A FORMAL COMPONENT-BASED SOFTWARE ENGINEERING APPROACH FOR DEVELOPING TRUSTWORTHY SYSTEMS

MUBARAK SAMI MOHAMMAD

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY (COMPUTER SCIENCE)
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 2009

© MUBARAK SAMI MOHAMMAD, 2009

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Mubarak Sami Mohammad**

Entitled: **A Formal Component-Based Software Engineering Approach for Developing Trustworthy Systems**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science) (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair
Dr.

______________________________ External Examiner
Dr. Nicholas Graham

______________________________ External to Program
Dr. Jamal Bentahar

______________________________ Examiner
Dr. Joey Paquet

______________________________ Examiner
Dr. Juergen Rilling

______________________________ Supervisor
Dr. Vasu Alagar

______________________________ Co-supervisor
Dr. Olga Ormandjieva

Approved ______________________________
Chair of Department or Graduate Program Director

__________ 20 ______ ______________________________
Dr. Robin A.L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

A Formal Component-Based Software Engineering Approach for Developing Trustworthy Systems

Mubarak Sami Mohammad, Ph.D.
Concordia University, 2009

Software systems are increasingly becoming ubiquitous, affecting the way we experience the world. Embedded software systems, especially those used in smart devices, have become an essential constituent of the technological infrastructure of modern societies. Such systems, in order to be trusted in society, must be proved to be trustworthy. Trustworthiness is a composite non-functional property that implies safety, timeliness, security, availability, and reliability. This thesis is a contribution to a rigorous development of systems in which trustworthiness property can be specified and formally verified.

Developing trustworthy software systems that are complex and used by a large heterogenous population of users is a challenging task. The component-based software engineering (CBSE) paradigm can provide an effective solution to address these challenges. However, none of the current component-based approaches can be used as is, because all of them lack the essential requirements for constructing trustworthy systems. The three contributions made in this thesis are intended to add to the expressive power needed to raise CBSE practices to a rigorous level for constructing formally verifiable trustworthy systems.

The first contribution of the thesis is a formal definition of the trustworthy component model. The trustworthiness quality attributes are introduced as first class structural elements. The behavior of a component is automatically generated as an extended timed automata. A model checking technique is used to verify the properties of trustworthiness. A composition theory that preserves the properties of trustworthiness in a composition is presented.

Conventional software engineering development processes are not suitable either for developing component-based systems or for developing trustworthy systems. In order to develop a component-based trustworthy system, the development process must be reuse-oriented, component-oriented, and must integrate formal languages and rigorous methods

in all phases of system life-cycle. The second contribution of the thesis is a software engineering process model that consists of several parallel tracks of activities including component development, component assessment, component reuse, and component-based system development. The central concern in all activities of this process is ensuring trustworthiness.

The third and final contribution of the thesis is a development framework with a comprehensive set of tools supporting the spectrum of formal development activity from modeling to deployment.

The proposed approach has been applied to several case studies in the domains of component-based development and safety-critical systems. The experience from the case studies confirms that the approach is suitable for developing large and complex trustworthy systems.

To Allah, He who nurtured me and taught me all that which I know.

Acknowledgments

When I met Professor Vasu Alagar for the first time five years ago, I had no knowledge about scientific research. He taught me the alphabet of research, and throughout the past years he taught me how to be a good researcher not only by words but also by the quality of work, thoughts, and the discussions that he provides. He worked very hard, even at times work extends to early hours of the morning where he reviews my articles and gives advices and guidance. I'm profoundly thankful and grateful for his teachings. Although my work is positioned humbly in front of his teachings, it gives glimpses of what I learned from him. He has a great personality that kept me attracted to him. His spirituality, sincerity, quality of work, and alignment to the Truth are only a few traits, among so many others, that I admire a lot in him. I feel very fortunate for being his student and working under his supervision.

Also, I would like to thank my co-supervisor Dr. Olga Ormandjieva for her continuous support throughout the work of my Masters and PhD.

Finally, I would like to acknowledge with love the great emotional and spiritual support of my parents, my brothers, and my sisters. Although I lived most of my life away from them, they have been always in my heart and near my soul. I'm deeply grateful for their love and prayers.

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

1.1 Trustworthiness 2

1.2 Component-Based Software Engineering (CBSE) 5

1.2.1 Component Model 5

1.2.2 Component-based Development Process Model 6

1.3 Research Motivation 6

1.3.1 Analyzing Component Models 7

1.3.2 Analyzing Component-Based Development Process Models 7

1.3.3 Evaluation 8

1.4 Thesis Goals 8

1.5 Thesis Outline 9

2 Literature Survey **11**

2.1 Component Models 12

2.1.1 Koala 12

2.1.2 PIN 13

2.1.3 PECOS 14

2.1.4 KobrA 16

2.1.5 Fractal 16

2.1.6 SOFA 2.0 18

2.1.7 SaveCCM 20

2.2 Analyzing Current Component Models 22

2.2.1 Comparison . . . 22
2.2.2 Services . . . 24
2.2.3 Component Contract . . . 24
2.2.4 Encapsulation and Composition Theory . . . 27
2.3 ADL Related Work . . . 28
2.3.1 Acme . . . 28
2.3.2 Secure-xADL . . . 29
2.3.3 AADL . . . 29
2.4 Specifying Trustworthiness Properties . . . 30
2.4.1 Combining Safety and Security Properties . . . 31
2.4.2 Reliability . . . 32
2.5 Process Models for Component-based Development . . . 32
2.5.1 Discussion . . . 34
2.6 Summary . . . 37

3 Research Methodology **39**
3.1 Research Objectives . . . 39
3.2 Research Methodology . . . 39
3.2.1 Phase 1: Defining A Formal Component Model for Trustworthy Systems . . . 40
3.2.2 Phase 2: Defining A Process Model for Developing Trustworthy Component-Based Systems . . . 45
3.2.3 Phase 3: Developing a framework with comprehensive tool support 46
3.3 Summary . . . 48

4 Trustworthy Component Model **49**
4.1 Event and Data Parameters . . . 49
4.2 Services and Contracts . . . 52
4.3 Component Architecture . . . 57
4.4 Security Mechanism . . . 64
4.5 Behavior . . . 70
4.6 System Definition . . . 72
4.7 Composition . . . 73
4.8 Summary . . . 78

5 **TADL: Trustworthy Architecture Description Language** **80**
5.1 Meta-Architecture . . . 80
5.2 TADL . . . 82
5.2.1 Event and data parameter . . . 83
5.2.2 Contract . . . 84
5.2.3 Component architecture . . . 85
5.2.4 Security mechanism . . . 88
5.2.5 System definition . . . 88
5.3 Summary . . . 90

6 **Model Transformation and Formal Verification** **94**
6.1 Verifying Safety and Security . . . 95
6.1.1 UPPAAL . . . 95
6.1.2 Transformation Rules . . . 96
6.1.3 Example . . . 100
6.1.4 Preserving the requirements of safety and security . . . 101
6.2 Real-Time Analysis . . . 105
6.3 TADL Semantics . . . 106
6.4 Summary . . . 107

7 **Reliability and Availability** **108**
7.1 Service Failures and Repairs . . . 108
7.2 Defining Reliability and Availability . . . 110
7.3 Verifying Reliability and Availability . . . 111
7.4 Steam Boiler Controller Case Study . . . 112
7.4.1 System specification . . . 112
7.4.2 System operation . . . 113
7.4.3 Failure and repair operations . . . 113
7.5 Summary . . . 118

8 **Process Model for Developing Trustworthy Systems** **119**
8.1 Domain Engineering . . . 119
8.2 Component Development . . . 128
8.3 Component Assessment . . . 129
8.4 Component Reuse . . . 130

8.5 System Development 131
8.6 Traceability 133
8.7 Certification 134
8.8 Automated Component Testing 135
8.9 Run-Time Monitoring 136
8.10 Accomplishments 136

9 Framework Architecture **138**
9.1 Design-Time Tools 138
9.1.1 Visual modeling tool 138
9.1.2 Compiler and model transformation 139
9.1.3 Transformation analysis 142
9.1.4 Simulation and model checking 142
9.1.5 Real-time analysis 143
9.1.6 Architectural analysis 143
9.2 Implementation Tools 143
9.2.1 Component repository 143
9.2.2 Code generation 144
9.2.3 Traceability analysis 144
9.3 Run-time Tools 144
9.3.1 Run-time environment 144
9.3.2 Run-time analysis 145
9.4 Summary 145

10 Conclusion **147**
10.1 Summary 148
10.2 Assessment 149
10.3 Case Studies 152
10.3.1 The common component modeling example 152
10.3.2 The steam boiler controller case study 154

Bibliography **159**

List of Figures

1 Composite Component 6
2 Koala component and its CDL description 13
3 Pin component 14
4 PECOS component model 15
5 Fractal component 17
6 Example SOFA CDL specification 21
7 SaveCCM component 22
8 Trustworthy development life-cycle 37
9 Structure of Trustworthy Component 50
10 The different types of events 53
11 Service 55
12 Connecting two components 57
13 Component Definition 60
14 Structure of the ABS 62
15 ABS component instances 63
16 Role-based Access Control 65
17 Trustworthy Component Model 81
18 The TADL syntax of Element Type 82
19 The TADL syntax of Parameter Type, Event Type, and Attribute 83
20 An example definition of an event type 84
21 The TADL syntax of Time Constraint, Data Constraint, Service, Safety Property, and Contract 86
22 An example of a contract specifications 87
23 The TADL syntax of Interface Type, Connector Role Type, Connector Type, Architecture Type, and Component Type 89
24 An example of a composite component specification 91

25 The TADL syntax of RBAC specification 92
26 An example RBAC specification using TADL 93
27 The TADL syntax of system configuration specification 93
28 An example configuration specification using TADL 93
29 The process of transformation and analysis 94
30 Example Transformation 101
31 The UETA of the controller component 102
32 TADL Semantics 107
33 The process of modeling and verifying reliability and availability 111
34 The UETA of the steam-boiler controller component 116
35 Domain Engineering and Component Engineering (development, assessment, and reuse) 120
36 An ontology for domain analysis 123
37 Car ontology example focusing on the fingerprint security system 124
38 OWL specification of the controller concept 125
39 TADL specification of the Controller component 127
40 Component-based system development 130
41 Framework Architecture 139
42 Visual Modeling Tool 140
43 The compiler and model transformation tool 141
44 The implemented and adopted tools 145
45 Store System Components 153
46 The architecture of the cash desk 155
47 Part of the TADL specification of CocoMe 156
48 Part of the TADL specification of CocoMe 157
49 Extended timed automata of the Cashier component 158

List of Tables

1 Comparison of static and dynamic aspects of component models 25
2 Comparison of analysis tools support 26
3 A summary of the component and system development activities - Part 1 . . 35
4 A summary of the component and system development activities - Part 2 . . 36

Chapter 1

Introduction

Software systems are increasingly becoming ubiquitous affecting the way we experience life and perform work. For example, smart devices and intelligent sensors are currently used to capture information about human activities along with their physiological and psychological status and communicate it through wireless connections [DSS08]. The collected information triggers adaptation in a pervasive environment according to predefined preferences. Such systems are being used in the health-care sector to improve its services. Another example can be found in avionics. Currently, aircrafts are being controlled fully by *autopilot*, a software that guides the aircraft. Moreover, modern day cars contain up to 67 processors that implement around 270 user functions that a driver interacts with [PBKS07]. Modern day cars are expected to contain up to one gigabyte of embedded software [PBKS07]. Some of these software units perform safety critical missions such as controlling the engine, brakes, and steering.

These examples show the current advancement of software development in areas that affect our daily lives. At the same time, it raises questions about the ability of the current software development paradigms to cope with the risky trends of *pervasive computing*, which provide highly customizable and personalized services that must have the capability to run anytime, anywhere and on any device with minimal user attention. Pervasiveness also raises concerns on trustworthiness: *to which extent the current software development paradigms are capable of producing trustworthy systems that control the lives of people and manage their private data?*

1.1 Trustworthiness

Trustworthiness is a moral value concept which implies commitment and ability to be relied and depended on. Trust is a social aspect that is hard to define formally. It is a relation between two parties in which the trusting party places confidence, reliance, and dependance, whereas the trustee commits to take responsibility and never betray the trust.

In social aspects, it is difficult to measure trust because it is based on beliefs, feelings, and accumulated experiences. In all cases, trust is relative. There is no absolute trust. It is always bound to defined tasks. For example, we trust the postman to deliver our mail on time and to the correct address. However, we may not trust him to run our business or perform our private financial transactions. Trust implies a factor of risk. However if the level of reliance exceeds the level of risk we are inclined to trust.

In the domain of technology and computing, we rely on technology on which aircrafts, trains, traffic controllers, automated teller machines, and elevators serve us in our daily life although it fails from time to time causing many inconveniences, some even causing damage to property and humans. Yet, we continue to use them because they have been tried and tested for long periods of time that they seem to have passed our minimum acceptance level. Many embedded software systems have also become an essential part of the technological infrastructure of modern societies. Hence, there is a need to design these systems such that they are provably trustworthy. Towards this purpose, the credentials of trust should be formally defined along with their level of acceptance.

There are many important questions arising from the user perspective. These are the following:

- Is the system doing what it is supposed to do?
- Is the system available when the user needs it?
- Is the system protecting the private data?
- Is the system safe for use? Is it likely to cause damage to the environment in which it is deployed or the user who is using it?
- Will the system respond to user requests in a timely fashion.
- Is it possible to repair the system in real-time, if it fails? How often the system is likely to fail? How long will it stay in failure mode?

These questions form an envelope to defining the essential requirements of trustworthy systems. In the literature, trustworthiness is defined as the system property that denotes the degree of user confidence that the system will behave as expected [SBI99, ALRL04]. The terms *trustworthiness* and *dependability* are used interchangeably [Som07]. Dependability is defined as "the ability to deliver services that can justifiably be trusted" [ALRL04]. A comparison between the two terms presented in [ALRL04] has concluded that the two properties are equivalent in their goals and address similar concerns. The goals of dependability are providing justifiably trusted services and avoiding outage of service that is unacceptable to the consumer. The above definitions emphasize the importance of justifying trust. In order to justify trust, we should define trustworthiness formally.

There is a common consensus [SBI99, ALRL04, MdVHC02] that trustworthiness is a composite concept and that the essential quality properties contributing to trustworthiness are *safety*, *security*, *reliability*, and *availability*. Since many of the current systems are real-time, we also include *timeliness* to the quality attributes of trustworthiness. These properties are defined below.

- **Safety** is the quality of the operational behavior of the system in which no system action that may lead to catastrophic consequences will happen. Safety includes a set of properties that describe the correct and safe behavior of the system. Any violation to a safety property may cause dangerous consequences on the users and the environment. For example, modern vehicles have an *anti-lock brake system* (ABS) which prevents the wheels from locking while braking. The safety property states that if one wheel is rotating significantly slower than the others then the hydraulic pressure to the brake at the affected wheel must be reduced within a fraction of a second. On the other hand if the wheel is turning faster than the others, then the brake hydraulic pressure to the wheel should be increased so the braking force is reapplied and the wheel slows within a fraction of a second.

- **Security** is a composite property that includes *confidentiality*, *integrity*, and *availability*. Confidentiality ensures that system services and information are not exposed or disclosed to unauthorized users. Integrity ensures that there is no improper alteration to the system state or the information. For example, in a banking system, confidentiality means that only the client or one of his authorized people can perform transactions and view information related to this client. Integrity means that when the client deposits $500 then his account should be increased by exactly $500 not

less or more.

- **Reliability** is the quality of continuing to provide correct services despite any failure. It is possible to have an accepted frequency of failures. In this case the accepted *mean time between failures* should be precisely defined and respected. For example, many avionic systems have a required reliability of 10^9 hours mean time between failures [PBKS07].

- **Availability** means readiness for correct service. It is the quality of operation in which there is no unforeseen or unannounced disruption of service. A temporary outage of service may not cause big problems for a non-critical system. The required services can be requested at a later point of time when the system becomes available. However, any service outage for a safety-critical system may lead to catastrophic consequences. When a system fails, availability specifies the maximum accepted time of repair until the service returns back to operate correctly.

- **Timeliness** refers to bounded time constraint behavior. It means, when a request for service is received, the system should respond within acceptable limits of time. Timeliness is an essential safety requirement for real-time systems. In these systems the correctness of system behavior depends not only on providing services but also on the time at which the services are provided. It is possible to regard timeliness as one of the safety properties.

Some interesting questions are *how can these properties be satisfied collectively in one development process?*, and *can the current state of the art of software development paradigms collectively address their requirements?*

In the literature, there has been a tremendous research effort resulting in many publications about safety, security, reliability, and availability. However, research in specifying and verifying safety and security and estimating reliability and availability properties at the system architectural level have progressed only independently. This is due to many reasons such as (1) the early finding that safety and security properties cannot be formally specified, composed, and verified together in any one formal method [McL96], (2) the conventional ways of estimating reliability at a system architecture using stochastic methods which are based on uncertain and inaccurate parameters [Gok07], and (3) the lack of research in analyzing availability [IN08]. There is no published work that we are aware of which has successfully managed to combine all these attributes in one formal approach. In order to

develop trustworthy systems, all these properties must be combined together in one formal approach. *This thesis provides a novel formal approach which uses* component-based software engineering *(CBSE) for developing trustworthy systems. Our approach enables the specification and verification of safety, security, reliability, and availability properties.*

1.2 Component-Based Software Engineering (CBSE)

CBSE promises many advantages to software development including reuse, managing complexity, and reducing development time, effort, and cost. CBSE is widely adopted in the software industry as the mainstream approach to software engineering [Som07]. It is increasingly used to develop software systems, especially embedded systems, deployed in safety critical environments. Complexity is effectively managed by dividing the problem into smaller problems of manageable magnitudes, each of which handled separately in CBSE. The cost of development is reduced by reusing existing solutions to solve these subproblems. The essential constituents of CBSE are *component model* and *component-based development process model* [Som07]. The following subsections briefly discuss these two elements.

1.2.1 Component Model

A component model defines what components are (their syntax and semantics), their composition to develop component-based systems, and their deployment [LW07]. A component is defined as "a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard" [HC01]. Components *provide* and *require* services through public interfaces. The provided services are the operations performed by the component. The required services are the operations that the component need in order to complete its provided services and produce results. The interfaces of a component provide specification of the public services that are provided and required by the component. Component models describe the internal structure of components. A component can be *primitive* or *composite* [SG96]. A primitive component is the basic unit that can not be further divided. It is specified by its implementation. Primitive components can be composed together to form composite components. Connectors are used to bind the interfaces of the constituent components. Figure 1 depicts a composite component.

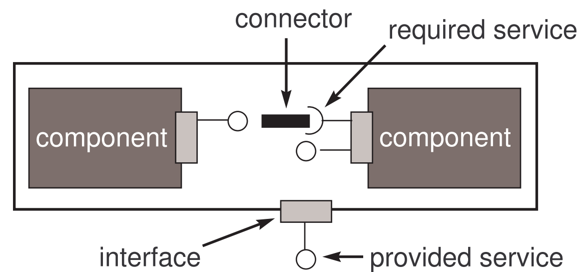


Figure 1: Composite Component

1.2.2 Component-based Development Process Model

A software development process defines the set of activities along with their interdependencies and relations that lead to the production of software systems. A typical process includes requirements definition and analysis, design, implementation, testing, deployment, and maintenance. A *component-based development process* (CBD) is a special type of software development process tailored for developing reusable components and building systems by integrating existing components. A conventional software development process is not suitable for developing component-based systems. This is because CBD is biased towards reuse. In order to achieve a successful reuse and integration of the developed components, the development process should be tailored to CBSE [Som07]. Therefore, a different development process is required to develop component-based systems than the development process used for conventional software.

CBD addresses the activities involved throughout the entire component and system life-cycles. It comprises two parallel activities: *software component development* and *component-based systems development* [CCL06, Pre05]. The former addresses the issues of components' specification, development, qualification, documentation, cataloguing, and adaptation and selection for reuse. The later addresses the issues related to assembling components to develop component-based systems.

1.3 Research Motivation

Pervasive computing raises major concerns about the ability of current development paradigms to develop trustworthy systems. Since CBSE is a mainstream approach to software engineering [Som07], an important question is : *can CBSE be used to develop systems which*

are provably trustworthy? In order to answer this question, we first investigate whether or not CBSE has fulfilled its initial intended promises.

1.3.1 Analyzing Component Models

In general, software systems implement functional and non-functional requirements. This implies that component specification methods and qualification techniques should support both functional and non-functional requirements. However, generally, current component-based development approaches have limited or no support for non-functional requirements. Furthermore, non-functional requirements and environmental constraints should be defined as contracts at the interfaces of a component. This is because environmental assumptions and non-functional requirements might not be valid when components are used in different deployment environments. For example, in real-time embedded systems, time constraints that define the maximum amount of time for a safe execution of a service might be different depending on hardware and software configurations of the deployment environment. The separation between the computation part of the component and its contract enables components to be reused in different environments by changing only its contract. However, when studying current component models, there is limited or no support for contracts. Therefore, current component engineering practices can only support limited reusability of components. Moreover, a study of current component models [LW07] revealed that components are composed using direct method calls or indirect message passing through connectors. Thus, these models produce tightly coupled components that are difficult to reuse. Also, when assembling components, special composition rules should be applied to ensure that the non-functional requirements of the constituent components are preserved in the assembly. This requires a defined composition theory. However, there is no component model that defines a composition theory for both the structural and nonfunctional parts of components [LW07]. Therefore, current component models are not suitable as is for specifying trustworthy systems.

1.3.2 Analyzing Component-Based Development Process Models

Safety and secure critical systems require a special type of software development process, preferably one based on formal methods for the representation and analysis of software specification. The primary goals of this process are to ensure the correctness of system

specification and design, and help to verify that the system implementation is consistent with its specification. Formal methods include specification, verification, and testing techniques throughout the different stages of system development. In order to develop systems that can be certified to be trustworthy, validation and verification of trustworthiness features should be made a core activity linked to all activities in the development process. Typical component-based development processes presented in the literature focus on the general activities involved in developing component-based systems with emphasis on reuse and integration testing. There is no work, that we are aware of, which presents a rigorous development process that is suitable for developing trustworthy component-based systems. Therefore, current component-based development processes are not suitable as is for developing trustworthy systems.

1.3.3 Evaluation

The above discussion about the current component models and component-based development processes confirms that:

1. they lack support for non-functional requirements,
2. they lack composition theory, and
3. current CBSE practices are not based on rigorous process models.

Therefore, despite the wide adoption of CBSE in software industry and the tremendous number of publications about it in academic research, it is still lacking essential formal foundations for the specification, composition, and verification of non-functional requirements. Therefore, current CBSE practices do not provide the essential needs for developing trustworthy systems.

1.4 Thesis Goals

This thesis is a contribution to a rigorous CBSE and trustworthy systems. This thesis investigates the challenges of defining trustworthy components, composing trustworthy components, and verifying trustworthiness in a unified model. The thesis provides a formal CBSE approach which satisfies the requirements of trustworthy systems. The approach provides a

remedy to the shortcomings of current component models by providing a component definition which collectively addresses the requirements of trustworthiness and component-based development. We do not intend to create a new component model, give it a new name, and add it to the list of component models in the literature. The goals of this thesis are:

1. Provide a formal definition that can be adopted by other component models to enhance their support for trustworthiness,

2. Show that it is possible to provide a single component definition that includes specifications of structural, functional, and non-functional requirements, especially the properties of trustworthiness,

3. Show that it is possible to define a composition theory which includes rules for composing both structural requirements and trustworthiness properties,

4. Show that it is possible to use one formal verification technique for safety, security, reliability, and availability properties, and

5. Provide a rigorous process model with tool support for the development of trustworthy component-based systems.

1.5 Thesis Outline

This thesis is organized as follows: Chapter 2 presents a detailed literature survey that covers the work done in the areas of component models and component-based process models. Chapter 3 introduces our research methodology. We present our contributions and provide a detailed discussion of the research problems and research questions that are related to the development of trustworthy component-based systems. Then, we provide our proposed solutions. Chapter 4 introduces our trustworthy component model. We provide formal definitions of the structural, functional, and trustworthiness properties. Also, it introduces a composition theory that preserves the properties of trustworthiness. Chapter 5 introduces an architecture description language (TADL) that is based on the trustworthy component model. The TADL specification provides a high level description of systems to make it easy for software architects to use our formal approach. Chapter 6 provides an automated model transformation technique for generating component behavior and real-time models. Chapter 7 presents a novel approach for the specification and verification of reliability

and availability using model checking. Chapter 8 presents a process model for developing trustworthy systems. Chapter 9 presents a framework of tools support for implementing the process model. We discuss different kinds of tools that has been development or under development. Finally, Chapter 10 concludes the thesis. It provides summary and assessment of the presented approach.

Chapter 2

Literature Survey

This chapter provides a survey of different component models and component-based development process models that have been presented in the literature of component-based software engineering. Section 2.1 surveys the related component models. We study different component models and describe their structural and behavioral definitions. Section 2.2 provides an analysis of the current component models and shows the lack of support for non-functional requirements. Since both component models and *architecture description languages* (ADL) share the component concept, Section 2.3 surveys the related work in ADLs and analyze their support for non-functional requirements. Section 2.4 briefly surveys the work done to specify trustworthiness properties. It describes the research efforts of the security and realtime research communities in putting forth one unified composition theory for trustworthiness properties, in particular how they failed to achieve it. Also, it give pointers to the work done in specifying and measuring reliability at an architecture level. The section provides arguments for the need to find a new formal method for ensuring reliability. Then, Section 2.5 surveys the related work in defining component-based development process models. We highlight the main activities used for developing component-based systems and provide a component-based process model for developing trustworthy systems. Section 2.6 explains the motivation behind our work.

2.1 Component Models

There is a common agreement [CL02, Szy02] that component specification should include both structural and behavioral descriptions. Structural description includes, but is not limited to, specifying *interfaces*, *connectors*, and *composition*. These are central concepts in component-based development. An interface defines access points to the services provided and requested by components. A connector is a special component that defines the communication between two components. Composition allows building systems by connecting existing components in such a way that preserves their essential properties.

In the literature, there is a large number of different component definitions. However, only a few of them have been considered as component models in a taxonomy of software component models [LW05, LW07]. These are SOFA 2.0 [BHP06], Fractal [BCL+06], KobrA [APRS01], Koala [vOvdLKM00], PECOS [NAD+02], and Pin [HIPW05]. We add SaveCCM [ÅCF+07] to this list. These component models provide a wide variety of component definitions and contributions to the advancement of CBD. In the following sections we provide an overview of these component models and analyze their relative merits.

2.1.1 Koala

Koala [vOvdLKM00] is a component model used for specifying and developing embedded systems in consumer electronics. Koala provides strict separation between component and configuration development. Components are developed with no assumption about the deployment configuration in which the component will be used. A component definition in Koala includes a set of interfaces. There are two types of interfaces: *requires* and *provides*. *Requires* interfaces are used to access functionality, whereas, *provides* interfaces are used to provide functionality. *Diversity* interfaces are special required interfaces that are attached to components and used to get configuration parameters that are controlled centrally. Switches allow *requires* interfaces to be bound to multiple different *provides* interfaces based on configuration parameters. Then, when the selection is resolved, only one binding is selected depending on the values returned through the diversity interfaces. Consequently, the switch will direct calls to one of the required interfaces bound to it. It is possible to define *optional* interfaces and query if they are available or not before trying to connect to them. A configuration is a set of components bound together to form a

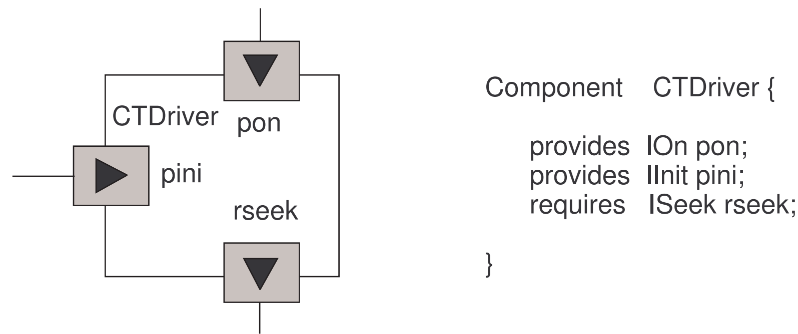


Figure 2: Koala component and its CDL description

product. Modules are used to implement functions of interfaces. A module is a component with no interfaces. It is bound to interfaces of a component to provide initialization and implementation code.

Koala has a component description language (CDL) used to specify systems and components. A design viewer is used to view CDL descriptions. Koala provides code synthesis by mapping CDL to an implementation programming language. Figure 2 shows a Koala component and its corresponding CDL description.

The above description shows that Koala provides only structural description for components. It does not provide behavior specification or non-functional contract. Therefore, it is not a suitable model for specifying and verifying trustworthiness properties.

2.1.2 PIN

Pin [HIPW05] is a component model and runtime environment. It provides a basic and simple component technology suitable for building embedded safety critical systems. Pin components are fully encapsulated by applying the container concept. Containers provide a “prefabricated shell” in which the custom code of the component executes and through which all interactions between the custom code and its external environment are mediated. Systems are assembled by selecting components and connecting their interfaces, called pins. Component interfaces receive stimuli through *sink pins* and respond through *source pins*. Figure 3 depicts a Pin component.

Each Pin component is implemented as a distributable dynamic link library (DLL). Pin

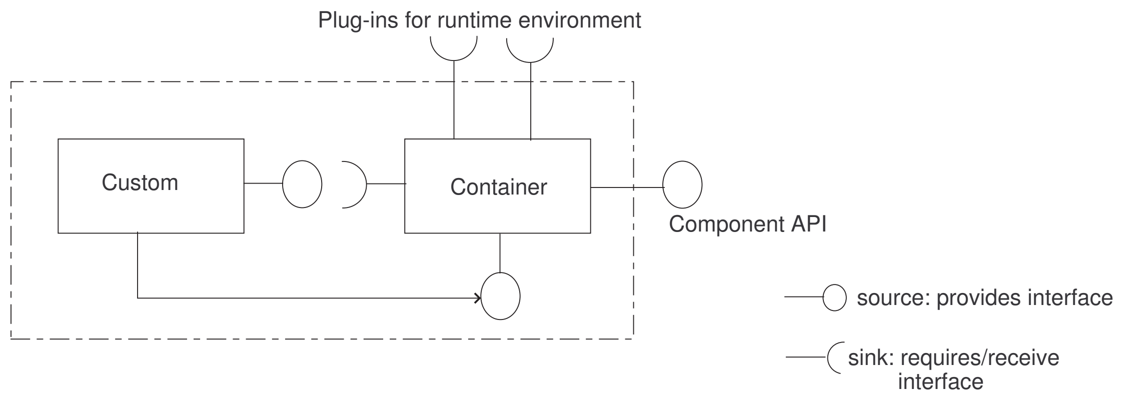


Figure 3: Pin component

supports a model of pure assembly. Applications are constructed by connecting components using connectors. A connector may impose coordination policies beyond those provided by containers. For example, a connector may impose a queuing policies on message buffers.

Pin component technology includes a component runtime environment which provides services and enforces component interaction policies. Services include access to the underlying platform; for example, timers, interrupts, and input devices. Interaction policies governing shared resources, such as process scheduling and inter-component communication, are also provided by the runtime environment. Lastly, the runtime provides a portability layer for components and their assemblies.

Similar to Koala, Pin does not provide support for non-functional properties. Hence, it is not a suitable model for specifying and verifying trustworthy component-based systems.

2.1.3 PECOS

PECOS [NAD+02] is used for specifying and developing component-based embedded systems of field devices such as sensors, actuators, and positioners. In PECOS, a component has a name, a number of property bundles, and ports. The ports of a component represent data that may be shared with other components. The behavior of a component consists of a procedure or an algorithm that reads internal data or the data available at its ports. Then, it produces data on the component ports or produce effects in the physical world.

Figure 4 provides an overview of PECOS component model. Components can be either simple or composite. A simple component can not be further defined by a model but

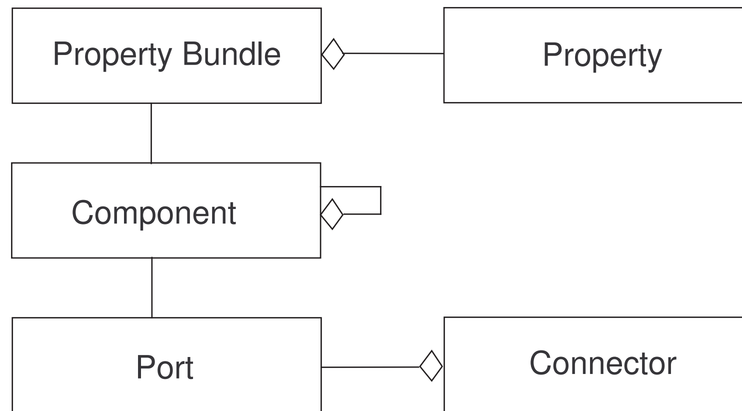


Figure 4: PECOS component model

rather directly implemented in a programming language. A component may have multiple property bundles, where each of property bundle consists of one or more properties. A composite component contains two or more subcomponents connected together. Connectors are used to connect ports of subcomponents.

A port is specified using a unique name, the type of the data passed over the port, and the range of the correct values (minimum and maximum values) that can be passed through the port. A port is implemented as a shared variable which allows communication between components. The ports of subcomponents can be connected only if they have the same data type. A connector is specified using a name, a type, and a list of ports it connects.

A property in PECOS is a tagged value, where the tag is an identifier and the value is typed. A collection of properties are grouped using a property bundle. It is used to characterize aspects of a component such as timing or memory usage.

A system is specified in CoCo language, which can be easily translated into target languages such as C++ or Java. The component structure from the CoCo specification can be mapped directly to an identical class structure in the target language. The behavior of the component has to be implemented by programmers. Thus, PECOS does not have behavior specification method.

PECOS provides a means to define simple non-functional requirements such as memory usage and timing. It is possible to perform real-time schedule analysis based on PECOS specification. Therefore, it provides only very limited support to non-functional requirements. Therefore, it is not a suitable model for specifying and verifying the trustworthiness properties.

2.1.4 KobrA

KobrA [ABB+02, ABB+07] is not a formal language, but rather a methodology for modeling architectures. It comprises a set of principles for using mainstream modeling languages, such as UML, to describe and break down the design of complex systems in a component-based way.

KobrA divides the full specifications of a component into two parts: a *specification* and a *realization*. A specification describes what a component does; hence, it focuses on the external view of a component. On the other hand, a realization describes how the component does its functionality in terms of interaction with other components; therefore, it focuses on the internal design of subcomponents and interactions between them. UML diagrams are used to describe these parts. The specification description includes three main models: (1) one or more static structure diagram giving the structural view that describes the nature of classes and their relationships, (2) a set of operation specifications giving the functional view in a tabular form specifying name, description, receives, sends, rules, changes, assumes, and result, and (3) a state chart diagram giving the behavioral views in terms of events, operations and states. The realization description includes three main models: (1) a static structure diagram presenting the design level structural view, (2) a set of interaction diagrams (collaboration or sequence diagrams) giving the interaction-oriented view, and (3) a set of activity diagrams giving the algorithmic view.

KobrA does not have tool support to ensure that a created model is compliant with the KobrA methodology. The process of modeling and analysis can be done only manually. Consequently, using the KobrA modeling technique for any large and complex project is both tedious and error-prone.

2.1.5 Fractal

A component in Fractal [BCL+06, BBC+07] consists of a *content* part and a *control* part. The content part contains either a hierarchical composition of subcomponents for a composite component or a java implementation for a primitive component. A component has a number of possible interfaces. Each interface is an instance of an interface type which states the signature of the interface, its kind, contingency, and cardinality as follows:

- The interface is a server (required) or a client (provided)
- Interface contingency defines whether an interface is mandatory or optional.

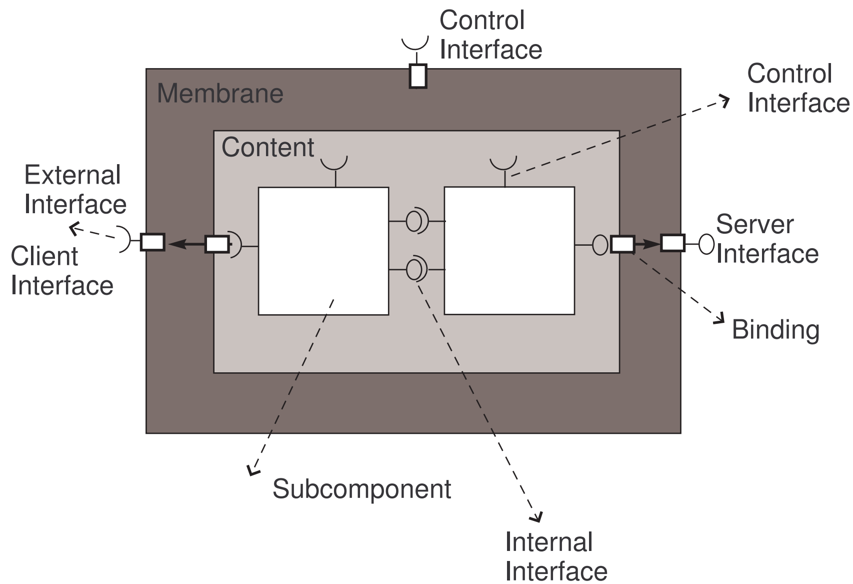


Figure 5: Fractal component

- Interface cardinality defines the number of possible instances that can be created of a specific interface type.

The control part is a composition of controllers, where each controller performs a particular management functionality such as creating components, binding client interfaces, managing sub-components, manipulating component's attributes, and managing the component's life cycle.

Figure 5 depicts a fractal component. The content part represents a component composed of two sub-components. The external interfaces are visible outside the components, whereas the internal interfaces are used to compose the constituent components of the content. Control interfaces are used to deliver management commands to components. Client and server interfaces are bound with internal interfaces that are not part of composition.

The behavior of Fractal applications is specified using SOFA [BHP06] behavior protocols, which specify the valid sequences of method calls on component's interfaces. The FractalBPC, *behavior protocol checker*, platform is used for specifying behavior protocols and verifying whether or not the behavior of a component complies with its stated behavior protocol. A correct behavior means absence of communication errors. There are three identified types of communication errors:

- Bad activity: an issued method call cannot be accepted.

- No activity: all of the ready method calls are prefixed with input sign (deadlock).
- Infinite activity: the composed protocols can not reach their final method calls, which means that the composed behavior contains an infinite trace.

FractalBPC is made of two behavior protocol checkers: static (code) checking using Java Pathfinder [Pat] and run-time checking. Static checking allows exhaustive analysis to verify whether the implementation of each primitive component corresponds to its defined behavior protocol. Run-time checking is done by monitoring method calls on the component's external interfaces at run-time to check whether or not the traces of component execution comply with the stated behavior protocol.

FractalADL is an XML-based architecture description language for Fractal component model. Also, it is the name of a tool-chain that parses the ADL, which describes how to instantiate components, and builds the application accordingly using Fractal API.

The behavior protocol specification is limited to functional requirements, where sequences of valid invocation of methods are specified. It does not support specifying non-functional requirements over those valid sequences such as security or reliability. Also, it does not support timing requirements. This is because Fractal is focused on the domain of distributed systems, not real-time systems.

2.1.6 SOFA 2.0

SOFA 2.0 [BHP06] is a hierarchical component model that inherits structure from its ancestor SOFA [PV02]. The main features of SOFA 2.0 include: (1) a meta-model based design of components, (2) support for dynamic reconfiguration of architectures using predefined patterns that allow adding/removing components and connecting to external services, (3) support for different communication styles by defining connectors as first class components, (4) defining the control part of components using micro-components, (5) seamless support for version control, (6) provision of design time and runtime environments for the development and deployment of component-based systems, and (7) support for behavior specification and verification of compliance.

