of four software components.
- There are over **forty** assumptions made by the software components about each other, about the physical plant and the environment. (even though, we did not study the electrical characteristics of the DC motor and the assumptions made on it).
- The system will fail if **any** of the assumptions listed above are violated.

[1]This statistic does not consider the DC motor assumptions, the full details of the motor constants are available in [70]

- Assumptions exist even between components that do not interact with each other through software interfaces.
- In directly interacting components, assumptions are made by the caller about the provider and vice-versa.
- Some assumptions are made about the system as a whole (global assumptions) and the operating environment

A.2 List of Defects Considered in Iperf

The table below indicates the list of defects considered in Iperf after interpreting the mails in the mailing lists. Relative links to the mail threads are given as identification for the defects. Interested readers can get further information from the website:

http://archive.ncsa.uiuc.edu/lists/iperf-users/ .

For all defects that have a score of 10, (defects that have the root cause as an invalid assumption), and a few defects with a score of 5, the classification information for the assumption and guarantee that is used to prevent the defect in future is given. The classification information is abbreviated as follows. For assumptions, for time-frame of validity, the notations are St for static, $SyCn$ for system configuration, Dy for dynamic. For criticality, the notations are Cr - critical and $NoCrX$ - non-critical at a level X. For guarantees, the notations are $HmEn$ for human-entered values, $McGn$ machine-generated values, Hyb for hybrid values.

Link (Iperf)	**Brief Analysis**	**Score**	**Classification**
/feb03 /msg00036.html	Incorrect throughput summaries. Assumes transfers are less than 4GB when using Linux 7.1. Did not find equivalent of int64 in Linux 7.1	10	$A \in SyCn, Cr$ $G \in McGn$
/feb03 /msg00031.html	User assumption on output format of Iperf. Causes problems while parsing Iperf output in a machine.	5	

Link (Iperf)	**Brief Analysis**	**Score**	**Classification**
/feb03 /msg00018.html	Assumes compatibility between IPv4 and IPv6 in FreeBSD APIs. Causes UDP mode to fail.	5	
/feb03 /msg00016.html	Assumes the first available NIC is connected to the multicast network	10	$A \in SyCn, Cr$ $G \in McGn$
/feb03 /msg00005.html	Iperf expects ICMP messages on UDP errors. WinNT discards ICMP messages on UDP errors.	10	$A \in SyCn, NoCr3$ $G \in McGn$
/mar03 /msg00008.html	User application assumes bandwidth reports on clients and server are the same. Not an Iperf defect, but explicit documentation required.	10	$A \in St, NoCr3$ $G \in HmEn$
/mar03 /msg00002.html	User application assumes a different definition of jitter. Iperf uses the standard definition of jitter as in RTP	5	
/mar03 /msg00000.html	User application assumes a different formula for data-rate calculation. Not an Iperf defect.	5	
/apr03 /msg00008.html	Iperf tight-loop timers increases CPU utilization. May need to make explicit the CPU requirements of Iperf	5	
/apr04 /msg00013.html	User application unable to set minimum window size for Iperf transfer. Actually, different operating systems have different statutes. For example, Solaris has a minimum of 4.5K, Linux 256 bytes. Iperf also has limitations on minimum window size of 2K to prevent poor performance (2K).	10	$A \in SyCn, Cr$ $G \in McGn$

Link (Iperf)	**Brief Analysis**	**Score**	**Classification**
/apr04 /msg00023.html	User application assumes Iperf can launch over 250 threads, but there are OS specific limits	10	$A \in SyCn, Cr$ $G \in McGn$
/apr04 /msg00001.html	Another jitter definition related assumption in the user application regarding PerfSocket UDP version.	10	$A \in St, NoCr3$ $G \in HmEn$
/apr04 /msg00028.html	For bi-directional tests, the conditions that are required for the tool to function correctly are not specified explicitly	5	
/apr03 /msg00000.html	Iperf assumes operating system support for Type Of Service (ToS)bits, which is not true for Win2K	5	$A \in SyCn, NoCr3$ $G \in McGn$
/jun03 /msg00025.html	Iperf assumes that the OS kernel allows setting different MTUs	5	$A \in SyCn, Cr$ $G \in McGn$
/jun03 /msg00004.html	Read buffer size set by user-app must be greater than the OS default MTU size.	10	$A \in SyCn, NoCr3$ $G \in McGn$
/jun03 /msg00001.html	Bi-directional testing assumes data transferred is less than 4 GB	10	$A \in St, Cr$ $G \in HmEn$
/jul03 /msg00022.html	Depending on the amount of data transferred, the -f option (that allows a file to be transferred) and the -t option (that allows transfer at least for a particular duration of time) may be incompatible	10	$A \in Dy, Cr$ $G \in McGn$
/jul03 /msg00007.html	Iperf assumption on the OS for RFC1213 support for large window sizes	10	$A \in SyCn, NoCr3$ $G \in McGn$

Link (Iperf)	**Brief Analysis**	**Score**	**Classification**
/aug03 /msg00006.html	Iperf assumes no data compression in the entire path. Results in higher bandwidth reports in dial-up networks with link layer compression.	5	$A \in SyCn, Cr$ $G \in McGn$
/aug03 /msg00003.html	Iperf TCP mode may run for slightly longer than requested time. This is due to waiting of final acknowledgment.	5	
oct03 /msg00011.html	Assumes multi-threaded UDP port, which is invalid for many versions of Win32	10	$A \in SyCn, NoCr3$ $G \in McGn$
/oct03 /msg00004.html	Sometimes, the host system configuration is the bottleneck for high-bandwidth transfers. Based on system configuration and bandwidth reported, Iperf needs to give users a hint on potential invalid reports	5	$A \in Dy, Cr$ $G \in McGn$
/feb04 /msg00008.html	When user application specifies both -n (number of packets) and -t (time to run) one of the option is silently ignored by Iperf.	10	$A \in St, NoCr3$ $G \in HmEn$
/mar04 /msg00047.html	Iperf assumes $getSockName()$ API returns the bound multicast address. In Win32, the local address not the bound multicast address,is returned by the API	10	$A \in SyCn, NoCr3$ $G \in McGn$
/mar04 /msg00025.html	Maximum TCP window size is dependent on not just on the OS but also on a system administrator set value.	10	$A \in SyCn, NoCr3$ $G \in McGn$

Iperf defects determined as algorithmic defects	/feb03/msg00007.html, /feb03/msg00000.html, /jun03/msg00012.html, /jun03/msg00007.html, /jul03/msg00029.html, /feb03/msg00011.html

A.3 List of Defects Considered in TinyOS

The following is the list of defects considered in TinyOS. Since TinyOS (unlike Iperf) has a SourceForge defects database, the defects below are those which are not marked as duplicate and which are marked as *closed*. Selecting only *closed* defects provides a higher confidence in the defect analysis for determining whether or not the defect is related to invalid assumptions. The table below lists the defects that were related to invalid assumptions with a brief analysis. A list of defects that were related to installation and version control is also provided. Similarly, a list of defects due to algorithmic defects is given.

ID (TinyOS)	**Brief Analysis**	**Score**	**Classification**
1084879	PowerTOSSIM module assumes by default that the sensor board type is Mica.	10	$A \in SyCn, Cr$ $G \in McGn$
1055439	*Bcast* module assumes maximum sized payload and all packets are transmitted as maximum length (28 bytes) packets	10	$A \in Dy, NoCr$ $G \in McGn$
1052596	Deluge module does not work under heavy load; results in not sending replies to ping requests via TOS_BCAST_ADDR.	5	
1119642	In *Bcast* module, TinyOS timers do not function well if timer granularity is $<$ 10ms.	10	$A \in St, Cr$ $G \in McGn$

ID (TinyOS)	Brief Analysis	Score	Classification
1033732	*OP*, a Java module depends on the variable *maxMessageSize* in *MoteIf* module to determine maximum transfer size, which is invalid	5	$A \in St, NoCr$ $G \in HmEn$
1017509	TOSSIM, the simulator for TinyOS ignores parity errors. In some rare cases, broken packets pass through CRC check and garbage is delivered to the application	5	
979119	*MintRoute* module assumes Node ID is no more than 8 bits. Causes lookups to fail when more than lower 8 bits is used to determine Node ID	10	$A \in St, Cr$ $G \in HmEn$
960088	*QueueSendMsg* module assumes message length is always > 0. There can be messages with no payload, which can cause this module to crash.	10	$A \in St, Cr$ $G \in HmEn$
947169	Assumption on the mode of running. In simulation, the node state is ignored and packets are delivered when the node is turned off, which can never happen in an actual run	10	$A \in Dy, NoCr$ $G \in HmEn$
938259	Invalid assumption in TOSSIM about a node's address causes all listeners to reply to unicast packets. Cannot happen in an actual run	10	$A \in Dy, NoCr$ $G \in HmEn$
913123	An undocumented assumption that allows only posting of 7 tasks at a time.	10	$A \in Dy, Cr$ $G \in McGn$