A component in SOFA is specified by a *frame* and *architecture*. The frame specifies a black-box view of the component. It defines the *requires* and *provides* interfaces and properties of the component. An interface is an instance of an interface type, which specifies the signature of the provided or requested methods. An interface definition contains the

following specification attributes: (1) *connection type*, which determines whether an interface can be used for reconfiguration control or not, (2) *is collection*, which captures the cardinality of an interface, and (3) *communication style*, which denotes the communication paradigm used at deployment time for the methods in the interface.

The architecture specifies the implementation of the frame. A frame can be implemented by several architectures. An architecture describes whether a component is composite or primitive. It specifies the internal structure of composite components using connectors and bindings. Connectors are first-class entities in the SOFA component model. There are three possible bindings: (1) *delegation*, which connects a provides interface of a component to one of its subcomponent's provides interfaces, (2) *subsumption*, which connects a subcomponent's requires interface to a requires interfaces of the component, and (3) *connector*, which represents a connection between two or more subcomponents.

SOFA 2.0 allows dynamic reconfiguration to components during run-time. There are three reconfiguration patterns allowed in SOFA 2.0: (1) *factory*, which allows adding new component to the architecture, (2) *removal*, which allows removing a component from the architecture, and (3) *utility interface*, which allows accessing interfaces across the component boundaries.

The behavior of a component is specified as a set of traces of events (method call requests and their corresponding responses) appearing on component interfaces. A behavior protocol is an expression built using events, classical regular operators (';', '+', '*'), and parallel composition by interleaving events. It defines the set of valid sequences. Tool support is provided to verify the compliance of component behavior to its defined behavior protocol. Static and run-time checking is provided.

The *component definition language* (CDL) is used to describe interfaces, frames and architectures of SOFA components. Figure 6 shows an example CDL specification. The example shows the specification of a frame, *DataAccess*, which consists of one requires interface of type *IAuthorize* and another provides interface of type *IAccess*. The specification of each interface type includes signatures of methods and the behavior protocol specification of the corresponding interface type. For example, the behavior protocol of *IAccess* represents all the possible traces in which the method *init* is executed first and followed by zero or more executions of either *executeQuery* or *executeReport* and terminated by the method *finish*. An architecture specification is provided to implement the composite frame *DataProcess* by composing the two components *DataAccess* and *AuthorizeData*. SOFA

2.0 is supported by a tool-suite for developing, assembling, deploying, and controlling run-time of applications.

Similar to Fractal, SOFA modeling is targeted for the domain of distributed systems. Thus, it provides powerful support for defining hierarchical components, different communication styles, and dynamic reconfiguration. However, it does not support specifying and verifying the trustworthiness properties neither in the structural definition nor the behavioral specification.

2.1.7 SaveCCM

SaveCCM [ÅCF+07] is a component model specially designed for vehicular systems. The model includes components, interfaces, switches, and assemblies. The SaveCCM model is based on the control flow paradigm using the pipe-and-filter architecture style. Interfaces are divided into input and output ports. A port can be used for data, control, or both data and control. Data ports are used to read and write data, whereas control ports are used to activate components. An interface may contain a set of name-value attributes such as the worst-case execution time value and the estimated reliability value. The functionality provided by a component is implemented by a single entry function using the C language. Therefore, each component in SaveCCM provides only one function, where its data ports act as data parameters for this function and its control ports activate the execution of the code. A component's function can be executed only if data has been received at all its input data ports and its input trigger ports are active. The execution model is read-execute-write: read input data from input ports, execute the component function, and write data to data output ports. Figure 7 depicts a SaveCCM component and shows the different possible ports that can be associated with it.

The switch construct in SaveCCM is similar to its synonym in Koala. It allows changing the component interconnection structure using predefined conditions specified as logical expressions. An assembly is used for naming a set of components connected sequentially according to pipe-and-filter architecture style.

The internal behavior of components is modeled using timed automata.
The UPPAAL [BDL04] model checker is used to verify timeliness and safety properties. The component function is modeled as a real-time task associated with computation time, deadline, and sequence of variable assignments. The Times [AFM+03] tool is used to perform real-time schedule analysis. A transformation tool is used to automate the process

```
interface IAccess {
    void init();
    void executeQuery(in string q);
    void executeReport(in string r);
    void finish();
    protocol: init; (executeQuery + executeReport)*; finish
    };
interface IAuthorize {
    void requestAuthorization();
    void requestStatus();
    protocol: requestStatus; requestAuthorization
}
frame DataAccess {
    provides: IAccess p;
    requires: IAuthorize r;
    protocol: !r.requestAuthorization; ?p.init;
              (?p.executeQuery + ?p.executeReport)*;
              ?p.finish; !r.requestStatus
}
frame DataProcess {
    ...
}
frame AuthorizeData {
    ...
}
architecture DatabaseAccess
            implements DataProcess {

            inst DataAccess da;
            inst AuthorizeData az;

            bind da.r to az.authorize;
            delegate processQuery to da.p;
            }
```

Figure 6: Example SOFA CDL specification

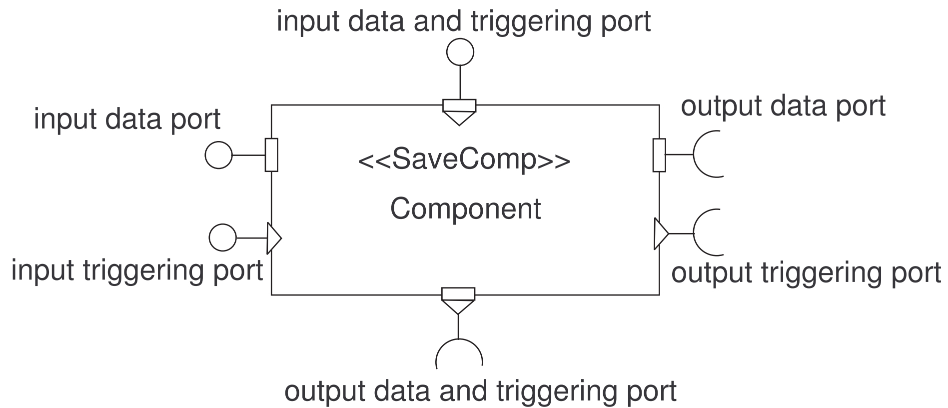


Figure 7: SaveCCM component

of analyzing components and generating extended timed-automata with tasks suitable for the Times tool. Also, an automatic code generation tool is used to create all the low-level platform dependent code, which is necessary for run-time.

SaveCCM components are primitive with only one provided function. Its execution model is very restrictive. The architecture style used is limited. It is very well suited for its intended domain, which is the domain of vehicular systems. However, it cannot be used for modeling other types of component-based systems. Also, it provides little support for specifying non-functional requirements.

2.2 Analyzing Current Component Models

2.2.1 Comparison

A detailed comparison between the component models mentioned above is presented in Table 1 and Table 2. In Table 1 we compare the static and dynamic aspects of the component models. The model includes several views. These are : (1) *structural view:* a view of the component structure, (2) *contract view:* a view which includes description of the non-functional requirements, (3) *behavioral view:* a specification of the behavior protocol, and (4) *composition:* a formal rule to compose simple components into a composite component. In Table 2 we compare the ability of the component models to ensure trustworthiness during different phases of software development. This includes the type of tools used to perform analysis during: (1) design, (2) implementation, (3) run-time, and (4) dynamic

configuration at run-time.

Table 1 compares the static and dynamic aspects of component models. It shows that:

- Component structures are either hierarchical or flat, primitive, including interfaces and connectors with the exception that Fractal does not have connectors.
- Only PECOS and SaveCCM define some non-functional requirements. All other component models have no support for non-functional requirements in their component definitions.
- PECOS and SaveCCM support only temporal requirements. PECOS's contract view is limited to execution-time, cycle-time, and memory consumption. SaveCCM enables defining named values; however, it is not clear how invariants and non-functional constraints can be defined using these named values.
- In SaveCCM the temporal requirements are encoded in the behavioral protocol. This limits the expressiveness of non-functional requirements by the behavior specification language.
- Component compositions define structural composition only. Although the component definition mentioned earlier highlights the importance of the composition rule, there is no current component model that defines a compositional theory that includes both structural and non-functional requirements [LW05]. Such compositional theory is essential for ensuring trustworthiness.

Table 2 compares the analysis ability and tool support of component models. It shows that:

- PECOS and SaveCCM support real-time analysis at design time using formal verification. All other component models don't have support for non-functional requirements analysis.
- The non real-time component models, except KobrA, provide compilers to check the syntax of system definitions.
- SOFA and Fractal support analysis at all development phases for protocol compliance (functional properties) to ensure that components behavior is restricted to the defined behavior protocol. However, their compliance analysis at implementation and run-time can handle only non-parameterized protocols.

- Only SOFA and Fractal supports run-time reconfiguration of components.

Therefore we find that, generally, current component models have limited or no support for non-functional requirements, specifically the properties of trustworthiness.

2.2.2 Services

Component can be defined generally as a software unit which provides or requires services. Service definitions are used to classify, search, and select components. A service can be defined as a function that maps a parameterized input request into an output action. We argue that non-functional requirements are generally concerned with component services. The following examples illustrate our hypothesis:

- Component security is concerned with ensuring that only authorized users have access to component data and services. It is possible to have different security classification and access levels to different services. However, integrity requirements are concerned generally with the component as a whole not a specific service.

- Component safety is concerned with ensuring that service executions do not cause catastrophic consequences to users or the environment. Timeliness requirements are concerned with execution time of services.

- Reliability is concerned with "the continuity of correct services" [ALRL04].

- Availability is concerned with "the readiness for correct service" [ALRL04].

These examples show that service definition is a central concept. Therefore, component definitions should include service definitions and enable assigning non-functional requirements, specially the properties of trustworthiness, to services. In current component models, services are defined indirectly as signatures in interface definitions. This makes service definitions a secondary concept with no attributes or properties assigned to them, which makes it difficult to specify and verify the non-functional requirements that are related to services.

2.2.3 Component Contract

We argue that non-functional requirements and *environmental constraints*, which are constraints related to the environment in which the component will be deployed, should be

Table 1: Comparison of static and dynamic aspects of component models

	Structural View	Contract View	Behavioral View	Composition
Koala	hierarchical components, interfaces, switches, and modules			structural
Pin	flat components, container, interfaces, and connectors		stimulus-response behavior using UML statechart	structural
PECOS	hierarchical components, ports, and connectors	execution and cycle time, and memory consumption		structural
KobrA	hierarchical component defined using UML and interfaces		UML interaction diagram	binding interfaces
Fractal	hierarchical components, controllers, and interfaces		behavior protocol	binding interfaces and behavioral composition
SOFA 2.0	frame, architecture, controllers, interfaces, and connectors		behavior protocol	structural and behavioral composition
SaveCCM	flat components, interfaces, switches, and assemblies	name-value properties	timed automata	structural

Table 2: Comparison of analysis tools support

	Tools	Design-Time Analysis	Implementation Analysis	Run-Time Analysis	Dynamic Configuration
Koala	design viewer, code synthesis, and ADL compiler				
Pin	run-time environment				
PECOS	code synthesis	real-time schedule analysis			
KobrA					
Fractal	FractalADL and FractalBPC		static checking of behavior protocol compliance and correctness of communication	run-time checking to verify that a component code obeys its behavior protocol	component life cycle management and interface binding control
SOFA 2.0	runtime environment, designer, code synthesis, and static checking		static checking of behavior protocol compliance and correctness of communication	run-time checking to verify that a component code obeys its behavior protocol	adding and remove components at run time
SaveCCM	designer, verification tool, and code synthesis	model checking safety and performing real-time schedule analysis			

defined in a non-functional contract associated with the component. The contract specification should not be included as part of the component definition but rather *assigned* to it. This is because environmental assumptions and non-functional requirements might not be valid when the component is used in different deployments. For example, in real-time embedded systems, time constraints that define the maximum amount of time for a safe execution of a service might be different depending on hardware and software configurations of the deployment environment. The separation between the computation part of the component and its contract enables components to be reused in different environments by changing only its contract. However, when studying current component models, we find that there is no support for non-functional contracts. In current component models, non-functional requirements, if they exist, are defined either as attributes in the component definition or as part of the behavior protocol specification. For example, in PECOS, memory consumption and worst-case execution time are specified as attributes in the component definition. On the other hand, in SaveCCM, timing requirements are specified in the timed automata. In the former case, specifying non-functional requirements inside the component definition makes it difficult to reuse the component for different deployments. In the later case, specifying non-functional requirements in the behavior protocol restricts the specification by the limitation of the behavior protocol specification language. Thus, the lack of contract specification in current component models severely limits the reuse of components, the very essential motivating factor for propounding component technology.

2.2.4 Encapsulation and Composition Theory

A component model addresses the issues of assembling components to develop component-based systems. There are two major concerns when assembling components: encapsulation and composition of non-functional properties.

First, components should be self contained and loosely coupled. The composition mechanism should preserve encapsulation of component's data and control. This is done by separating the computation part of the component from its interactional specification. A study of current component models [LW07] revealed that components are composed using direct method calls or indirect message passing through connectors. Thus, these models produce tightly coupled components that are difficult to reuse.

Second, when assembling components, special composition rules should be applied to ensure that the non-functional requirements of the constituent components are preserved

in the assembly. This requires a defined composition theory. Composition is a central concept in component-based development. However, when analyzing the aforementioned component models we found that they define only structural composition by linking interfaces together. There is no component model that defines a composition theory for both structural and non-functional parts of a component. This finding has also been described in [LW07].

2.3 ADL Related Work

This section discusses the related work in the architecture description languages (ADLs) literature. Both component models and ADLs share the component concept. A comprehensive study of ADLs was presented in [MT00]. Recent studies have shown the limited support of ADLs for non-functional requirements. Garlan and Schmerl [GS06] remark that, "despite the notable progress and concern for ways to represent and use software architecture, specification of architecture designs remains relatively informal, relying on graphical notations with weak or non-existent semantics that are often limited to expressing only the basic of structural properties". This section presents a brief review of three ADLs: Acme [GMW00], secure-xADL [RT05], and AADL [AAD].

2.3.1 Acme

Acme is a second generation architectural description language. It provides support for specifying the canonical set of structural elements of an architectural design. It includes definitions of component, port (runtime interface), connector, connector role, system, property (attribute), constraint, and representation (substructure of a component or a connector). Acme is extensible. Therefore, it serves as a basis for developing new domain specific ADLs and integrating existing architectural design analysis tools. Acme is intended to provide a unifying ADL for interchanging architecture descriptions between different ADLs. It includes only the essential items that are common among ADLs. Since non-functional requirements are not common in ADLs, component definition in Acme does not include non-functional requirements. Hence, it is not suitable for defining architecture of trustworthy component-based systems.

2.3.2 Secure-xADL

A secure architectural description language (secure-xADL) was introduced in [RT05]. The proposed language is based on extending xADL, an XML based extensible ADL. The basic structural elements of architectural modeling in xADL are components and connectors. Secure-xADL uses an access control mechanism for modeling security at the architectural level. The access control model [Bis03] precisely defines the rights of every subject with respect to every other secured entity. In order to ensure security, components and connectors play different, yet complementary, roles. Component types provide security *contracts* that specify the subject it acts for, the principals it can take, the privileges it possesses, and the safeguard it requires. These are defined as part of its interface specifications. On the other hand, connectors regulate and enforce the defined security contracts of the communicating components. Connectors check the contracts at the two ends of the communicating components and decide whether they have sufficient privilege to communicate. Then it either lets the communication passes through or rejects it.

Although Secure-xADL introduces a promising approach for modeling security at the architectural level, there are four major issues that, to the best of our knowledge, have not been addressed by Secure-xADL. First, it does not provide a solution to the compositional problem which is a major concern in component-based development. Second, Secure-xADL does not provide mechanism for the formal verification of security policies in a component and in the whole system. Third, it does not provide a mechanism for performing consistency checking to ensure that the defined policies does not include conflicting specifications. Fourth, security contracts are maintained at a component level; therefore, there is a need to prove that the subject actually posses the claimed privileges on the objects. Otherwise, any component can make false claims and violate security by accessing restricted resources. There is a need for either a centralized or distributed authentication authority to confirm the legitimacy of the claimed privileges. To the best of our knowledge, Secure-xADL is the only secure ADL proposed in the literature.

2.3.3 AADL

The *architecture analysis and design language* (AADL) [AAD] is a textual and graphical architecture description language used to specify the design of real-time systems. AADL provides formal modeling abstractions for the specification of complex real-time embedded

systems. The structure of a system is specified as an assembly of communicating components. Their interfaces, functional and data, and non-functional properties, such as timing requirements and space requirements, are precisely defined. Component specification in AADL includes an identity, possible interfaces with other components, properties, and subcomponents defining the internal structure of the component's implementation. Components are divided into three categories: application software, execution platform (hardware), and system. Each component category has its own predefined set of properties. Interactions between components are defined as flows of control and information through defined connections. Multiple predefined configuration settings and interactions between components can be defined using operational modes. The language enables deployment specification by allocating software components to execution platform components. It is possible to extend the language with more properties and analysis specific constructs that can be associated with components. The *error model annex* is a standard AADL extension that supports fault/reliability modeling and analysis.

In AADL a primitive component represents a single service for which the defined *data ports* specify the stimulating input or triggered output event and the input and output data parameters. Therefore, AADL does not provide a clear distinction between a component and a service. In AADL, the critical safety requirements are specified as properties that define timing requirements, period, worst-case execution time, deadline, space requirements, and arrival rates. These properties are included in the component specification. This hinders the reuse of components for different deployments. In our view, specifying contracts outside component definition enables reuse of components and contracts definitions and allows changes to contract without affecting component specification. AADL does not support security specification, but it supports reliability specification by defining *error models* annotated with probability parameters. However, the values of the probability parameters are based on assumptions. Therefore, the accuracy and precision of these values can not be proven or justified. AADL does not provide a mechanism to analyze availability.

2.4 Specifying Trustworthiness Properties

This section reviews the research efforts made in the past for formally specifying trustworthiness properties and summarize their conclusions.

2.4.1 Combining Safety and Security Properties

In the literature, safety and security properties are formally specified and composed using different methods. This is due to the common consensus that while safety properties are defined as sets of "safe" sequences, security properties cannot be expressed as sets of sequences[McL96, Zak96, Man02]. It is well known [AL93] that safety properties can be preserved in a composition; however, some security properties are not preserved by any composition [McC88]. Hence it was concluded that it would not be possible to neither express safety and security using one formal logic nor use one compositional theory for both safety and security. This implies that different formal methods have to be used for the specification and verification of security independent of safety.

Many security properties have been proposed as *information flow properties* [GM82, McC88, McL90, McL94, FG95, Man00] which attempt to prevent a low-level user from *inferring* some thing that is confidential to a *high-level* user [Zak96]. Many interface security properties that were presented early on were proved to be weak in later research and were replaced by stronger ones. See [GM82, McC88, McL90, McL94, FG95] for a history of the research related to the introduction of new information flow properties and an account of how they were either proven to be weak subsequently or proven that they failed to preserve security in composition. Most importantly, the use of this type of security properties doesn't allow combining it with safety properties within one formal specification method so that composition, and verification can be formally achieved [McL96, Zak96, Man02]. Finding a single composition rule and a formalism to assure the satisfaction of trustworthiness in composite components has been an open problem until now.

In this thesis we propose a composition rule that unifies both *access control* and *interface security* models. Access control models restrict access to component services, and validate user requests of authorized users. We apply this restriction at the interfaces of components. We argue that the access control security properties suggested by this thesis can be expressed as sets of sequences. Hence, these security properties can be expressed in any mathematical logic in which safety properties are expressed. Therefore, one compositional theory can be used for both safety and security.

2.4.2 Reliability

In the literature, there is a large number of publications aiming to predict the reliability of systems at architecture level [IN08, Gok07]. The comprehensive analysis of these approaches reveals that these practices suffer from a serious defect which is the fact that the estimated quantitative value of reliability is based on inaccurate or unjustified assumptions about component reliability [CRMG08, IN08, Gok07]. This is because the only way to quantitatively measure the exact accomplished reliability is by using operational profiles, sets of execution sequences of component behavior. Since the reliability prediction is done at design time and since the operational profiles are available only after deployment and execution time, many assumptions are made in order to quantify reliability. These assumptions are uncertain and unjustifiable. This is the motivation for seeking a new approach for defining reliability and availability in this thesis. We believe that reliability and availability must be architected at design time, specified in the component's contract, and ensured by the implementation. Model checking technique can be used to verify the correctness of architecting reliability and availability at design time.

2.5 Process Models for Component-based Development

A software engineering process defines a set of integrated activities to develop software systems. A typical software engineering process groups the related activities into stages of requirements acquisition and analysis, design, implementation, testing and verification, deployment, and maintenance. The type of activities involved may vary from one system to another depending on the type of system and the development methodology used. For example, the activities involved in designing a safety critical system differs from those used to design a library classification system. Rigor must be applied in the former. Also, the design of a system using the object-oriented methodology differs from the design of the same system using procedural programming.

In order to achieve the benefits of component-based development (CBD), a component-based process must be used. The design of a component-based system differs than the design of other types of systems. CBD is a reuse oriented process. Therefore, the development of reusable components and the integration of components to create systems are the main concerns in a CBD process. A major difference between conventional software engineering process and a CBD process is that the former results in a software system, whereas the

later results in a software system as well as a system with reusable components [TGG07]. A typical component-based development process comprises two parallel activities: software component development and component-based system development [CCL06, Pre05]. The former addresses the issues of component's specification, development, qualification, documentation, cataloguing, and adaptation and selection for reuse. The later addresses the issues of assembling components to develop component-based systems. Several component-based process models exist in the literature [Chr95, DT03, TGG07, CCL06]. The following presents an overview of these process models highlighting their main features. The overview is arranged in a chronological order based on their presentation time.

In [Chr95], a reuse-based software development process model was presented. This process is based on the hypothesis that domain engineering is the foundation for a reuse-based software system development. A domain is a set of applications that share similar requirements, capabilities, and data. Domain engineering is the set of activities that create and support a model, architectures, components, and applications specific to the domain. The *domain analysis* defines the requirements that are common for all products in the domain and the requirements that vary for each product. These requirements are used to develop a *domain model* that includes requirements of all products. From the domain model, a *domain architecture* is developed to form the basis for all domain products. The architecture is further refined to define the constituent *reusable components*. *Domain applications* are designed based on the domain architecture and developed by reusing existing domain components.

In [DT03], a component-oriented development process was presented. The process focusses on system development by integrating existing components. It is based on the *abstract design paradigm* which suggests decomposing a system structure hierarchically into components and associating data, functions, and controls to each component. A design modeler starts with system requirements and uses a recursive structural decomposition to arrive to the definition of composite or simple components. Then, activities of component specification, search, modification, or creation starts. After that, the system is built by integrating components.

In [CCL06] a process of three parallel tracks was presented: component development, component assessment, and system development. The activities in component assessment include finding and evaluating existing components. It yields a repository of components that includes the components' specifications, descriptions, documented tests, and

executable components. In [TGG07] two independent processes are defined for component and system development.

Tables 3 and 4 provides a summary of the activities suggested in [Chr95, DT03, CCL06, TGG07] for both component and system development. Reviewing these models helps us to extrapolate the main activities involved in a general component-based development process, which can be extended with rigorous methods to define a component-based process for developing trustworthy systems.

2.5.1 Discussion

From the above summary and Tables 3 and 4, we find that there are four major activities involved in component-based development. These are *domain engineering*, *component development*, *component assessment*, and *system development*. These activities are important, however might not all be required at the same time. For example, it is possible to have a company which focuses only on developing and selling components. Therefore, there is neither domain engineering nor system development. On the other hand, there could be a company which has a domain engineering and system development but no component development because it buys the required components from others using the component assessment activities. Also, it is possible to have a single project which uses CBD; therefore, it requires component development and system development only. It is quite possible, however, to have an enterprise which uses all the four types of activities for developing complex systems. Such examples may be found in avionics, automotive, and product-line development industries. *Therefore, a component-based development process should address all four types of activities.*

Component assessment through testing and verification is an important factor for the success of reuse. The assessment should be done at a component level, in which the functional and nonfunctional requirements are tested and verified, and at a system level, in which composability tests are used to test the successful integration of reusable components. Integration testing should assess not only structural assembling but more importantly nonfunctional requirements, specially unwanted properties that may emerge at system level but remain invisible at component level. In safety/security critical systems, the issue of emergent properties is critical. Integration may violate safety or security properties. Verification of such properties is challenging [CCL06]. *Therefore, a component-based development process for trustworthy systems must use rigorous formal methods, including*

Table 3: A summary of the component and system development activities - Part 1

Phase	Component Development	System Development
Requirements	Domain analysis is used to identify required components [Chr95]. The defined requirements should address ranges of requirements and the possibility of reuse [TGG07].	In addition to requirements acquisition, existing components' information and documentation are reused [CCL06]. System requirements are captured and component requirements are defined to help in searching and selecting existing components [TGG07].
Analysis and Design	Assumptions are made about the environment in which the component will operate. A component technology is selected for components, such as .NET, J2EE, COM+, etc. The design should be general to enable reuse. Design adaptations to existing components to fit into the system [CCL06]. Detail component specification are designed including functional, structural, and nonfunctional specifications [TGG07].	The overall system architecture is designed. Then, the architecture is refined and the constituent components are identified and specified in details [DT03, CCL06]. A component-oriented architecture is selected and components are identified. Detail design of new components is performed. Verification and validation of functionalities are conducted [TGG07].
Implementation	The selected component technology determines the implementation details [CCL06]. The methods and events of components are implemented [TGG07]. Reuse is encouraged whenever possible.	The emphasis in implementation is put on component selection and adaptation [DT03, CCL06]. Components must be assessed before reuse [CCL06]. New components are developed and glue code is written [TGG07].

Table 4: A summary of the component and system development activities - Part 2

Phase	Component Development	System Development
Integration	Integration considerations must be continuously in focus through all phases [CCL06].	Architectural matches should be tested, and functional and nonfunctional behavior should be verified thoroughly to insure successful integration [CCL06]. Connectors are used to integrate components [DT03].
Testing	Extensive tests such as unit and integration testing should be done to verify functional and nonfunctional requirements. Test results should be delivered with the component to system developers [CCL06].	Tests must be performed during component selection and integration [CCL06]. Integration, system, and acceptance testings are required [TGG07].
Maintenance	Strategies should be defined for component maintenance [CCL06].	Replace old components by new ones or add new components into the system whenever necessary [CCL06].

verification and testing techniques, in component development, assessment, and system integration to ensure a correct and trustworthy system behavior.

Since software requirements form the foundation from which the development process starts and quality attributes can be assessed, there is a need for a formal specification language that collectively and precisely define the software requirements related to functional, nonfunctional (such as trustworthiness), and structural parts of the software. The combination of rich formal specification language and a rigorous development process provide a high assurance level of trustworthiness. *Trustworthiness must be a central concern throughout the different activities in the component and system life-cycle* as depicted in Figure 8. In every stage, established methods for the verification of trustworthiness are to be used. Iterative cycles exist between sequential phases to ensure that the trustworthiness requirements are satisfied. Although formal methods may seem complex and costly, they are inherently supported by automation tools. *Therefore, a rich specification language and tools support are essential for the success of the development process of trustworthy systems.*

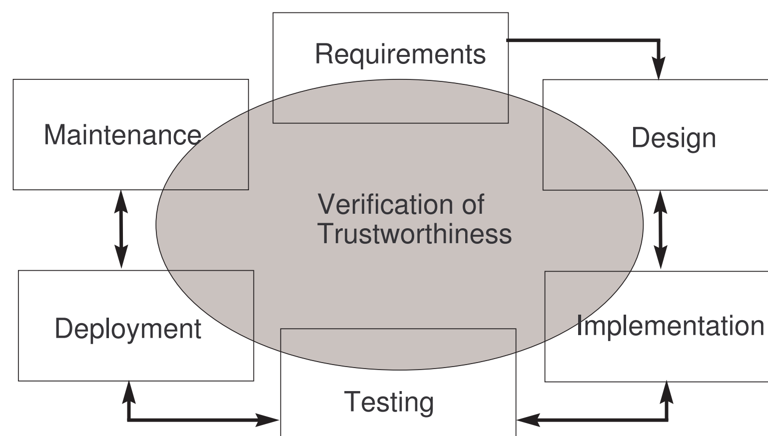


Figure 8: Trustworthy development life-cycle

2.6 Summary

From the above discussion, we conclude that there is a need for: (1) a rich and unified formal specification language for defining trustworthy components and systems, (2) a rigorous development process model that collectively includes domain engineering, component development, component assessment, component reuse, and system development activities,

and (3) a framework of tool support to support formal specification and the development process model. In these elements, trustworthiness must be a central concern. From the above survey and analysis it is clear that none of the existing component modeling technique nor a development process model can fit our needs. This is both a challenge and opportunity for creating a new component modeling technique that is both formal and practical.

Chapter 3

Research Methodology

This chapter presents the objectives of this research and shows the different research steps, which together formulate the research methodology used to reach the objectives.

3.1 Research Objectives

The goal of this research is to develop a formal component-based software engineering approach for developing trustworthy systems. The objective is achieved by putting together the following three contributions:

- A formal component model that collectively addresses the requirements of trustworthiness and component-based development,
- A formal development process model that describes component engineering and component-based development of trustworthy systems, and
- A framework with a comprehensive set of tools that support the formal development process.

3.2 Research Methodology

The research methodology is divided into three phases. The first phase is concerned with defining a formal component model for trustworthy systems. The second phase is concerned with defining a process model for the component-based development. Finally, the third phase is concerned with defining and implementing a development framework. The

following three sections describe the research problems, research questions, the solutions provided by our research for the stated problems, the limitations, and the future work of each phase.

3.2.1 Phase 1: Defining A Formal Component Model for Trustworthy Systems

This section presents the research problems in defining a formal component model for trustworthy systems. There are four research problems outlined in this section. For each problem, we discuss the corresponding challenging research questions and provide our proposed solution.

Research Problem 1-A: The lack of support for non-functional requirements

Problem Statement: *Current component models provide limited or no support for defining non-functional requirements. Therefore, they do not fit the need to define trustworthy component-based systems.* As discussed in our literature survey, current component models focus on the structural part of components. In order to profit form component technology to develop trustworthy systems, there is a need to extend component definitions with specification of trustworthiness.

Research Questions: Below we discuss the questions related to the research problem 1-A and provide solutions.

Q1: *What are the essential properties of trustworthiness?* We define trustworthiness as a composite concept that involves safety, security, reliability, and availability properties. The safety property is an invariant property of the system. Thus the property is true at all system states. In general, a safety property is a constraint that will prevent a system state in performing an action that might injure the environment as well as damage its internals. For real-time systems that are safety-critical, both liveness and timeliness become part of safety property. Security properties include confidentiality and custom defined security policies that are concerned with security of the services provided by the component and its local data. Reliability and availability properties include constraints that will limit, if not prevent, the frequency of system failures and set acceptable duration on non-availability of service due to repairs.

Q2: *What are the essential constituents of the component model?* Components provide and request services at public interfaces. A component can have local data variables that are used by its services. Also, a component may have constraints that define invariants over its behavior or its structural definition. A component can have non-functional requirements. Components interact with each other through connectors. A component can be primitive or composite. An architecture defines the internal structure of components. Therefore, in our perspective, a component definition should include structural, contract, and behavioral parts. The structural part defines the following concepts: component, interface, connector, architecture, attributes, constraints. The contract part defines the services, trustworthiness properties, and other non-functional requirements. The behavioral part defines the inter-play communication at the interfaces of a component and the trustworthiness restrictions.

Q3: *How can we define a trustworthy component model?* In order to develop trustworthy systems, rigorous methods should be applied for specifying and verifying systems. We use mathematical notations for formally specifying the structure, contract, and behavior of component-based systems. We formally specify the requirements of safety, security, reliability, and availability in the contract. The formalism provides abstract and precise definitions of the component model and trustworthiness properties. The formalism is the foundation of our engineering approach. It provides a unified language for describing the structure and the properties of trustworthiness. Also, it enables automatic techniques for formal analysis. Based on the formalism, verification of trustworthiness properties can be automated.

Q4: *How can we make the formalism easily accessible to non experts?* Realizing that formal notations are difficult to use and communicate among software architects, we created an architecture description language (TADL), which is based on our formalism. Our architecture description language provides complete descriptions for our trustworthy component model in a high level language that can be understood by software engineers.

Q5: *How can we define the behavior of trustworthy components?* In order to analyze and reason about the behavior of trustworthy components, there is a need for a behavior specification method that combines all the trustworthiness properties. We provide an extended timed automata for describing the behavior of trustworthy components. In

our solution, timed automata are augmented with timing constraints, security policies, and failure and repair behaviors. This rich behavioral model is generated automatically by analyzing the component structure and contract specification.

Limitations and Future work: In our component model, the security mechanism focuses only on confidentiality and authorization aspects. However, security is a broad concept that includes more considerations such as data integrity, encryption, intrusion detection, denial of service, and impersonalization. Some of these consideration must be guaranteed by the implementation of the component and the connector. For example, it is possible to implement connectors that analyze the communication and perform intrusion detection. Also, it is possible to implement encryption connectors that encrypt any communicated data or services. Moreover, our component model assumes a centralized security mechanism, where a central authority authenticates and authorizes users. However, in distributed systems, it is possible to have distributed authorization. In order to support such requirements, we need to add a *control unit* to the component definition. This control unit will be responsible for trust management. It operates as a controlling filter through which all out-going requests are encrypted and digitally signed and assigned a public key. Also, It decrypts the received requests and verifies the identity and credentials of the requesters. In this case, distributed authorities are needed, which host lists of certified identities with credentials. The control unit can communicate with those distributed authorities to verify identities. For every component, in order to operate, it should seek a digital certificate along with a public key from one of the distributed authorities. Then, it can use this certificate when communicating with other components. The certificate involves information about the contract guaranteed by the components, some of its quality attributes along with their values, and its security credentials.

Research Problem 1-B: The strong coupling of components

Problem Statement: *In current component models, components are composed using interface binding or connectors. This binding makes components tightly coupled. The strong coupling of composite components in current component models severely limit the reuse of components, the very essential motivating factor for propounding component technology.* Therefore, there is a need for a solution to define loosely coupled components that can be reused easily.

Research Question: Below we discuss the questions related to the research problem 1-B and provide solutions.

Q6: *How can we develop reusable components?* We provide two solutions for this question. First, the non-functional requirements of a component should be defined in a contract, which is associated with the component definition. This enables reconfiguring the contract to fit different deployments and systems. Second, the relation between the requested services and the provided services is specified in the contract, outside the component definition. The binding between services should not be included inside the component definition. This enables configuring different behaviors, relations between requests and responses, without affecting the component definition. Therefore, it increases reuse of the component definition.

Research Problem 1-C: The need for a composition theory

Problem Statement: Current component models define only structural composition by binding interfaces, either directly or using connectors. No composition theory exists for components. The concept of component composition goes beyond connecting components. Composition allows analysis about the non-functional properties of the constituent components after the composition. It ensures that the composition does not violate the properties that are already satisfied by the constituent components. Therefore, there is a need for a composition theory.

Research Questions: we discuss the questions related to the research problem 1-C and provide solutions.

Q7: *How can we define a composition theory for trustworthy components?* A composition theory should include both structural and contract composition. We provide a composition theory that defines two types of composition rules: composition rules for the structural part and composition rules for the contract part of our component model. The structural composition rules describe how a new component structure is formed by gluing compatible interfaces using connectors. It is possible to have multiple architectures for the new composite component. The contract composition rules specify how the services and trustworthiness properties are composed to form a new contract. There is only one possible contract for a component. Our proposed

solution is based on our ability to define both the structural and contract part of a component using one formalism.

Q8: *Is it possible to use one composition theory for safety and security properties?* In the literature, it was not possible to compose safety and security properties together because they were defined in different ways. The security properties were defined as *information flow* properties, which can not be composed in one method with safety properties. However, in our approach we use a role-based access control security mechanism. In this mechanism, security policies are specified using predicate logic. This makes it possible to specify, compose, and verify these policies with safety properties using one approach.

Limitations and Future work: The composition theory proposed by our research provides rules for composing safety and security properties only. We need to extend the composition theory with rules for composing reliability and availability. When a service fails at one component, it either produces erroneous results or become unavailable. This will affect the other components that request this service. Therefore, failures can propagate from one component to another. If a service becomes unavailable then it may violate the timeliness requirements because the component will not respond to requests in within the safe limits of time. Therefore, there is a need to extend the component definition to prevent the propagation of failures. In this regard, it is possible to define a control unit to monitor the behavior of services. If a service failure occurs then this service must be hidden from the public interface of the component until it is corrected. Dynamic reconfiguration may provide solution in this case. For example, the control part may create a new instance of the component, rebind all connectors to the interfaces of the new component, and delete the faulty component.

Research Problem 1-D: The need for an approach for specifying and verifying reliability and availability at architecture level

Problem Statement: *The current approaches for analyzing reliability and availability at an architecture level are based on inaccurate or unjustified assumptions.* Component reliability can be measured quantitatively by analyzing its execution sequences at run time. Since this information is not available at design time, the current approaches in the literature are trying to only estimate, but not measure, reliability. The estimation requires

assumptions about the deployment environment and the behavior of the component at run time. Since it is difficult to prove the accuracy and correction of the assumptions, there is a need for a qualitative approach for specifying and verifying reliability and availability.

Research Question: Below we discuss the questions related to the research problem 1-D and provide solutions.

Q9: *How can we provide a formal approach for specifying and verifying reliability and availability?* In our trustworthy component model, the definition of reliability and availability are based on service failures and repair durations. A failure is a deviation from the correct service behavior. It is indicated by any violation to the functional or non-functional requirements including those of safety and security. A repair is a change from the state of service failure to the state of correct service. The acceptable level of reliability is defined based on the frequency and severity of service failures. The acceptable level of availability is defined based on the duration of service failure time. The component implementation and maintenance should guarantee the repair time. The failures, repairs, and the acceptable levels of reliability and availability are formally defined in the component contract. Also, the behavior model is extended with failure and repair specification. This enables us to use formal model checking to verify safety, security, reliability, and availability in a one approach.

3.2.2 Phase 2: Defining A Process Model for Developing Trustworthy Component-Based Systems

This section presents the research problems in defining a formal process model for developing trustworthy component-based systems. We discuss the corresponding challenging research questions and provide our proposed solution.

Research Problem 2-A: The need for a process model for developing trustworthy systems

Problem Statement: *A conventional software engineering development process is not suitable for developing component-based systems. Also, a conventional component-based development process is not suitable for developing trustworthy systems.* Current component technologies are focusing on the structural and implementation aspects of component-based

systems. The typical component-based development processes presented in the literature focus on the general activities involved in developing component-based systems with emphasis on reuse and integration testing. On the other hand, safety-critical development processes are not suitable for component-based systems. This is because component-based development generates two products: a component-based system and reusable components. Therefore, there is a need for a rigorous component-based process for developing trustworthy systems.

Research Questions: Below we discuss the questions related to the research problem 2-A and provide solutions.

Q10: *What are the major activities in a component-based development process?* Our research found that a component-based development process should address four major types of activities. These are domain engineering, component development, component assessment, and system development.

Q11: *What are the requirements of a component-based development process for trustworthy systems?* We provide a process that inclusively blends the activities of component-based development and those of rigorous critical systems development. Our process includes the following merits: (1) it uses rigorous formal methods, including verification and testing techniques, in component development, assessment, and system integration to ensure a correct and trustworthy system behavior, (2) trustworthiness is a central concern throughout the different activities in the component and system life cycle, and (3) it has tool support for specifying and verifying component-based systems.

3.2.3 Phase 3: Developing a framework with comprehensive tool support

This section presents the research problems in developing a framework that supports the development process model. We discuss the corresponding challenging research questions and provide our proposed solution.

Research Problem 3-A: The need for tools support

Problem Statement: *Defining a rigorous development process is not sufficient to ensure trustworthiness. It requires tools to support the schematic implementation of the various tasks as in the process model.* Formal development of systems without tools is a difficult task. Designing systems formally requires expertise and takes long time and effort. Therefore, there is a need for building tools to automate and support the various activities defined in the rigorous development process model.