ID (TinyOS)	**Brief Analysis**	**Score**	**Classification**
900058	TOSSIM simulator assumes time can be measured in *int*, while actual values run into the range of *long long*.	10	$A \in St, Cr$ $G \in HmEn$
896022	Inconsistencies between the opcode names used by Bombilla and the actual set of opcodes provided	5	
891749	LossyBuilder assumes TOSSIM can read values provided in exponential notation (like 8.99E-4)	10	$A \in St, NoCr$ $G \in HmEn$
833450	$TimeUtil$ assumes time can only be added and not subtracted (allows no negative values for its $addUint32()$ function	10	$A \in St, Cr$ $G \in HmEn$

TinyOS defects determined as algorithmic defects	1119642, 1004206, 1011809, 995187, 990531, 935051, 922122, 909284, 870244

TinyOS defects determined as installation and version control related defects. *Not considered for statistics*	1160877, 1042429, 1040869, 1040856, 1014456, 992043, 959059, 952785, 952784, 959059, 952785, 952784, 945583, 928680, 907793, 896021, 828525, 814805, 808643 814752, 809601

APPENDIX B

The AMF Language Grammar rules in Bachus-Naur form

The grammar for the AMF parser is given below. The rule for *logicalOrExpression* is from standard Java Language Syntax as described in the ANTLR Parser generator for Java. This maintains compatibility between basic built-in types of AMF and the corresponding basic types in Java. Also, only the grammar rules are given, the actions taken on successfully parsing the rule are omitted for brevity. Tokens are in all upper-case letters and all parser rules start with lower-case letters. The AMF parser accepts the rule *componentDefinitionsSet* .

```
// The set of component definitions
componentDefinitionsSet : (componentDefinitions)+;

// Component definition
componentDefinitions: COMPONENT_DEFINITION componentName componentBody;

// name {= id}
componentName: NAME assignID;

// The body for component definition
componentBody: LCURLY optionalDecl assnGuarSets RCURLY SEMI;

// Optional declarations
optionalDecl: ( LCURLY (langDecl)? (importsDecl)* RCURLY)? ;

// Language declaration
langDecl: LANGUAGE ASSIGN languageName SEMI;

// Currently assumptions are generated in Java, only language supported
```

```
languageName: JAVA;

// Imports declaration
importsDecl: IMPORT  IDENT (DOT IDENT)* (DOT! STAR)? SEMI! ;

// = identfier
assignID: ASSIGN! IDENT;

// Assumption guarantee sets for a component
assnGuarSets: (assumptionSet)+ ;

// One about clause per dependent component
assumptionSet: ABOUT idAssignSet ;

// name of dependent component and its definitions
idAssignSet: IDENT LCURLY (assumption|guarantee)+ RCURLY SEMI ;

// An assumption
assumption : ASSUMES assumesParamsAndBody;

// Assumption parameters and body
assumesParamsAndBody : IDENT  assumesParams assumesBody
                      (critStmt)? (scopeStmt)?
                      (changeIntStmt)? (impactStmt)? SEMI ;

// Parameters for an assumption
assumesParams: LPAREN ((param) (COMMA param)*)? RPAREN ;

// Body of an assumption
assumesBody: LCURLY logicalOrExpression RCURLY ;

// Parameter declaration
param:  primitiveDataType IDENT ;

// Primitive data type
primitiveDataType:  BYTE_KW | SHORT_KW | INT_KW | LONG_KW | FLOAT_KW
                  | DOUBLE_KW | CHAR_KW | BOOLEAN_KW |STRING_KW ;
```

```
critConsts :  CRITICAL_LEVEL_5 | NON_CRITICAL_LEVEL_4 | NON_CRITICAL_LEVEL_3
| NON_CRITICAL_LEVEL_2 | NON_CRITICAL_LEVEL_1 ;

critStmt   : LCURLY! CRITICALITY^ ASSIGN! critConsts RCURLY! ;

scopeConsts : PUBLIC_ASSUMPTION | PRIVATE_ASSUMPTION ;

validityTimeFrameConsts: STATIC_CI | DYNAMIC_C | SYSTEM_CONFIGURATION_CI ;

scopeStmt  : LCURLY SCOPE ASSIGN scopeConsts RCURLY;

changeIntStmt : LCURLY VALIDITY_TIME_FRAME ASSIGN validityTimeFrameConsts RCURLY;

// Guarantee
guarantee : GUARANTEES assumptionNameAndBody ;

// for a guarantee
assumptionNameAndBody : IDENT guaranteesBody;

// body for guarantee
guaranteesBody : LCURLY (paramValue)+ RCURLY SEMI ;

// Parameter type, name and value