Research Questions: Below we discuss the questions related to the research problem 3-A and provide solutions.

Q12: *What are the necessary tools for realizing the trustworthy development process model?* We propose a blueprint of a framework for the development of trustworthy systems. The framework can be viewed in three layers: design, implementation, and deployment. Taken as a whole, the framework describes the tools necessary for the different stages outlined in the process model. As of now, we have implemented the visual modeling and the transformation tools which includes the activities of designing systems, translating design to ADL notation, generating behavior models, and generating real-time models. Also, we have successfully adopted UPPAAL and Times tool in our framework.

Q13: *How can we make the formalism accessible to software architects* We provide a *visual modeling tool*, developed in our research lab, to design trustworthy component-based systems. The tool provides a graphical user interface that allows the user to select, drag, and drop different elements of our trustworthy component model on a design canvas. The user can specify the structural and contract requirements with no knowledge of the underlying formalism. The tool provides syntactic checking and report any design errors to the user. The tool automatically generates formal system specification in accordance with our TADL language. Zhou Yun [Yun09] has implemented the visual modeling tool.

Q14: *How can we specify the behavior of trustworthy components easily?* We use model transformation techniques to automate the process of generating behavior specification and real-time models. We provide a *transformation tool*, developed in our

research lab, to automatically analyze the TADL specification, generated by the *visual modeling tool*, and generate a behavior model as extended timed automata. The behavior model contains functional and non-functional specifications including the requirements of safety, security, reliability, and availability. The transformation tool produces extended timed automata that conforms to the specification language of *UPPAAL* and *Times* model checkers. Naseem Ibrahim [Ibr08] has implemented this transformation tool.

Q15: *How can we verify the properties of trustworthiness?* We use model checking to verify the trustworthiness properties. Safety, security, reliability, and availability requirements are verified using *UPPAAL* model checker. Real-time requirements, such as schedule analysis, are verified using the *Times* tool. The extended timed automata that is produced by the transformation tool is input to UPPAAL and Times to conduct the formal verification.

Limitations and Future work: The current implementation of the framework includes only the design time tools. We need to build a repository to host components definitions and the results of their testing and verification.

3.3 Summary

In this section, we presented our research methodology for achieving a formal software engineering approach for developing trustworthy systems. We provided our solution to the research problems and their corresponding research questions. We provided our solutions based on our conducted research and the results we found. We are not aware of any other approach to answer these problems. In the rest of the thesis the technical details of the proposed solutions are given.

Chapter 4

Trustworthy Component Model

This chapter introduces a formal model of a trustworthy component. Informally, a component has a structure and behavior. The structure of a component is shown in Figure 9. This component structure is different from other models proposed earlier [LW07]. The novel contribution of the component model is the formal way in which safety contract and security mechanism are combined and in which reliability and availability are defined and verified. The rationale for the new model arises from the essential need for defining non-functional requirements, composing them in such a way that preserves these non-functional requirements, and verifying them. Therefore, the new model is designed in such a way that it collectively addresses the requirements of CBD and trustworthiness. Also, the model is designed in a formal way that allows automated analysis and verification of trustworthiness. The formal specification of reliability and availability is presented in Chapter 7. The remaining elements that make up the component model are formalized in the following sections.

4.1 Event and Data Parameters

Components provide and request services through public interfaces. Component technology provides a means to define, implement, deploy, maintain, and reuse related services in one entity. Service definitions are used to classify, search, and discover components. Therefore, it is important to model services independently and include their definitions in components. We model a service as a *function* mapping an *input request* to an *output response*. Requests and responses are parameterized events where a parameter is either a *data*

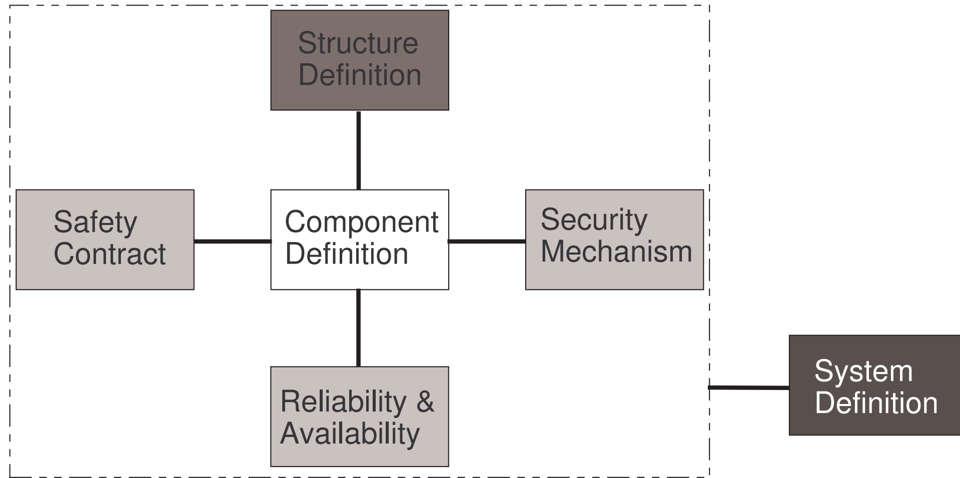


Figure 9: Structure of Trustworthy Component

parameter or an *attribute*. A parameter has a *type*, *value*, and a set of *constraints* defined over the value.

A component interacts with other components through *stimulus* and *response*. Let Σ denote a finite nonempty set of events. An event in Σ denotes either a stimulus, *request for service*, or a response, *service provision*, but not both. A request for service is an *input event* representing an information flow from outside the component to the inside. On the other hand, a service provision is an *output event* representing an information flow from inside the component to the outside. Input and output events are *external* events. Internal processing inside the component is done using *internal* events. Output events are divided into two types: *output response* and *output request*. An output response models the actual response to a request. An output request models an event sent to request additional processing from outside the component. Therefore, Σ is divided into a set of input events Σ_{input}, a set of output events Σ_{output}, and a set of internal events $\Sigma_{internal}$ where $\Sigma_{output} = \Sigma_{response} \cup \Sigma_{OutRequest}$. Formally, $\Sigma = \Sigma_{input} \cup \Sigma_{output} \cup \Sigma_{internal}$, $\Sigma_{input} \cap \Sigma_{output} \cap \Sigma_{internal} = \emptyset$. An event representation has 4 tuples in it; one tuple denotes the set of data parameters, one tuple denotes the set of attributes, one tuple denotes the set of constraints, and one tuple denotes the event flow.

Constraints: A constraint is a logical expressions, defined over data parameters and attributes, which is an invariant associated with an event. Constraints are used to define the valid values of event's parameters. Let $\mathcal{C}$ denote the set of all logical expressions. A logical

expression $\chi \in \mathcal{C}$ is defined using first order predicate logic (FOPL).

We use the following notation in all subsequent definitions:

- $\mathbb{T}$ denotes the set of all data types.
- $\mathcal{D} \in \mathbb{T}$ means $\mathcal{D}$ is a data type such as $\mathbb{N}$.
- $\nu : \mathcal{D}$ denotes that v is either a constant or variable of type $\mathcal{D}$.
- χ_ν is a constraint on ν. If ν is a constant then χ_ν is true.

Data parameters: A data parameter is information carried by an event. The definition of a data parameter includes name, data type, and value. In principle, abstract data types can be data parameter types; however, we restrict only to simple data types such as integer, char, boolean, and float and arrays defined over them. The set of data parameters is $\Lambda = \{\lambda = (\mathcal{D}, \nu, \chi_\nu) \mid \mathcal{D} \in \mathbb{T},\ \nu : \mathcal{D},\ \chi_\nu \in \mathcal{C}\}$. Modeling data parameters as architectural elements has three important implications. These are:

- It allows modeling different types of simple and complex data communicated at the interfaces of a component, which results in a rich communication specification.
- It provides a mechanism for securing the information passed through the interfaces of a component. Security is essential for both the services and the data communicated at the interfaces. Therefore, explicit modeling of data parameters enables designing information security at architectural level.
- It enables rich specification of safety contracts by regulating reactions of the component based on values of data parameters.

Attributes: An attribute qualifies the semantic content associated with an element in the component model. A quality attribute has a type, which can be either simple or complex. As an example, attributes can be used to qualify real-time information, such as priority and worst-case execution time. These attributes are necessary for performing real-time schedule analysis. The set of attributes is $\mathcal{A} = \{\alpha = (\mathcal{D}, \nu_\alpha, \chi_{\alpha\nu}) \mid \mathcal{D} \in \mathbb{T},\ \nu_\alpha : \mathcal{D},\ \chi_{\alpha\nu} \in \mathcal{C}\}$.

Event is formalized in Definition 1.

Definition 1 *Let* $SY = \{?, !, !!, \epsilon\}$. *The set of events is formally defined by* $\Sigma = \{e = (\Lambda_e, \mathcal{A}_e, \chi_e, \varrho_e) \mid \Lambda_e \subseteq \Lambda,\ \mathcal{A}_e \subseteq \mathcal{A},\ \chi_e \in \mathcal{C}, \varrho_e \in\ SY\}$ *where*

$$\varrho_e = \begin{cases} ?, & \textit{if } e \textit{ is an input event;} \\ !, & \textit{if } e \textit{ is an output response;} \\ !!, & \textit{if } e \textit{ is an output request;} \\ \epsilon, & \textit{if } e \textit{ is an internal event.} \end{cases}$$

Example 1 *A sensor sends the current temperature in the room, where it is installed, to a centralized controlling unit. The upper and lower bounds on sensor reading constrain the temperature value sent to the controller. In order to formalize this event, which is a service request received by the controller, we let* x *be the current temperature and [-25,40] be the range for sensory observations. The behavior of the sensor's output is either periodic, sporadic, or controlled.*

Let $\Lambda = \{\lambda_{temp} = (\mathbb{N}, x, x < 40 \wedge x > -25)\}$.
Let $Behavior = \{Periodic,\ Controlled,\ Sporadic\}$ *be the set of types to model the message output by the sensor. We define* $Periodic \equiv \mathbb{N}$, $Sporadic \equiv [Low, High]$ *where* $Low : \mathbb{N},\ High : \mathbb{N},\ Low < High$, *and* $Controlled \equiv \{0, 1\}$. *The set of attributes is* $\mathcal{A} = \{\ \alpha_{behavior}, \alpha_{priority},\ \alpha_{WCET}\}$ *where* $\alpha_{behavior} = (Periodic, 10, true)$, $\alpha_{priority} = (\mathbb{N}, y, true)$, *and* $\alpha_{WCET} = (\mathbb{N}, z, true)$. *The values* y *of worst-case execution time (WCET) and* z *of priority are assigned by the designers of the system. No additional system constraint is imposed on the event. Therefore, the event specification is the tuple* $(\Lambda, \mathcal{A}, true, ?)$.

4.2 Services and Contracts

The behavior of a component must be predictable, deterministic. When a request for service arrives at an interface, the component must react by providing a response. When a component receives a request for service at an interface, which will be discussed later, it reacts by doing one of the following actions:

- performing internal processing and becoming silent, a log monitoring component for example;
- performing an internal processing and sending a response to the calling component, a query analyzer component for example; or

- performing an internal processing, sending an output request to another component to get more information or perform further processing, and finally, sending the response to the initial caller.

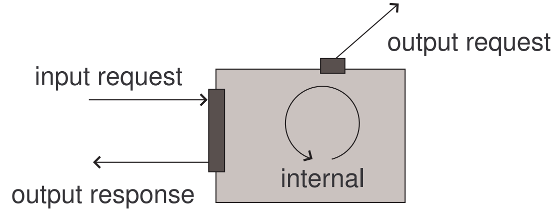


Figure 10: The different types of events

Therefore, *response events* are either *internal, output response,* or *output request* events. On the other hand, the request for service, *stimulus*, can be an *external input* request or an *internal* event. Having internal events as stimulus enables modeling *periodic* events that stimulate the component to perform monitoring or self control activities. Therefore, stimulus events are either internal or input requests. Figure 10 depicts the different kinds of events occurring at a component. *A service is defined as a function that maps stimulus to response with the help of data and time constraints* as described bellow.

Data Constraint: In general, a stimulus may have more than one possible response. *Data constraints* are used to avoid this nondeterminism. For each possible response, a data constraint is defined such that only one data constraint can be true at an instant. Therefore, only one response will be selected. A data constraint is a special type of constraint that is used to decide whether or not a specific response for a requested service should be sent. The decision is based on evaluating a logical expression defined over the values of the data parameters associated with the stimulus and the attributes of the stimulus and the component. The response is given only if the constraint evaluates to true. The set of data constraints is $\Omega = \{\omega = (\mathcal{A}_\omega,\ s,\ r,\ \chi_\omega) \mid \mathcal{A}_\omega \subseteq \mathcal{A}_s \cup \mathcal{A}_\emptyset,\ s \in \Sigma_{stimulus},\ r \in \Sigma_{response}, \chi_\omega \in \mathcal{C}\}$ where $\mathcal{A}_s$ is the set of attributes in the stimulus and $\mathcal{A}_\emptyset$ is the set of attributes of the component, which will be defined later in this chapter. If s has n responses than there must be n number of mutually exclusive data constraints defined for the responses of s in Ω. This ensures that the responses of s are mutually exclusive which ensures determinism.

We define $dc : \Sigma_{stimulus} \rightarrow \mathbb{P}\,\Omega$ which extracts the data constraints for a given stimulus. $dc(s) = \Omega_s \subseteq \Omega$ such that $\Omega_s = \{\omega_i = (\mathcal{A}_{\omega_i}, s_i, r_i, \chi_i \mid \omega_i \in \Omega \wedge s_i = s)\}$.

Time Constraint: The correct behavior of real-time systems does not depend only on the provided services but also on the time at which the services are provided. Therefore, service provision can be governed by *time constraints*. A time constraint specifies the maximum amount of time allowed to elapse between the time of receiving a stimulus and the time of sending the response. This is an essential requirement for safety critical systems where timeliness is a critical factor in defining safety. The set of time constraints is $\Gamma = \{\gamma = (\mathcal{A}_\gamma,\ \chi_\gamma,\ s,\ r,\ \delta) \mid \mathcal{A}_\gamma \subseteq \mathcal{A},\ \chi_\gamma \in \mathcal{C},\ s \in \Sigma_{stimulus},\ r \in \Sigma_{response},\ \delta : \mathbb{N}\}$ where δ defines the *maximum safe time*, the maximum safe time interval between the occurrence of a stimulus and the occurrence of its corresponding response. If $\chi_\gamma(\mathcal{A}_\gamma)$ evaluates to true then the maximum safe time is enforced on the response. However, if $\chi_\gamma(\mathcal{A}_\gamma)$ evaluates to false, then the maximum safe time need not be enforced.

A response event can be accompanied by executing several update statements that set the values of local attributes. Also, a response can be accompanied by several actions, which are internal or external events that occur after a response. Service is formalized in Definition 2.

Definition 2 *Let Γ be a finite set of time constraints, Ω be a finite set of data constraints, $\Sigma_{stimulus} = \Sigma_{internal} \cup \Sigma_{input}$, and $\Sigma_{response} = \Sigma_{internal} \cup \Sigma_{output}$. A service is defined as a function $\Theta : \Sigma_{stimulus} \times \Omega \times \Gamma \times \mathbb{N} \rightarrow \Sigma_{response} \times \mathbb{P}\mathcal{U} \times \mathcal{S} \times \mathbb{N}$ where $\mathcal{U}$ is a set of update statements defined using the function $assign : \mathcal{D} \rightarrow \mathcal{A}$ such that $assign(\alpha) = \nu$ assigns a value ν from the domain $\mathcal{D}$ to an attribute $\alpha \in \mathcal{A}_r \cup \mathcal{A}_\emptyset$ where $\mathcal{A}_r$ is the set of attributes of the response and $\mathcal{A}_\emptyset$ is the set of attributes of the component, and $\mathcal{S} \subset \Sigma_{output} \cup \Sigma_{internal}$ is a set of actions triggered by the service.*

The precondition for the function is defined as follows:
Let $s \in \Sigma_{stimulus},\ s = \{\Lambda_s,\ \mathcal{A}_s,\ \chi_s,\ ?\}$ such that:

- $\Lambda_s \subseteq \Lambda,\ \Lambda_s = \{\lambda_{s_i} = (\mathcal{D}_i, \nu_i, \chi_i) \mid \mathcal{D}_i \in \mathbb{T},\ \nu_i : \mathcal{D}_i,\ \chi_i \in \mathcal{C}\}$
- $\mathcal{A}_s \subseteq \mathcal{A},\ \mathcal{A}_s = \{\alpha_{s_i} = (\mathcal{D}'_i, \nu'_i, \chi'_i) \mid \mathcal{D}'_i \in \mathbb{T},\ \nu'_i : \mathcal{D}'_i,\ \chi'_i \in \mathcal{C}\}$

The stimulus occurs if the following conditions are satisfied:

- *the constraints defined for the data parameters are satisfied. That is, $\bigwedge_{(\mathcal{D}_i,\nu_i,\chi_i)} \chi_i(\nu_i) = true$,*
- *the constraints defined for the attributes are satisfied. That is, $\bigwedge_{(\mathcal{D}'_i,\nu'_i,\chi'_i)} \chi'_i(\nu'_i) = true$, and*

- *the additional constraint defined in the stimulus specification is satisfied. That is,* $\chi_s = true$

$\Theta(s, \omega_{s_r}, \gamma_{s_r}, t_1) = (r, \mathbb{R}, t_2)$ *where* t_1 *is the time at which the stimulus occurs,* t_2 *is the time at which the response occurs, and:*

1. r *is extracted from the tuple* $\omega_{s_r} = (\mathcal{A}_{\omega_{s_r}}, s, r, \chi_i) \in \Omega_s$ *such that* χ_i *evaluates to true,*

2. *select* $\gamma_{s_r} = (\mathcal{A}, \chi_{\gamma_{s_r}}, s, r, \delta) \in \Gamma$

3. *The post condition of the function is:*

   a. $r = (\Lambda_r, \mathcal{A}_r, \chi_r, SY_r)$, *where* $SY_r \in \{!, !!, \epsilon\}$,

   b. $\Lambda_r \subseteq \Lambda$, $\mathcal{A}_r \subseteq \mathcal{A}$, $\mathbb{R} \subseteq \Sigma_{output} \cup \Sigma_{internal}$, *and*

   c. $|t_2 - t_1| \leq \delta$.

For convenience, we define the function $\phi : \Sigma_{stimulus} \rightarrow \mathbb{P}\,\Sigma_{response}$ *such that* $\phi(s) \neq \emptyset$. *This function maps each stimulus to the set of responses associated with it.*

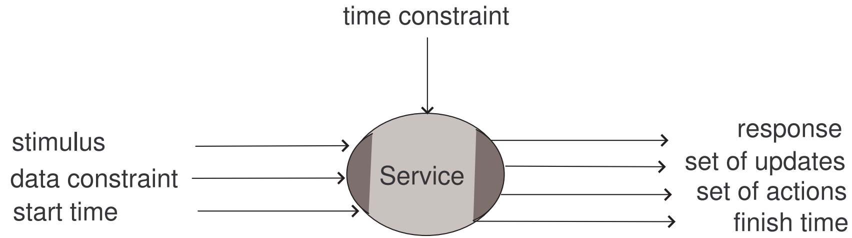


Figure 11: Service

Figure 11 Depicts a service.

Safety Property: Safety properties can be defined at a component level to enforce safe behavior. A component safety property is an invariant over the behavior of a component. The behavior can be defined using timed automata which will be discussed later. A safety property is regarded as a special type of constraint over the services provided by the component. A *contract* defines a nonempty set of services and safety properties. The rationale behind specifying the contract outside of the component type definition is to allow reuse of

a contract for other components that provide similar services and to enable reconfiguration of its specification. The reconfiguration updates maximum safe time, data constraints, and services for different system configurations and deployment plans.

Definition 3 *We define a set of safety properties* $\mathcal{P} = \{p = (\Sigma_p,\ \chi) \mid \Sigma_p \subseteq \Sigma, \chi \in \mathcal{C}\}$*. A contract* Ξ *is defined as a tuple* $\Xi = (\Theta,\ \Omega,\ \Gamma,\ \mathcal{P})$*.*

The service definition must satisfy the following conditions:

- $\forall \gamma_1, \gamma_2 \in \Gamma,\ \gamma_1 = (\mathcal{A}_1,\ \chi_1,\ s_1,\ r_1,\ \delta_1) \wedge \gamma_2 = (\mathcal{A}_2,\ \chi_2,\ s_2,\ r_2,\ \delta_2) \rightarrow s_1 \neq s_2 \vee r_1 \neq r_2$. That is, it is not possible to define two different time constraints for the same stimulus-response (service).

- $\forall \omega_1, \omega_2 \in \Omega,\ \omega_1 = (\mathcal{A}_1,\ s_1,\ r_1,\ \chi_1) \wedge \omega_2 = (\mathcal{A}_2,\ s_2,\ r_2,\ \chi_2) \rightarrow s_1 \neq s_2 \vee r_1 \neq r_2$. That is, it is not possible to define two different data constraints for the same stimulus-response (service).

- $\forall s \in \Sigma_{stimulus}, |\phi(s)| > 1 \rightarrow \exists \Omega_s \subseteq \Omega \bullet \forall \omega_1, \omega_2 \in \Omega_s,\ \omega_1 = (\mathcal{A}_1,\ s,\ r_1,\ \chi_1)$, $\omega_2 = (\mathcal{A}_2,\ s,\ r_2,\ \chi_2),\ \omega_1 \neq \omega_2 \wedge \chi_1 \oplus \chi_2$, i.e. if a stimulus has multiple possible responses then we must define a service for every stimulus-response relation and assign it a different data constraint. The data constraints must be mutually exclusive.

Example 2 *Assume a real-time Continuous Glucose Monitoring system which consists of (1) a sensor inserted subcutaneously in the abdominal area to measure interstitial fluid glucose levels, and (2) a small mobile monitoring device. The sensor takes glucose readings regularly and relays it to the monitoring device. The monitoring device has an attribute which specifies an alarm threshold glucose level. If the current glucose reading is above the defined threshold then the monitoring device should trigger the alarm within 5 units of time to alert the patient to potentially dangerous glucose level and display the level. Otherwise, the monitoring device should just display the current glucose level. The following defines only the service definition part of this example for the monitoring device.*

Let $\Lambda = \{\lambda_{glucose}\}$ *where* $\lambda_{glucose} = (\mathbb{N},\ \nu,\ true)$ *is the data parameter defining the current glucose level.*
Let $\mathcal{A}_1 = \{threshold\}$ *where* $threshold = (\mathbb{N},\ \nu,\ true)$*.*
Let $\Sigma = \{Level,\ Alarm,\ Display\}$ *be the set of events where* $\Sigma_{input} = \{Level\}$ *and* $\Sigma_{Internal} = \{Alarm,\ Display\}$ *such that:* $Level = (\Lambda,\ \emptyset,\ true,\ ?)$*,*
$Alarm = (\Lambda,\ \mathcal{A}_1,\ true, \epsilon)$*, and* $Display = (\Lambda,\ \mathcal{A}_1,\ true, \epsilon)$*.*

Let $\Gamma = \{\gamma\}$ *where* $\gamma = (\emptyset,\ true,\ Level?,\ Alarm,\ 5)$.
Let $\Omega = \{\omega_{alarm}, \omega_{normal}\}$ *where* $\omega_{alarm} = (\emptyset, Level?, Alarm, \lambda_{glucose} \geq threshold)$ *and* $\omega_{normal} = (\emptyset, Level?, Display, \lambda_{glucose} < threshold)$. *The service definitions are:*
$\Theta(Level?, \omega_{alarm}, \gamma, t_1) = (Alarm,\ \emptyset,\ \{Display\}, t_2)$ *and*
$\Theta(Level?, \omega_{normal}, \gamma, t_1) = (Display,\ \emptyset,\ \emptyset, t_2)$.

4.3 Component Architecture

The structural description of a component includes definitions of *interface types, connector role types, connector types, architecture types,* and *component types*.

Interface Types: Interfaces are access points to the services provided and requested by components. An *interface type* is an enumerated type whose elements are events from Σ. An *interface* is an instance of an interface type, it inherits the events listed in the type definition. We define Π as the set of interface types where each interface type π is a triple $\pi = (\mathcal{A}_\pi, \chi_\pi, \sigma)$ such that $\mathcal{A}_\pi \subseteq \mathcal{A}$ is a set of attributes defined for the interface type, $\chi_\pi \in \mathcal{C}$ is a constraint over the events and attributes of the interface type, and $\sigma : \Pi \rightarrow \mathbb{P}\Sigma$ is a function that associates a finite non-empty subset of external events to each interface-type such that $\forall \pi_1, \pi_2 \in \Pi,\ \pi_1 \neq \pi_2,\ \sigma(\pi_1) \cap \sigma(\pi_2) = \emptyset$. Two interface types π_1 and π_2 are *compatible* if and only if for every event $s \in \pi_1$ there exists exactly one event $\overline{s} \in \pi_2$ such that $\varrho(s) =!$ and $\varrho(\overline{s}) =?$ are *complementary*. That is, in designing component interactions both s and $\overline{s}$ will be assigned to occur simultaneously at component interfaces of interacting components. We define the predicate $Compatible(\pi_1, \pi_2)$ which is true if and only if π_1 and π_2 are compatible.

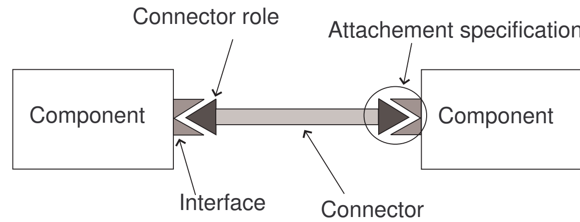


Figure 12: Connecting two components

Connector Type: A connector defines the connectivity between two or more components. A connector type definition includes a non-empty finite set of connector role types in addition to attributes and constraints. A connector role type serves as an interface to a connector. It links a connector to a component interface. Figure 12 presents an *attachment specification* showing how two components can be attached together using a connector, two interfaces, and two connector role types. The attachment specification is inspired by the work in ACME [GMW00]. Abstracting the connector role from the connector specification enables abstracting the *communication method* used in the connector from its access points. Therefore, it is possible to define different communication methods such as RPC, HTTP, and SOAP using the same access points (connector roles). Also, introducing connection points at the ends of a connector can help to reason about the integrity of the communication method by comparing representations of the data before and after the communication. A connector role type is defined as a triple $\rho = (\mathcal{A}_\rho, \chi_\rho, \pi)$ where $\mathcal{A}_\rho \subseteq \mathcal{A}$ and $\chi_\rho \in \mathcal{C}$. A Connector type is defined as a tuple $\mathrm{K} = (\mathcal{A}_k, \chi_k, \mathcal{R}, \mathcal{M})$ where $\mathcal{R}$ is a finite set of connector role types, $\mathcal{A}_k \subseteq \mathcal{A}$ is a set of attributes associated with the connector, and $\chi_k \in \mathcal{C}$ is a constraint that can be used to specify an invariant or restriction that controls whether or not the communication is allowed. The communication method $\mathcal{M}$ specifies the type of communication used by the connector to deliver services. There are a number of common communication styles to choose from [SG96] such as procedure call, message passing, remote procedure calls, etc. The communication is bidirectional. Details of the communication method fall outside the scope of this thesis. The attachment specification is a tuple $(\mathrm{K}, \rho, \pi, CT)$, where $\rho \in \mathcal{R}_\mathrm{K}$ is a connector role type in the connector type K, CT is a component type (definition will follow), and π is an interface type that is defined in both ρ and CT. The attachments are specified outside the connector type definition to make the connector specification independent from how it is used. This abstraction enhances reuse and reconfiguration of connector type specification. A connector $\tilde{\mathrm{K}}$ is an instance of a connector type K. It inherits all the connector roles defined at the connector role type, implement the communication method, and restrict communication to the defined constraints.

Architecture Type: A component can be primitive or composite [SG96]. A composite component is built by assembling existing components and specifying their connectors. An *architecture type* defines the structure of a composite component in which the constituent

components and their internal connections are specified. A component type can have multiple possible architecture types. An architecture type comprises connector types, attributes, constraints, and attachments specifications. An attachment specifies how components are connected. This is specified by linking the interface type of a connector role type with an interface type of a component at the ends of a connector. The attachment specification allows us to define the structural composition of components. We use the notation $\propto = \rho @ \mathrm{K} \bowtie \pi @ CT$, instead of tuple notation, to introduce an attachment specification that links interface type π of the connector role type to its corresponding one in the component type. Each attachment specification defines the connection at one end of a connector. For example, a binary connector which connects two component types requires two attachment specifications each of which specifying the connection point at one end. We use the notation $\propto_1 \boxplus \propto_2$ to introduce a structural composition of two components at an architecture type definition using one connector type. Defining the architecture outside of the component type definition increases reuse and allows reconfiguration of architecture without changing the component definition.

Component Type: A component type definition includes definitions of events, interface types, architecture types, a contract, attributes, and constraints. If no architecture is specified then the component type denotes a primitive component. In a composite component's type definition, the list of interface types that are not attached to connector role types form the external interface types, whereas the attached ones form the internal interface types.

Definition 4 *A Component Type is a tuple $CT = (\Sigma, \Pi, \sigma, \Lambda, \xi, \Xi, \mathcal{A}_c, \mathcal{C}_c, \mathcal{T})$ where Σ is the set of events, Π is the set of interface types through which the events are accessed, σ is a function that associates events to interface types such that $\forall \pi_1, \pi_2 \in \Pi, \sigma(\pi_1) \cap \sigma(\pi_2) = \emptyset$, Λ is the set of data parameters, $\xi : \Sigma \rightarrow \mathbb{P}\, \Lambda$ is a function that associates with each service request or provision a set of data parameters, Ξ is a contract, $\mathcal{A}_c \subseteq \mathcal{A}$ is a set of attributes, $\mathcal{C}_c \subseteq \mathcal{C}$ is a set of constraints, and $\mathcal{T}$ is a set of architecture types describing the possible internal structures of composite component types. If $\mathcal{T} = \emptyset$ then the component is primitive. An architecture type τ is defined as a tuple $\tau = (\mathcal{O}, \mathcal{N}, \mathcal{H}, \mathcal{A}_\tau, \mathcal{C}_\tau)$ where $\mathcal{O}$ is the set of constituent component types, $\mathcal{N}$ is the set of connector types used to connect these component types, $\mathcal{H}$ is the set of attachment specifications used to attach connector types to interface types of component types, $\mathcal{A}_\tau \subseteq \mathcal{A}$ is a set of attributes, and $\mathcal{C}_\tau \subseteq \mathcal{C}$ is a set of constraints.*

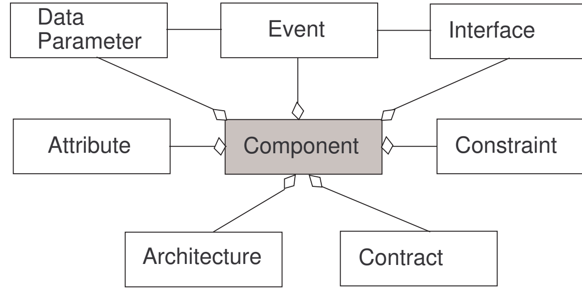


Figure 13: Component Definition

Figure 13 depicts the component definition.

Definition 5 *A component* $\o$ *is an instance of a component type* CT*. The set of its external interface types* Π_I *is created from the set of interface types* Π *of* CT *by specifying for each interface type* π *in* Π *the number of interfaces (#*π*) of type* π *required in* $\o$*. If #*π *=* n*, we let* $\o_\pi = \{\pi_1, \ldots, \pi_n\}$ *denote the interfaces created. A specification of* $\o$ *is* $(\Sigma,\ \Pi_I,\ \sigma,\ \Lambda,\ \xi,\ \Xi,\ \mathcal{A}_\o,\ \mathcal{C}_\o,\ \mathcal{T})$*, where* $\Pi_I = \bigcup_{\pi \,\in\, \Pi} \o_\pi$*. The* σ *function is extended to interfaces:* $\forall \pi_i \in \pi \bullet \sigma(\pi_i) = \sigma(\pi)$*. This means that an event can be provided at multiple instances (interfaces) of the same interface type. Each instantiated interface will be used to provide the event to a specific component using a specific connector (instance of a connector type).*

A component architecture o_τ *from a component type's architecture type* τ *is created by defining (1) a set of components instantiated from the component types, (2)* n *connectors in* $\o_\tau$ *for each connector type in* τ *if* n *interfaces have been created in* $\o$ *corresponding to the interface type(s) in the connector type's role type definitions, and (3) a set of attachments instantiated from the attachment specifications* $\mathcal{H}$ *in* τ *where each attachment is of the form* $\tilde{\propto} = \rho\ @\ \tilde{\mathrm{K}}\ \bowtie p\ @\ \o$ *where* $\tilde{\mathrm{K}}$ *is a connector instantiated from* K*,* p *is an interface instantiated from* π*, and* $\o$ *is a component instantiated from* CT*.*

Component type and architecture type definitions satisfy the following properties:

- $\forall \o : CT\ \bullet\ \o = (\Sigma_\o,\ \Pi_\o,\ \sigma_\o,\ \Lambda_\o,\ \xi_\o,\ \Xi_\o,\ \mathcal{A}_\o,\ \mathcal{C}_\o,\ \mathcal{T}) \rightarrow \forall \pi_1, \pi_2 \in \Pi_\o\ \bullet\ \sigma(\pi_1) \cap \sigma(\pi_2) = \emptyset$ i.e. a service request and a service response events can be defined only once at only one interface type.

- $\forall c : \mathrm{K}\ \bullet\ (c = (\mathcal{A}_c, \chi_c, \mathcal{R}, \mathcal{M})) \rightarrow (\forall \rho_1, \rho_2 \in \mathcal{R} \bullet \rho_1 = (\mathcal{A}_1, \chi_1, \pi_1) \wedge \rho_2 = (\mathcal{A}_2, \chi_2, \pi_2) \rightarrow Compatible(\pi_1, \pi_2))$ i.e. the interface types used for connecting

components must be compatible.

Example 3 *The anti-lock brake system (ABS) is a safety critical system which prevents the wheels from locking while braking. The system is composed of (1) a central electronic control unit (ECU), (2) four wheel speed sensors, one for each wheel, (3) four hydraulic valves to release pressure, and (4) four hydraulic pumps to increase pressure. The sensors continuously sense the speed of the wheels and inform the ECU. The ECU continuously monitors the rotational speed of each wheel. When the ECU detects that a wheel is rotating significantly slower than the others it actuates the corresponding valve to reduce the hydraulic pressure; hence, it reduces the braking force on that wheel. Then the wheel turns faster and if the ECU detects that it is turning significantly faster than the other wheels then it increases the pressure so the wheel slows. The process is repeated many times.*

In this example, we focus only on the architecture part of the system. The architecture of the ABS system is composed of 4 different component types: ECU*,* Sensor*,* Valve*,* Pump*. It also contains 3 different connector types:* ECU-sensor*(*K_s*),* ECU-valve*(*K_v*), and* ECU-pump*(*K_p*), and 3 different interface types:* ISensing*,* IReleasing*,* IPressing*.* ECU *contains all the interface types,* Sensor *contains only* ISensing*,* Valve *contains only* IReleasing*, and* Pump *contains only* IPressing*. The system consists of one instance of* ECU*, 4 instances of* Sensor*, 4 instances of* Valve*, and 4 instances of* Pump*. It also contains 12 connectors where 4 instances of each connector type are created to link the ECU instance component with the other instances of each type. The ECU instance includes 4 interface instances of each interface type so that each of which can be attached to a connector.*

Formally: let $\Pi_e = \{ISensing,\ IReleasing,\ IPressing\}$*,* $\Pi_s = \{ISensing\}$*,* $\Pi_v = \{IReleasing\}$*, and* $\Pi_p = \{IPressing\}$*.*
Let $ECU = ((\Sigma_c,\ \Pi_c,\ \sigma_c,\ \Lambda_c,\ \xi_c,\ \Xi_c,\ \emptyset,\ \emptyset,\ \emptyset)$*,*
$Sensor = ((\Sigma_s,\ \Pi_s,\ \sigma_s,\ \Lambda_s,\ \xi_s,\ \Xi_s,\ \emptyset,\ \emptyset,\ \emptyset)$*,*
$Valve = ((\Sigma_v,\ \Pi_v,\ \sigma_v,\ \Lambda_v,\ \xi_v,\ \Xi_v,\ \emptyset,\ \emptyset,\ \emptyset)$*, and*
$Pump = ((\Sigma_p,\ \Pi_p,\ \sigma_p,\ \Lambda_p,\ \xi_p,\ \Xi_p,\ \emptyset,\ \emptyset,\ \emptyset)$*.*

The connector role types are $\rho_{s1} = (\emptyset,\ true, ISensing)$*,* $\rho_{s2} = (\emptyset,\ true, ISensing)$*,* $\rho_{v1} = (\emptyset,\ true, IReleasing)$*,* $\rho_{v2} = (\emptyset,\ true, IReleasing)$*,* $\rho_{p1} = (\emptyset,\ true, IPressing)$*, and* $\rho_{p2} = (\emptyset,\ true, IPressing)$*. The connector types are* $\mathrm{K}_s = (\emptyset,\ true,\ \{\rho_{s1}, \rho_{s2}\},\ \mathcal{M})$*,* $\mathrm{K}_v = (\emptyset,\ true,\ \{\rho_{v1}, \rho_{v2}\},\ \mathcal{M})$*, and* $\mathrm{K}_p = (\emptyset,\ true,\ \{\rho_{p1}, \rho_{p2}\},\ \mathcal{M})$*. Figure 14 shows the structure of the ABS.*

The following attachment specifications are used to link the component types in the

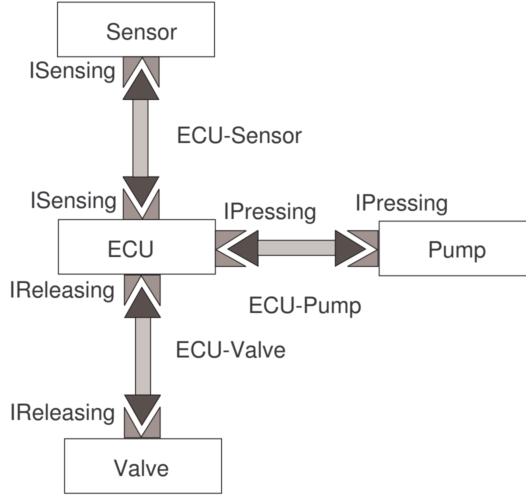


Figure 14: Structure of the ABS

system architecture:

$\propto_1 = \rho_{s1}$ @ $\mathrm{K}_s \bowtie ISensing$ @ ECU,

$\propto_2 = \rho_{s2}$ @ $\mathrm{K}_s \bowtie ISensing$ @ $Sensor$,

$\propto_1 \boxplus \propto_2$,

$\propto_3 = \rho_{v1}$ @ $\mathrm{K}_v \bowtie IReleasing$ @ ECU,

$\propto_4 = \rho_{v2}$ @ $\mathrm{K}_v \bowtie IReleasing$ @ $Valve$,

$\propto_3 \boxplus \propto_4$,

$\propto_5 = \rho_{p1}$ @ $\mathrm{K}_p \bowtie IPressing$ @ ECU,

$\propto_6 = \rho_{p2}$ @ $\mathrm{K}_p \bowtie IPressing$ @ $Pump$,

$\propto_5 \boxplus \propto_6$.

The architecture type of the system is $\tau = (\{ECU, Sensor, Valve, Pump\}, \{\mathrm{K}_s, \mathrm{K}_v, \mathrm{K}_p\}, \{\propto_1, \propto_2, \propto_3, \propto_4, \propto_5, \propto_6\}, \emptyset, \emptyset)$.