paramValue : primitiveDataType IDENT ASSIGN (constant |
             (primaryExpression dotMethodSubExpr)) SEMI!;

// logical or (||)  Lowest precedence
logicalOrExpression : logicalAndExpression (LOR logicalAndExpression)* ;

// logical and (&&)
logicalAndExpression : inclusiveOrExpression (LAND inclusiveOrExpression)* ;

// bitwise or (|)
inclusiveOrExpression : exclusiveOrExpression (BOR exclusiveOrExpression)* ;

// exclusive or (^)
exclusiveOrExpression : andExpression (BXOR andExpression)* ;
```

```
// bitwise (&)
andExpression : equalityExpression (BAND^ equalityExpression)* ;

// equality/inequality (==/!=)
equalityExpression : relationalExpression ((NOT_EQUAL^ | EQUAL^ relationalExpression)*;

// boolean relational expressions
relationalExpression : shiftExpression (( LT | GT | LE | GE ) shiftExpression)* ;

// bit shift expressions
shiftExpression : additiveExpression ( (SL | SR | BSR ) additiveExpression)*;

// binary addition/subtraction
additiveExpression : multiplicativeExpression ((PLUS | MINUS) multiplicativeExpression)*;

// multiplication/division/modulo
multiplicativeExpression : unaryExpression ((STAR | DIV | MOD) unaryExpression)*;

// Unary expression
unaryExpression : INC unaryExpression | DEC unaryExpression
                 | MINUS unaryExpression | PLUS unaryExpression
                 | unaryExpressionNotPlusMinus ;

unaryExpressionNotPlusMinus : BNOT unaryExpression | LNOT unaryExpression |
                              postFixExpression ;

postFixExpression : primaryExpression ( dotMethodSubExpr | (dotArraySubExpr)?);

// After a primary expression
dotMethodSubExpr : DOT IDENT LPAREN (expressionList)? RPAREN ;

dotArraySubExpr : LBRACK additiveExpression RBRACK;

expressionList : primaryExpression (COMMA! primaryExpression)*;

// the basic element of an expression
primaryExpression: constant | primitiveDataType (LBRACK RBRACK)*
              |IDENT | LPAREN logicalOrExpression RPAREN;
```

```
constant : NUM_INT | CHAR_LITERAL | STRING_LITERAL | NUM_FLOAT | NUM_LONG |
           NUM_DOUBLE | TRUE | FALSE | NULL;

validityTimeFrameClause : ‘{’ ‘ViolationImpact’ ‘=’ (impactConsts| IDENT) ‘}’ ;
impactConsts : ‘VALUE_ERROR’ | ‘CRASH’ | ‘LOCKUP’ | ‘SUSPEND’ | ‘OMISSION’
    ‘STATE_TRANSITION_ERROR’ | ‘BYZANTINE’ | ‘DEADLINE_MISS’ | ‘BUDGET_OVERRUN’
     | ‘RESOURCE_SHARING_ERROR’ ;

// Standard Java/AMF Tokens
// NUM_INT | CHAR_LITERAL | STRING_LITERAL | NUM_FLOAT | NUM_LONG | NUM_DOUBLE
// recognize the standard java integer, character, string, float, long and double
// IDENT recognizes an indentifier

// Operators
LPAREN:’(’; RPAREN : ’)’; LBRACK : ’[’; RBRACK : ’]’; LCURLY : ’{’;
RCURLY: ’}’; COMMA : ’,’; DOT : ’.’; ASSIGN :’=’; EQUAL : "=="; LNOT:’!’;
BNOT : ’~’; NOT_EQUAL :"!="; DIV : ’/’; PLUS :’+’; MINUS : ’-’; STAR:’*’;
MOD : ’%’; SR :">>"; BSR : ">>>"; GE : ">="; GT :">"; SL :"<<";
LE : "<="; LT :’<’; BXOR :’^’; BOR:’|’; LOR : "||"; BAND :’&’;
LAND:"&&"; SEMI:’;’;

// AMF Constants / Tokens
COMPONENT_DEFINITION = "componentDefinition"; NAME = "name"; ABOUT = "about";
ASSUMES="assumes"; CRITICALITY="Criticality"; CRITICAL_LEVEL_5 = "CRITICAL_LEVEL_5";
NON_CRITICAL_LEVEL_4 ="NON_CRITICAL_LEVEL_4"; NON_CRITICAL_LEVEL_3="NON_CRITICAL_LEVEL_3";
NON_CRITICAL_LEVEL_2 ="NON_CRITICAL_LEVEL_2"; NON_CRITICAL_LEVEL_1="NON_CRITICAL_LEVEL_1";
VALIDITY_TIME_FRAME = "ValidityTimeFrame"; STATIC_CI = "STATIC"; DYNAMIC_CI = "DYNAMIC";
SYSTEM_CONFIGURATION_CI = "SYSTEM_CONFIGURATION"; SCOPE = "Scope";
PUBLIC_ASSUMPTION = "PUBLIC_ASSUMPTION"; PRIVATE_ASSUMPTION = "PRIVATE_ASSUMPTION";
BYTE_KW = "byte"; SHORT_KW = "short"; INT_KW = "int"; LONG_KW = "long";
FLOAT_KW = "float"; DOUBLE_KW = "double"; CHAR_KW = "char"; BOOLEAN_KW = "boolean";
STRING_KW = "String"; TRUE = "true"; FALSE = "false"; NULL = "null";
GUARANTEES = "guarantees"; COMPONENT_NAME = "componentName";
ASSUMPTION = "assumption"; LANGUAGE = "language"; JAVA = "JAVA"; IMPORT = "import";
```