The following components are created:

1. An instance of ECU*:* $\varnothing_E = (\Sigma_c, \{ISensing_1, ISensing_2, ISensing_3, ISensing_4, IReleasing_1, IReleasing_2, IReleasing_3, IReleasing_4, IPressing_1, IPressing_2, IPressing_3, IPressing_4\}, \sigma_c, \Xi_c, \emptyset, \emptyset, \emptyset)$,

2. 4 instances of $Sensor$*:* $\varnothing_{s1} = ((\Sigma_s, \{ISensing_{s1}\}, \sigma_s, \Xi_s, \emptyset, \emptyset, \emptyset)$,

$\varnothing_{s2} = ((\Sigma_s, \{ISensing_{s2}\}, \sigma_s, \Xi_s, \emptyset, \emptyset, \emptyset)$,

$\varnothing_{v3} = ((\Sigma_s, \{ISensing_{s3}\}, \sigma_s, \Xi_s, \emptyset, \emptyset, \emptyset)$,

and $\varnothing_{s4} = ((\Sigma_s, \{ISensing_{s4}\}, \sigma_s, \Xi_s, \emptyset, \emptyset, \emptyset)$,
3. 4 instances of $Valve$*:* $\varnothing_{v1} = ((\Sigma_v, \{IReleasing_{v1}\}, \sigma_v, \Xi_v, \emptyset, \emptyset, \emptyset)$,
$\varnothing_{v2} = ((\Sigma_v, \{IReleasing_{v2}\}, \sigma_v, \Xi_v, \emptyset, \emptyset, \emptyset)$,
$\varnothing_{v3} = ((\Sigma_v, \{IReleasing_{v3}\}, \sigma_v, \Xi_v, \emptyset, \emptyset, \emptyset)$,
and $\varnothing_{v4} = ((\Sigma_v, \{IReleasing_{v4}\}, \sigma_v, \Xi_v, \emptyset, \emptyset, \emptyset)$,
4. 4 instances of $Pump$*:* $\varnothing_{p1} = ((\Sigma_p, \{IPressing_{p1}\}, \sigma_p, \Xi_p, \emptyset, \emptyset, \emptyset)$,
$\varnothing_{p2} = ((\Sigma_p, \{IPressing_{p2}\}, \sigma_p, \Xi_p, \emptyset, \emptyset, \emptyset)$,
$\varnothing_{p3} = ((\Sigma_p, \{IPressing_{p3}\}, \sigma_p, \Xi_p, \emptyset, \emptyset, \emptyset)$,
and $\varnothing_{p4} = ((\Sigma_p, \{IPressing_{p4}\}, \sigma_p, \Xi_p, \emptyset, \emptyset, \emptyset)$.

Attachments are created to link component instances. For example,
$\tilde{\propto}_1 = \rho_{s1} @ \tilde{\mathrm{K}}_{s1} \bowtie ISensing_1 @ \varnothing_E$,
$\tilde{\propto}_2 = \rho_{s2} @ \tilde{\mathrm{K}}_{s1} \bowtie ISensing_{s1} @ \varnothing_{s1}$,
$\tilde{\propto}_1 \boxplus \tilde{\propto}_2$
is created to link the component $\varnothing_E$ *with the component* $\varnothing_{s1}$ *using the connector* $\tilde{\mathrm{K}}_{s1}$. *Other attachment specifications are created in the same manner to link all components.*

The architecture of the system is $(\{\varnothing_E, \varnothing_{s1} \ldots \varnothing_{s4}, \varnothing_{v1} \ldots \varnothing_{v4}, \varnothing_{p1} \ldots \varnothing_{p4}\}, \{\tilde{\mathrm{K}}_{s1} \ldots \tilde{\mathrm{K}}_{s4}, \tilde{\mathrm{K}}_{v1} \ldots \tilde{\mathrm{K}}_{v4}, \tilde{\mathrm{K}}_{p1} \ldots \tilde{\mathrm{K}}_{p4}\}, \{\tilde{\propto}_1 \ldots \tilde{\propto}_{24}\})$. *Figure 15 depicts the component instances.*

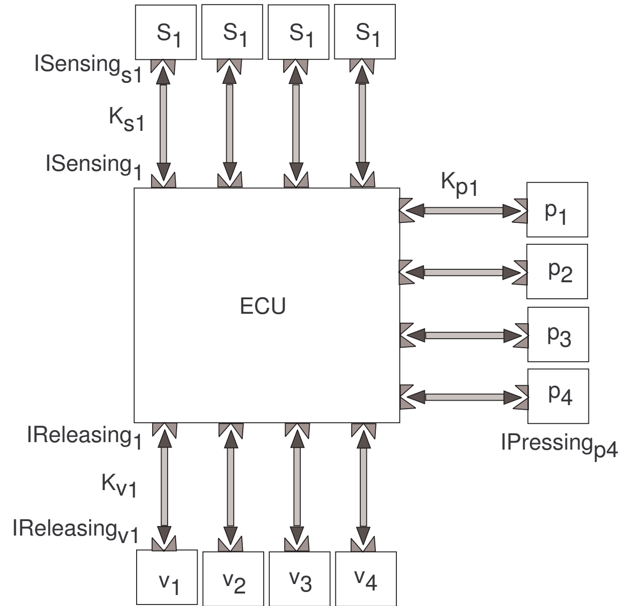


Figure 15: ABS component instances

4.4 Security Mechanism

There is a general consensus [ALRL04] that security is a composite concept that comprises *confidentiality*, ”the prevention of any unauthorise discloser of information”, *integrity*, ”prevention of the unauthorized amendment or deletion of information”, and *availability*, ”the prevention of the unauthorized withholding of information”. This section focuses on confidentiality.

The component type definition includes a *user* attribute. This attribute is set at component’s instantiation time with a value that denotes the identity of the client on whose behalf the component executes. The value of user identity is assigned from a domain of user identities defined at system level. In computer security [Bis03] the *identity* of the entity executing a process is the basis for assigning and checking security access rights. We assume a list of all possible identities defined at the system level. In our discussion, the user identity, henceforth called user, is associated with the component at its instantiation time. All access control to system resources assume that the association is correct. Verifying the correctness of the identity and describing how it is associated to components falls outside the scope of this thesis.

The security mechanism is based on role-based security access control (RBAC). The mechanism restricts access of services and data parameters to authorized users only. In [AM07a] we defined security property in terms of *service security* and *data security*.

- **Service security** states that: (1) for every request received at the interfaces of a component, the request should be received from a user who has permission to request the service, and (2) for every response sent by the component, the user who will receive the response should have permission to receive it.

- **Data security** states that: (1) for every request received, for every data parameter in the request, the user sending the request should have permission to access the data parameter, and (2) for every response sent, for every data parameter associated with the response, the user receiving the response should have permission to access the data parameter.

If a user does not have a permission to send a request then the request will be ignored. Also, if a user does not have a permission to receive a response, the response will not be sent. On the same manner, if a user does not have a permission to access a data parameter, the data parameter value is set to *null* value.

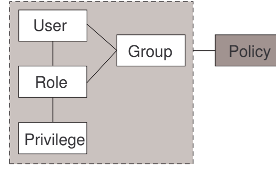


Figure 16: Role-based Access Control

The main concepts in RBAC are user, *group*, *role*, and *privilege*. Figure 16 depicts the elements of RBAC and their relations. A group defines a set of related users. A user can be individual or belong to one or more groups. A role defines a security responsibility that a user or a group of users can take in the system. A privilege defines a permission to access a service or a data parameter. A role comprises many privileges. A privilege can be assigned to many roles. The functions *Group-User-Assignment*, *User-Roles-Assignment*, and *Group-Roles-Assignment* are used to assign users to groups, roles to users, and roles to groups accordingly.

There are two types of privileges: service privilege and data parameter privilege. A service privilege defines an access right for a service. Hence, it is associated with services and roles using the function *Role-Service-Assignment*. A data parameter privilege defines an access right for a data parameter. Therefore, it is associated with data parameters and roles using the function *Role-Data-Assignment*.

Definition 6 *Let User, Role, Group, and $\mathsf{Privilege}$ be defined as finite sets of users, roles, groups, and privileges respectively. The following functions are used to define the RBAC:*

- *Group-User-Assignment: $GUA : \mathsf{Group} \rightarrow \mathbb{P}\ \mathsf{User}$ assigns for a group $g \in \mathsf{Group}$ the users $GUA(g) \in \mathbb{P}\ \mathsf{User}$. A user may belong to more than one group. The function $UG : \mathsf{User} \rightarrow \mathbb{P}\ \mathsf{Group}$ gives for each user $u \in \mathsf{User}$ the set of groups $UG(u) \in \mathbb{P}\ \mathsf{Group}$ that he belongs to.*

- *User-Role-Assignment: $URA : \mathsf{User} \rightarrow \mathbb{P}\ \mathsf{Role}$ assigns for a user $u \in \mathsf{User}$ the roles $URA(u) \in \mathbb{P}\ \mathsf{Role}$. The function $RU : \mathsf{Role} \rightarrow \mathbb{P}\ \mathsf{User}$ gives for each role $r \in \mathsf{Role}$ the set of users $RU(r) \in \mathbb{P}\ \mathsf{User}$ that has r.*

- *Group-Role-Assignment: $GRA : \mathsf{Group} \rightarrow \mathbb{P}\ \mathsf{Role}$ assigns for a group $g \in \mathsf{Group}$ the roles $GRA(g) \in \mathbb{P}\ \mathsf{Role}$. All users of a group are equally assigned the same set*

of roles. The function RG : $\textsf{\textit{Role}} \to \mathbb{P}$ $\textsf{\textit{Group}}$ *gives for each role* $r \in \textsf{\textit{Role}}$ *the set of groups* $RG(r) \in \mathbb{P}$ $\textsf{\textit{Group}}$ *that has* r.

- *Role-Service-Assignment:* RSA : $\textsf{\textit{Role}} \times \Sigma \to \mathbb{P}$ $\textsf{\textit{Privilege}}$ *assigns for a role* $r \in \textsf{\textit{Role}}$ *and a service* $s \in \Sigma$ *the set of privileges* $RSA(r, s) \in \mathbb{P}$ $\textsf{\textit{Privilege}}$. *A service access control matrix is* $SAC = \mathbb{P}(\textsf{\textit{Role}}, \Sigma, \mathbb{P}(\textsf{\textit{Privilege}}))$.

- *Role-Data-Assignment:* RDA : $\textsf{\textit{Role}} \times \Lambda \to \mathbb{P}$ $\textsf{\textit{Privilege}}$ *assigns for a role* $r \in \textsf{\textit{Role}}$ *and a data parameter* λ *the set of privileges* $RDA(r, \lambda) \in \mathbb{P}$ $\textsf{\textit{Privilege}}$. *A data access control matrix is* $DAC = \mathbb{P}(\textsf{\textit{Role}}, \Sigma, \mathbb{P}(\textsf{\textit{Privilege}}))$.

- *Functions:*

  - US : $\textsf{\textit{User}} \times \Sigma \to boolean$ *returns true if a user* u *has a privilege to access a service* s *i.e.* $US(u, s) \to \exists r \in \textsf{\textit{Role}} \bullet r \in URA(u) \wedge RSA(r, s) \neq \emptyset$.

  - USP : $\textsf{\textit{User}} \times \Sigma \times \textsf{\textit{Privilege}} \to boolean$ *returns true if a user* u *has privilege* v *on service* s. *The function* UD : $\textsf{\textit{User}} \times \Lambda \to boolean$ *returns true if a user* u *has a privilege to access a data parameter* λ *i.e.* $UD(u, \lambda) \to \exists r \in \textsf{\textit{Role}} \bullet r \in URA(u) \wedge RDA(r, \lambda) \neq \emptyset$.

  - UDP : $\textsf{\textit{User}} \times \Lambda \times \textsf{\textit{Privilege}} \to boolean$ *checks whether or not a user has a specific privilege on a data parameter.*

  - GU : $\textsf{\textit{User}} \times \textsf{\textit{Group}} \to boolean$ *returns true if a user* u *is part of a group* g *i.e.* $g \in UG(u)$.

  - UR : $\textsf{\textit{User}} \times \textsf{\textit{Role}} \to boolean$ *returns true if a user* u *has role* r *i.e.* $r \in RU(u)$.

It is possible to extend the protected data to include not only the data parameters but also the local attributes of a component. This enables protecting the inner state variables of a component from any unauthorized change.

The tuple $(\textsf{User}, \textsf{Group}, \textsf{Role}, \textsf{Privilege}, SAC, DAC)$ defines the state of the security mechanism at any instance. The state changes with time as different security specifications are modified, for example when users are assigned or denied roles or when the privileges associated with roles are changed. We assume that there is a *security officer* component which is responsible for maintaining the state of the security mechanism. Every system that requires security mechanism has this component as part of it. A security policy Υ comprises service security and/or data security requirements. It is defined as an expression,

using first order predicate logic, that involves any conjunctions of the RBAC functions in Definition 6. The result of a security policy expression reflects the state of the security mechanism at the time of evaluating the expression.

A component which has no security restrictions will respond to every stimulus received by it. The introduction of security mechanism will enrich its behavior by forcing (1) an analysis of the stimulus received before processing it internally, and (2) an analysis of the response before sending it. Therefore, the service definition will be extended to include security policies as follows.

Definition 7 *The set of security properties is* $\Psi = \{\psi = (s, x, \Upsilon) \mid s \in \Sigma_{stimulus}, x \in \Sigma_{response} \cup \Lambda_s \cup \mathcal{A}\}$ *where* Λ_s *is the set of data parameters of the stimulus* s *and* Υ *is a security policy defined for the relation between* s *and* x*. There are three possible relations between* s *and* x*: either* $x \in \phi(s)$ *is the response for* s*,* $x \in \Lambda_s$ *is a data parameter associated with* s*, or* $x \in \mathcal{A}$ *is an attribute.* Ψ *is divided into a set of service security* $\Psi_{service}$ *and a set of data security* Ψ_{data} *such that* $\Psi = \Psi_{service} \cup \Psi_{data}$*.*

The service definition is extended to include secure services:
$\Theta : \Sigma_{stimulus} \times \Omega \times \Gamma \times \mathbb{N} \times \Psi_{service} \times \mathbb{P}\Psi_{data} \rightarrow \Sigma_{response} \times \mathbb{P}\mathcal{U} \times \mathcal{S} \times \mathbb{N} \times \mathbb{P}\Psi_{service} \times \mathbb{P}\Psi_{data}$
Note that, in the service definition:

- *On the left hand side,* $\Psi_{service}$ *includes service security of the stimulus and* $\mathbb{P}\Psi_{data}$ *includes the data security properties of the data parameters associated with it.*
- *On the right hand side,* $\mathbb{P}\Psi_{service}$ *includes service security for the response and the actions in* $\mathcal{S}$*, whereas* $\mathbb{P}\Psi_{data}$ *includes data security for the data parameters associated with the response and actions and the update statements in* $\mathbb{P}\mathcal{U}$*.*

The contract Ξ *is extended to include the security properties:* $\Xi = (\Theta,\ \Omega,\ \Gamma,\ \mathcal{P},\ \Psi)$.

Filtering services:

The following part explains how security properties are used to filter the behavior of a service.

The reactions of a component are filtered by the security properties defined for its services. As mentioned earlier, the identity of the user on whose behalf the component is executing is assumed to be associated with the component at its initialization time. Therefore, when a component requests a service, it provides the identity of the user as part of the

request. Let s be a stimulus and r be its corresponding response, let Λ_s be the set of data parameters associated with s and Λ_r be the set of data parameters associated with its corresponding response r. For convenience, we define the predicates: $PROCESS(x)$ which means that event x will be processed by the component, $ACCESS(\lambda)$ which means that the value λ will be accessed and used by the service, $IGNORE(x)$ which means the event x will be ignored i.e. it will not trigger any service processing or change in the state of a component, $NULL(v)$ which means v will be set to a *null* value. We use the function $assign(\alpha) = v$ which means that the value of the attribute α will be set to v. The behavior of a service
$\Theta : \Sigma_{stimulus} \times \Omega \times \Gamma \times \mathbb{N} \times \Psi_{service} \times \mathbb{P}\Psi_{data} \rightarrow \Sigma_{response} \times \mathbb{P}\mathcal{U} \times \mathcal{S} \times \mathbb{N} \times \mathbb{P}\Psi_{service} \times \mathbb{P}\Psi_{data}$
is determined according to the following rules (R1 and R2 are related to service requests, whereas R2,R3,R4 are related to service provision):

R1: $\forall \psi \in \Psi_{service},\ \psi = (s, r, \Upsilon),\ (\ (\Upsilon \rightarrow PROCESS(s)) \vee (\neg\Upsilon \rightarrow IGNORE(s))\ )$ i.e. if the security policy associated with the service evaluates to true then the stimulus will be accepted and processed, otherwise, the stimulus will be ignored.

R2: $\forall \psi \in \Psi_{data},\ \forall \lambda \in \Lambda_s,\ \psi = (s, \lambda, \Upsilon) \rightarrow (\ (\Upsilon \rightarrow ACCESS(\lambda)) \vee (\neg\Upsilon \rightarrow NULL(\lambda))\ )$ i.e. for all the data parameters associated with the stimulus, if the security policy associated with a data parameter evaluates to true then the data parameter will be used, otherwise, the data parameter will be set to null value.

R3: $\forall \psi \in \Psi_{data},\ \forall \lambda \in \Lambda_r,\ \psi = (s, \lambda, \Upsilon) \rightarrow (\ (\Upsilon \rightarrow ACCESS(\lambda)) \vee (\neg\Upsilon \rightarrow NULL(\lambda))\ )$ i.e. for all the data parameters associated with the response, if the security policy associated with a data parameter evaluates to true then the data parameter will be used, otherwise, the data parameter will be set to null value.

R4: $\forall \psi \in \Psi_{service},\ \forall y \in \mathcal{S},\ \psi = (s, y, \Upsilon) \rightarrow (\ (\Upsilon \rightarrow PROCESS(y)) \vee (\neg\Upsilon \rightarrow IGNORE(y))\ )$ i.e. if there is a security policy associated with triggering an action within the service then the action will be triggered only if the security policy evaluates to true.

R5: $\forall \psi \in \Psi_{data},\ \forall y \in \mathcal{U},\ y = (assign(\alpha) = v) \wedge \psi = (s, y, \Upsilon) \rightarrow (\ (\Upsilon \rightarrow assign(\alpha) = v) \vee$
$(\neg\Upsilon \rightarrow NULL(\alpha))\ )$ i.e. if there is a security policy associated with an update statement then the update statement will be executed only if the security policy evaluates to true, otherwise, the attribute will be set to null value.

Example 4 *Consider a* fingerprint-based car security system *mounted on the door of a car. The system consists of three components: (I) a remote control which comprises a* biometric sensor *that collects user fingerprint,* buttons *that trigger the required actions such as starting the car and locking/unlocking the doors, and a* small monitor *to show information such as the current status of the car, next maintenance time, and details about the last trip, (II) a controller which is responsible for starting the car, locking, unlocking the doors, and sending information about the car, and (III) the security officer component which is responsible for defining and maintaining the security configuration. The security configuration at an instance includes (1) two roles:* driver *and* passenger*, (2) one group :* family*, (3) four privileges:* start the car, lock, unlock*, and* view information*, and (4) 4 users: father, mother, son, daughter. A driver has all privileges, whereas a passenger has only the privilege to lock and unlock the doors. The father has a driver role, whereas the other family members have passenger role. In this example we focus only on the representation of the security configuration and the secure service specification.*

Let $\mathsf{User} = \{father, mother, son, daughter\}$ *be the set of users,*
$\mathsf{Group} = \{family\}$*, and* $GUA(family) = \{mother, son, daughter\}$ *assigns the mother, son, and daughter to the family group.*

Let $\mathsf{Role} = \{driver, passenger\}$ *be the set of roles,* $GRA(family) = \{passenger\}$ *assigns passenger role to family group, and* $URA(father) = \{driver\}$ *assigns driver role to the father.*

Let $\mathsf{Privilege} = \{start, lock, unlock, view\}$ *be the set of privileges,*
$RSA(driver) = \{start, lock, unlock, view\}$ *assigns privileges to the driver role, and*
$RSA(passenger) = \{lock, unlock\}$ *assigns privileges to the passenger role.*

Let $\Sigma = \{start, lock, unlock, view, switchOn, open, close, show\}$ *be the set of events,* $\Lambda = \{status, info\}$ *be the set of data parameters, and* $\xi(show) = \{status, info\}$ *assigns data parameters to the show event.*

Let $u \in \mathsf{User}$ *be the current user using the remote control (the identity is recognized when the user swipes his finger on the scanner), the security policies are:*

- $\psi_1 = (start, switchOn, US(u, start) \vee UR(u, driver))$ *is a service security property associated with the stimulus* start.*The security policy states that: either the user has the* start *privilege or he has the* driver *role;*

- $\psi_2 = (lock, close, US(lock) \vee UG(u, family) \vee UR(u, driver))$ *is a service security property associated with the stimulus* lock. *The security policy states that:*

either the user has the lock *privilege, he is member of the* family *group, or he has the* driver *role;*

- $\psi_3 = (unlock, open, US(lock) \ \vee \ UG(u, family) \ \vee \ UR(u, driver))$ *is a service security property associated with the stimulus* unlock. *The security policy states that: either the user has the* lock *privilege, he is member of the* family *group, or he has the* driver *role;*

- $\psi_4 = (view, show, US(u, show) \ \vee \ UR(u, driver))$ *is a service security property associated with the stimulus* view. *The security policy states that: either the user has the* show *privilege or he has the* driver *role;*

- $\psi_5 = (show, status, UD(u, status) \ \vee \ UR(u, driver))$ *is a data security property associated with the data parameter* status. *The security policy states that: either the suer has the* status *privilege or he has the* driver *role; and*

- $\psi_6 = (show, info, UD(u, info) \ \vee \ UR(u, driver))$ *is a data security property associated with the data parameter* info. *The security policy states that: either the suer has the* info *privilege or he has the* driver *role.*

Thus the set of security properties $\Psi = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5, \psi_6\}$. *Let* $\Omega = \{\omega_1, \omega_2, \omega_3, \omega_4\}$ *where* $\omega_1 = (\emptyset, start, switchOn, true)$, $\omega_2 = (\emptyset, lock, close, true)$, $\omega_3 = (\emptyset, unlock, open, true)$, $\omega_4 = (\emptyset, view, show, true)$. *Let* $\Gamma = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$ *where* $\gamma_1 = (\emptyset, true, start, switchOn, \infty)$, $\gamma_2 = (\emptyset, true, lock, close, \infty)$, $\gamma_3 = (\emptyset, true, unlock, open, \infty)$, $\gamma_4 = (\emptyset, true, view, show, \infty)$. *Service specifications are:*

- $\Theta(start, \omega_1, \gamma_1, t_1, \psi_1, \emptyset) = \ (switchOn, \emptyset, \emptyset, t_2, \emptyset, \emptyset\})$,

- $\Theta(lock, \omega_2, \gamma_2, t_1, \psi_2, \emptyset) = \ (close, \emptyset, \emptyset, t_2, \emptyset, \emptyset\})$,

- $\Theta(unlock, \omega_3, \gamma_3, t_1, \psi_3, \emptyset) = \ (open, \ \emptyset, \ \emptyset, t_2, \emptyset, \emptyset\})$, *and*

- $\Theta(view, \omega_4, \gamma_4, t_1, \psi_4, \emptyset) = \ (show, \ \emptyset, \ \emptyset, t_2, \emptyset, \{\psi_5, \psi_6\})$.

4.5 Behavior

The behavior of a component is determined by its services (stimulus and response) and the constraints defined over them. For a component $\o = (\Sigma_\o, \ \Pi_\o, \ \sigma_\o, \ \Lambda_\o, \ \xi_\o, \ \Xi_\o, \ \mathcal{A}_\o, \ \mathcal{C}_\o, \ \mathcal{T}_\o)$

where $\Xi_{\emptyset} = (\Theta_{\emptyset}, \Omega_{\emptyset}, \Gamma_{\emptyset}, \mathcal{P}_{\emptyset}, \Psi_{\emptyset})$ we define the *behavior* at an interface π as a set $S(\pi)$ of timed sequences, where each sequence $\varpi \in S(\pi)$ contains only stimulus and response events (in addition to actions triggered by the response and occurrence time of each event) belonging to $\sigma(\pi) \cup \Sigma_{\emptyset\ internal}$, and satisfies the following conditions:

S1 for every stimulus $s\ in\ \varpi$, $s \in \sigma(\pi)$, there exists exactly one response r such that $r \in \phi(s) \wedge r \in \varpi$. The stimulus s may occur at many different times in ϖ; therefore, let $s[i]$ denote an occurrence of s in ϖ then for every $s[i]$ there exists exactly one response $r[i]$ where $r[i] \in \phi(s), i : \mathbb{N}, i <$ number of events in ϖ. It is possible to have different responses for different occurrences of the same stimulus (based on data constraints in $\Omega_{\emptyset}$);

S2 $t(r[i]) \geq t(s[i])$, where $t(.)$ is the time function for event occurrences and $r[i], s[i]$ denote an occurrence of s and r in ϖ. Also, $t(s[i]) \geq t(s[j]) \wedge t(r[i]) \geq t(r[j]), i, j : \mathbb{N}, i > j \wedge i, j <$ number of events in ϖ. This means that an event may occur at different times in the timed sequence where always the time of the later occurrence is greater than the time of the former occurrence of the same event. Moreover, if there is an action a, $a \in \mathcal{S} \wedge a \in \Sigma_{\emptyset\ internal}$, defined in the reactivity then $t(a) \geq t(s)$;

S3 for every stimulus $s \in \varpi$, response $r \in \varpi \wedge r \in \phi(s)$: if there is a time constraint $\gamma = (\mathcal{A}, s, r, \delta) \wedge \gamma \in \Gamma_{\emptyset}$ then $|t(r) - t(s)| \leq \delta$;

S4 for every stimulus $s \in \varpi$, response $r \in \varpi \wedge r \in \phi(s)$: if there is a data constraint $\omega = (\mathcal{A}, s, r, \chi) \wedge \omega \in \Omega_{\emptyset}$ defined over the data parameters of s then the data constraint is satisfied i.e. $\chi \rightarrow true$. If there are many data constraints defined on the data parameters of s then only one of them is satisfied;

S5 for every stimulus $s \in \varpi$, response $r \in \varpi \wedge r \in \phi(s)$: if there is a security property $\psi = (s, r, \Upsilon)$ defined for the stimulus then $\Upsilon \rightarrow true$;

S6 for every stimulus $s \in \varpi$, response $r \in \varpi \wedge r \in \phi(s)$: for every data parameter $d \in \xi(s)$ and $d' \in \xi(r) \wedge r \in \phi(s)$ if there is a security property $\psi_s = (s, d, \Upsilon_s)$ or $\psi_r = (r, d', \Upsilon_r)$ then $\Upsilon_s \wedge \Upsilon_r \rightarrow true$;

S7 for every action a, $a \in \varpi \wedge a \in \mathcal{S}$: if there is a security property $\psi = (s, a, \Upsilon)$ defined for the action then $\Upsilon \rightarrow true$;

S8 for every update statement u for an attribute α, $u \in \mathcal{U}$: if there is a security property $\psi = (s, \alpha, \Upsilon)$ defined for the update then $\Upsilon \rightarrow\ true$; and

S9 for every safety property $p \in \mathcal{P}_{\o}$, ϖ satisfies p ($\varpi \vdash p$).

Notice that [S1] assures predictability, [S2] and [S3] assure timeliness, [S4] and [S9] assert that safety requirements are satisfied, and [S5], [S6], [S7], and [S8] asserts that security properties are satisfied.

Definition 8 *The behavior of a component is the arbitrary interleaving of the behaviors at the interfaces of the component. Let $\Pi_{\o} = \{\pi_1, \pi_2, \ldots, \pi_n\}$ be the set of interfaces and $S(\Pi) = \{S(\pi_1), S(\pi_2), \ldots S(\pi_n)\}$ be the set of its corresponding behaviors where $S(\pi_i)$ is the behavior at interface π_i. We define the behavior of a component as the set $\mathfrak{S}(\Pi_{\o})$ of timed sequences where each sequence $\varpi \in \mathfrak{S}(\Pi_{\o})$ is constructed by interleaving sequences from $S(\Pi)$ such that $\varpi = \varpi_1 \mathbin{|||} \varpi_2 \mathbin{|||} \cdots \mathbin{|||} \varpi_n$ where $\varpi_1 \in S(\pi_1) \wedge \varpi_2 \in S(\pi_2) \wedge \cdots \wedge \varpi_n \in S(\pi_n)$ satisfying the following conditions:*

B1 : if a stimulus s is defined at one interface $s \in \sigma(\pi_i)$ and the response is defined at another interface $r \in \phi(s) \wedge r \in \sigma(j),\ i, j : \mathbb{N},\ i, j \leq n \wedge i \neq j$, this means that $s \in \varpi_i \wedge r \in \varpi_j$, then in the interleave sequence ϖ: $s, r \in \varpi \wedge t(s) \leq t(r)$ always; and

B2 : similar to [B1], all the above conditions ([S1],[S2],[S3],[S4],[S5],[S6],[S7],[S8] and [S9]) can be redefined for the interleave sequences and must be satisfied. Note that: $\mathcal{S} \subseteq \Sigma$.

4.6 System Definition

The system definition includes two types of components, *hardware* and *software*, and *configuration*. A hardware component is a special type of component on which the software components will be deployed i.e. a deployment unit. Resource capabilities of deployment units are specified as attributes. For example, a hardware component definition can include attributes such as the number of CPUs and the memory capacity. We define a standard attribute called *type* where its value is either *software* or *hardware*. If this attribute is not defined in a component then it is assumed to be a software component. The system configuration specification includes instances of the defined software and hardware component

types and deployment specification. Deployment Specification are assignments of software instances to hardware instances using the *Deploy* function, $Deploy : CT_s \rightarrow CT_h$ where $\exists type_s \in \mathcal{A}_s, \exists type_h \in \mathcal{A}_h \bullet assign(type_h) = hardware \wedge assign(type_s) = software$. Configuration is defined as a triple $(CT_s, CT_h, Deploy)$ where CT_s is a finite set of software components, CT_h is a finite set of hardware components.

4.7 Composition

Informally, composition means "gluing together" two or more components to form a new component. A given set of components can be composed in different ways to achieve different results. However, the challenging aspect is to develop a set of rules for a stated requirements of trustworthiness to be preserved in a composition. It should be possible to reason about the properties of the composite component relative to the properties of the constituent components. In this respect composition of components is different from component integration [CL02].

In this section we propose a composition rule that composes both the structural part and the contract part of components. For example, composing two components $\varnothing_1$ and $\varnothing_2$ results in a new composite component $\varnothing$ such that: (1) the structural part of $\varnothing$ results from gluing the compatible interfaces of $\varnothing_1$ and $\varnothing_2$, and (2) the contract part of $\varnothing$ results from composing the contracts of $\varnothing_1$ and $\varnothing_2$. The composition should preserve the requirements of trustworthiness ($[S1], [S2], [S3], [S4], [S5], [S6], [S7], [S8], [S9]$). In this section we define the composition of two component types CT_1 and CT_2.

Definition 9 *Let* $CT_1 = (\Sigma_1, \Pi_1, \sigma_1, \Lambda_1, \xi_1, \Xi_1, \mathcal{A}_1, \mathcal{C}_1, \mathcal{T}_1)$ *where* $\Xi_1 = (\Theta_1, \Omega_1, \Gamma_1, \mathcal{P}_1, \Psi_1)$ *and* $CT_2 = (\Sigma_2, \Pi_2, \sigma_2, \Lambda_2, \xi_2, \Xi_2, \mathcal{A}_2, \mathcal{C}_2, \mathcal{T}_2)$ *where* $\Xi_2 = (\Theta_2, \Omega_2, \Gamma_2, \mathcal{P}_2, \Psi_2)$*, their corresponding architecture types* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *are hidden. The compositional rule defines a unique* CT *which can have many architectures* $\mathcal{T}$*.*
The composition $CT = (\Sigma, \Pi, \sigma, \Lambda, \xi, \Xi, \mathcal{A}_{CT}, \mathcal{C}_{CT}, \mathcal{T})$ *where* $\Xi = (\Theta, \Omega, \Gamma, \mathcal{P}, \Psi)$ *is defined using the following rules:*

C1 ***(Interfaces)****:* $\Pi = \{\pi | (\pi \in \Pi_1 \wedge \nexists Q \in \Pi_2 \bullet Compatible(\pi, Q)) \vee (\pi \in \Pi_2 \wedge \nexists Q \in \Pi_1 \bullet Compatible(\pi, Q))\}$*: compatible interface types are used to connect the components together. They form the internal interface types of the composite component, whereas non-compatible interface types form the external interface types.*

C2 ***(Events)****:* $\Sigma = \Sigma_1 \cup \Sigma_2$ *such that:*

- $\Sigma_{internal} = \Sigma_{1\ internal} \cup \Sigma_{2\ internal} \cup \{s \mid (s \in \Sigma_{1\ input} \cup \Sigma_{1\ output},\ s \in \sigma(\pi) \wedge \pi \in \Pi_1 \wedge \exists Q \in \Pi_2 \bullet Compatible(\pi, Q)) \vee (s \in \Sigma_{2\ input} \cup \Sigma_{2\ output},\ s \in \sigma(\pi) \wedge \pi \in \Pi_2 \wedge \exists Q \in \Pi_1 \bullet Compatible(\pi, Q))\}$*: the events defined at the compatible interface types are regarded as internal events for the composite component.*
- $\Sigma_{output} = \Sigma_{1\ output} \cup \Sigma_{2\ output} \setminus \{s \mid (s \in \Sigma_{1\ output},\ s \in \sigma(\pi) \wedge \pi \in \Pi_1 \wedge \exists Q \in \Pi_2 \bullet Compatible(\pi, Q)) \vee (s \in \Sigma_{2\ output},\ s \in \sigma(\pi) \wedge \pi \in \Pi_2 \wedge \exists Q \in \Pi_1 \bullet Compatible(\pi, Q))\}$
- $\Sigma_{input} = \Sigma_{1\ input} \cup \Sigma_{2\ input} \setminus \{s \mid (s \in \Sigma_{1\ input},\ s \in \sigma(\pi) \wedge \pi \in \Pi_1 \wedge \exists Q \in \Pi_2 \bullet Compatible(\pi, Q)) \vee (s \in \Sigma_{2\ input},\ s \in \sigma(\pi) \wedge \pi \in \Pi_2 \wedge \exists Q \in \Pi_1 \bullet Compatible(\pi, Q))\}$

C3 ***(Data Parameters)****:* $\Lambda = \Lambda_1 \cup \Lambda_2$

C4 ***(Event's data parameters)****:* $\forall s \in \Sigma,\ \xi(s) = \{\xi_1(s) \mid s \in \Sigma_1\} \cup \{\xi_2(s) \mid s \in \Sigma_2\}$

C5 ***(Interface's events)****:* $\forall \pi \in \Pi,\ \sigma(\pi) = \{\sigma_1(\pi) \mid \pi \in \Pi_1\} \cup \{\sigma_2(\pi) \mid \pi \in \Pi_2\}$

C6 ***(Attributes)****:* $\mathcal{A}_{CT} = \mathcal{A}_1 \cup \mathcal{A}_2$

C7 ***(Constraints)****:* $\mathcal{C}_{CT} = \mathcal{C}_1 \cup \mathcal{C}_2$

C8 ***(Architecture)****:* $\mathcal{T} = \{\tau\}$ *where* $\tau = \{\mathcal{O}, \mathcal{N}, \mathcal{H}, \mathcal{A}_\tau, \mathcal{C}_\tau\}$ *such that:*

- *The set of component types:* $\mathcal{O} = \{CT_1, CT_2\}$
- *The set of connector types:* $\forall \pi_1 \in \Pi_1, \forall \pi_2 \in \Pi_2 \bullet Compatible(\pi_1, \pi_2) \rightarrow \exists \mathrm{K} \in \mathcal{N},\ \mathrm{K} = (\mathcal{A}_k, \mathcal{C}_k, \{\rho_1, \rho_2\}, \mathcal{M}) \wedge \rho_1 = (\mathcal{A}_1, \chi_1, \pi_1) \wedge \rho_2 = (\mathcal{A}_2, \chi_2, \pi_2)$
- *The set of attachments:* $\forall \pi_1 \in \Pi_1, \forall \pi_2 \in \Pi_2,\ Compatible(\pi_1, \pi_2) \rightarrow \exists \propto_1 \boxplus \propto_2, \exists \mathrm{K} \in \mathcal{N},\ \propto_1 = \rho_1 \ @\ \mathrm{K} \bowtie \pi_1 \ @\ CT_1 \wedge \propto_2 = \rho_2 \ @\ \mathrm{K} \bowtie \pi_2 \ @\ CT_2$

There could be many architecture types for CT *in* $\mathcal{T}$ *because not all the interfaces in the resulting connector types should be linked. Also, different component instances can have different number of connector and interface instances which enables the component to have different possible dynamic architectures.*

C9 ***(Services)****:* $\Theta = \Theta_1 \cup \Theta_2$

C10 ***(Data Constraints)****:* $\Omega = \Omega_1 \cup \Omega_2$

C11 ***(Time Constraints)****:* $\Gamma = \Gamma_1 \cup \Gamma_2$

C12 ***(Safety Properties)****:* $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$

C13 ***(Security Properties)****:* $\Psi = \Psi_1 \cup \Psi_2$

The security officer is defined at a system level. Therefore, RBAC functions are not affected by the composition.

We assert that the composition rule stated in Definition 9 preserves the requirements of safety and security ($[S1] \ldots [S9]$).

Theorem 1 *The composition of two components that satisfy the requirements of safety and security ($[S1] \ldots [S9]$) results in a component that satisfies these requirements.*

Proof 1 *Let $\emptyset_1$ and $\emptyset_2$, instances of CT_1 and CT_2 respectively, be two components that satisfy the requirements of safety and security ($[S1] \ldots [S9]$). Let $\emptyset$ be an instance of CT, the composition of CT_1 and CT_2 according to Definition 9. Let $\mathfrak{S}_1$ and $\mathfrak{S}_2$ be behaviors representing the set of all possible observed sequences of $\emptyset_1$ and $\emptyset_2$ respectively, $\mathfrak{S}$ be the behavior of the composite component $\emptyset$ representing the set of all possible observed sequences of $\emptyset$, $S(\pi)$ be the behavior at an interface instantiated from the interface type $\pi \in \Pi$ in $\emptyset$. We want to proof that $\mathfrak{S}$ satisfies the requirements of safety and security ($[S1] \ldots [S9]$).*

We use the following properties in the proof, which are derived from Definition 9:

Prop.1 *From C5 in Definition 9 and from Definition 4, Every event is associated with only one interface type: $\forall \pi_1, \pi_2 \in \Pi, \sigma(\pi_1) \cap \sigma(\pi_2) = \emptyset$,*

Prop.2 *From C1, every interface type in the composite component belongs only to one component definition CT_1 or CT_2: $\forall \pi \in \Pi \bullet (\pi \in \Pi_1 \ \vee \ \pi \in \Pi_2) \wedge (\pi \notin \Pi_1 \cap \Pi_2)$, and there are no two interface types that are compatible in the composite component: $\nexists \pi_1, \pi_2 \in \Pi, Compatible(\pi_1, \pi_2)$.*

Prop.3 *The behavior of a component is the arbitrary interleaving of the behaviors at the interfaces of a component. From [Prop.2], $\Pi \subseteq \Pi_1 \cup \Pi_2$, excluding the compatible interface types. From Definition 8, the observed behavior of the composite component is the arbitrary interleaving of the timed sequences of the non-compatible interface types.*

Prop.4 *From Definition 8, the behavior of an interface* π *is the set of timed sequences* $S(\pi)$*. From [Prop.2] an interface in the composite component either belongs to* CT_1 *or to* CT_2*. Since,* CT_1 *and* CT_2 *satisfy (*$[S1] \dots [S9]$*) then* $\forall \pi \in \Pi_1 \cup \Pi_2,\ \forall \varpi \in S(\pi),\ \varpi \vdash ([S1] \dots [S9])$ *i.e. every timed sequence in the behavior of every interface satisfies (*$[S1] \dots [S9]$*).*

[Prop.3] defines the behavior of the composite component and [Prop.4] shows that the timed sequences of the behavior of the composite component satisfy ($[S1] \dots [S9]$*). Therefore, the behavior of the composite component is an interleaving of sequences which already satisfy (*$[S1] \dots [S9]$*).* Therefore,we need to prove that the composition rule doesn't violate any of ($[S1] \dots [S9]$) so that the sequences remain to satisfy ($[S1] \dots [S9]$) after interleaving.

S1 ***for every stimulus there is a response****: We need to show that stimulus and response relations exist after composition and stimulus and response events are still available in interface definitions after composition: From C2, the events of* CT_1 *and* CT_2 *are preserved in* CT *after composition. From C5, interface definitions preserve their events after composition. From C9, service definitions doesn't change after composition. Therefore, stimulus-response relations doesn't change after composition. This means that in every timed sequence of every interface behavior, for every stimulus there exists one response. Since the interleaving doesn't change the time sequences then the behavior of the composite component is an interleaving of timed sequences where for every stimulus in each sequence there exists one response. Therefore, in the result interleaved sequence, for every stimulus there exists one response.*

S2 ***response occurs after stimulus****: From B1 in Definition 8, the interleaving preserves the timing of events. Therefore, building on the previous proof, for every timed sequence in the behavior of the composite component the time of the response occurs after the time of the stimulus.*

S3 $|t(r) - t(s)| \leq \delta$*: From the previous two proofs, stimulus and response relations are preserved and their timing order is preserved. Therefore, we need to prove that time constraints are preserved and their associations with stimulus-response relations are preserved in the composition: From C11, time constraints are preserved in the composition. From C9, service definitions are preserved; therefore, the associations between time constraints and services are preserved in the composition. Since*

the interleaving doesn't change the occurrence time of events in a sequence then $t(s)$, the time of a stimulus s, before the interleaving equals $t(s)$ after the interleaving. Also, $t(r)$, the time of the corresponding response remains the same. Therefore, $|t(r) - t(s)|$ is the same before and after the interleaving and since time constraints are preserved then δ is the same. Thus $|t(r) - t(s) \leq \delta|$ is preserved.

S4 ***data constraints are preserved****: We need to prove that the data constraints are preserved and that the evaluation of the logical expression doesn't change before and after the interleaving. (1) From C10, data constraints are preserved in the composition. From C9, service definitions are preserved. Therefore, the association between data constraints and stimulus-response are preserved. Similar to the previous proof, for every sequence, any data constraint defined before the interleaving will remain after the interleaving. (2) The evaluation of the logical expression depends on the values of event's data parameters and on the attributes. Therefore, we need to prove that the sequences of the other component doesn't affect the values of data parameters and attributes so that the evaluation of the logical expression doesn't change before and after the interleaving: First, each event defines a set of data parameters. The values of the data parameters are not affected by other events because they are defined locally for the event. Therefore, the interleaving doesn't affect the values of data parameters of events. Second, each component define its own set of attributes. The services defined in each component affect only the attributes defined for that specific component. Since, CT_1 and CT_2 satisfy [S4], the time sequences of each component satisfy [S4]. Therefore, each interleaved sequence consists of two parts. One part from CT_1 and another part from CT_2. Since each part doesn't affect the attributes of the other part then the evaluation of logical expressions remains the same after interleaving. Therefore, the composition preserves [S4].*

S5,S6,S7,S8 ***security properties are preserved****: We need to prove that the security properties are preserved in the composition and that the behavior of the composite component does not violate the defined security properties. First, From C3, the data parameters are preserved. From C4, the associations between events and data parameters are preserved. From C9 and C13, the defined security properties are preserved in the composition. Second, from [Prop.4], each sequence of $\emptyset_1$ and $\emptyset_2$ satisfies the security properties before the interleaving. Therefore, in each sequence of $\emptyset_1$ and $\emptyset_2$, for every security policy, if the security policy of a service security evaluates to false then*

an event will be filtered out and not included in the sequence. Also, if the security policy of a data security evaluates to false then the value of the data parameter will be set to Null *value. Since interleaving does not change events and does not introduce new events then any event that exists in the sequences of will exist in the interleaving and any blocked event that does not exist in the sequences of $\emptyset_1$ and $\emptyset_2$ will not exist in the interleaving. Therefore, for any event that exists in an interleaved sequence, either there is no service security defined for it or the security policy evaluates to true in the original sequence before the interleaving. Data security can be proved in similar way. Information flow is prevented because response events and data parameters associated with them are filtered based on the user to whom the service is provided. Therefore, in the interleaved sequence, if the destination user doesn't have privilege then events and data will be filtered.*

S9 ***safety properties are preserved****: We need to proved that safety property definitions are preserved and that the behavior of the composite component does not violate a safety property which is satisfied by any of its constituent components. First, from C13 and [Prop.4], safety property definitions are preserved. Second,* $\forall p \in \mathcal{P},\ p \in \mathcal{P}_1 \ \vee\ p \in \mathcal{P}_2$*. Case 1 (*$p \in \mathcal{P}_1$*): 1)* p *is defined over the attributes and evens of* CT_1*, 2)* $\forall \varpi_1 \in \mathfrak{S}_1,\ \varpi_1 \vdash p$*, 3)* $\forall \varpi \in \mathfrak{S},\ \varpi = \varpi_1 \parallel\mid \varpi_2$ *where* $\varpi_2 \in \mathfrak{S}_2$ *i.e. each sequence of the behavior of the composite component is defined from two parts: one part comes from the behavior of* $\emptyset_1$ *and the second part from the behavior of* $\emptyset_2$*. From 1,* $\varpi_1 \vdash p$*. Since a safety property is defined over events and attributes of a specific component then* p *is defined over* Σ_1 *and* $\mathcal{A}_1$*. Since* ϖ_2 *consists of events that belong to* Σ_2 *then these events doesn't violate* p*. Therefore,* p *is satisfied after the composition. Case 2: it can be proved in a similar way to case 1.*

4.8 Summary

This chapter presented a formal description of the structure, contract, and behavior of trustworthy components. The structural definition included the essential elements of component-based development. The contract definition included timeliness, safety and security properties. The behavior specification of components are generated automatically as extended timed automata. Chapter 6 provides detail description of the behavior specification. Reliability and availability properties will be discussed on Chapter 7. This chapter introduces

a composition theory that includes rules for composing the structural part and the contract part of trustworthy components. We provided a proof that the composition theory preserves the defined requirements of timeliness, safety, and security.

Chapter 5

TADL: Trustworthy Architecture Description Language

Formal notations are difficult to use and communicate among software architects. In order to comprehend and use the formally expressed content, some mathematical background and expertise in the formal language's semantics are essential. Therefore, we created an architecture description language (TADL), which is a light-weight formal notation based on our formalism. TADL provides complete descriptions of our trustworthy component model in a high level language that can be understood by software engineers. This chapter introduces the TADL syntax for architectural elements.

The formalism described in Chapter 4 is an abstract description of a component. Figure 17 shows the meta model of a component as well as a system that can be composed of components. There is a one to one correspondence between the formal elements and the elements shown in Figure 17.

5.1 Meta-Architecture

Our component model is a meta-architecture, an architecture type from which different system architectures can be created. Figure 17 depicts the component model. The main building blocks of the component model are *component definition*, *component architecture definition*, *contract*, *security mechanism*, *system definition*, *package*, *constraint*, and *attribute*. All the elements in the model inherit from the *System Element* which contains

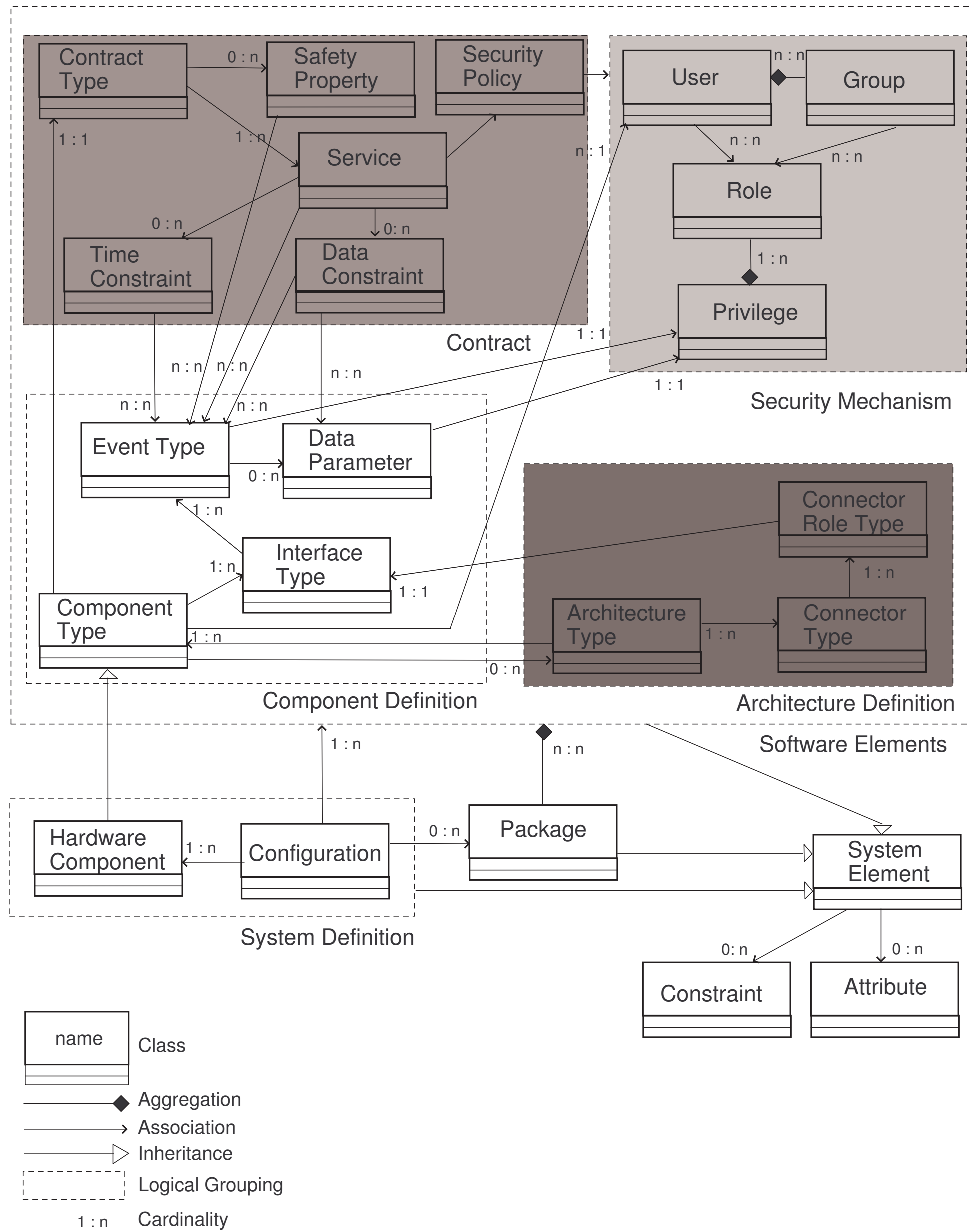


Figure 17: Trustworthy Component Model

```
ElementType < name > {
        (Attribute < name >)*;
}
```

Figure 18: The TADL syntax of Element Type

basic class definition along with attributes and constraints. A component definition includes an architecture definition, the internal structure of the component implementation, and contract specification, a description of services together with restrictions that constrain the behavior of component interactions. In addition, a component definition includes a security mechanism to filter the services and data that are communicated through interfaces. The system definition contains *hardware components* and system *configuration* specification. A package contains a collection of definitions of related elements. The formal definitions of the elements of the component model has already been introduced in Chapter 4. The following section describes the TADL syntax definition of the elements of the meta architecture shown in Figure 17. Note that, reliability and availability properties are not shown in Figure 17. They will be introduced in Chapter 7.

5.2 TADL

In this section we give a concrete syntax of a component description and call the language of description TADL. Consequently, corresponding to each element of the component model in Figure 17 there is a description in TADL. Moreover, the abstract formal description given in Chapter 4 for each element is written in a concrete syntax within the structural element of this unit.

In TADL, every element of the meta-architecture is described separately. The rationale behind this is to increase reuse of elements for designing different systems and allow reconfiguration without affecting other definitions. The description of an element contains: (1) element type, (2) element name, and (3) specification of the contents of the element. Figure 18 gives an example of an element specification. Note that $(Attribute < name >)*$ means that 0 or more attributes can be defined as part of the element.

```
ParameterType < name > {
        < DataType >< name >;
        Default < value >;
        Constraint < FOPL >;
}

EventType < name > {
    (Attribute < name >)*;
    Constraint < FOPL >;
    (ParameterType < name >)*;
    Direction < name >;
}

Attribute< name > {
      < DataType >< name >;
      Default < value >;
      Constraint < FOPL >;
}
```

Figure 19: The TADL syntax of Parameter Type, Event Type, and Attribute

5.2.1 Event and data parameter

The formal description of a data parameter, attribute, and event, which were given in Chapter 4, are: $\lambda = (\mathcal{D}, \nu, \chi_\nu)$, $\alpha = (\mathcal{D}, \nu_\alpha, \chi_{\alpha\nu})$, and $e = (\Lambda_e, \mathcal{A}_e, \chi_e, \varrho_e)$. The TADL syntax of the *data parameter*, *attribute*, and *event type* are presented in Figure 19. The TADL syntax of the data parameter and attribute include *data type*, *default* value, and constraint. The TADL syntax of the event type includes a set of attributes, a constraint, data parameters, and the *direction* of the event.

Figure 20 shows an example specification of an event type called *ControlTemperature*. It has one data parameter of integer type, *CurrentTemperature*, and two attributes, *priority* and *WCET*. The event type definition includes one constraint stating that the value of the *current temperature* data parameter must not exceed 100 or be less than -25.

```
ParameterType CurrentTemperature{
              DataType Integer;}

Attribute Priority{
        DataType Integer;
        Default 1;}

Attribute WCET{
        DataType Integer;
        Default 30;}

EventType ControlTemperature{
          CurrentTemperature temp;
          Priority p;
          WCET w;
          Constraint temp ≥ −25 ∧ temp ≤ 100;
          Direction input;
}
```

Figure 20: An example definition of an event type

5.2.2 Contract

The formal description of a time constraint, data constraint, service, safety property, security property, and contract, which were given in Chapter 4, are: $\omega = (\mathcal{A}_\omega,\ s,\ r,\ \chi_\omega)$, $\gamma = (\mathcal{A}_\gamma,\ \chi_\gamma,\ s,\ r,\ \delta)$, $\Theta :\ \Sigma_{stimulus} \times \Omega \times \Gamma \times \mathbb{N} \times \Psi_{service} \times \mathbb{P}\Psi_{data} \rightarrow \Sigma_{response} \times \mathbb{P}\mathcal{U} \times \mathcal{S} \times \mathbb{N} \times \mathbb{P}\Psi_{service} \times \mathbb{P}\Psi_{data}$, $p = (\Sigma_p,\ \chi)$, $\psi = (s, x, \Upsilon)$, and $\Xi = (\Theta,\ \Omega,\ \Gamma,\ \mathcal{P},\ \Psi)$. Figure 21 shows the TADL syntax of a time constraint, data constraint, service, safety property, security policy, and contract.

The TADL syntax of time constraint, data constraint, and service include a set of attributes and two event types defining the *request event* and the *response event* and two predicates specifying which event is the request ($RequestService$) and which event is the response ($ResponseService$). The TADL syntax of time constraint includes the *maximum safe time*, whereas the TADL syntax of data constraint includes a logical expression stated using first order predicate logic (FOPL). The syntax definition of a service includes a data

constraint, time constraint, security policies, update statements, and action event types associated with the service. The TADL syntax of a safety property and security policy include a set of event types, a set of data parameter types, set of attributes, and a logical expression. The TADL syntax of a contract includes one or more services, a set of safety properties, and a set of security policies.

An example of a contract specification is presented in Figure 22. It includes three Event types: *ControlTemperature* (defined earlier), *Raise*, and *Lower*. *ControlTemperature* represents a request for service aimed to control the current temperature in a room. In response, either *Raise* or *Lower* should be executed. Therefore, there is a need to define two data constraints to specify the conditions based on which either *Raise* or *Lower* will be selected as a response to *ControlTemperature*. The two data constraints are *RaiseDataConstraint* and *LowerDataConstraint*. The first constraint requires the current temperature, which is a data parameter defined in *ControlTemperature*, to be less than or equal to 20. The second constraint requires the current temperature to be more than 20. Two services are defined: *Control-Raise*, which will be activated if the data constraint *RaiseDataConstraint* evaluates to true, and *Control-Lower*, which will be activated if the data constraint *LowerDataConstraint* evaluates to true. Time constraint specification mandates the interval of time between the two actions "request to control the temperature" and "raising the temperature to the desired level" be less than or equal to 45 units of time. The time constraint is associated with the *Control-Raise* service. Also, a safety property is defined as part of the contract.

5.2.3 Component architecture

The structural description of a component includes definitions of *interface types, connector role types, connector types, architecture types,* and *component types*. Their corresponding formal definitions are: $\pi = (\mathcal{A}_\pi, \chi_\pi, \sigma)$, $\rho = (\mathcal{A}_\rho,\ \chi_\rho,\ \pi)$, $\mathrm{K} = (\mathcal{A}_k,\ \chi_k,\ \mathcal{R}, \mathcal{M})$, $\tau = (\mathcal{O},\ \mathcal{N},\ \mathcal{H},\ \mathcal{A}_\tau,\ \mathcal{C}_\tau)$, and $CT = (\Sigma,\ \Pi,\ \sigma,\ \Lambda,\ \xi,\ \Xi,\ \mathcal{A}_c,\ \mathcal{C}_c,\ \mathcal{T})$. Figure 23 presents the TADL syntax of these elements. An interface definition includes a set of attributes, a constraint, and one or more event types. The TADL syntax of a connector role type includes a set of attributes, a constraint, and an interface type. The TADL syntax of a connector type includes one or more connector role types in addition to a set of attributes and a constraint. Also, it includes communication method. The TADL syntax of an architecture type includes one or more component types, one or more connector types, a set of attributes, a set

```
TimeConstraint < name > {
    (Attribute < name >)*;
    Constraint < FOPL >;
    EventType < request − name >;
    RequestService
    (< request − name >);
    EventType < response − name >;
    ResponseService
    (< response − name >);
    float MaxSafeTime;
}

DataConstraint < name > {
    (Attribute < name >)*;
    EventType < request − name >;
    RequestService
    (< request − name >);
    EventType < response − name >;
    ResponseService
    (< response − name >);
    Constraint < FOPL >;
}

ContractType < name > {
        (Service < name >)+;
        (SafetyProperty< name >)*;
        (SecurityPolicy< name >)*;
}
```

```
Service < name > {
    EventType < request − name >;
    RequestService
    (< request − name >);
    EventType < response − name >;
    ResponseService
    (< response − name >);
    DataConstraint < name >;
    TimeConstraint < name >;
    (Update statements)*;
    (EventType < action − name >)*;
    (SecurityPolicy< name >)*;
}

SafetyProperty < name > {
      (EventType < name >)*;
      (ParameterType < name >)*;
      (Attribute< name >)*;
      Constraint < FOPL >;
}

SecurityPolicy < name > {
        (EventType < name >)*;
        (ParameterType < name >)*;
        (Attribute< name >)*;
        Constraint < FOPL >;
}
```

Figure 21: The TADL syntax of Time Constraint, Data Constraint, Service, Safety Property, and Contract

```
EventType Raise{...};
EventType Lower{...};


DataConstraint RaiseDataConstraint{
    ControlTemperature Control;
    RequestService(Control);
    Raise raise;
    ResponseService(raise);
    Constraint Control.temp ≤ 20;
    }

DataConstraint LowerDataConstraint{
    ControlTemperature Control;
    RequestService(Control);
    Lower lower;
    ResponseService(lower);
    Constraint Control.temp > 20;
    }

TimeConstraint RaiseTimeConstraint{
    ControlTemperature Control;
    RequestService(Control);
    Raise raise;
    ResponseService(raise);
    float MaximumSafeTime = 45;
    }
```

```
Service Control-Raise{
    ControlTemperature Control;
    RequestService(Control);
    Raise raise;
    ResponseService(raise);
    RaiseDataConstraint dcl;
    RaiseTimeConstraint tc1;}

Service Control-Lower{
    ControlTemperature Control;
    RequestService(Control);
    Lower lower;
    ResponseService(lower);
    LowerDataConstraint dcl;}

SafetyProperty Safety{
    ControlTemperature Control;
    (Control.temp > 20 →
    Control − Lower)∨
    (Control.temp ≤ 20 →
    Control − Raise); }

ContractType Contract{
    Control-Raise cr;
    Control-Lower; cl
    Safety p; }
```

Figure 22: An example of a contract specifications

of constraints, and a set of attachment specification. Each attachment specification binds a connector role type of a connector to an interface type of a component. The TADL syntax of a component type includes sets of event types, attributes, constraints, and architecture types. Also, it includes a contract, one or more interface types, and a special attribute called user, which identifies the client on whose behalf the component is executing.

Figure 24 shows an example of a composite component specification. It shows two components types, *DataStore* and *QueryManager* which are connected together using a connector type *DBConnector*. The architecture specification is defined using the architecture type *DatabaseArchitecture*. Each constituent component defines an interface type. *DataStore* contains *IDataProvider* and *QueryManager* contains *IDataReader*. Two connector role types are used to connect *DBConnector* to the interface types of the two components. The system is defined as a composite component, *DatabaseSystem*, whose architecture is of type *DatabaseArchitecture*.

5.2.4 Security mechanism

Figure 25 presents the TADL syntax of *role based access control* (RBAC) specifications. The syntax of RBAC includes assigning users to groups, users to roles, roles to groups, privileges to roles, *service privileges* and *data parameter privileges* to roles. The formal description of a user, role, group, and privilege is identical to the formal description of an attribute. The formal description of the assigning functions are described in Chapter 4.

Figure 26 shows an example of RBAC specification using TADL. The user *Anne* is assigned the role *Accountant*. This role has the privilege *GenerateReport*, which allows Anne to generate the account receivable report *GenerateAccountReceivableReport*.

5.2.5 System definition

Figure 27 includes the TADL syntax of system configuration. It includes one or more system elements and a deployment specification, which assign software component types to hardware component types.

Figure 28 shows an example configuration specification using TADL. It includes one software component *DatabaseSystem* which is deployed on a hardware component, called *server*. This hardware component is configured with Xenon processor and 4GB memory.

The meta-architecture elements are units of reuse. Towards this purpose we include a

```
InterfaceType < name > {
    (Attribute < name >)*;
    (EventType < name >)*;
    Constraint < FOPL >;
}

ConnectorRoleType < name > {
    (Attribute < name >)*;
    Constraint < FOPL >;
    InterfaceType < name >;
}

ConnectorType < name > {
    (ConnectorRoleType < name >)+;
    (Attribute < name >)*;
    Constraint < FOPL >;
    Communication Method < name >;
}
```

```
ArchitectureType < name > {
    (ComponentType < name >)+;
    (ConnectorType < name >)+;
    (Attribute < name >)*;
    (Constraint < FOPL >)*;
    (Attachment
    (ConnectorType.RoleType.InterfaceType,
    ComponentType.InterfaceType))*;
}

ComponentType < name > {
    (EventType < name >)*;
    (Attribute < name >)*;
    (Constraint < FOPL >)*;
    User u;
    (InterfaceType < name >)+;
    (ArchitectureType < name >)*;
    ContractType < name >;
}
```

Figure 23: The TADL syntax of Interface Type, Connector Role Type, Connector Type, Architecture Type, and Component Type

repository in the design. All meta-architecture elements' definitions are stored in a repository, a storage place to store and reuse the specified and developed meta-architectural elements. The repository provides storage facilities for system specification, development source code, and compiled, execution ready assembly of components. The repository allows storing and retrieving different versions of the same component. Detailed discussion of the repository is provided in Chapter 9.

5.3 Summary

This chapter introduced TADL, an architecture description language that is based on our trustworthy component model. We described the TADL syntax for the elements of our trustworthy component model. A visual modeling tool [Yun09] is used to design component-based systems based on our component model. The tool provides user interface to design systems and automatically generates the corresponding TADL specification of the system. We use an automatic model transformation technique to analyze the resulting TADL specification and generate behavior specification and real-time models.

```
InterfaceType IDataProvider{}
InterfaceType IDataReader{}
ConnectorRoleType Provider{
IDataProvider iRoleDataProvider;}
ConnectorRoleType Reader{
IDataReader iRoleDataReader;}

ConnectorType DBConnector{
        Provider providerRole;
        Reader readerRole;}

ComponentType DataStore {
IDataProvider idataProvider;}
ComponentType QueryManager{
IDataReader idataReader;}

ArchitectureType DatabaseArchitecture{
       DataStore storeDB;
       QueryManager queryDB;
       DBConnector connect;
       Attachment
       (connect.providerRole.iRoleDataProvider,
       storeDB.idataProvider);
       Attachment
       (connect.readerRole.iRoleDataReader,
       queryDB.idataReader);
}

ComponentType DatabaseSystem{
DatabaseArchitecture dBArchitecture;}
```

Figure 24: An example of a composite component specification

```
User< name > {
    (Attribute < name >)*;
    Constraint < FOPL >;}

Group< name > {
    (Attribute < name >)*;
    Constraint < FOPL >;}

Role< name > {
    (Attribute < name >)*;
    Constraint < FOPL >;}

Privilege< name > {
        (Attribute < name >)*;
        Constraint < FOPL >;
}
```

```
RBAC< name > {
  (User < name >)*;
  (Group < name >)*;
  (Role < name >)*;
  (Privilege < name >)*;
  (User − Groups − Assignment(User,Group))*;
  (User − Roles − Assignment(User,Role))*;
  (Group − Roles − Assignment(Group,Role))*;
  (ServiceType < name >)*;
  (ParameterType < name >)*;
  (Privileges−for − services
            (Service,Privilege,Role))*;
  (Privileges−for − data − parameters
            (DataParameter,Privilege,Role))*;
}
```

Figure 25: The TADL syntax of RBAC specification

```
RBAC Accounting{
  User Anne;
  Role Accountant;
  Privilege GenerateReport;
  User-Roles-Assignment(Anne,Accountant);
  ServiceType GenerateAccountReceivableReport{}
  Privileges-for-services(GenerateAccountRecievableReport,
  GenerateReport,Accountant);
}
```

Figure 26: An example RBAC specification using TADL

```
Configuration< name > {
     (SystemElement < name >)+;
     (Deploy(HardwareComponentType,ComponentType))+;
}
```

Figure 27: The TADL syntax of system configuration specification

```
ComponentType DatabaseSystem{}

Attribute Processor{
          DataType String;
          Default "Xenon";}

Attribute RAM{
          DataType String;
          Default "4GB";}

HardwareComponent Server{
            Processor Xenon;
            RAM memory;}

Configuration  Deployment{
     DatabaseSystem DBS;
     Server server;
     Deploy(server,DBS);}
```

Figure 28: An example configuration specification using TADL

Chapter 6

Model Transformation and Formal Verification

Our goal is to achieve a uniform specification language for specifying and analyzing the different kinds of trustworthiness properties. The contract specification enables regulating services through time constraints, restricting services through constraints, and specifying safety properties. The security mechanism specification enables filtering services and information so that only authorized users can request services and view information. Therefore, These features enable specifying trustworthiness properties at the architecture level.

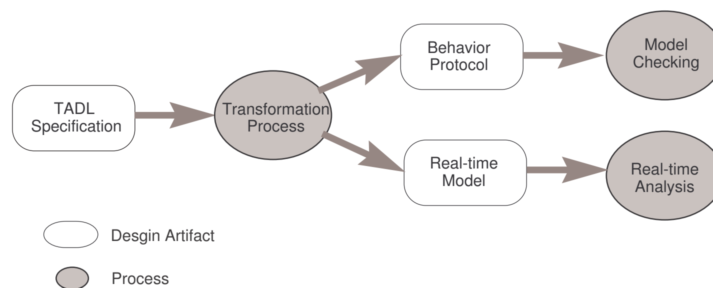


Figure 29: The process of transformation and analysis

The next step is to reason about safety and security in a uniform manner. Figure 29 depicts the transformation and analysis process. During this process, the specification is analyzed and undergoes an automatic transformation process. The transformation process automatically generates behavior protocols and real-time models. The behavior is generated as an extended finite-state machine. The resulting state machine is input into a model checker to verify safety and security. The real-time model is generated as a finite-state

machine augmented with real-time tasks specification. The resulted state machine and task specification are input into a real-time schedule analysis tool to verify timeliness requirements.

The next sections describe how the specification is transformed into behavior protocols and real-time models and how safety and security properties are verified.

6.1 Verifying Safety and Security

This section introduces transformation rules from our component model into UPPAAL model checker [BDL04]. We explain how the formal specification of a component is transformed into UPPAAL extended timed automata. First, we present brief information about UPPAAL model checker. Then, we introduce transformation rules for the automatic generation of component behavior. Finally, we describe how the verification process is conducted using UPPAAL model checker.

6.1.1 UPPAAL

UPPAAL [BDL04] is a mature model checker that has been used successfully for more than a decade to model check several types of concurrent real time systems. The UPPAAL modeling language is based on timed automata $TA = (L, l_0, K, A, E, I)$ where L is the set of locations denoting states, l_0 is the initial location, K is the set of clocks, A is the set of actions denoting events that cause transitions between locations, E is the set of edges, and I is the set of invariants. Formally, $E \subseteq L \times A \times B(K) \times 2^K \times L$ where $B(K)$ is the set of clock and data constraints denoting guard conditions that restrict transitions, 2^K is the set of clock initializations to set clocks whenever required, and $I : L \rightarrow B(K)$ is a function assigning clock constraints to locations as invariants. UPPAAL extends timed automata with additional features. We present some of these features that are relevant to the transformation process:

- **Templates:** Timed automata are defined as templates with optional parameters. Parameters are local variables that are initialized during template instantiation in system declaration.
- **Global variables:** Global variables and user defined functions can be introduced in a global declaration section. Those variables and functions are shared and can be

accessed by all templates.

- **Binary synchronization:** Two timed automata can have a synchronized transition on an event when both move to a new state at the same time when the event occurs. An event that causes synchronous transition is defined as a *channel*, a UPPAAL data type. A channel can have two directions: input(labeled with ?) and output(labeled with!).

- **Committed Location:** Time is not allowed to pass when the system is in a committed location. If the system state includes a committed location, the next transition must involve an outgoing edge from the committed location.

- **Expressions:** There are three main types of expressions: (1) *Guard* expressions are evaluated to boolean and used to restrict transitions; guard expressions may include clocks and state variables, (2) *Assignment* expressions are used to set values of clocks and variables, and (3) *Invariant* expressions are defined for locations and used to specify conditions that should be always true in a location.

- **Edges:** Edges denote transitions between locations. An edge specification consists of four expressions: *Select:* assigns a value from a given range to a defined variable, *Guard:* an edge is enabled for a location if and only if the guard is evaluated to true, *Synchronization:* specifies the synchronization channel and its direction for an edge, and *Update:* assignment statements that reset variables and clocks to required values.

In UPPAAL, system properties are expressed formally using a simplified version of CTL [BDL04] as follows:

- **Safety property** is formulated positively stating that some thing good is invariantly true. For example, let φ be a formula, $A\Box\ \varphi$ means that φ should be always true.

- **Liveness property** states that some thing good will eventually happen. For example, $A \diamond\ \varphi$ means that φ will eventually be satisfied.

6.1.2 Transformation Rules

In this section, we introduce the transformation rules for the automatic generation of component behavior based on the analysis of component's structure and contract defined in its

specification. A component-based system is a network of connected components. Every component is mapped to a UPPAAL template in a one to one manner. We assign a parameter to every UPPAAL template to denotes the identifier of the user on whose behalf the component is running. This parameter will be used for ensuring service and data security. Let $O = \{\o_1, \ldots, \o_n\}$ be the set of components in a system where
$\o_i = (\Sigma_i,\ \Pi_i,\ \sigma_i,\ \Lambda_i,\ \xi_i,\ \Xi_i,\ \mathcal{A}_i,\ \mathcal{C}_i,\ \mathcal{T}_i)$ and $\Xi_i = (\Theta_i, \Omega_i, \Gamma_i, \mathcal{P}_i, \Psi_i)$. Let $TA = (L, L_0, K, A, E, I, u)$ be the definition of UPPAAL timed automata where u denotes the user identity parameter associated with the template at its instantiation. Then, the transformation rules construct $T = \{t_1, \ldots, t_n\}$, a set of UPPAAL templates, where t_i is the template constructed from component $\o_i$. In brief, during the process of constructing $TA = (L, l_0, K, A, E, I)$ from the component specification:

- Σ_i is used to construct L where every location in L denotes the state of processing a service request in Σ_i,
- Γ_i is used to construct K and I where a clock in K and an invariant in I are defined for every time constraint in Γ_i,
- Σ_i is used to construct A where an action in A is defined for every service request and response in Σ_i, and
- $\Sigma_i, \Lambda_i, \xi_i, \Theta_i, \Omega_i,$ and Ψ_i are used to construct E and its associated expressions. More precisely, Λ_i defines data parameters in ξ_i which in turn are used in defining data constraints in Ω_i that are used along with Υ to define $Guard$ conditions for edges. Σ_i and Θ_i are used in defining $Sync$ expressions. Ψ is used to control data parameters access in $Update$ expression.

We extend the UPPAAL formal template by adding security features. In the global declaration section, we define: (1) lists of groups, roles, privileges, and a list of representative users where a user defined for each role and group, (2) a *service-access control matrix*, SAC, that defines role access rights to services, (3) a *data-access control matrix*, DAC, that defines role access rights to service data parameters, and (4) implementation of the security functions defined in Definition 6.

The steps for constructing $TA = (L, L_0, K, A, E, I, u)$ re given bellow:

Locations [L]: We use locations to denote the states for processing services. The function $\Delta : \Sigma \rightarrow L$ constructs for an event e a location $\Delta(e)$ in L. The location is the state

for processing the event e. The set of locations L can be constructed with the help of Σ as follows:

- [L.1] Create an initial location l_0 to denote the *idle* state where the component is waiting for a stimulus.
- [L.2] For every stimulus event, create a location to represent the service of processing the stimulus.
- [L.3] For every output request event, create a location to represent the state of requesting that service.
- [L.4] For every action in the service definition, create a location to represent the state of processing the service action.

Clocks [K]: Time constraints in Γ can be represented by clocks in K and invariants representing clock constraints in I. The set of clocks K can be constructed by creating a clock for every time constraint that constrains the response of a stimulus. Clocks are defined as template's local variables.

Invariants [I]: Time constraints are defined as location invariants in I. We create an invariant in I for each time constraint in Γ and assign it to $\Delta(e)$ to form the set of invariants for that state, $I_{\Delta(e)}$. Also, the constraint that is defined for service request (stimulus) is added to I: $\forall x \in \Sigma,\ x = (\Lambda, \mathcal{A}, \chi, \varrho),\ \chi \in I \wedge \chi \in I_{\Delta(e)}$.

Actions [A]: The set of actions A can be constructed by creating an action in A for every stimulus and response in $\Sigma_{stimulus} \cup \Sigma_{response}$. Actions are defined as synchronous channels. Input actions are decorated with $?$ and output actions are decorated with $!$.

Edges [E]: The behavior of a component is based on stimuli and responses. Therefore, E can be constructed using Σ according to the rules [E.1], [E.2], and [E.3] defined bellow. The specifications of edge expressions are derived from the data parameters Λ and the constraints in Ω and Ψ that are related to the action a, which is the stimulus, service action, or response that causes the transition, according to the following rules [E.Ex]:

- *Select*: It is used to get a value in a temporary variable for each data parameter in $\xi(a)$. These values will be assigned to their corresponding data parameters in the *Update* expression. The data type of the parameter, $\mathcal{D}$, is used to specify the type of the temporary variable.

- *Guard*: A guard condition is a conjunction $\omega \wedge \Upsilon$ such that: $\omega \in \Omega$ is a predicates over the value of data parameters in $\xi(a)$ and of component, service, and reactivity attributes, and Υ is a service security policy related to a where $(a, r, \Upsilon) \in \Psi$.

- *Sync*: the action, which is the event causing the transition.

- *Update*: It includes assignment statements that: (1) update the value of data parameters in $\xi(a)$, (2) reset the clock in K related to the time constraint in Γ that is defined for a, and (3) update the value of component attributes. In order to ensure data security, update statements are constrained by Υ as follows:
  $\forall d \in \xi(a),\ d := \Upsilon?Select(d) : Null$, which means that if there is a data security property $\psi_d = (a, d, \Upsilon)$ associated with the data parameter then d will be assigned the selected value only if Υ evaluates to true; otherwise, d will be set to $Null$. Also, $\forall u \in \mathcal{U}, u := assign(\alpha, \nu) \wedge (a, \alpha, \Upsilon) \in \Psi \rightarrow \alpha := \Upsilon?Select(\alpha) : Null$ which means that in the update statements of the service, the value of the attribute will be set only if the security policy evaluates to true.

The following rules are used to construct template edges. Constructing edges is based on the service definitions in Θ; therefore, the following steps will be repeated for every service definition:

- [E.1] For every stimulus e, an edge from the initial location l_0 to $\Delta(e)$ is created. If there is a time constraint defined for the service then the defined clock of this time constraint should be reset in the update expression, as defined above in [E.Ex].
  After finishing the processing of e, the response is sent and the component can go back to idle state waiting for the next stimulus. Therefore, for every response, an edge from $\Delta(e)$ back to l_0 is created.

- [E.2] In order to provide the required services, the component may request services from other components. When a stimulus e has a response $r \in \Sigma_{OutRequest}$ then an edge from $\Delta(e)$ to $\Delta(r)$ is created and a second edge from $\Delta(r)$ to l_0 is also created
  .

- [E.3] If the stimulus has multiple possible responses where data constraints are used to select the proper response then an edge for each case is created and the data constraint defined in the service is added to the guard condition of the edge. The edge

is created from $\Delta(e)$ to: (1) I_0, if the response is internal or output response i.e. $r \in \Sigma_{internal} \cup \Sigma_{output}$, or (2) $\Delta(r)$, if r is an output request i.e. $r \in \Sigma_{OutResponse}$.

- [E.4] the component may have a concurrent behavior. It can receive stimuli while processing others. Therefore, an edge is created from every location that represents stimulus processing location l_{p1} to the other stimulus processing locations l_{p2}. An intermediate committed locations is used to split the edge into two edges: (1) an edge from l_{p1} to the committed location labeled with the stimulus and (2) an edge from the committed location to l_{p2} labeled with the response of l_{p1}. The reason for having two edges is that UPPAAL doesn't allow having two synchronous channels on an edge.

- [E.5] If there is one or more actions defined in the service i.e. $\mathcal{S} \neq \emptyset$ then a location $\Delta(x_i)$ is created for each action $x \in \mathcal{S}$ where $i : \mathbb{N}, 0 \leq i \leq n, n = |\mathcal{S}|$. Then, the edge created for the response (in E.1, E.2, and E.3) is pointed to $\Delta(x_1)$ and edges are created between $\Delta(x_1) \ldots \Delta(x_n)$. Then a final edge is created back to l_0.

After constructing each edge, the rules in [E.Ex] are used to define its expressions. All attributes are defined as local variables of the component.

6.1.3 Example

Figure 30 shows the extended timed automata generated for the following service definition:

$\Theta(s, \omega, \gamma, t_1, \psi_s, \{\psi_d\}) = (r, \{k := x\}, \{a\}, t_2, \{\psi_a\}, \{\psi_u\})$ where $\xi(s) = \{d\}$ and $\gamma = (s, r, 5)$

The construction is done as follows:

Locations: $idle$ is created according to rule [L.1], $\Delta(s)$ according to [L.2] and [E.5], the invariant $c \leq 5$ at $\Delta(s)$ according to [I], and $\Delta(a)$ according to [L.4].