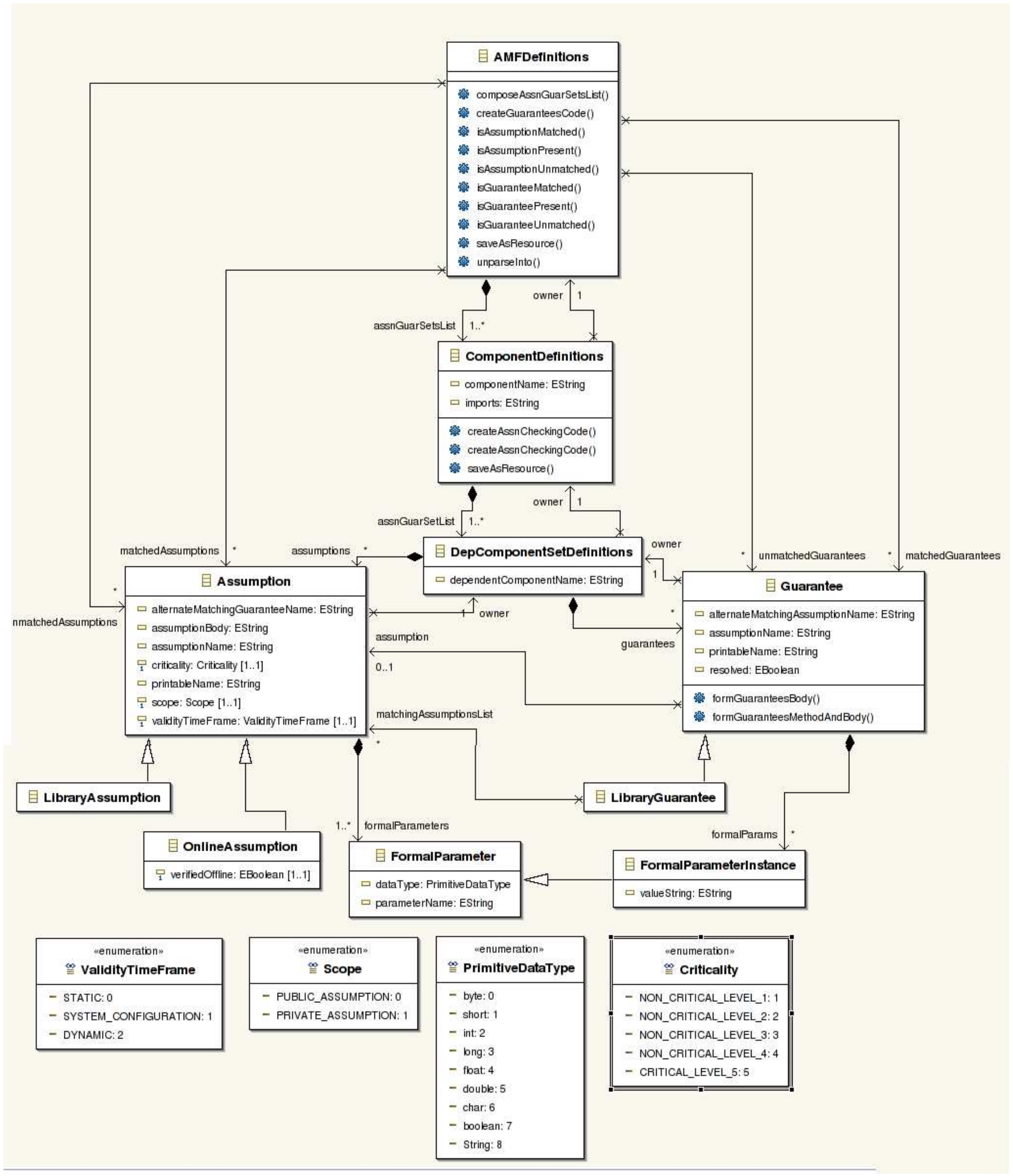


Figure B.1 The AMF Design Using UML Notation

REFERENCES

[1] B. Hailpern and P. Santhanam, “Software debugging, testing and verification,” *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[2] *ARIANE 5 Failure - Full Report [Online].* http://www.ima.umn.edu/ arnold/ disasters/ariane5rep.html.

[3] S. Leue, “Baby death due to software-controlled air bag deactivation?,” *ACM Risks Digest*, vol. 20, March 1999.

[4] G. Booch, “From small to gargantuan,” *IEEE Software*, vol. 23, no. 4, pp. 14–15, 2006.

[5] *FlexRay - The communication system for advanced automative control applications.* http://www.flexray-group.com.

[6] T. Pfarr and J. Reis, “The integration of COTS/GOTS within NASA’s HST command and control system.,” in *Proceedings of the First International Conference on COTS-Based Software Systems.*, Februrary 2002.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, ““System architecture directions for networked sensors”,” in *Proceedings of ACM ASPLOS*, Nov 2000, pp. 93–104.

[8] *IBM Rational Software [Online].* http://www.ibm.com/software/rational.

[9] E. Dubois, P. D. Bois, and A. Rifaut, “Structuring and expressing formal requirements for composite systems,” in *Proceedings of the Fourth International Conference on Advanced Information Systems Engineering (CAiSE 92)*, 1992.

[10] S. Greenspan, J. Mylopoulos, and A. Borgida, “On formal requirements modeling languages: RML revisited,” in *Proceedings of the 16th International Conference on Software Engineering*, May 1994.

[11] A. Tirumala, T. Crenshaw, L. Sha, G. Baliga, S. Kowshik, C. L. Robinson, and W. Witthawaskul, "Prevention of failures due to assumptions made by software components in real-time systems," *ACM SIGBED Review*, vol. 02, July 2005.

[12] K. Beck, "Embracing change with extreme programming," *IEEE Computer*, vol. 32, no. 10, pp. 70–77, 1999.

[13] M. Morisio, C. Seaman, A. T. Parra, V. Basili, and S. E. K. amd S.E. Condon, "Investigating and improving a COTS-based software," in *Proceedings of the International Conference on Software Engineering*, 2000, pp. 32–41.

[14] *Risks Forum Digest [Online].* http://catless.ncl.ac.uk/Risks.

[15] Q. Jones, "Software disaster leaves new australian submarine unfit," *ACM Risks Digest*, vol. 23, Nov 2003.

[16] *Mars Climate Orbiter, Mishap Investigation Board, Phase I Report*, Nov 1999. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.

[17] *XML Path Language.* http://www.w3.org/TR/xpath.

[18] L. Gu, D. Jia, P. Vicaire, T. Yan, L. Luo, A. Tirumala, Q. Cao, T. He, J. Stankovic, T. Abdelzaher, and B. Krogh, "Lightweight detection and classification for wireless sensor networks in realistic environments," in *Proceedings of Sensys*, 2005, pp. 205–217.

[19] F.Chen and G.Rosu, "Towards Monitoring-Oriented Programming: A paradigm combining specification and implementation," in *Electronic Notes in Theoretical Computer Science*, vol. 89, 2003.

[20] *"Software considerations in airborne systems and equipment certification"*, 1993. Radio Technical Commission for Aeronautics RTCA DO-178B/EUROCAE ED-12B.