Edges: created according to the following rules and [E.Ex]: (1) $(idle, s, \Delta(s))$ is created according to [E.1], (2) $(\Delta(s), r, \Delta(a))$ and $(\Delta(a), a, idle)$ is created according to [E.5].

Clocks: c and the invariant at $\Delta(s)$ are created according to rules [K] and [I].

Actions: $s?, r!$ are created according to [A] and [E.Ex]

The security properties, events security and data security, can be specified in UPPAAL language in the following way:

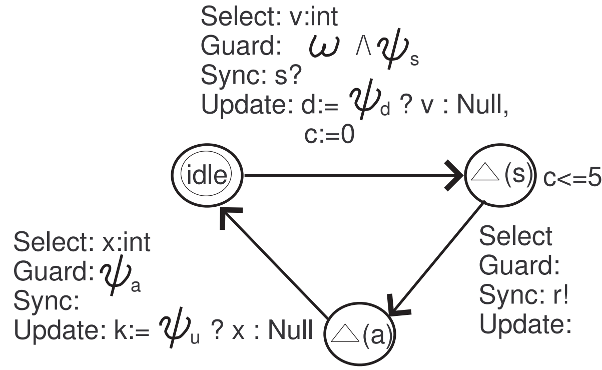


Figure 30: Example Transformation

- **Event security**: An event can be triggered only by a user who has access right. This is expressed as the CTL formula:
  $A\Box$ *for all(i:int[1,NoOfUsers]) C.user==i && $C.event_x$ imply US(i,$event_x$).* It means: invariantly, in all system executions, $event_x$ can be triggered by authorized users only. Each user is identified with a unique identifier.

- **Data security**: A data parameter value should be visible only to authorized users. This is expressed as the CTL formula:
  $A\Box$ *for all(i:int[1,NoOfUsers]) C.user==i && DataParameter!=Null imply UD(i,DataParameter).* It means: invariantly, in all system executions, the value of $DataParameter$ can be visible only to authorized users; otherwise, it is set to $Null$;

Figure 31 shows the extended timed automata generated for the controller component of the *fingerprint car security system* presented in Example 4.

6.1.4 Preserving the requirements of safety and security

Theorem 2 *The transformation rules preserve the requirements of safety and security ($[S1] \ldots [S9]$).*

The following proof shows that the transformation process preserves safety and security requirements:

Proof 2 *In order to proof that the transformation process preserves the safety and security requirements we need to show that the sequences generated by the UPPAAL extended*

Figure 31: The UETA of the controller component

timed automata (UETA) satisfy $[S1] \ldots [S9]$. This can be shown by proving that the transformation rules preserves $[S1] \ldots [S9]$. In this proof we use the rules $[S1] \ldots [S9]$ as the equivalency conditions. If these rules are satisfied in the component behavior and in the UETA then we say that the component behavior and its UETA are equivalent.

[S1] for every stimulus, there is exactly one response: *in the component model, services are defined by Θ. In UETA, transitions are defined as edges. The transformation rules [E.1] and [E.2] ensure that for every stimulus there is two edges and a state. The first edge is labeled with the stimulus and the second edge is labeled with the response. Therefore, every sequence generated by the timed automata will have exactly one occurrence of a response for each occurrence of a stimulus.*

[S2] a stimulus occurs before a response in any sequence: *since the edge created for a stimulus precedes the edge created for a response([E.1] and [E.2]), a stimulus always occurs before its corresponding response.*

[S3] time constraints are respected: *in the component model, time constraints are defined in Γ as part of the contract definition Ξ. In the UPPAAL template, a time constraint is created by defining a clock and assigning the time constraint as an invariant to a location. The transformation rules [K] and [I] ensure that a clock is defined for every time constraints and that the time constraint is assigned to the location that represents the state of processing the stimulus. Therefore, for every sequence generated by UETA, time constraints are enforced.*

[S4] data constraints are respected: *in the component model, data constraints are defined in Ω as part of the contract definition Ξ. In UETA, data constraints are defined as guard conditions restricting transitions. The transformation rules [E.Ex - Guard] and [E.3] ensure that a guard condition is defined over the edge that represents the response transition. Therefore, for every sequence generated by UETA, data constraints are enforced and only the proper responses are included in the sequences.*
[S5 and S7] event security properties are enforced: *in the component model, security policies are defined in Ψ and included in the service definition Θ. In UETA, event security policies are defined as guard conditions restricting transitions. The transformation rule [E.Ex - Guard] ensures that the security property is defined as part of the guard condition of the edge that represents the response transition. Therefore, for every sequence generated by UETA, security properties are enforced.*
[S6 and S8] data security properties are enforced: *in the component model, security policies are defined in Ψ and included in the service definition Θ. In UETA, data security policies are defined as conditions restricting the update statements of an edge. The transformation rule [E.Ex - Update] ensures that the data security property is defined as a condition on updating data values. Therefore, for every sequence generated by UETA, data security properties are enforced.*

Theorem 3 *The composition of two UPPAAL extended timed automata (UETA) preserves the safety and security requirements ($[S1] \ldots [S9]$).*

The following proof shows that the composition of two UPPAAL extended timed automata (UETA) preserves the safety and security requirements:

Proof 3 *let $M_1 = (L_1, l_{01}, K_1, A_1, E_1, I_1)$ and $M_2 = (L_2, l_{02}, K_2, A_2, E_2, I_2)$ be two UETAs. The composition of M_1 and M_2 is $M = (L, l_0, K, A, E, I)$ where: $L = L_1 \times L_2$, $l_0 = (l_{01}, l_{02})$, $k = K_1 \cup K_2$, $I = I_1 \cup I_2$ such that the invariant of a composite location is the conjunction of the invariants of all its constituent locations i.e. $\forall l = (l_1, l_2) \in L \bullet I(l) = I(l_1) \wedge I(l_2)$, and E is given as follows (*Edge Composition Rule*): an edge is defined as a tuple (l, a, g, r, u, l') where l is the initial location, a is an action, g is a guard condition, r resets clock values, u is an update expression, and l' is the destination location. For $(l_1, a_1, g_1, r_1, u_1, l'_1) \in E_1$ and $(l_2, a_2, g_2, r_2, u_2, l'_2) \in E_2$:*
if $a_1 = a_2$ then E includes $((l_1, l_2), a_1, g_1 \wedge g_2, r_1 \cup r_2, u_1 \wedge u_2, (l'_1, l'_2))$;

if $a_1 \notin A_1 \cap A_2$ then E includes $((l_1, l_2), a_1, g_1, r_1, u_1, (l'_1, l_2))$; and
if $a_2 \notin A_1 \cap A_2$ then E includes $((l_1, l_2), a_2, g_2, r_2, u_2, (l_1, l'_2))$.

Since M_1 and M_2 satisfy $[S1] \ldots [S9]$, we need to show that M satisfies those requirements:

[S1] for every stimulus, there is exactly one response: *since M_1 and M_2 satisfy [S1] then for every stimulus $a \in A$: $a \in A_1 \ \vee \ a \in A_2 \ \vee \ a \in A_1 \cap A_2$ and there is a location $l \in L_1 \vee l \in L_2 \vee L_1 \cup L_2$ that represents the state of processing a. Since $L = L_1 \times L_2$ then there exists a set of states $L_a \subset L$ such that $\forall x \in L_a \bullet (x = (l, y) \wedge l \in L_1, y \in L_2) \vee (x = (y, l) \wedge l \in L_2, y \in L_1) \ \vee \ (x = (l, l) \wedge l \in L_1 \cup L_2)$. Also, since M_1 and M_2 satisfy [S1] then there exists two edges $e_1 = (l_0, a, g, r, u, l), \ e_2 = (l, s, g', r', u', l') \wedge s \in \phi(a)$ such that $e_1, e_2 \in E_1 \ \vee e_1, e_2 \in E_2 \ \vee e_1, e_2 \in E_1 \cup E_2$ that represents receiving the stimulus and triggering response to it. From the* Edge Composition Rule*, there exists edges in E corresponding $e1, e_2$. Therefore, the composition preserves the reactivity requirement.*

[S2] a stimulus occurs before a response in any sequence: *in the previous proof, e_1, which is created for the stimulus, precedes e_2, which is created for the response; therefore, a stimulus always occurs before its corresponding response in M.*

[S3] time constraints are respected: *since M_1 and M_2 satisfy [S3], then there exists a clock $c \in K_1 \ \vee \ c \in K_2$ and an invariant $i \in I_1 \ \vee \ i \in I_2$ for every time constraint. Since $k = K_1 \cup K_2$, $I = I_1 \cup I_2$, and $\forall l = (l_1, l_2) \in L \bullet I(l) = I(l_1) \wedge I(l_2)$ then the composition preserves time constraints and every sequence in the composed model respects time constraints.*

[S4,S5,S7] data constraints and event security are respected: *since M_1 and M_2 satisfy [S3], then for every edge $e = (l, a, g, r, u, l'), \ e \in E_1 \ \vee \ e \in E_2 \ \vee e \in E_1 \cup E_2$ the guard condition g is a conjunction $\omega \wedge \Upsilon$. From the* Edge Composition Rule*, guard conditions are preserved in the composition. Therefore, data constraints and event security are preserved in the composition.*

[S6 and S8] data security properties are enforced: *since M_1 and M_2 satisfy [S3], then for every edge $e = (l, a, g, r, u, l'), \ e \in E_1 \ \vee \ e \in E_2 \ \vee e \in E_1 \cup E_2$ the update expression u contains the data security properties. From the* Edge Composition Rule*, update expressions are preserved in the composition. Therefore, data security are preserved in the composition.*

Model checking has been used effectively for verifying safety critical systems. However, model checking falls short when used for large and complex systems because it suffers inherently from the state-space explosion problem. This problem limits the application

of model checking to small size problems. Our formal approach provides a technique to overcome the state-space explosion problem by using incremental model checking. Primitive components can be verified for trustworthiness properties. Then, large and complex systems can be built by composing these components. There is no need to perform model checking on the composite component because the composition preserves the trustworthiness properties. Therefore, Theorem 3 provides an important contribution for incremental model checking. Thus, the model checking problem is tractable despite the size of the component-based system.

6.2 Real-Time Analysis

It is possible to conduct real-time scheduling analysis relative to criticality, priority, and other real-time non-functional properties based on the formal specification. Times [AFM$^+$03] is used to perform the real-time analysis. First, the formal specification are transform into timed automata extended with tasks, which is the language used by Times tool. Then, times tool performs the scheduling analysis.

Times uses a timed automata extended with tasks to model real-time systems. The structure of the timed automata is similar to the structure of UPPAAL timed automata. In addition, Times extends UPPAAL model by defining real-time tasks and their real-time attributes, which are explained bellow. In order to define real-time models, events must be annotated with real-time task attributes. These attributes are used to build the real-time characteristics of the service when executed at run-time. Therefore, we define a *standard set of attributes* that must be defined for every stimulus and response events. These attributes are:

- *Behavior*: an enumeration type whose value is: *Controlled*, *Periodic*, or *Sporadic*. This attributes specifies the execution behavior of the service type.
- *Priority*: an integer value specifying the priority of the service type.
- *Computing time*: an integer vale specifying the computation time of a service, the total time required for the service to finish executing.
- *Deadline*: the maximum safe time before which the service must finish its execution. It is similar to the specified time constraint.

- *Period*: this attributes specifies the time before two consecutive executions of the same service type.
- *Offset*: this attribute specifies the variable time allowed for the service to start after its period time occurs.
- *Max number of tasks*: this attribute specifies the number of concurrent instances of the same task.

Since Times model is similar to UPPAAL model, the transformation process follows the same rules as those specified earlier for UPPAAL with few added rules that are related to the new concept of *Task*. These are:

- Defining tasks: a task is created for every stimulus.
- Assigning attributes: the service attributes are mapped into task attributes.
- Generating templates: templates are created in the same way used for UPPAAL including their timed automata. Note that some templates are environmental templates; therefore, an attribute is used at the component type level to specify if a component is environmental or not.
- Global declaration: similar to the former transformation rules of UPPAAL, global declarations are added to the system.
- Instantiation: an instance for each template is created and added to the system.
- Local declaration: a local declaration is created for each template, same as UPPAAL.
- Assigning tasks to locations: for every stimulus, its task is associated with the location which represents the state of processing the stimulus.
- Deciding policy: a scheduling policy is selected.

6.3 TADL Semantics

In this section we give an argument as to why TADL has a formal semantics. The semantics basis of the formalism is grounded on *set theory*. Let the transformation from the

formal model to UPPAAL formal model, described in this Chapter, be called f. The implementation of f has a TADL file as input. Therefore, we have Figure 32 where u is the automatic transformation developed within UPPAAL model checker and g is the transformation described in Chapter 5. Therefore, for every formal model m we have the equation $f(g(m)) = u(f(m))$. Since f and u are both sound and complete, it follows that g is sound and complete. That is, TADL has a set theoretical formal semantics.

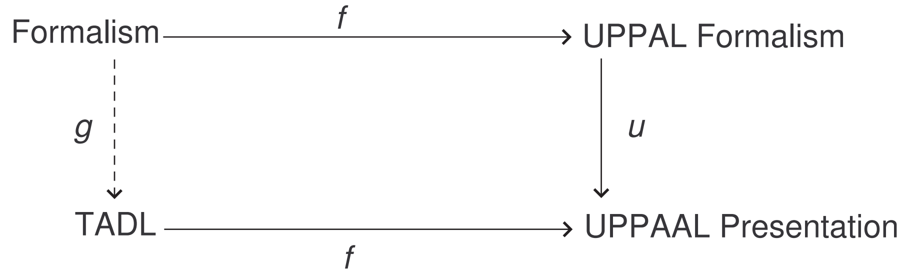


Figure 32: TADL Semantics

6.4 Summary

This chapter introduced a model transformation technique for analyzing systems built using our trustworthy component model and generating two types of extended timed automata. One type is suitable for UPPAAL model checker and another type is suitable for the Times tool. Formal transformation rules were provided to describe how the extended timed automata is generated from the formal component specification. This approach allows us to formally verify the requirements of safety, security, and real-time systems using model checking. A model transformation tool [Ibr08] has been developed to automatically generate UPPAAL and Times timed automata from TADL specification based on the transformation rules. Detail description of the tool is available in Chapter 9.

Chapter 7

Reliability and Availability

This section introduces a novel formal approach for specifying and verifying reliability and availability properties using model checking. The section includes modeling service failures and repairs. An example case study is provided to explain our approach.

7.1 Service Failures and Repairs

Reliability and availability are two related trustworthiness attributes. In the literature [ALRL04], reliability is defined as the "continuity of correct servic", whereas availability is defined as the "readiness for correct service". A service *failure* is defined as a deviation from the correct service behavior [ALRL04]. The deviation is defined with respect to the required functional and non-functional requirements of the system. We assume that the requirements have been reviewed and validated at the system analysis phases. Therefore, the stated specification is used as the basis for deciding whether or not the behavior complies with the required and stated specification. A service has functional and non-functional requirements. While in operation, the service can change the values of attributes and local variables. It can also trigger events. A service failure is indicated by any violation to the requirements of safety and security stated in ($[S1] \ldots [S9]$):

1. If a service request arrives but no response is triggered;
2. If a service request arrives but an event other than the expected response is triggered;
3. If the time of the response precedes the time of the request;
4. If a service delivers results later than the maximum safe time;

5. If a service request violates a data constraint (precondition);

6. If one of the constraints associated with data parameters or events is violated;

7. If service security is violated; and

8. If data security is violated.

These failures can be categorized into 4 classes: *reaction*(1 and 2), *timing*(3 and 4), *condition*(5 and 6), and *security*(7 and 8) failures. A service failure has the following attributes:

- Class: an enumerated value that defines the class of a service failure (*reaction*, *timing*, *condition*, *security*);

- Conditions: a set of logical expressions that define the situation leading to the failure;

- Severity: an enumerated value that classifies the consequences of a service failure (*critical*, *minor*);

- Persistence: an enumerated value that classifies the duration of a service failure (*permanent*, *transient*); and

- Acceptable frequency: a pair (ν, t) where ν defines the number of occurrences of a service failure during time t. This pair defines the acceptable frequency of service failure. $(0, \infty)$ means that the service failure is not accepted at all.

We define the set of service failures:
$F = \{f = (\Sigma_{stimulus}, \Sigma_{response}, Class_f, Conditions, Severity, Persistence, Frequency)\}$

A service *repair* is defined as a change from incorrect service to correct service. A repair can be *internal* or *external*. Internal repairs are done by internal events automatically when a failure is detected. An external repair is done by an external entity such as human or system control component. An external repair can be part of system *maintenance*. Maintenance includes all modifications to the system throughout its deployment and execution phase [ALRL04]. A service repair has the following attributes:

- Class: an enumerated value that defines the class of a service repair (*internal*, *external*);

- Actions: a set of events executed to remedy the effect of a service failure;

- Updates: a set of update statements to reset the values of attributes affected by the service failure;
- Time to recover: an integer value specifying the mean time to recover the service failure starting from the time in which a failure is detected; and
- Type of recover: an enumerated value that states the type of recovery (*Full*, *partial*). Full recover means that the failure has been remedied and the service will return back to correct behavior. Partial recover means the service will continue to work in degraded mode, during the failure period, until a full recover repair arrives.

We define $R = \{r = (Class_r, Action, Updates, Type, Time)\}$ as a set of service repairs.

7.2 Defining Reliability and Availability

The acceptable level of reliability is defined based on the frequency and severity of service failures. The acceptable level of availability is defined based on the duration of service failure time. In order to assure trustworthy services, there should be a repair or set of repairs defined for each failure. Repairs are reactions to failures aimed to recover failures and return to correct service behavior. The component implementation must guarantee the failure-repair relations and acceptable levels of reliability and availability of services. The component implementation and maintenance must guarantee the repair time. The acceptable levels of reliability and availability should be added to the component contract. Then, operational profiles are used to assess the validity of the values specified in the contract. Reliability and availability are formally defined in Definition 10.

Definition 10 *We define a total function* FR : $F \rightarrow \mathbb{P}R$ *that associates a set of repairs to every failure such that* $\forall f \in F, \mathrm{FR}(f) \neq \emptyset$.

The following defines reliability and availability requirements: let $f_1, f_2 \in F$ be service failures and $rp_1, rp_2 \in R$ be service repairs such that *FR*$(f_1) = \{rp_1\}$, *FR*$(f_2) = \{rp_2\}$

- Reliability is defined as a set of invariants Re where each invariant is a logical expression defined over the severity and frequency of service failures. For example the reliability invariant *frequency*$(f_1) \leq 5/100$ means that the occurrence frequency of the service failure f_1 should be less than or equal to 5 times every 100 units of time. Another reliability invariant example is $severity(f_2) = critical \rightarrow$ *frequency*$(f_2) \leq$

$1/100$ which means that if a service failure f_2 has a $critical$ severity then the occurrence frequency of f_2 should be less than or equal to once every 100 units of time.

- Availability is defined as a set of time constraint Av where each time constraint specifies the maximum time allowed between the occurrence of a failure and its corresponding repair. For example, $t(rp_1) - t(f_1) \leq 5$ means that the service repair rp_1 should occur within 5 units of time from the occurrence time of the service failure f_1, where t(.) means the occurrence time of the failure or repair. This ensures that the service will be available within 5 units of time in case of failure.

The contract definition of components is extended to include the requirements of reliability and availability.

Definition 11 *Let F be the set of service failures, R be the set of service repairs, Re be the set of reliability invariants, Av be the set of availability time constraints, and* FR *be the total function that maps failures to repairs. The contract definition is extended to include reliability and availability as follows* $\Xi = (\Theta, \Omega, \Gamma, \mathcal{P}, \Psi, F, R, Re, Av, \text{FR})$.

7.3 Verifying Reliability and Availability

Figure 33: The process of modeling and verifying reliability and availability

Figure 33 presents our qualitative approach to specify and verify reliability and availability properties. The process includes the following steps:

1. The failure and repair specifications are formally defined. These information are part of the system requirements. Domain knowledge helps extrapolate these requirements.

2. The UPPAAL extended timed automata for each component is extended to include not only the correct service behavior but also the failure and repair behavior. A service failure is a transition from a state of processing the service request to the state

that represents the service failure. A service partial repair is a transition from a state of service failure to a state of partial repair. A service full repair is a transition from a state of service failure or partial repair to a state of correct service behavior. The destination of the transition depends on the actions specified in the repair specification tuple. Clock variables are used to model availability time constraints. Local variables are used to hold the frequencies of failure occurrences.

3. The requirements of reliability and availability are specified as UPPAAL CTL formulas.

4. UPPAAL model checker is used to formally verify that component's behavior satisfies the requirements of availability and reliability.

The next section provides an example case study that illustrates this process.

7.4 Steam Boiler Controller Case Study

The steam boiler controller case study [ABL96] is a benchmark case study for modeling real-time systems. We adopt a simplified component-based version of the case study to explain reliability and availability modeling.

7.4.1 System specification

The steam boiler controller system consists of hardware and software components. Hardware components are: (1) A *steam boiler* characterized by two safe limits of water: minimum (*min*) and maximum (*max*). The minimum value indicates the lowest safe level of water under which the steam boiler will be in danger. The maximum value indicates the maximum safe level of water above which the steam boiler may be in danger. (2) A water *pump* to pour water inside the steam boiler in order to increase the level of water. (3) A *valve* to evacuate water from the steam boiler in order to reduce the level of water. (4) A *water level measuring sensor* which is continuously measuring the current quantity of water ($q - water$) inside the steam boiler. (5) A *steam level measuring sensor* which is continuously measuring the current quantity of steam ($q - steam$) coming out of the steam boiler.

Software components are: (1) A controller component responsible for maintaining a safe level of water inside the steam boiler. (2) A control monitoring component responsible for monitoring and managing the system in cases of failure.

7.4.2 System operation

When the system is initialized, the controller sends a stimulus $Program_Ready$ to check if all hardware components are ready. If all hardware components are ready then the controller will receive a response from each hardware component within 5 units of time. In this case the controller will set the value of the local attribute *operational mode* to $Normal$ and be ready to receive stimulus from sensors. However, if one of the hardware components is not ready then the system will operate in failure modes as explained later.

If the controller is in the normal operating mode then the water level measuring sensor reads the current quantity of water ($q - water$) inside the steam boiler, and the steam level measuring sensor reads the current quantity of steam $q - steam$. They send stimulus to the controller component informing it about the current quantities. To simplify the requirements, we assume that one stimulus $Level$ will carry the readings of the two sensors. The stimulus is parameterized by $q - water$ and $q - steam$. The controller component should react to the stimulus within 5 units of time. The reaction depends on the value of $q - water$. If the value is bigger than the maximum allowed i.e. $q - water > max$, the controller will send a stimulus to instruct the valve to open and evacuate water. However, if the value is less than the minimum allowed i.e. $0 \leq q - water < min$, the controller will send a stimulus to instruct the pump to open and pour water inside the steam boiler. If the value is within the safe limit i.e. $min \leq q - water \leq max$, the controller does nothing and waits for the next stimulus.

7.4.3 Failure and repair operations

In this simplified version, we define the following failures:

F1 : If the controller sends $Program_Ready$ to the hardware components and does not receive $Pump_Ready$, $Valve_Ready$, $Water_Ready$, or $Steam_Ready$ within 5 units of time then this indicates that there is a failure in the pump, valve, water level measuring sensor, or steam measuring sensor. In this case there are three possible

repairs R1, R2, and R3 which switch the operational mode according to the following conditions:

R1 : if the responses $Pump_Ready$, $Valve_Ready$, or $Steam_Ready$ are not received within 5 units of time since $Program_Ready$ is sent then an internal repair switches the controller to $Emergency_Stop$ mode. This is a partial internal repair.

R2 : if the responses $Pump_Ready$, $Valve_Ready$, and $Steam_Ready$ are all received within 5 units of time but $Water_Ready$ is not received then an internal repair switches the controller to $Rescue$ mode. In this mode, the value of $q - steam$ is used to estimate the quantity of water inside the boiler. This is a partial internal repair.

R3 : if the controller is in $Emergency_Stop$ then a full external repair fixes the defective hardware components and switches the mode to $Initialize$.

F2 : The water level measuring should always send a positive non-zero value indicating the current quantity of water. If $q - water < 0$ then the sensor is malfunctioning and it indicates a failure. In this case, if all other hardware components are working properly then an internal partial repair switches the mode into $Rescue$. The full external repair R3 fixes the defective water level measuring and switches the controller back to $Initalize$.

F3 : The steam level measuring should also send a positive value indicating the current quantity of steam. If $q - steam \leq 0$ then the steam measuring sensor is malfunctioning and it indicates a failure. In this case an internal partial repair switches the controller to $Emergency_Stop$ mode.

Local variables (attributes) are used to hold the current status of each hardware component. These are ***PumpOK, ValveOK, SteamOK,*** and ***WaterOK***. Initially these attributes are set to zero to indicate that the ready response is not received yet from each hardware component. When a ready response is received, its corresponding attribute is set to 1 to indicate that it has responded. If 5 units of time passed and the value of an attribute is still 0 then this indicates that no response was received.
The set of failures is $F = \{f_1, f_2, f_3, f_4, f_5, f_6\}$ where:
$f_1 = (Program_Ready,\ \{Pump_Ready\},\ condition,$

$(t(Pump_Ready) - t(Program_Ready) > 5 \wedge PumpOK = 0)$, $critical$, $transient$, $(F1_{max}, F1_{time}))$;
$f_2 = (Program_Ready$, $\{Valve_Ready\}$, $condition$,
$(t(Valve_Ready) - t(Program_Ready) > 5 \wedge ValveOK = 0)$, $critical$, $transient$, $(F1_{max}, F1_{time}))$;
$f_3 = (Program_Ready$, $\{Steam_Ready\}$, $condition$,
$(t(Steam_Ready) - t(Program_Ready) > 5 \wedge SteamOK = 0)$, $critical$, $transient$, $(F1_{max}, F1_{time})$;
$f_4 = (Program_Ready$, $\{Water_Ready\}$, $condition$,
$(t(Water_Ready) - t(Program_Ready) > 5 \wedge SteamOK = 1)$, $critical$, $transient$, $(F1_{max}, F1_{time}))$;
$f_5 = (Level, \emptyset, condition, (q - water < 0), critical, transient, (F2_{max}, F2_{time}))$;
$f_6 = (Level, \emptyset, condition, (q - steam \leq 0), critical, transient, (F1_{max}, F1_{time}))$.

For example, f_1 defines a failure that occurs due to the following scenario: the controller issues $Program_Ready$ and resets a clock to measure the time passed starting from $Program_Ready$ until it receives the response $Pump_Ready$. If 5 units of time passed and the controller did not receive $Pump_Ready$, which means the value of the attribute $PumpOK$ is still set to 0, then the controller will assume that the pump is not working and a failure is detected. The severity of this failure is critical and the acceptable frequency is indicated by the constant $F1_{max}$ during $F1_{time}$ units of time. The other failures can be explained in the same manner.

The set of repairs is $R = \{rp_1, rp_2, rp_3, rp_4\}$ where:
$rp_1 = (internal, \emptyset, \{mode := Emergency_Stop\}, Partial, 1)$;
$rp_2 = (internal, \emptyset, \{mode := Rescue\}, Partial, 1)$;
$rp_3 = (external, Fix_Level, \{mode := Normal\}, Full, 100)$; and
$rp_4 = (external, Fix_Units, \{mode := Initialize\}, Full, 100)$.

The first two repairs are internal partial repairs, no action assigned to them. They switch the operational mode of the controller to either $Emergency_Stop$ or $Rescue$ within one unit of time. The third and fourth repairs are external events issued by the control monitoring component to indicate that the defective hardware components have been fixed. The component stays in a repair state until a full repair is received. The associations between the failures and repairs are:
$FR(f_1) = FR(f_2) = FR(f_3) = FR(f_6) = \{rp_1, rp_4\}$;

$\mathit{FR}(f_4) = \mathit{FR}(f_5) = \{rp_2, rp_3\}$;

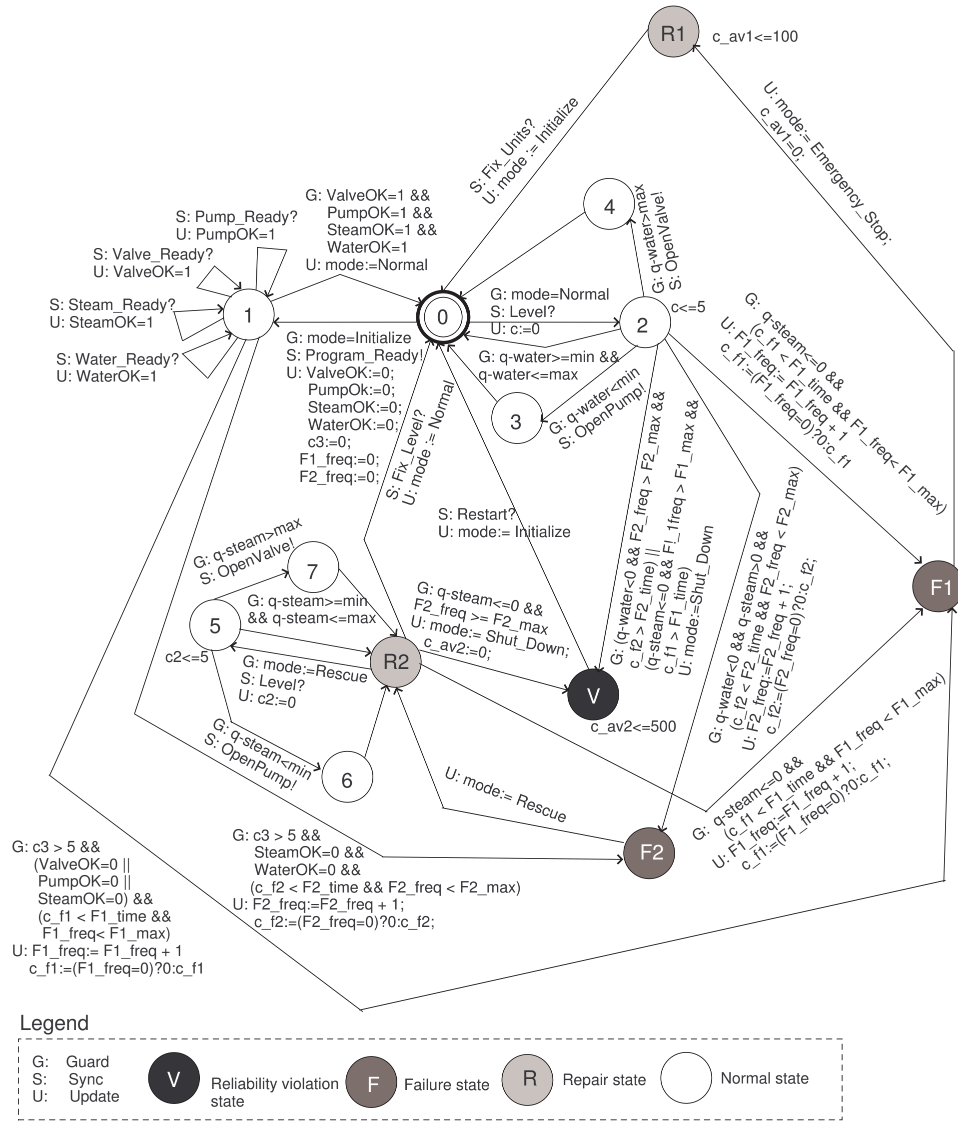


Figure 34: The UETA of the steam-boiler controller component

Figure 34 depicts the UPPAAL extended timed automata of the controller component. In this Figure, state 0 is the initial state. States 1,2,3, and 4 represent the correct behavior of the component. Since there are two equivalent types of associations between failures and repairs, we create two states F1 and F2 that represent the occurrence of failures and two states R1 and R2 that represent the occurrence of repairs corresponding to the failures. There are 4 transitions that indicate the occurrence of a failure: (1) from state 1 to F1 if f_1, f_2, or f_3 occurs, (2) from state 1 to F2 if f_4 occurs, (3) from state 2 to F2 if f_5 occurs,

and (4) from state 2 to F1 if f_6 occurs. There are 4 transitions that indicate the occurrence of a repair: (1) from state F1 to R1 when rp_1 occurs, (2) from state F2 to R2 when rp_2 occurs, (3) from state R1 to 0 when rp_4 occurs, and (4) from state R2 to 0 when rp_3 occurs. There is one state V which represents the state of violating reliability requirement. There are two transitions to state V: (1) from R2 to V if the occurrence frequency of failure f_6 exceeds the maximum allowed number within the maximum allowed time period, and (2) from state 2 to V if the occurrence frequencies of failures f_5 or f_6 exceed the maximum allowed number. The transitions to failure states are guarded by conditions that check if the occurrence frequencies of failures exceed the maximum allowed numbers. For example, the transition from state 2 to state F1 has the following guard condition: $q - steam \leq 0 \wedge (c_{f1} < F1_{time} \wedge F1_{freq} < F1_{max})$ this means that if the value of steam quantity is invalid and if the frequency of failure F1 ($F1_{freq}$) has not exceeded the maximum allowed number $F1_{max}$ within $F1_{time}$ then F1 occurs; otherwise, if F1 exceeds those limits then the system moves to V states indicating a violation to the stated reliability requirement. The clock variable c_{f1} is used to keep track of the time since the first occurrence of F1. Similar conditions are used for F2. Figure 34 includes two availability requirements: (1) the full repair rp_4 should occur within 100 units of time from the occurrence of the failure. This is specified by the time constraint $c_{av1} \leq 100$ which is associated with the repair state R1 indicating that the system should move from this state within 100 units of time. (2) the system should be restarted within 500 units of time after the violation of reliability requirements. This is specified by the time constraint $c_{av2} \leq 500$ which is associated with the V state.

Reliability and availability properties can be stated as CTL formulas in UPPAAL as follows:

- Reliability: "$F1 \rightarrow F1_{freq} < F1_{max}$ && $c_{f1} < F1_{time}$", which means if failure F1 occurs then the frequency is less than the maximum allowed and the value of the clock variable associated with the failure is within the acceptable period of time.
- Availability: "$F1 \rightarrow state0$ && $c_{av1} \leq 100$", which means if failure F1 occurs then the system will recover and move back to the initial state within 100 units of time i.e. the services will be available within 100 units of time after the occurrence of the failure.

Therefore, using the extended state machine depicted in Figure 34, along with the other state machines of the other components, and the stated reliability and availability properties,

it is possible to use the UPPAAL model checker to verify the availability and reliability of the steam boiler controller case study. This shows that it is possible to use model checking techniques to provide a qualitative approach to ensure reliability and availability.

7.5 Summary

This chapter provided an extension to our trustworthy component model. Formal definitions of reliability and availability were provided based on the failure and repair models. A model checking technique is used to verify reliability and availability properties.

Chapter 8

Process Model for Developing Trustworthy Systems

This chapter introduces our proposed rigorous process model for developing trustworthy systems. The process is based on the formal component model which is introduced in Chapter 4, the TADL which is introduced in Chapter 5, the model transformation process which is introduced in Chapter 6, and the specification and verification of reliability and availability which is introduced in Chapter 7. The process integrates these formal methods in the phases of systems life-cycle. In particular, it incorporates incremental design using TADL, validation and formal verification using our model checking technique, iterative development, traceability analysis, and certification.

The entire development process is divided into several tracks that can run in parallel. The tracks are domain engineering, component development, component assessment, component reuse, and system development. Figures 35 and 40 depict the rigorous development process tracks. Figures 35 presents the domain engineering and component development, assessment, and reuse tracks. Figure 40 presents the system development track. The activities along these tracks are explained in the following sections.

8.1 Domain Engineering

We propose an ontology-based approach to domain engineering which consists of two phases: building ontology and deriving components and component-based systems specifications from it.

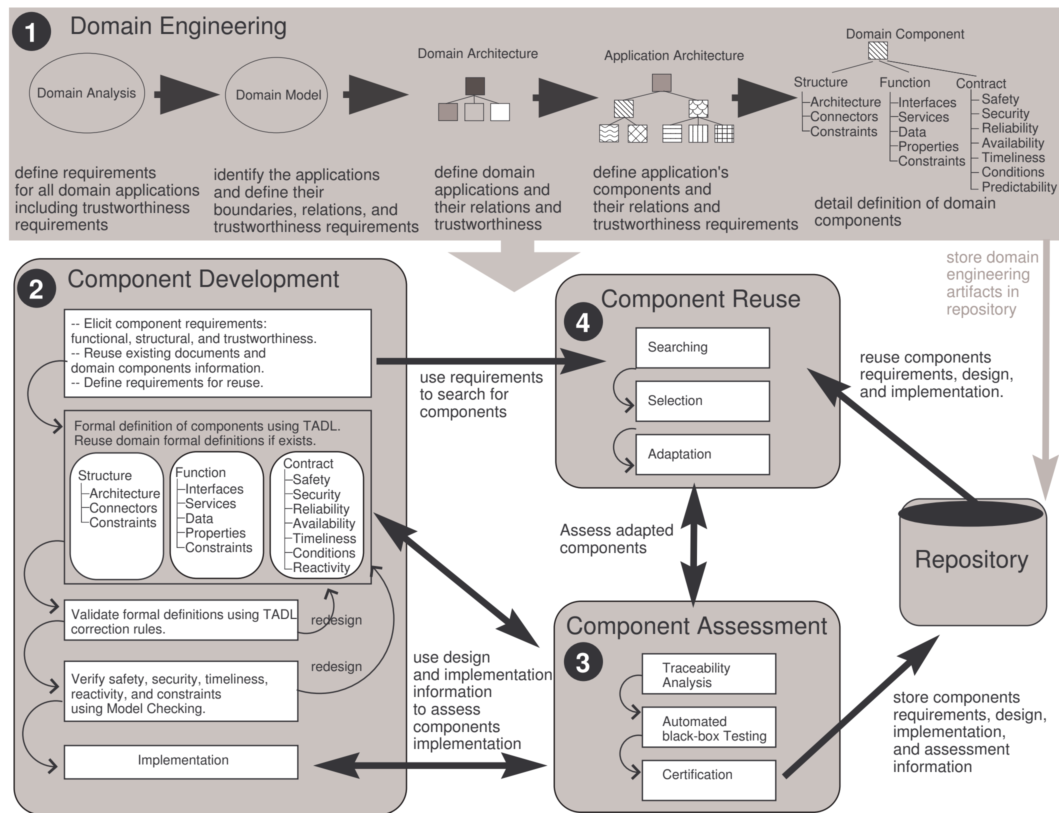


Figure 35: Domain Engineering and Component Engineering (development, assessment, and reuse)

The goal of domain engineering is to define, model, construct, and catalogue a set of artifacts that can be reused in all applications within a specific domain [Pre05]. Thus, domain analysis involves identification and analysis of the applications, their detailed requirements, and the relations and data that exist in a specific domain. For example, the domain of automotive industry deals with designing, manufacturing, and marketing motor vehicles. A car, for example, contains many control systems such as *cruise control*, *cooling and heating*, *stability control*, *anti-lock braking*, and *fingerprint-based security systems*. Domain analysis aims towards understanding each system, its interactions with other systems, the constituent components in the system, their functional and non-functional requirements, and the data and events stored and communicated between them. For example, the car control systems mentioned above share the usage of smart sensors which collect relevant data about the current status of the car and use it to perform control actions. The results of the domain analysis is a domain model which consists of knowledge about the domain and

all its applications and its reusable components. This knowledge can be stored in a knowledge base. Domain analysis plays a key role in software reuse, which has been recognized earlier on in the literature [Nei84].

An Ontology is a ”content theory about the sorts of concepts, their properties, constraints, and the relations between concepts that are possible in a specified domain of knowledge”[CJB99]. It provides terms for describing the knowledge about a domain capturing the intrinsic conceptual structure of the domain[CJB99]. Building ontologies is a major approach for capturing and representing reusable knowledge. Many methodologies, tools, and languages are available for building and maintaining ontologies [CFLGP03]. In order to allow sharing and reusing ontologies, a common ontology language was developed and named *ontology web language* (OWL) [SWM04, GHM+08]. OWL allows representation of and reasoning about ontologies. Reasoning involves: (1) syntax checking, (2) consistency checking, ensuring that the ontology doesn’t contain contradictory facts (3) subsumption, checking whether a class description is more general than another class description, and (4) query answering, retrieving knowledge from the knowledge base.