[21] *"The SAE Architecture Analysis and Design Language (AADL) Standard"*, Nov 2004. SAE International Document Number AS5506.

[22] P. Feiler, R. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, K. S. D. Schmidt, and K. Wallnau, *Ultra-Large-Scale Systems: The Software Challenge of the Future*, June 2006.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*, Jan 1995.

[24] R. Thompson, "Microsoft patches their patched patches," *ACM Risks Digest*, vol. 20, July 1999.

[25] G. Baliga, S. Graham, L. Sha, and P.R.Kumar, "Etherware: Domainware for wireless control networks," in *Proceedings of International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2004.

[26] H. Ding and L. Sha, "Dependency algebra: A tool for designing robust real-time systems," in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 2005.

[27] *Eclipse Modeling Framework [Online:].* http://www.eclipse.org/emf.

[28] *The ANTLR Parser Generator [Online:].* http://www.antlr.org.

[29] *Extensible Markup language.* http://www.w3.org/XML.

[30] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE Software*, vol. 12, no. 4, pp. 52–62, 1995.

[31] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, K. Gibbs, and M. Gates, *Iperf - End-to-end Bandwidth Measurement Tool*, Mar 2003. http://dast.nlanr.net/Projects/Iperf/.

[32] A. Tirumala, L. Cottrell, and T. Dunigan, "End-to-end bandwidth measurement using iperf and web100," in *Passive and Active Measurements*, 2003.

[33] L.S.Brakmo and L.L.Peterson, "TCP Vegas: end-to-end congestion avoidance on a global internet," *IEEE Journal in Selected Areas in Communication*, vol. 13, pp. 1465–80, October 1995.

[34] J. Mo, R. La, V. Anantharam, and J.Walrand, "Analysis and comparison of tcp reno and vegas," in *Proceedings of IEEE INFOCOMM*, 1999.

[35] M. Mathis, J. Heffner, and R. Reddy, "Web100: Extended TCP Instrumentation," *ACM Computer Communications Review*, vol. 33, July 2003.

[36] *The Internet End-to-end Performance Monitoring, Bandwidth to the world (IEPM-BW) [Online:].* http://www-iepm.slac.stanford.edu/bw/index.html.

[37] *Iperf quick mode measurements [Online:].* http://www-iepm.slac.stanford.edu/bw/iperf_res.html.

[38] S. Vinoski, "CORBA: integrating diverse applications within distributed heterogeneous environments," *IEEE Communications Magazine*, vol. 35, pp. 46–55, Feb 1997.

[39] P. Levis, N. Lee, M. Welsh, and D. Culler, ""TOSSIM: Accurate and scalable simulation for entire TinyOS applications"," in *Proceedings of Sensys*, 2003, pp. 126–137.

[40] B. Meyer, "Applying 'design by contract'," *IEEE Computer*, vol. 25, pp. 40–51, Oct 1992.

[41] J. Jazequel and B. Meyer, "Design by contract: the lessons of ariane," *IEEE Computer*, vol. 30, pp. 129–130, Jan 1997.

[42] D. Schmidt and F. Kuhns, "An overview of the Real-Time CORBA specification," *IEEE Computer*, vol. 33, pp. 56–63, Jun 2000.

[43] D. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "An overview of the real-time CORBA specification," *IEEE Communications Magazine*, vol. 35, pp. 72–77, Feb 1997.

[44] Y. Krishnamurthy, I. Pyarali, C. Gill, L. Mgeta, Y. Zhang, Torn, and D. Schmidt, "The design and implementation of real-time CORBA 2.0: dynamic scheduling in TAO," in *Proceedings of Real-Time and Embedded Technology and Applications Symposium*, May 2004, pp. 121– 129.

[45] J. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis, "VEST: an aspect-based composition tool for real-time systems," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003, pp. 58–69.

[46] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," in *Proceedings of WISP*, May 2001.

[47] G. Lee, J. Howard, and P. Anderson, *Safety Critical Requirements Specification using SpecTRM.* http://www.safeware-eng.com.

[48] N. Leveson, *Safeware: System Safety and Computers.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1995.

[49] N. Leveson, "Intent specifications: an approach to building human-centered specifications," in *Proceedings of Third International Conference on Requirements Engineering*, 1998, pp. 204–213.

[50] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese, "Requirements specification for process-control systems," *Software Engineering, IEEE Transactions on*, vol. 20, no. 9, pp. 684–707, 1994.

[51] M. Heimdahl and N. Leveson, "Completeness and consistency in hierarchical state-based requirements," *Software Engineering, IEEE Transactions on*, vol. 22, no. 6, pp. 363–377, 1996.

[52] J. Li and P.H.Feiler, "Impact analysis in real-time control systems," in *Proceedings of IEEE International Conference on Software Maintenance*, 1999, pp. 443–452.

[53] *PLCOpen.* http://www.plc-open.org.

[54] R. Ommering, "Building product populations with software components," in *In Proceedings of the 24th ACM International Conference on Software Engineering*, 2002, pp. 255–265.

[55] R. Ommering, F. Linden, and J. Kramer, "The Koala component model for consumer electronics software," *IEEE Computer*, vol. 33, pp. 78–85, Mar 2000.

[56] A. Fioukov, E. Eskenazi, D. Hammer, , and M. Chaudron, "Evaluation of static properties for component-based architectures," in *In Proceedings of the 28th Euromicro Conference*, 2002, pp. 33–39.

[57] D. Isovic and C. Norstrom, "Components in real-time systems," in *In Proc. RTCSA 2002, 8th International Conference on Real-Time Computing Systems and Applications*, Mar 2002.

[58] *PECOS Project.* http://www.pecos-project.org/publications.html.

[59] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[60] C. Hoare, “Monitors: An Operating System Structuring Concept,” *Communications*, 1974.

[61] P.Lago and H.V.Vliet, ““Explicit assumptions enrich architectural models”,” in *Proceedings of the 27th International Conference on Software Engineering*, 2005, pp. 206–214.

[62] D. Giannakopoulou, C. Pasareanu, and J. Cobleigh, ““Assume-Guarantee Verification of Source Code with Design-Level Assumptions”,” in *Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 211–220.

[63] T. Ball and S. Rajamani, “Automatically validating temporal safety properties of interfaces,” *Proceedings of the 8th international SPIN workshop on Model checking of software*, pp. 103–122, 2001.

[64] T. Henzinger, S. Qadeer, and S. Rajamani, “You assume, we guarantee: Methodology and case studies,” *CAV*, vol. 98, pp. 440–451, 1998.

[65] T. Henzinger, S. Qadeer, S. Rajamani, and S. Tasiran, “An assume-guarantee rule for checking simulation,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 24, no. 1, pp. 51–64, 2002.

[66] C. Pasareanu, M. Dwyer, and M. Huth, “Assume-guarantee model checking of software: A comparative case study,” *Theoretical and Practical Aspects of SPIN Model Checking*, vol. 1680, pp. 168–183, 1999.

[67] E. Clarke and J. Wing, “Formal methods: state of the art and future directions,” *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[68] L. Sha, “Upgrading embedded software in the field: Dependability and survivability,” in *In Proceedings of EMSOFT*, 2002.

[69] T. Crenshaw, A. Tirumala, S. Hoke, and M. Caccamo, “A fault-tolerant asynchronous protocol for wireless communication among collaborating units,” in *Euromicro Conference on Real-time Systems (ECRTS)*, 2005.

[70] D. Seto and L. Sha, “A case study on analytical analysis of the inverted pendulum real-time control system,” tech. rep., SEI-CMU, 1996. ESC-TR-99-023.

Author's biography

Ajay Tirumala was born in Bangalore in 1977. He loves his hometown. He obtained his Bachelors degree in Computer Science and Engineering from University of Mysore in 1998, and later dabbled as a Senior Software Engineer for Novell. He decided to pursue graduate studies in the other side of the world, surrounded by corn-fields, at University of Illinois at Urbana-Champaign, from the Fall of 2000. His time at Urbana-Champaign was spent mostly in falling in love with a charming girl, getting married; working with 'Asha for education' as a volunteer, treasurer and president; training for and completing the Chicago marathon; biking; and spending copious amounts of time having fun with friends and family. Along the way, he managed to obtain his Masters (in 2001) and Doctoral (in 2006) degrees in Computer Science from University of Illinois at Urbana-Champaign. He loves Urbana-Champaign too.

His research interests span the areas of embedded and real-time systems and software engineering. His conference publications/presentations include ACM Sensys, ACM MobiHoc, IEEE Real-time Systems Symposium, IEEE European Conference on Real-time Systems and IEEE/ACM Super Computing; and journal/magazine publications include ACM Transactions on Embedded Computer Systems and ACM SIGBED Review.