Both ontologies and domain models are forms of models that result in detailed specifications of reusable knowledge. The former produces detailed specifications of reusable concepts and their relations and the later, when applied to component-based development, produces detailed specification of reusable components and component-based architectures. Achieving efficient component-based development depends on building an appropriate domain ontology. The ontology can be used as a basis for specification and development of domain applications. The captured conceptualization and relations should be formally specified. OWL can be used to formally represent the results of domain analysis. Consequently, this enables mapping the OWL ontology formalization into TADL. Thus, our proposed ontology-based approach for domain engineering consists of the following steps:

- Design an ontology for representing the knowledge captured during domain analysis.
- Represent in a precise and unambiguous way the elements that model the existing domain entities. Tools such as *Protege* [Pro], which provides a graphical user interface, can be used to capture the ontology of domain model and communicate it with domain experts. The tool uses OWL to describe the ontology.

- Analyze the resulting ontology and map its concepts and relations into TADL. Mapping occurs between OWL language constructs and their relevant TADL constructs. Details about this mapping are explained later in this section.

- Visualize the TADL model using a graphical editing tool.

The following example is used to illustrate our approach. We use the fingerprint-based car security system example, which was introduced earlier on, with slight modification.

Example 5 *Consider a* fingerprint-based car security system *mounted on the door of a car. The system consists of two entities: (I) a remote control which comprises a* biometric sensor *that collects user fingerprint and* buttons *that trigger the required actions such as starting the car and locking/unlocking the doors, and (II) a controller which is responsible for starting the car, locking, and unlocking the doors. The functional requirements includes:* start the car*,* lock*, and* unlock *doors. The security requirements state that only authorized drivers have access to these functions. The safety requirements state that the doors must be locked/unlocked within 1 unit of time. Reliability requirements states that if the remote control fails to lock/unlock then it must be possible to luck/unlock manually using the car key.*

The basic constructs that define an ontology are the *concepts*, *properties*, and instances of concepts (*individuals*). Properties are binary relations on individuals. The first step in designing an ontology is defining what these constructs represent. Figure 36 depicts our proposed abstract design of ontology for capturing domain analysis. In this Figure, rectangles represent concepts, arrows represent properties which model relations between concepts, and dash-arrows indicate that a concept is *sub-class-of* another concept. The domain consists of multiple applications where each of which consists of entities that perform functions and are restricted by non-functional requirements. An entity can be composed of multiple entities. This is indicated by the *is-part-of* relation. Also, an entity can be sub-class-of another entity, which means that any instance of the sub-class is an instance of the super-class. Non-functional requirements include safety, security, reliability, availability, and other non-functional requirements.

Figure 37 presents an ontology for Example 5. It is an instance of the abstract ontology model presented in Figure 36. In this example, the domain is car and it comprises multiple applications: anti-lock brake, stability control, cruise control, and fingerprint security. The

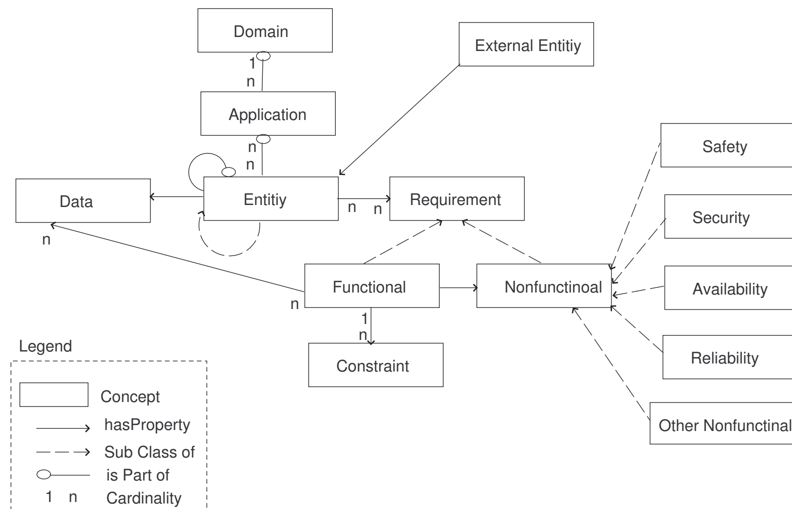


Figure 36: An ontology for domain analysis

example focuses on the fingerprint security application and shows its entities. Two individuals instantiated from the entity concept are shown in the figure: remote control and controller. The controller contains individual requirements instantiated from the functional and non-functional requirement concepts. Relations between individuals are represented by properties. Two kinds of properties exist in the model: *has-property* and *request-property*. For example, the controller has five functional requirements: lock, unlock, start car, manual lock, and manual unlock. These functional requirements has some non-functional requirements. For example, the lock function has a security requirement (*LockDoorSecurity*) and a safety requirement (*LockUnlockOnTime*). The request-property relation relates an individual of type entity or functional to and individual of type functional to indicate that the former is requesting the function provided by the later. When creating this ontology example in the Protege tool, the OWL language specification are generated automatically. Figure 38 shows part of the OWL language generated for the example. It focuses on the controller entity and its functional and non-functional requirements and their relations.

TADL specification can be generated by mapping OWL language constructs to TADL constructs as follows:

- Entities are mapped to *components*. The *part-of* relation between entities is mapped to composite components where a component consists of multiple constituent components. Note that the *sub-class-of* relation is not supported in the current version of TADL.

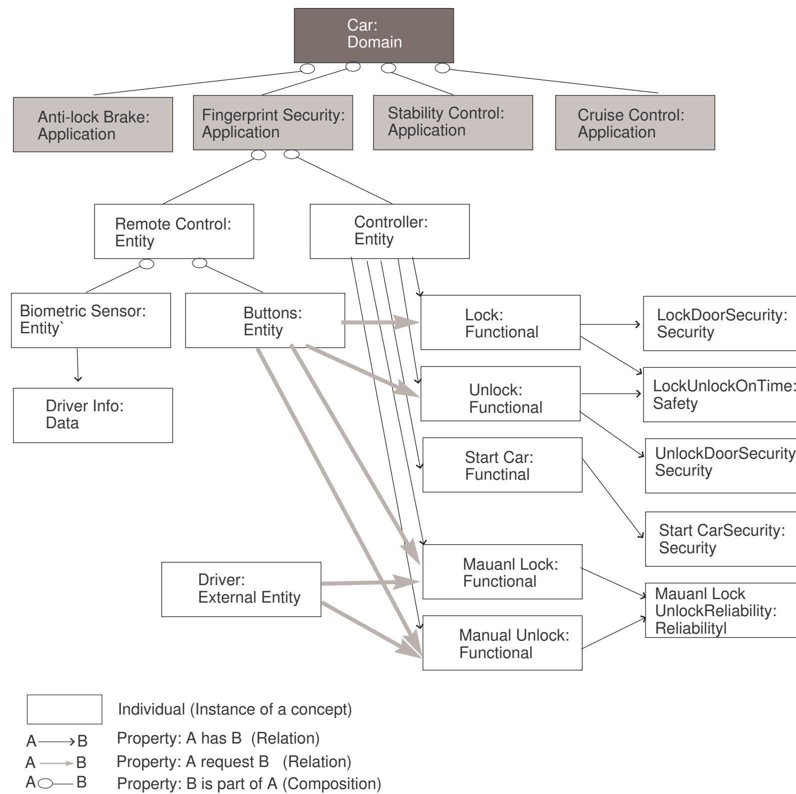


Figure 37: Car ontology example focusing on the fingerprint security system

- Data are mapped to *attributes*. An attribute is a data element that can be associated with any construct in TADL.

- Functional requirements are mapped to *services*. For every functional requirement, a service is created in TADL. Also, two events are created for each service: a request for service and a response of the service. The *has-property* and *request-property* relations help identify which component is providing the service and which components are consuming it. A service is provided by the component which is related to the functional requirement by the has-property relation. An interface is created for each component. The request and response events are associated with this interface. The services provided and consumed by the component are provided and requested at this interface. A connector is created for every request-property relation to provide a means to communicate requested and provided services. If two components are related by multiple service requests then it is sufficient to create one connector for the communication between the two components.

```
// some classes
<owl:Class rdf:about="#Functional">
    <rdfs:subClassOf rdf:resource="#Requirements"/>
    <owl:disjointWith rdf:resource="#Non_Functional"/>
</owl:Class>
<owl:Class rdf:about="#Non_Functional">
    <rdfs:subClassOf rdf:resource="#Requirements"/>
</owl:Class>
<owl:Class rdf:about="#Reliability">
    <rdfs:subClassOf rdf:resource="#Non_Functional"/>
</owl:Class>
<owl:Class rdf:about="#Requirements">
    <rdfs:subClassOf rdf:resource="&owl;Thing"/>
</owl:Class>
<owl:Class rdf:about="#Safety">
    <rdfs:subClassOf rdf:resource="#Non_Functional"/>
</owl:Class>
<owl:Class rdf:about="#Security">
    <rdfs:subClassOf rdf:resource="#Non_Functional"/>
</owl:Class>
//some  Individuals
/////////////////
<Entity rdf:about="#Controller">
    <hasRequirements rdf:resource="#Lock"/>
    <hasRequirements rdf:resource="#ManualLock"/>
    <hasRequirements rdf:resource="#ManualUnlock"/>
    <hasRequirements rdf:resource="#StartCar"/>
    <hasRequirements rdf:resource="#Unlock"/>
</Entity>
<Functional rdf:about="#Lock">
    <hasSecurityProperty rdf:resource="#LockDoorsSecurity"/>
    <hasSafetyProperty rdf:resource="#LockUnlockOnTime"/>
</Functional>
<Security rdf:about="#LockDoorsSecurity"/>
<Safety rdf:about="#LockUnlockOnTime"/>
<Functional rdf:about="#ManualLock">
    <hasReliabilityProperty rdf:resource="#ManualLockUnlockReliability"/>
</Functional>
<Reliability rdf:about="#ManualLockUnlockReliability"/>
<Functional rdf:about="#ManualUnlock">
    <hasReliabilityProperty rdf:resource="#ManualLockUnlockReliability"/>
</Functional>
<Functional rdf:about="#StartCar">
    <hasSecurityProperty rdf:resource="#StartCarSecurity"/>
</Functional>
<Security rdf:about="#StartCarSecurity"/>
<Functional rdf:about="#Unlock">
    <hasSafetyProperty rdf:resource="#LockUnlockOnTime"/>
    <hasSecurityProperty rdf:resource="#UnlockDoorsSecurity"/>
</Functional>
<Security rdf:about="#UnlockDoorsSecurity"/>
```

Figure 38: OWL specification of the controller concept

- Non-functional requirements are used to define the *contract* of each component. The contract contains services, safety, security, reliability, availability, and any other non-functional requirements. A one to one mapping occurs between elements of these types of non-functional requirements. For example, a safety property is created in the component contract for every safety requirement in the ontology. A manual intervention is required in this step of generating TADL. Domain experts should translate the non-functional requirements, such as safety, to their corresponding representation in TADL, such as *first order predicate logic* expressions.

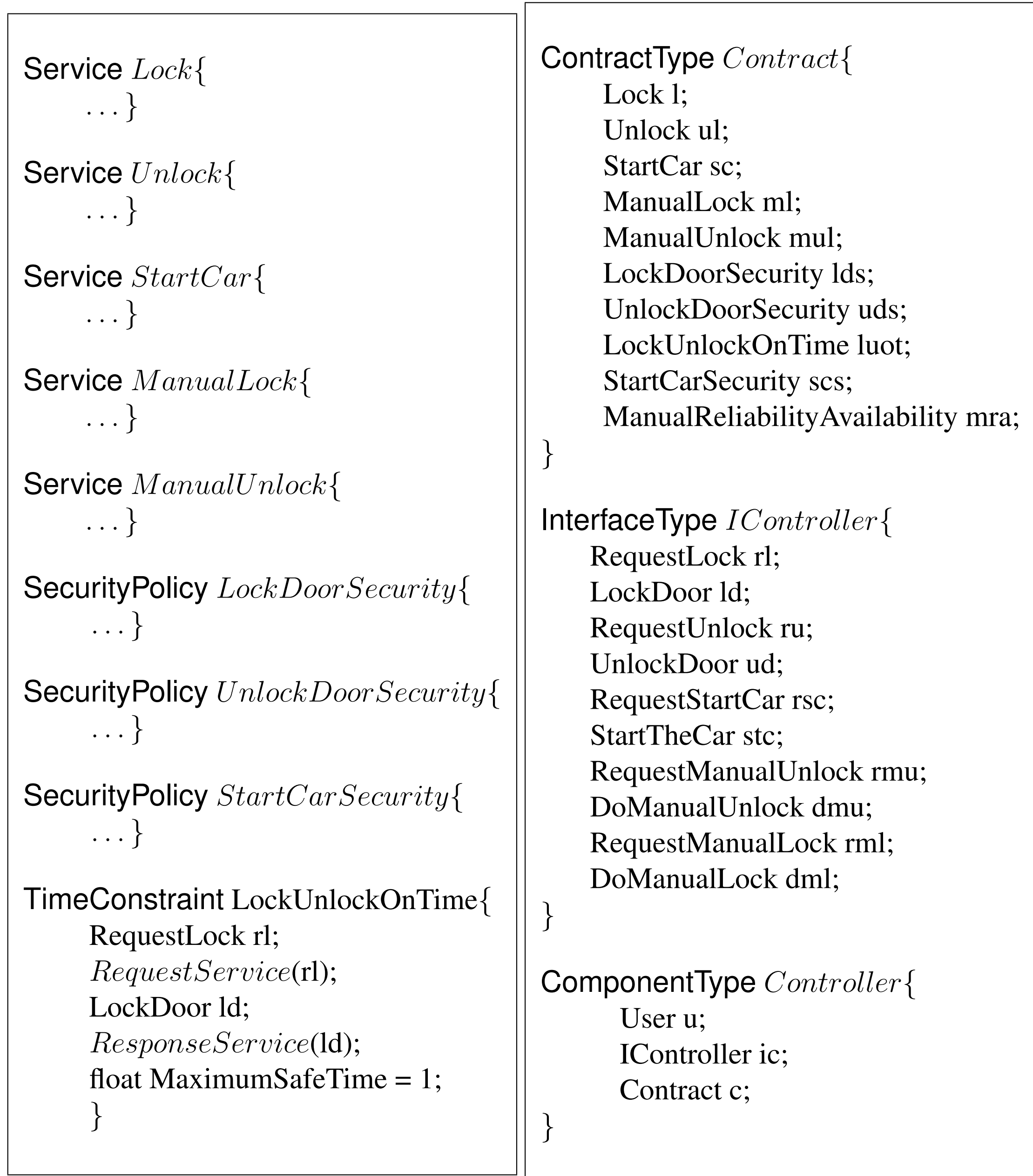
- Constraints are mapped into their corresponding synonym in TADL. A constraint is an invariant on services. Here also, a manual intervention is required to translate the constraints specified in the requirements to first order predicate logic suitable for TADL.

Figure 39 presents the TADL specification of the fingerprint security example. The specification includes only the controller component.

Therefore, domain engineering yields an ontology representing the knowledge base of the domain. The domain architecture can be deduced from the ontology. It includes the applications and their relations. Then, an architecture is created for each application. This architecture is specified by TADL specification which is generated from the ontology. The constituent domain components and their detail specifications are also defined in TADL. A component's definition includes details about functional, data, non-functional, and structural requirements. This knowledge and the resulting TADL specification will be used for both component and system development processes. The formal specification of TADL will enable formal analysis and reasoning about trustworthiness properties in the following steps. The ontology and TADL specification will be stored in a "repository" and reused by the next steps.

Domain engineering is a challenging task. We have provided an ontology-based approach for domain engineering. The approach is directly related to TADL which has been introduced in Chapter 5. There are many challenges in domain engineering. Further research is required in order to address the following questions:

- how to transform the system requirements which are collected by system analysts into the ontology?



```
Service Lock{
    ...}

Service Unlock{
    ...}

Service StartCar{
    ...}

Service ManualLock{
    ...}

Service ManualUnlock{
    ...}

SecurityPolicy LockDoorSecurity{
    ...}

SecurityPolicy UnlockDoorSecurity{
    ...}

SecurityPolicy StartCarSecurity{
    ...}

TimeConstraint LockUnlockOnTime{
    RequestLock rl;
    RequestService(rl);
    LockDoor ld;
    ResponseService(ld);
    float MaximumSafeTime = 1;
    }

ContractType Contract{
    Lock l;
    Unlock ul;
    StartCar sc;
    ManualLock ml;
    ManualUnlock mul;
    LockDoorSecurity lds;
    UnlockDoorSecurity uds;
    LockUnlockOnTime luot;
    StartCarSecurity scs;
    ManualReliabilityAvailability mra;
}

InterfaceType IController{
    RequestLock rl;
    LockDoor ld;
    RequestUnlock ru;
    UnlockDoor ud;
    RequestStartCar rsc;
    StartTheCar stc;
    RequestManualUnlock rmu;
    DoManualUnlock dmu;
    RequestManualLock rml;
    DoManualLock dml;
}

ComponentType Controller{
    User u;
    IController ic;
    Contract c;
}
```

Figure 39: TADL specification of the Controller component

- how to translate the requirements of trustworthiness from a human readable text into formal properties suitable for formal analysis?

- how to minimize the human intervention or guide it in such a way that will avoid errors in the specification?

8.2 Component Development

This section describes the activities done during component development, which are depicted in Figure 35. Component requirements are defined for new components or reused for existing domain components. The requirements are defined using TADL which is based on our formal component model introduced in Chapter 4. Therefore, formal component definitions are created using TADL or reused from the repository. The formal definitions specify component's structure, functional, and trustworthiness requirements in details using TADL syntax. Safety, timeliness, security, reliability, and availability properties should be defined formally using some mathematical logic which is compatible with the adopted model checking tool. The syntax of TADL specification is validated to check its correctness with reference to the correctness rules specified in Chapter 4. An iterative process occurs here until the specification passes the validation successfully. Then, the specification is analyzed and the component behavior is generated automatically as an extended time-automata using our approach which is presented in Chapter 6. The output is an extended time automata which is compatible with the UPPAAL modeling language. The transformation rules has been discussed in Chapter 6. After that, *verification* is conducted using UPPAAL model checking techniques to verify the correctness of the design. An iterative process of verification occurs until the design passes all functional, safety, security, reliability, availability, and timeliness requirements checks successfully. In case of errors or violation of any requirement, the component is redesigned using TADL specifications and the process starts over.

After finishing the iterative process of design and verification, the component is *implemented* by the developers. A component technology is selected to determine the implementation details. Emphasis during implementation should be on component reuse. Software engineering design concepts such as abstraction, hiding, functional independence, refinement, and structural programming can effectively help in developing reusable components [Pre05].

In this thesis we provided a formal approach to the analysis and design activities of the component development in Chapter 4, Chapter 5, Chapter 6, and Chapter 7. However, the correct design of components does not guarantee that its implementation is correct. Therefore, further investigation is needed to analyze the challenges in component implementation such as:

- what is the suitable programming language for developing trustworthy components?
- how to ensure that the developers will implement all the trustworthiness requirements?
- how to ensure that the implemented trustworthiness requirements are implemented correctly?
- how to minimize the developed code? Is it possible to generate code automatically?
- how to implement the component contract?
- how to keep the components loosely coupled?

8.3 Component Assessment

This section describes the activities used for component assessment, which are depicted in Figure 35. The implemented component undergoes iterative cycles of code inspection and traceability analysis to ensure that the implementation satisfies the verified design. An automated black-box testing method is applied to ensure the correctness and predictability of components' behavior. Then, the new component is certified and stored in the component repository. Traceability, testing, and certification are discussed later in Sections 8.6 and 8.8. The component's requirements documentation, design, implementation, testing reports and verification results, traceability analysis and certificates are all stored as meta-data in the repository . This requires a powerful and automated classification technique that eases storing and retrieving components and their meta-data. Examples of automated approaches for searching and retrieving reusable components in large repositories and on the Internet exist in the literature [YEV08].

8.4 Component Reuse

This section describes the activities involved in component reuse, which are depicted in Figure 35. During system design, designers and integrators can reuse existing components. If there exists no component which satisfies the requirements, a new component should be developed. System designers start *searching* for candidate components that could satisfy the stated requirements, both functional and non-functional. The meta-data stored in the repository along with the automated classification and retrieval approaches will facilitate an efficient searching of the repository. If the search is successful in finding some components, the *selection* task is carried out to qualify the candidate components and select the most appropriate one. Selection is based on domain knowledge and components meta-data retrieved from the repository. If the component requires some modifications to fit in the new deployment environment, the *adaptation* task is carried out to perform the required modifications. These modifications must be tested using the component assessment activities. After finishing the testing, the adapted component is certified and stored in the component repository for future reuse.

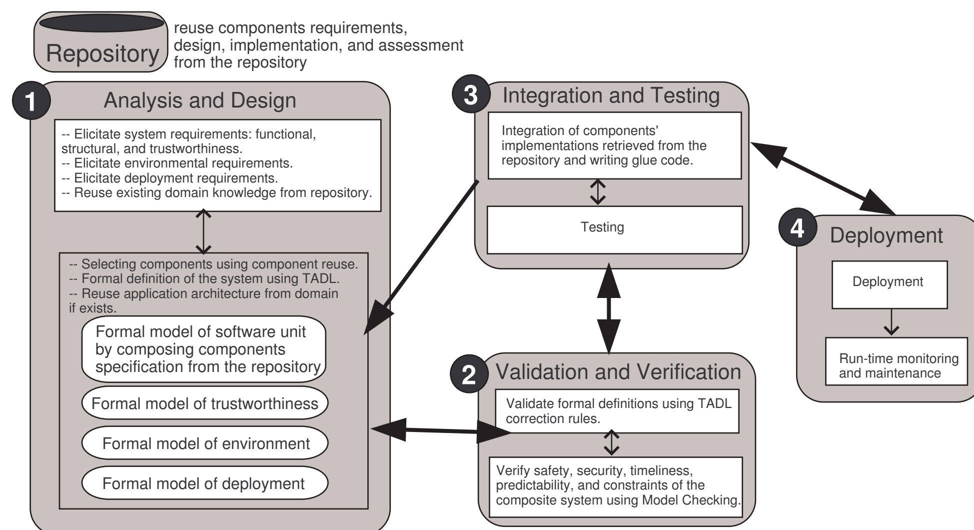


Figure 40: Component-based system development

Component Adaptation involves many challenges. System integrators need to evaluate the effectiveness of adapting an existing component versus implementing a new one. Further research is required to investigate the following questions that are related to component adaptation:

- how to ensure that the trustworthiness properties will be preserved during the adaptation process?
- how to create a sub type of an existing component type?

8.5 System Development

Figure 40 presents the system development process. It consists of four phases: analysis and design, validation and verification, integration and testing, and deployment. In system requirements definition, reuse of domain knowledge is to be encouraged whenever possible. We have outlined our approach for the reuse of domain knowledge in Section 8.1. Requirements should include functional, structural, and trustworthiness aspects of the system. The requirements are stated in TADL. The requirements analysis leads to selecting components to build the component-based system. The reuse of components is discussed in Section 8.4. Once the components are selected, their formal specifications are retrieved to build the software unit formal specification. The formal specifications are based on our formal component model introduced in Chapter 4. Domain application architecture can be used, if exists, to build the system architecture and define relations between components. The trustworthiness properties of the system must be defined using a formal logic language. There is a need to translate the trustworthiness properties from TADL specification into a formal logic language. The language depends on the kind of model checking tool used, such as UPPAAL [BDL04]. A formal model of the environment is required to test the boundaries of the system. A formal deployment model is required to verify the correctness of the deployment, relations between hardware and software components. The formal model of the environment is specified using TADL.

An iterative process of validation is conducted to ensure that the system design is syntactically and semantically correct with respect to TADL rules. Then, the formal models of software unit is formed by composing the extended time-automata for all the constituent components using UPPAAL according to our approach which is introduced in Chapter 6. Also, the trustworthiness properties are specified using UPPAAL's CTL language. Then, the formal verification process starts. The goal is to ensure that the composition of components does not violate the stated trustworthiness properties, which are already satisfied at a component level.

If the system design is proven correct, *integration* activities take place to integrate the

implementations of the selected components, which are retrieved from the repository, according to the system architecture. Glue code is written whenever required. The implementation undergoes an iterative cycle of extensive system testing. Different kinds of testing can be applied such as unit, integration, and acceptance testing. Then, the software is deployed according to the formal deployment plan in a run-time environment that allows dynamic reconfiguration. While in operation, the run-time software is continuously monitored and analyzed to ensure that its behavior respects the trustworthiness properties and conforms to the verified formal models. Run-time monitoring (see Section 8.9) is a powerful mechanism to ensure availability and analyze reliability. During maintenance, components can be substituted to fix bugs or install new upgraded versions that provide more services. In this case, the formal model of the new component must be composed with the software unit design and validation, verification, and integration testing activities should be applied to ensure that the new component does not compromise the trustworthiness of the system.

The formal component model and TADL are the basis for the formal specification of component-based systems. We provided a compositional theorem which preserves the requirements of trustworthiness when assembling components together. This theorem makes the model checking of large component-based systems tractable. There are many challenges in the component-based development for trustworthy systems that need further investigation such as:

- what is the effect of the glue code between components on their trustworthiness requirements?
- what the the requirements of a trustworthy deployment environment in which trustworthy systems can be deployed?
- how to perform run-time monitoring? what is the effect of runtime monitoring on the performance of the system?
- how to guarantee that the maintenance will be applied whenever it is needed?
- how to perform dynamic reconfiguration? what is the effect of the dynamic reconfiguration on the properties of trustworthiness?

8.6 Traceability

The goal of traceability is to analyze newly developed components and verify their conformance to design specifications. Traceability verifies that the code satisfies the functional, structural, and trustworthiness specifications. Traceability analysis is not a trivial task. It requires scrutinizing the generated and developed component code. We propose the following techniques to perform this operation:

Traceability of functional and structural specifications: During the automatic code generation or manual development of components, there is a need to maintain the relation between each functional and structural design element and its implementation construct. These relations can be kept in a *transformation file* in which the name and type of each design element is associated with the name and type of its actual implementation. For example, the services that are defined at the interfaces of a component in TADL are associated with implementation methods, functions in C# or Java. Then, model transformation analysis techniques, such as [CHM^{+}02], can be used to verify the completeness and correctness of the code generation and development. At the same time, information can be added to component's *meta-data* to link the implementation to its source design time specification. For example, current programming languages like $C\#$ and Java support defining *custom attributes*. These attributes can add semantic information to implementation constructs such as methods and classes. Then, *Reflection* techniques are used to read attributes and analyze component's meta-data. Therefore, the traceability uses attributes and reflection to analyze the conformance of component's implementation to its design specifications.

Traceability of real-time specification: *Worst case execution time* (WCET) of services can be specified as an attribute to a service at design time and as a custom attribute at implementation time. Then, during traceability analysis, the functions that implement real-time services can be executed to check if their measured execution time is bound by their specified WCET attribute.

Traceability of trustworthiness: The actual traceability of security, availability, reliability, and safety behavior can be analyzed using run-time analysis techniques described in Section 8.9.

Traceability analysis is a challenging task. Further research is needed to investigate the following directions:

- how to implement a custom attribute for the different language constructs of TADL?

- how to build an efficient reflection mechanism to verify that the developers have implemented all the requirements?
- how to verify that the implementation of each requirement is correct?
- how to link a language construct to its implementation? Is it sufficient to use names to link a requirement to an implementation?
- the execution time at a development or test environment may be different than the execution time at a real deployment environment. Therefore, how to validate the implementation of worst-case execution time?

8.7 Certification

After traceability analysis, there is a need to interact with a *certification authority* to obtain a certificate that indicates the trustworthiness of the component and the level of development conformity to design and quality attributes stated in its specifications. Certification authorities do exist for electronic components [ECC]. However, for software components, the issues of certification exists only in research. Despite many publications about this topic in the literature [AdAdLM05], there is no general official certification authority that currently exists. A certification authority could exist only at a domain level. For example, in the domain of avionics, the Federal Aviation Administration (FAA) and the European Aviation Safety Agency (EASA) use the *DO-178B guidance* to give certificates to software components. Also, it is possible that the same company develops and reuses its components; therefore, it can have a local certification body.

The certificate can be issued based on analyzing the following information: (i) the information of the software development firm, (ii) component's design specification and implementation code, (iii) the results of the conducted design-time verification. It is possible to submit the state space generated by the local model checker for external reviewer to verify properties, in case the company does not want to submit the detail design, (iv) the results of the traceability analysis of components implementation relative to its design, (v) the test results from the automated black-box testing, and (vi) detailed information about the tools used to generate the analysis reports. Then, the certification authority can verify the claimed analysis reports (may perform the verification and traceability checks again)

and issue the certificate. Certificates are stored with components along with their analysis reports in the component repository.

Certification is still a challenging task. Further research is required to investigate the following questions:

- what are the requirements of a trustworthy certification authority?
- how to ensure that a certified component has not been modified after certification?
- how to validate certificates?
- what is the effect of component modification on certificates? what type of modifications invalidate a certificate?

8.8 Automated Component Testing

The formal specification approach is necessary for having an automated, contract-based testing. The component formal definition, introduced in Chapter 4, includes specifications of the services provided or requested by the component along with the data parameters communicated through them. It is possible to use the constraints that are defined for data parameters to specify the valid ranges of values. Service specification defines the relation between requests for services and their corresponding responses. It can be used also to define the relation between the valid input and output values. These formal information can be used to define automatic, black-box test scenarios for each component. A selection of input data both from the valid and invalid ranges can be used to test the responses of components. Then these responses are analyzed according to the service specification to determine whether or not the correct responses ware issued and whether or not the valid outputs were attached to it.

The technique described above helps building automatic testing scenarios for the observable behavior of a component. However, there are more research problems that need to be investigated to develop an effective and automated component testing such as:

- how to test the composition and communication between components?
- how to ensure that the internal, non-observed, behavior of a component is correct?

8.9 Run-Time Monitoring

In this activity, a tool performs run-time analysis during system execution. The tool ensures that system behavior conforms to the stated functional and trustworthiness properties. This is done by observing input, output, and system states during program execution. Execution sequences can be monitored, logged, and visualized to ease analyzing system behavior. These sequences are used to build usage profiles for components. These profiles can be used to monitor the availability and analyze the reliability of components and system. The execution profiles can be subjected to formal verification. Verification is done by ensuring that system executions do not reach a state that violates trustworthiness. It can produce a counter-example in case of system failure.

Run-time monitoring is a challenging task. The following research problems require further investigation:

- what are the requirements of a run-time environment for trustworthy systems?
- how to ensure that the run-time environment is supplying all the essential needs of trustworthy components?
- how to intercept the interplay communication between components to build execution profiles?
- is it possible to monitor the internal behavior of a component?
- how to perform efficient formal verification using the execution profiles?

8.10 Accomplishments

This chapter provided our perspective of a process model for the development of trustworthy component-based systems. The process model consists of several phases that cover the different activities that are necessary for developing such systems. We have outlined the essential activities that are necessary for a rigorous component-based development process model. We provided a brief discussion about each activity. This thesis accomplished the following tasks:

- Domain Engineering: we provided an ontology oriented approach for domain ontology in Section 8.1. The approach includes providing an ontology for trustworthy

systems, designing systems using this ontology, and transforming the design into TADL. We provided an example to illustrate our approach.

- Component Development: we provided a formal component model in Chapter 4 and TADL in Chapter 5 which is based on the formal component model. TADL is used to specify trustworthy components. Specifically, it provides language descriptions for the functional, trustworthiness, and structural requirements. Validation rules have been defined in Chapter 4 to validate the syntactic correctness of component specification. We provided an automated model transformation technique for specifying the behavior of components automatically in Chapter 6. We provided a technique for the formal specification of reliability and availability in Chapter 7. We integrated two model checkers, UPPAAL and Times, for verifying the trustworthiness properties of the component.

- Component-based system development: the formal component model, TADL, model transformation technique, and model checking contributions that we provided for the component development are applicable also for the component-based system development. We provided a composition theorem which insures that the trustworthiness properties are preserved in the composition. Therefore, incremental model checking can be used effectively to model check component-based systems using our approach.

We provided a brief discussion and research directions about the other activities, which include component implementation, traceability analysis, automated black-box testing, component reuse, and deployment. These activities need to be explored further.

Chapter 9

Framework Architecture

The framework is being built on the rigorous process model. Tools are provided to practise CBSE to develop trustworthy systems. The framework can be viewed in three layers: design, implementation, and deployment. Taken as a whole, the framework describes the tools necessary for the different activities outlined in the process model, which was introduced in Chapter 8. Figure 41 depicts the framework architecture showing the tools in the three layers. This chapter gives a detailed description of the tools and highlights the merits of each tool.

9.1 Design-Time Tools

9.1.1 Visual modeling tool

This tool is used in the component development activity described in Section 8.2 and the system analysis and design activity described in Section 8.5. This tool provides a user friendly interface to model components and systems and specify functional and non-functional properties. It acts as an interface to perform design without being directly exposed to the formal notation. The tool projects both textual and visual representations of the design. Also, it projects the model into 3 different views for different users: CBD, real-time, and trustworthiness view. The tool has been implemented using Java [Yun09]. Every architectural element has a defined visual representation. The user designs a system by dragging and dropping visual elements into a design canvas. Relations, properties, attributes, and conditions can be associated with the design elements. The system specification are saved in an XML file according to TADL syntax. Figure 42 shows a screen shot

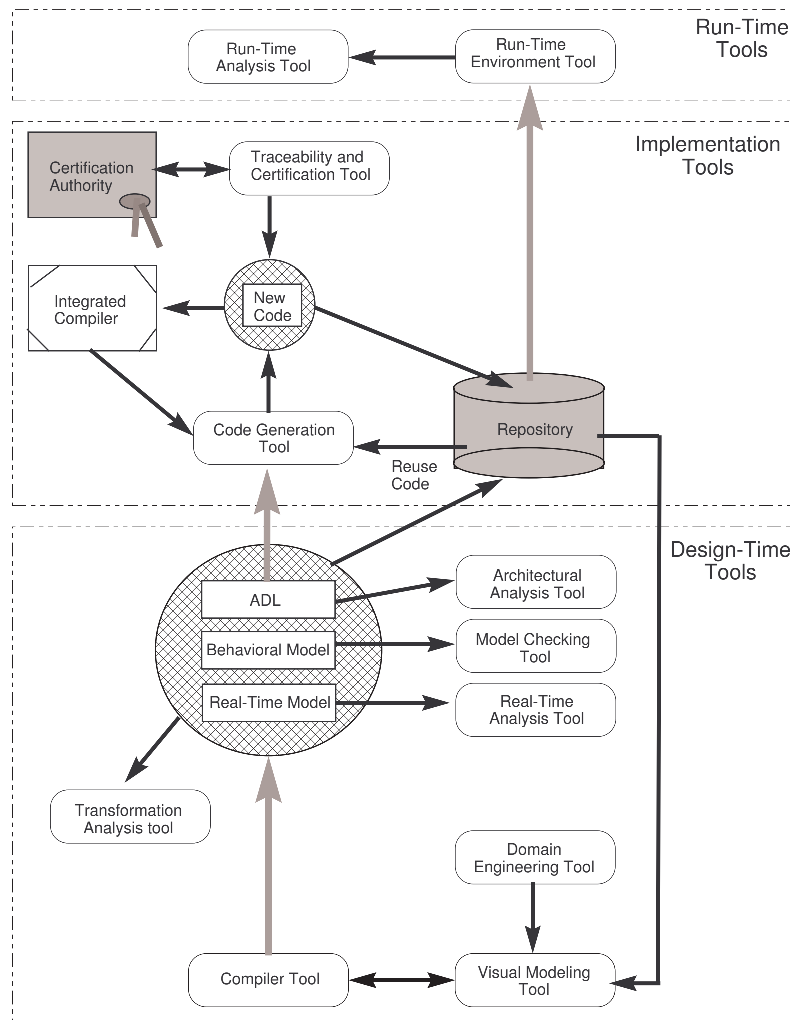


Figure 41: Framework Architecture

of the visual modeling tool.

9.1.2 Compiler and model transformation

This tool is used in the component development activity described in Section 8.2 and the system analysis and design activity described in Section 8.5. This tool checks the syntactic correctness of the visual modeling design with respect to its abstract definitions. The compositional correctness of component design elements and the architectural mismatches such as incompatibility of the interface types defined in the connector types or those used in the architectures of composite components are checked. Error messages are given when inconsistent or incompatible definitions appear in the design. If the design is syntactically

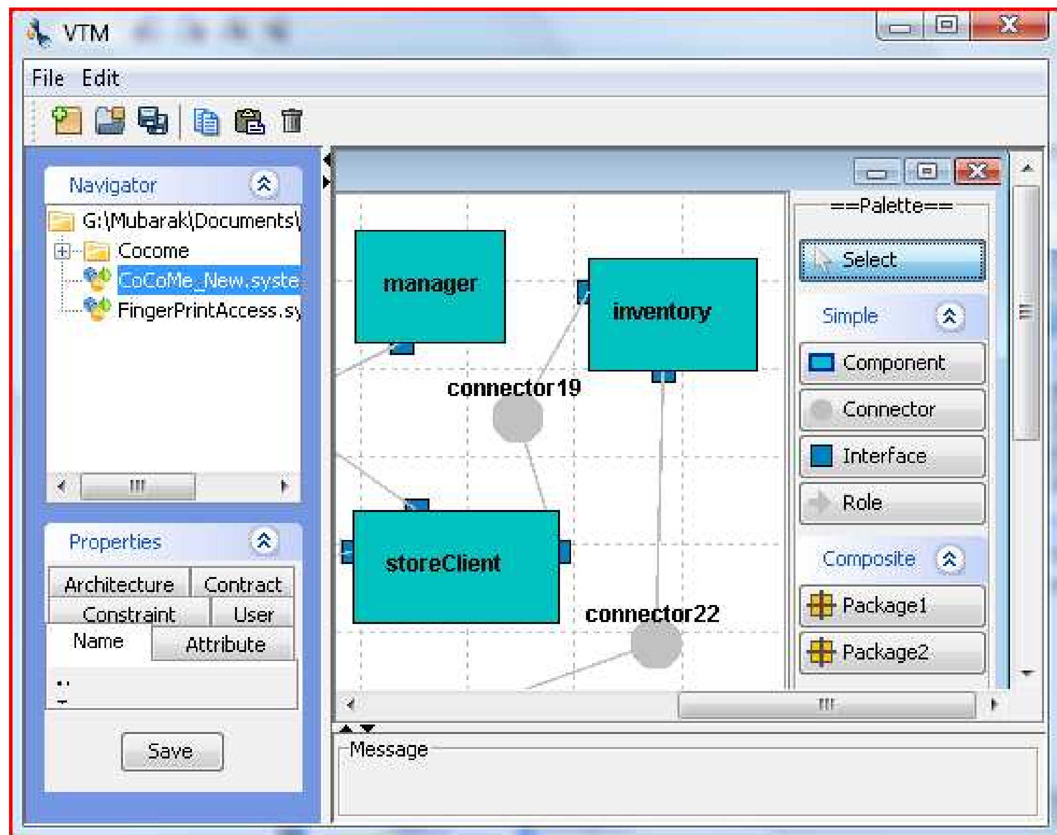


Figure 42: Visual Modeling Tool

correct, the compiler generates a formal descriptions of the visual model. The compiler generates different types of output by transforming the valid design according to formally defined transformation rules. The current version of the compiler generates three types of output:

- a textual description in TADL syntax,
- a behavioral model descriptions as UPPAAL extended timed automata, and
- a real-time model using timed automata extended with tasks.

Manual transformation of component specification at design time into other models is complex and error-prone. Therefore, applying automatic model transformation techniques is very important to ensure a highly convincing level of trustworthiness. The transformation process is implemented using XSLT [XSL], a standard mechanism for transforming XML documents into other types of documents. The process uses the formal transformation rules,

defined in Chapter 6, to transform the system specification from the saved XML file into the required output format. The transformation rules are implemented using XSLT instructions and XPath, an expression language for finding information in an XML document. The implementation of this tool can be easily extended to accommodate more views. This is because the transformation is implemented using XSLT. This means that the transformation rules can be maintained, updated, and extended without affecting the transformation process or requiring reimplementation. Figure 43 shows a snap shot of the compiler and model transformation tool [Ibr08]. The window is divided into two sections: system specification and model translation. The system specification part displays the TADL specification using XML tree or textual format. The model translation shows the generated UPPAAL or Times extended timed automata, which is resulted from the transformation process.

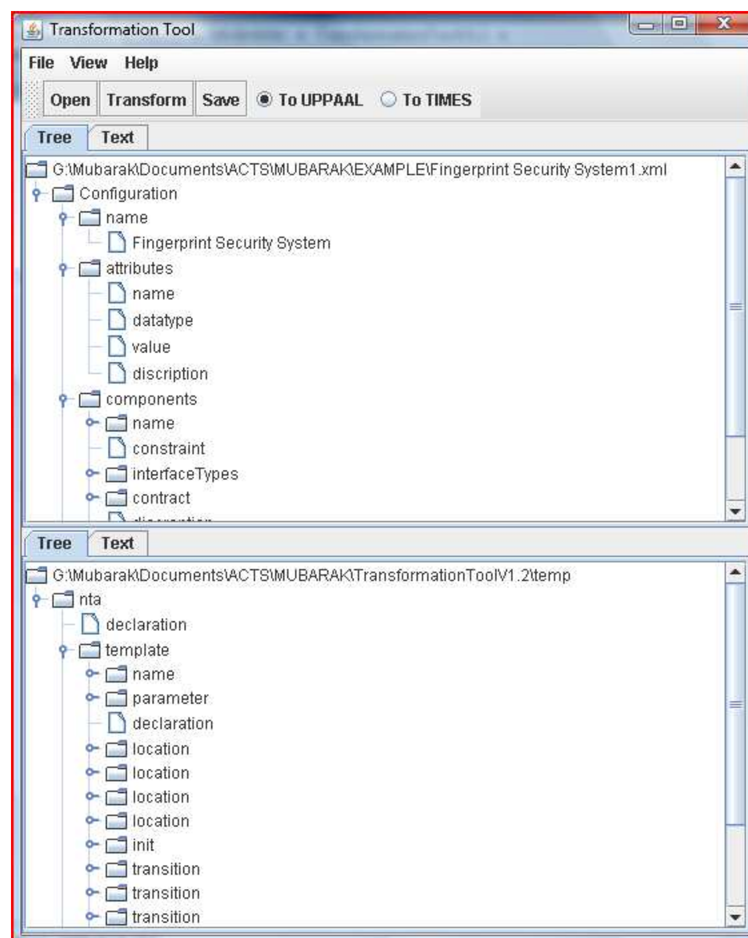


Figure 43: The compiler and model transformation tool

9.1.3 Transformation analysis

This tool is used in the component development activity described in Section 8.2 and the system analysis and design activity described in Section 8.5. Automatic model transformation is increasingly being adopted as a technique to reduce the complexity and faults of transformation. However, the correctness and completeness of the transformation process must be subject to reasoning. Design and implementation flaws are still possible during the development of the automatic transformation tools. Therefore, it is necessary to subject the transformation process to inspection in order to make it trustworthy. The transformation analysis tool is crucial to verify the correctness, completeness, and compatibility of the views produced by the transformation process of the compiler. A view is complete with respect to the visual model if every feature in the view is a feature in the visual model. That is, there is no extraneous feature in a view. A view is correct with respect to a visual model if the view is complete and every feature in the visual model is mapped to only one feature in the view. Two views are compatible if and only if both views are correct with respect to the visual model. Depending on the type of output (ADL, behavior protocol, or real-time model) and defined formal transformation rules, the tool will analyze the transformation process and produce the result to the user.

The transformation analysis is done by reversing the transformation process and validating the resulting output with the original XML file that specifies the system. This is done by defining an XSLT with reverse transformation rules, from extended timed automata to system specification in XML. The process takes a behavioral specification as input and produces a system specification as output. Then, the resulted XML file is validated against the original XML file. In this process, every component specification, including its services, attributes, data constraints, time constraints, and security specification, should match a component specification in the original XML file.

9.1.4 Simulation and model checking

This tool is used in the component development activity described in Section 8.2 and the system analysis and design activity described in Section 8.5.

The distinct advantage of the compiler tool is that it can generate the behavior model in different notations, thus allowing different model checkers to be integrated into the framework to perform formal verification. There exists no general purpose model checker, in the

sense that no existing model checker has the ability to model check any stated property. Since trustworthiness attributes can be defined differently by developers for different applications, a developer must be given the facility to plug in the model checker that is most suitable for verifying the chosen trustworthiness properties. This is the rationale behind our design decision to translate the model into different formal notations.

By design the translator in our tool is syntax-directed and hence extensional. The translator in our compiler will only require the grammar of the target language to produce an output in the target language. No change to the translator code is necessary.

In the current implementation, the compiler supports only UPPAAL [BDL04] format. Therefore, we use UPPAAL tool for simulation and model checking.

9.1.5 Real-time analysis

This tool is used in the component development activity described in Section 8.2 and the system analysis and design activity described in Section 8.5.

This tool supports real-time scheduling and real-time analysis relative to criticality, priority, and other real-time non-functional properties. We are currently using Times [AFM$^+$03] to perform real-time analysis.

9.1.6 Architectural analysis

This tool is used in the component development activity described in Section 8.2 and the system analysis and design activity described in Section 8.5.

This tool analyzes the correctness of the architectural style and system configuration specification relative to architectural constraints defined in the system design.

9.2 Implementation Tools

9.2.1 Component repository

This is a storage place to store and reuse developed trustworthy components. The repository provides storage facilities for: (i) component specification (structure and contract), (ii) development source code, (iii) compiled, execution ready assembly of the component, and (iv) usage profiles and certificates. The repository allows storing and retrieving different versions of the same component.

9.2.2 Code generation

This tool is used in the component development activity described in Section 8.2 and the system analysis and design activity described in Section 8.5.

This tool produces source code. It supports different programming languages such as C++, C#, and java. It analyzes the system design specification. Then, for every component or connector, if the component exists in the component repository then it should reuse it; otherwise, it should produce source code or skeleton for new components. The tool will also develop code for new components by refining existing implementations. The tool provides facilities to use language specific compilers such as C# or java to perform syntactic and semantic analysis of components code. Contracts will be handled as *cross-cutting concerns* implemented as *aspects*.

9.2.3 Traceability analysis

This tool performs the traceability activity, which is described in 8.6. It takes a component's specification and its corresponding implementation as input. Then, it queries the meta-data of the component implementation using reflection techniques to retrieve the custom attributes. After that, it compares the implementation constructs of the component with its specification. For example, it checks if all the defined services has been implemented or not.

9.3 Run-time Tools

9.3.1 Run-time environment

This tool supports running systems and dynamically reconfiguring executions. The tool is a *middleware* between the component repository and the run-time environment that communicates with the operating system (e.g., J2EE or .NET run-time environment). It communicates with the component repository to load component assemblies. The tool allows a controlled reconfiguration to the running system (e.g., adding a new component or replacing an existing one).

9.3.2 Run-time analysis

This tool performs the run-time analysis activity, which is desribed in 8.9. This tool monitors that interplay communications between components and logs it for further analysis. It is possible to extend the implementation of connectors by adding logging mechanism. This enables logging all the interactions between components. Then, the logs of all connectors of a component can be composed together to analyze the overall behavior of the component. These logs build operational profiles that can be used to perform run-time analysis of the trustworthiness properties.

9.4 Summary

This chapter introduced a framework which is designed to implement the rigorous process model. Currently, the design time tools has been implemented and tested on several case studies in the domains of component-based development and safety critical systems. The other tools are under different stages of design and development. Figure 44 shows the currently implemented tools. The visual modeling tools, which is implemented by Zhou Yun [Yun09], is used to design and specify trustworthy component-based systems. Then, the tools exports the TADL specification as an XML file. This file is input into the compiler tool, which is implemented by Naseem Ibrahim [Ibr08]. Then, the compiler produces two output XML files one with UPPAAL representation language and the other with Times representation language. After that, the two files are input into the model checkers to perform verification.

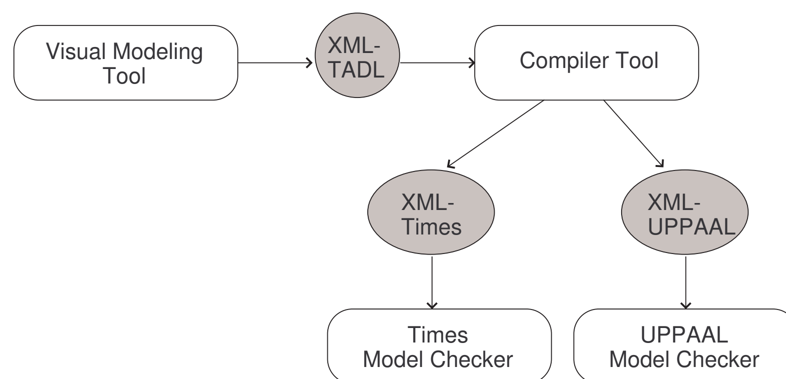


Figure 44: The implemented and adopted tools

These implemented tools have the following limitations:

- The visual modeling tool is used to design component-based systems. However, the whole component-based system specification is saved into one XML file. Therefore, it is not possible to reuse an existing component specification for another system. There is a need to improve the tool to allow reuse of system elements.
- The output of the compiler tool is limited to UPPAAL and Times representation languages. There is a need to extend the tool to support other compiler representation languages.
- There is a need to embed the compiler functionality inside the visual modeling tool. This will facilitate the design and avoid the use of an intermediate XML file between the two tools.

Chapter 10

Conclusion

In this thesis we have evaluated the state of the art of CBSE approaches. The analysis shows that the CBSE has not fully realized its objectives and still has a long way to go to fulfilling its promises. Hence, it is unlikely that current practices of CBSE can lead to developing trustworthy systems. Therefore, we have introduced a formal approach that aims to remedy the shortcomings of CBSE by proposing a trustworthy component model, a rigorous development process model, and a framework that implements the development process.

Component modeling techniques with whom we have compared our work, do not provide all the tools necessary for rigorous analysis at different stages of system life-cycle. The reason is that these component models are designed and implemented for different specific domains. For examples, SaveCCM, Pin, and PECOS are real-time component models. Hence they provide tools for real-time analysis and verification of safety and liveness. On the other hand, SOFA is a distributed component-based model focusing on distributed systems' architecture and communication aspects.

A virtue of the presented software engineering approach is that it can be a unified platform for developing component models, regardless of their application domain. The proposed component model provides both real-time elements and essential architectural features for hierarchical, as well as distributed systems. Also, it supports the specification and verification of trustworthiness properties.

It is reasonable not to claim that systems developed under this proposed framework will be absolutely trustworthy, but it is justifiable to claim that such systems can be provable to meet the trustworthiness criteria, provided that the tools in the framework are correct.

As of now, the visual modeling, the compiler, the automatic translator to ADL notation, and translating the model to UPPAAL language for model checking have been completed. We tested the translation to UPPAAL model checker on the *steam boiler controller case study* [ABL96] and the *common component modeling example* [RRMP08] and verified timeliness, safety and security properties. Times tool has been used to perform real-time analysis. We are optimistic in realizing the rest of the tools.

10.1 Summary

In this section we discuss and evaluate the results achieved in this thesis with respect to the goals stated in Chapter 3.

1. Defining "A Formal Component Model for Trustworthy Systems": there are 4 research problems stated for this goal. The research problems and the solution provided by this thesis are stated bellow:

   - The lack of support for trustworthiness requirements in component models: the solutions provided for this problem and their limitations are listed bellow:
     - Solution 1: in Chapter 4 we provided a formal component model which supports the specification of safety and security requirements.
     - Solution 2: in Chapter 7 we provided a formal specification for reliability and availability.
     - Solution 3: the trustworthiness properties are define in the component contract.
     - Solution 4: in Chapter 5 we provided an architecture description language for describing trustworthy component-based systems.
     - Solution 5: in Chapter 6 we provided an automated approach to specify the behavior of trustworthy components using extended timed automata.
     - Limitation 1: the security properties are limited to role-based access security. Further research is required to explore other security mechanisms and distributed authentication.
   - The strong coupling of components: the solution for this problem is provided in in Chapter 4. The dependence between components is defined in the component contract. Therefore, component specification does not need to include

information about the components it communicates with.

- The need for a composition theory:
  - Solution: in Chapter 4 we provided a composition theory for trustworthy components.
  - Limitation: the composition theory includes only safety and security requirements. Further research is needed to investigate the composition of reliability and availability.
- The need for an approach for specifying and verifying reliability and availability at architecture level: in Chapter 7 we provided a novel approach for the specification and verification of reliability and availability at architectural level.

2. Defining “A Process Model for Developing Trustworthy Component-Based Systems”: the solution provided for this goal and the limitations are stated bellow:

   - Solution: in Chapter 8 we provided process models for component engineering and component-based development of trustworthy systems.
   - Limitation: brief discussion was provided for the activities of component implementation, traceability analysis, automated black-box testing, component reuse, and deployment. Further research is required to explore these activities.

3. Developing “A Framework with Comprehensive Tool Support”: the solution provided for this goal and the limitations are stated bellow:

   - Solution: in Chapter 9 we introduced a framework with comprehensive set of tools for supporting the activities in the process model.
   - Limitation: We implemented only the design time tools. Further research is required to design and implement the rest of tools.

10.2 Assessment

The formal approach presented in this thesis is a contribution to CBSE. A model is a corner stone in any engineering practice. A formal model helps in understanding and reasoning about a problem very well. Our formal approach includes tools to support the engineering activities. In this section, we evaluate our formal approach with respect to the following

criteria: completeness, comprehensibility, modifiability, testability, reusability, scalability, and usability.

Completeness: *Are the elements of the formal component model sufficient to model trustworthy systems?* The following factors support our argument that the elements of the formal model are sufficient to express various trustworthy component-based systems:

- *Component Model*: When we analyze the various component definitions in the literature, we find that the essential defining elements of a component model are: component, interface, connector, attribute, architecture, and behavior specification. These elements might have different names or syntactic definitions but there is a common consensus about their semantics. The formal component model that is introduced in this thesis includes all these elements.

- *Trustworthiness*: When we analyze the definition of trustworthiness in the literature, we find that safety, security, reliability, and availability are the essential properties of trustworthiness. The contract of our component model inclusively defines these properties.

- *Case Studies*: We have tested our component model on two benchmark case studies: (1) *steam boiler controller case study* [ABL96], which is a benchmark case study in the domain of safety critical systems, and (2) *Common Component Modeling Example* [RRMP08], which is a benchmark case study for testing the modeling ability of component models. The results are provided in [Ibr08, AM07b]. It shows that our component model is capable of modeling such case studies. The early definition of our trustworthy component model appeared in [AM07a].

Comprehensibility: *Are the formal descriptions easy to understand?* Mathematical notations are not easy to understand for a non expert. This motivated us to create an architecture description language (TADL). TADL uses high level language to describe component-based systems. Therefore, it is easy to understand by non experts in our formalism. TADL appeared in [MA08]. The behavior specifications are generated in timed automata, which is widely used in the literature to describe the behavior of different types of systems, specially safety-critical ones. Trustworthiness properties are modeled using guard conditions and state invariants which makes it easy to the reader to understand their rule in governing system transitions.

Modifiability: *How easy it is to modify the specification?* Every element in our formal model is described separately. For example, a contract is specified separately from a component definition. This enables modifying the contract without affecting the definition of a component. Also, the definition of an architecture type is specified separately form a component. This enables customizing the structural definition of components without affecting their definition.

Testability: *Is it possible to validate whether or not a specification is right?* We have provided rules of well-formedness for the elements of the formal model. A visual modeling tool [Yun09] is used to design systems according to our component model. This tool checks the rules of well-formedness and report any error to the user.

Reusability: *Does the formal model support reuse?* Since every element in our component model is described separately, it is possible to reuse these definitions for different systems. We are currently designing a repository tool to host the component-based system specifications so that the elements can be reused.

Scalability: *Does the formalism scale up to handle large problems?* The scalability issue can be analyzed in the following two contexts:

- *Specification*: The formal specifications of component-based systems is scalable to large systems. It is possible to define hierarchical components and analyze it at different levels of granularity. Architectures can be used to specify the design of complex systems and encapsulate the details in one composite component specification.
- *Verification*: Our design time verification is based on model checking. Scalability of model checking is still an open challenge. The problem of state-space explosion limits the scalability of the verification process. Techniques such as symbolic model checking have been applied successfully to improve the scalability of model checking. In our model, incremental composition can be used to effectively address the state-space explosion problem.

Usability: *Is it easy to use the formal approach?* Our formal model can be used easily by software architects to specify and verify component-based systems. The user uses a visual modeling tool to design the system and configure its elements. Then, the visual modeling

tool will generate the corresponding TADL. After that, a model transformation tool will take the generated TADL as input and produce two types extended timed automata. The syntax of one extended time automata conforms to the syntax of UPPAAL model checker and the syntax of the other conforms to the syntax of the Times tool. Then, the design is verified against any trustworthiness properties. Thus, the whole process is supported by tools and the user needs to know only how to user the graphical user interface of the visual modeling tool and the property specification language of UPPAAL.

10.3 Case Studies

In this section we briefly describe the work done to apply our design methodology on two benchmark case studies in the fields of component-based development and safety-critical systems. The two case studies are: *the common component modeling example* [RRMP08] and *steam boiler controller* [ABL96]. The goals of applying our methodology on these two case studies are:

- test the expressiveness power of TADL,
- test the automatic model transformation process, which transforms TADL specification into extended timed automata,
- test the formal verification of the properties of trustworthiness using one unified model and one model checking tool, and
- test the visual modeling and compiler tools.

10.3.1 The common component modeling example

A common component modeling example (CoCoME) has been introduced by the component development community [RRMP08] to be used by different component models to evaluate and compare the practical application of existing component models using a common component-based system as a modeling example. The details of specification and verification of the case study was introduced in [Ibr08] and [Yun09]. In this section, we provide a brief description and show the results.

The CoCoME defines the Trading System, which is concerned with all aspects of handling sales at a supermarket, including the interaction with the customer at the cash

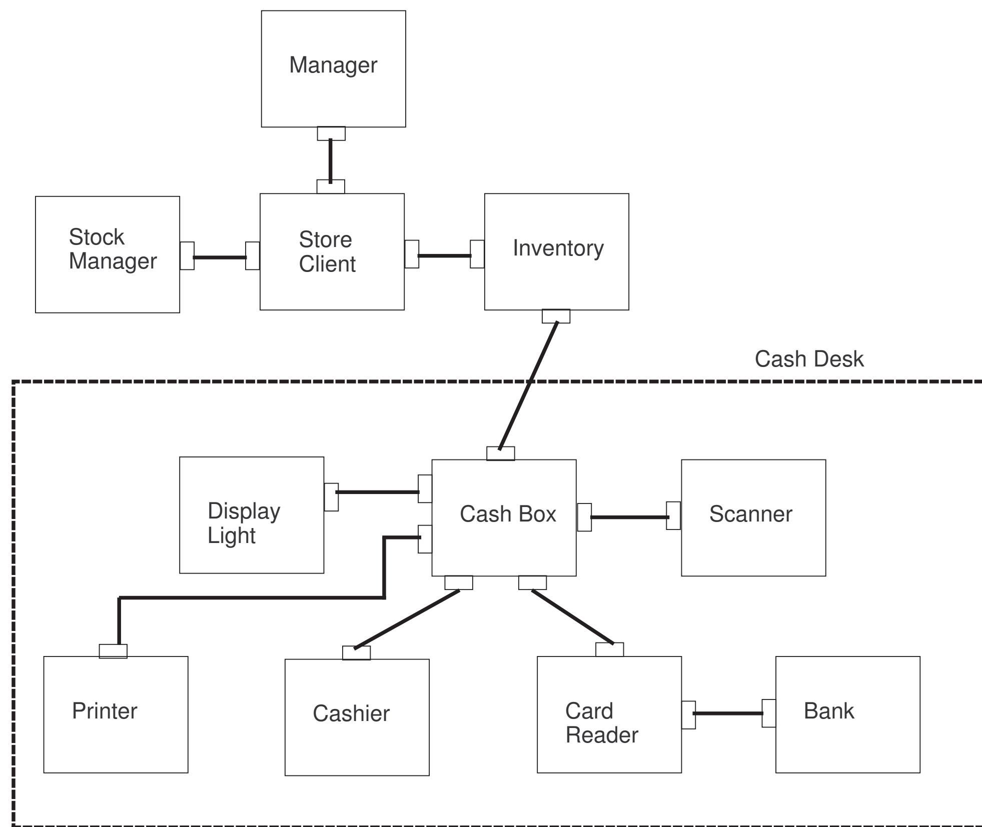


Figure 45: Store System Components

desk (product scanning and payment) and recording of the sale for inventory purposes. It also deals with ordering goods from wholesalers and generating various kinds of reports. Figure 45 shows the architecture of the trading system. It consists of the following components:

- *Cash Box*: this component is responsible of performing the selling operation by the cashier.

- *Bar Code Scanner*: this component is responsible for scanning the items to be sold and reading their bar code.

- *Card Reader*: this component is responsible for managing the process of card payment. It reads the card information and sends it to the bank to approve the payment.

- *Bank*: this component represents the financial institution that is responsible for performing and approving the card payment.

- *Printer*: this component is responsible for printing the sale receipt.
- *Stock Manager*: this component represents the stock manager which is responsible for receiving arriving orders and checking them then rolling them into the inventory.
- *Light Display*: this component represents the display light above the cash box which is used to indicate if this cash box is in express mode.
- *Inventory*: this component represents that store server (inventory).

First, we used the visual modeling tool to model the system. Figure 46 shows the architecture of the cash desk in the visual modeling tool. After that, the visual modeling tool automatically generated TADL specification for the case study. Figures 47 and 47 show parts of the generated TADL. The full TADL specification and verification is provided in [Ibr08] and [Yun09]. After that, the TADL was input into the compiler tool which generated the timed automata for all the components in the system using UPPAAL specification language. For example, Figure 49 shows the extended timed automata of the Cashier component. After that, we performed model checking to verify safety and security requirements.

The limitation of our methodology is that it does not handle the concurrency requirements which are part of the requirements of this case study.

Therefore, applying our methodology to the common component modeling example shows that the language constructs of TADL were sufficient to specify the requirements of the common component modeling example. Also, we were able to transform the requirements successfully to UPPAAL extended timed automata and perform verification of safety and security properties.

10.3.2 The steam boiler controller case study

In Chapter 7 we used the steam boiler controller case study to illustrate the expressive power of our formalism and design methodology in specifying and verifying reliability and availability requirements. We included the extended timed automata of the controller component which is the main component in this case study.

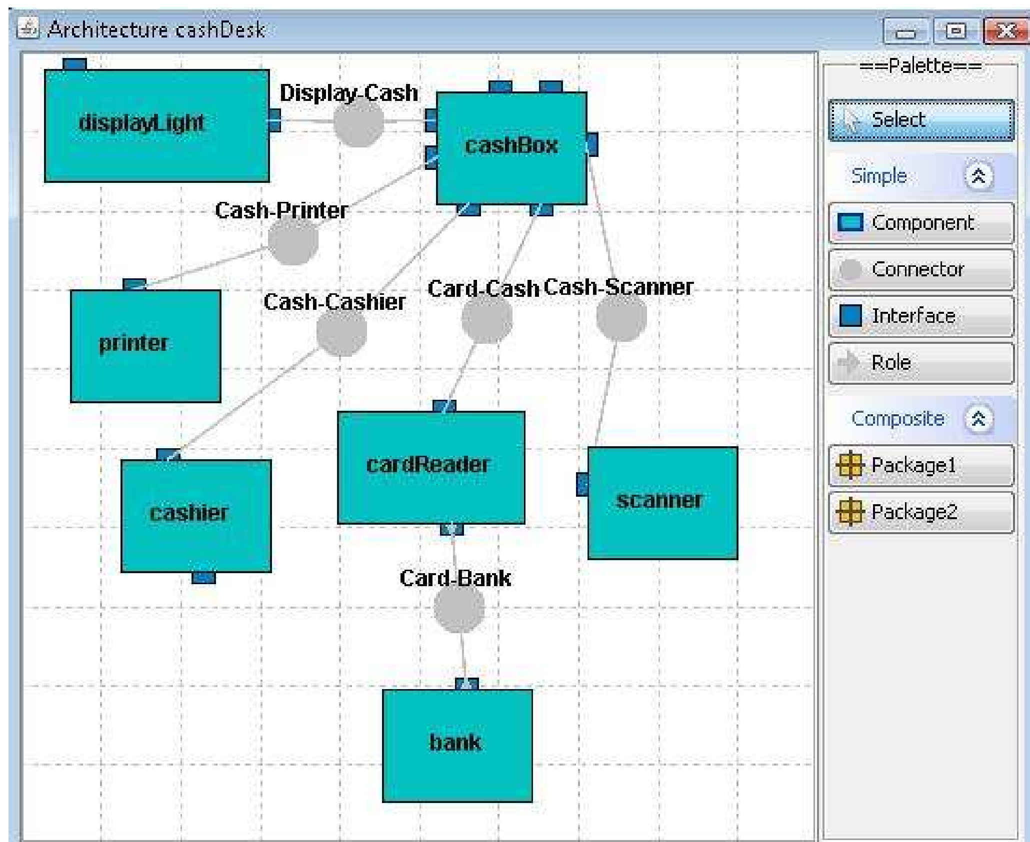


Figure 46: The architecture of the cash desk

```
  TimeConstraint ProcessCashPay{
    Cash cash;
    RequestEvent(cash);
    ReturnChange returnChange;
    ResponseEvent(returnChange);
    float MaxSafeTime=120;
}
DataConstraint CashBoxDataCons1{
    CheckIfExpress checkIfExpress;
    RequestEvent(checkIfExpress);
    CheckLastHour checkLastHour;
    ResponseEvent(checkLastHour);
    Constraint Mode==done;
}
Service CashReturn{
    Cash cash;
    RequestEvent(cash);
    ReturnChange returnChange;
    ResponseEvent(returnChange);
    TimeConstraint ProcessCashPay;
    Security Policy ;
    Update {
    }
    Action {
    }
}
Service ExpressHour{
    CheckIfExpress checkIfExpress;
    RequestEvent(checkIfExpress);
    CheckLastHour checkLastHour;
    ResponseEvent(checkLastHour);
    DataConstraint CashBoxDataCons1;
    Security Policy ;
    Update {
       Statement Mode:=waiting;
       Security Policy null;
    }
    Action {
    }
}
ContractType CashBox_Contract{
    Service CashReturn;
    Service ExpressHour;
    .....;
}
InterfaceType Cashier_cashBox{
    PassItem passItem;
    Cash cash;
    Card card;
    SaleFinished saleFinished;
    DisableExpress disableExpress;
    Pay pay;
    IsMoreItem isMoreItem;
}
InterfaceType CashBox_internal{
    CheckIfExpress checkIfExpress;
    ReturnChange returnChange;
    AddTotal addTotal;
    Ignore ignore;
    ChangeModeToNormal changeModeToNormal;
}
}
```

Figure 47: Part of the TADL specification of CocoMe

```
ComponentType CashBox{
    Contract CashBox_Contract;
    CashBox_scanner cashBox_scanner;
    CashBox_displayLight cashBox_displayLight;
    CashBox_printer cashBox_printer;
    CashBox_cashier cashBox_cashier;
    CashBox_cardReader cashBox_cardReader;
    CashBox_inventory cashBox_inventory;
    CashBox_internal cashBox_internal;
}
ArchitectureType CashDesk{
    DisplayLight displayLight;
    CashBox cashBox;
    Scanner scanner;
    Bank bank;
    CardReader cardReader;
    Cashier cashier;
    Printer printer;
    Cash-CashierCNN Cash-Cashier;
    Card-CashCNN Card-Cash;
    Card-BankCNN Card-Bank;
    Display-CashCNN Display-Cash;
    Cash-ScannerCNN Cash-Scanner;
    Cash-PrinterCNN Cash-Printer;
    Attachment(Cash-Cashier.Connector29Role30.cashier_cashBox,Cashier.cashier_cashBox);
    Attachment(Cash-Cashier.Connector29Role31.cashBox_cashier,CashBox.cashBox_cashier);
    Attachment(Card-Cash.Connector32Role33.cashBox_cardReader,CashBox.cashBox_cardReader);
    Attachment(Card-Cash.Connector32Role34.cardReader_cashbox,CardReader.cardReader_cashbox);
    Attachment(Card-Bank.Connector35Role36.cardReader_bank,CardReader.cardReader_bank);
    Attachment(Card-Bank.Connector35Role37.bank_CardReader,Bank.bank_CardReader);
    Attachment(Display-Cash.Connector23Role24.displayLight_cashBox,
    DisplayLight.displayLight_cashBox);
    Attachment(Display-Cash.Connector23Role25.cashBox_displayLight,
    CashBox.cashBox_displayLight);
    Attachment(Cash-Scanner.Connector20Role21.cashBox_scanner,CashBox.cashBox_scanner);
    Attachment(Cash-Scanner.Connector20Role22.scanner_cashBox,Scanner.scanner_cashBox);
    Attachment(Cash-Printer.Connector26Role27.printer_cashBox,Printer.printer_cashBox);
    Attachment(Cash-Printer.Connector26Role28.cashBox_printer,CashBox.cashBox_printer);
}
```

Figure 48: Part of the TADL specification of CocoMe

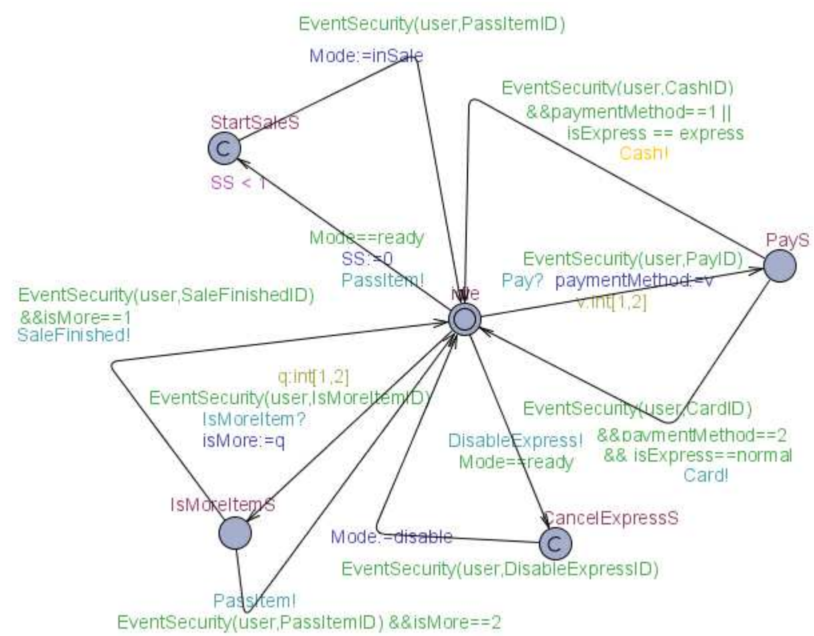


Figure 49: Extended timed automata of the Cashier component

Bibliography

[AAD] Aadl. http://www.aadl.info.

[ABB+02] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jurgen Wust, and Jorg Zettel. *Component-based product line engineering with UML*. Addison-Wesley, 2002.

[ABB+07] Colin Atkinson, Philipp Bostan, Daniel Brenner, Giovanni Falcone, Matthias Gutheil, Oliver Hummel, Monika Juhasz, and Dietmar Stoll. Modeling components and component-based systems in KobrA. In *The Common Component Modeling Example: Comparing Software Component Models [result from the Dagstuhl research seminar for CoCoME, August 1-3, 2007]*, volume 5153, pages 54–84. Springer, 2007.

[ABL96] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995)*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996.

[ÅCF+07] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, 2007.

[AdAdLM05] Alexandre Alvaro, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. Software component certification: A survey. In *Proceedings of the 31st EUROMICRO SEAA*, pages 106–113. IEEE, 2005.

[AFM+03] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: A tool for schedulability analysis and code generation of real-time systems. In *FORMATS*, pages 60–72, 2003.

[AL93] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):73–132, 1993.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[AM07a] Vasu Alagar and Mubarak Mohammad. A component model for trustworthy real-time reactive systems development. In *International Workshop on Formal Aspects of Component Software (FACS07)*, Sophia-Antipolis, France, September 2007.

[AM07b] Vasu Alagar and Mubarak Mohammad. Specification and verification of trustworthy component-based real-time reactive systems. In *SAVCBS'07, Specification and Verification of Component-Based Systems*, Dubrovnik, Croatia, September 2007. ACM.

[APRS01] Colin Atkinson, Barbara Paech, Jens Reinhold, and Torsten Sander. Developing and applying component-based model-driven architectures in KobrA. In *Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, page 212, 2001.

[BBC+07] Lubomír Bulej, Tomas Bures, Thierry Coupaye, Martin Decký, Pavel Jezek, Pavel Parizek, Frantisek Plasil, Tomás Poch, Nicolas Rivierre, Ondrej Sery, and Petr Tuma. Cocome in fractal. In *The Common Component Modeling Example: Comparing Software Component Models [result from the Dagstuhl research seminar for CoCoME, August 1-3, 2007]*, volume 5153, pages 357–387. Springer, 2007.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support

in java: Experiences with auto-adaptive and reconfigurable systems. *Software Practice & Experience*, 36(11-12):1257–1284, 2006.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In *Proceedings of the 4th International School on Formal Methods for Design of Computer, Communicadtion and Software Systems-Real Time*, number LNCS 3185, pages 200–236, 2004.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of the 4th International Conference on Software Engineering Research, Management and Applications*, pages 40–48, August 2006.

[Bis03] Matt Bishop. *Computer Security: Art and Science*. Addison Wesley, 2003.

[CCL06] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. Component-based development process and component lifecycle. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA'06)*, pages 321–327. IEEE Computer Society, 2006.

[CFLGP03] Oscar Corcho, Mariano Fernández-López, and Asunción Gómez-Pérez. Methodologies, tools and languages for building ontologies: where is their meeting point? *Data and Knowledge Engineering*, 46(1):41–64, 2003.

[CHM+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, page 267. IEEE, 2002.

[Chr95] Steven R. Christensen. Software reuse initiatives at lockheed. *CrossTalk*, 8(5):26–31, 1995.

[CJB99] B. Chandrasekaran, John R. Josephson, and V. Richard Benjamins. What are ontologies, and why do we need them? *IEEE Intelligent Systems*, 14(1):20–26, 1999.

[CL02] Ivica Crnkovic and Magnus Larsson, editors. *Building reliable component-based Software Systems*. Artech House Publishers, 2002.

[CRMG08] Leslie Cheung, Roshanak Roshandel, Nenad Medvidovic, and Leana Golubchik. Early prediction of software component reliability. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 111–120. ACM, 2008.

[DSS08] Nigel Daviel, Daniel Siewiorek, and Rahul Sukthanker. Activity-based computing. *IEEE Pervasive Computing*, 7(2):58–61, 2008.

[DT03] Ali H. Dogru and Murat M. Tanik. A process model for component-oriented software engineering. *IEEE Software*, 20(2):34–41, 2003.

[ECC] The electronic components certification board. http://www.eccb.org/.

[FG95] Riccardo Focardi and Roberto Gorrieri. A taxonomy of security properties for process algebras. *JOURNAL of Computer Security*, 3(1):5–34, 1995.

[GHM+08] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler. Owl 2: The next step for owl. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(4):309–322, 2008.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, 1982.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[Gok07] Swapna S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE Transactions on Dependable and Secure Computing*, 4(1):32–40, 2007.

[GS06] David Garlan and Bradley Schmerl. Architecture-driven modelling and analysis. In Tony Cant, editor, *Proceedings of the 11th Australian Workshop on Safety Related Programmable Systems (SCS'06)*, volume 69 of *Conferences in Research and Practice in Information Technology*, pages 3–17, Melbourne, Australia, 2006.

[HC01] George T. Heineman and William T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, June 2001.

[HIPW05] Scott Hissam, James Ivers, Daniel Plakosh, and Kurt C. Wallnau. Pin component technology (v1.0) and its c interface. Technical Report CMU/SEI-2005-TN-001, Software Engineernig Institute, Carnegie Mellon university, April 2005.

[Ibr08] Naseem Ibrahim. Transforming architectural descriptions of component-based systems for formal analysis. Master thesis, Concordia University, 2008.

[IN08] Anne Immonen and Eila Niemelä. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, February 2008.

[LW05] Kung-Kiu Lau and Zheng Wang. A taxonomy of software component models. In *31st EUROMICRO SEAA'05*, pages 88–95, 2005.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.

[MA08] Mubarak Mohammad and Vasu Alagar. TADL - an architecture description language for trustworthy component-based systems. In *2nd European Conference on Software Architecture (ECSA'08)*, LNCS, pages 220–228, Paphos, Cyprus, September 2008. Springer.

[Man00] Heiko Mantel. Possibilistic definitions of security - an assembly kit. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 185–199. IEEE Computer Society, 2000.

[Man02] Heiko Mantel. On the composition of secure systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 88–101, Berkeley, California, USA, 2002. IEEE Computer Society.

[McC88] Daryl McCullough. Noninterference and the composability of security properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 177–186. IEEE Computer Society, 1988.

[McL90] John McLean. Security models and information flow. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 180–189. IEEE Computer Society, 1990.

[McL94] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 79–93. IEEE Computer Society, 1994.

[McL96] John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, 1996.

[MdVHC02] Craig Mundie, Pierre de Vries, Peter Haynes, and Matt Corwine. Trustworthy computing. Microsoft White Paper, October 2002.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[NAD+02] Oscar Nierstrasz, Gabriela Arevalo, Stephane Ducasse, Roel Wuyts, Andrew Black, Peter Muller, Christian Zeidler, Thomas Genssler, and Reinier Born. A component model for field devices. In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment*, June 2002.

[Nei84] James M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, Sebtember 1984.

[Pat] Java PathFinder. http://javapathfinder.sourceforge.net/.

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 55–71. IEEE Computer Society, 2007.

[Pre05] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005.

[Pro] Protege. Stanford University and University of Manchester. http://protege.stanford.edu/.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil. *The Common Component Modeling Example: Comparing Software Component Models*. Springer, 2008.

[RT05] Jie Ren and Richard N. Taylor. A secure software architecture description language. In *Proceedings of SSATTM'05*, Long Beach, CA, USA., November 2005. ACM Press.

[SBI99] Fred B. Schneider, Steven M. Bellovin, and Alan S. Inouye. Building trustworthy systems: Lessons from the ptn and internet. *IEEE Internet Computing*, 3(6):64–72, 1999.

[SG96] Mary Show and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall Publishing, 1996.

[Som07] Ian Sommerville. *Software Engineering*. Addison Wesley, 8th edition, 2007.

[SWM04] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. Owl web ontology language guide. W3C Recommendation, February 2004. http://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[Szy02] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming Second Edition*. Addison-Wesley / ACM Press, 2002.

[TGG07] A. K. Tripathi, Ratneshwer Gupta, and Manjari Gupta. Some observations on software processes for CBSE. *Software Process: Improvement and Practice*, July 2007.

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[XSL] XSLT. http://www.w3.org/TR/xslt.

[YEV08] Haining Yao, Letha H. Etzkorn, and Shamsnaz Virani. Automated classification and retrieval of reusable software components. *Journal of the American society for information science and technology*, 59(4):613–627, 2008.

[Yun09] Zhou Yun. A visual modeling tool for the development of trustworthy component-based systems. Master thesis, Concordia University, 2009.

[Zak96] Aris Zakinthinos. *On the composition of security properties*. Phd thesis, University of Toronto, 1996.