Verification and Validation Techniques in Systems Engineering: Application to State-Chart Diagrams

Payam Kafashe Panjeh Shahi

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

December 2006

Your file Votre référence
ISBN: 978-0-494-28918-1
Our file Notre référence
ISBN: 978-0-494-28918-1

ABSTRACT

Verification and Validation Techniques in Systems Engineering: Application to State-Chart Diagrams

Payam Kafashe Panjeh Shahi

Verification and validation have become very important steps in systems engineering. This is due to the increasing complexity of nowadays systems. Verification and validation aims at detecting flaws early in the design process and/or to verify/validate design models of systems. The state of the art techniques in this field are mainly based on simulation and extensive testing. In this thesis, we propose a new paradigm for verification and validation in systems engineering. It is based on an established synergy between program analysis, software engineering techniques and automatic verification. To illustrate this paradigm, we present a technique for the verification/validation of state-chart diagrams in UML/SysML modeling languages.

To my dear family:

Javad,

Heshmat

and Raham.

And also I would like to appreciate Mr. Mohammed Rostami Safa, The president of Saveh Rolling and Profile Mills Company, for his generosity and gentleness. Without his aid, this thesis could not have been written.

ACKNOWLEDGEMENTS

I have been very fortunate to have Dr. Mourad Debbabi as my supervisor. I am deeply grateful for his strong support and encouragement throughout my master studies. His expertise and judicious advice have shaped the execution of my research.

I also wish to express my gratitude to the examination committee members for reviewing my thesis and giving me valuable feedback.

During my studies in Concordia University many people have encouraged me. I have enjoyed doing research and working with my colleagues. I thank all of them for their support and the nice time we have spent together.

I would like to reserve my deepest appreciations to my family for their perpetual love and encouragements. I can never thank them enough.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background and Motivations

As the size and the complexity of systems continue to increase, verification and validation will become even more essential in the systems development life cycle. Determining whether a system meets its predefined requirements and performs as it is required, have tremendous advantages, such as reduced cost of development, increased efficiency, and enhanced reliability. Exhaustive methods for testing and assuring good quality systems design have become more and more less practical because of the increasing size of and complexity of systems. Both competitiveness and time-to-market requirements mandate that verification and validation be completed within a reasonable period of time and cost. Moreover, as safety becomes a critical concern in many systems, more rigorous methods to verify and validate quality attributes become necessary. Currently, traditional verification and validation methods, such as simulation and semi-formal methods are used to perform the verification and validation tasks. However, these methods are neither rigorous nor exhaustive. Verification and validation are highly recommended, particularly for the development of high safe/secure systems, which require the establishment of their partial or total correctness. The development of these systems under such requirements means that a new paradigm

for verification and validation is needed. This paradigm should cater for automation and rigour. To this end, we propose, in this thesis, a new approach to verification and validation systems engineering. This approach is based on an established synergy between program analysis, software engineering techniques and automatic verification. The advantages of the proposed approach include automation, formality, rigour and cost effectiveness. By automation, we mean the execution of the verification and validation by automatic procedures. This is achieved by using formal automatic verification such as model checking together with program analysis techniques. The proposed approach rests on formal methods that are based on well-defined (syntax and semantics) specification languages and proving techniques (sound semantic based verification algorithms). Rigour comes as a downstream result of formality. The proposed approach is cost effective since it automates verification and validation tasks and abstracts away the underlying complexity in well-defined and automated techniques.

1.2 Objectives

The primary objectives of this thesis are:

- To report and present the main standardization bodies, initiatives and languages in the area of systems engineering.
- To compile and compare the state of the art in terms of verification and validation and systems engineering.
- To prepare a new approach to do verification and validation in systems engineering that will achieve more automation, rigour and formality.
- To illustrate in detail the proposed approach on state-chart diagrams in UML/SysML modeling languages.

1.3 Contributions

The main contributions of this thesis are:

- Study of the most prominent standardization bodies, initiatives and languages in systems engineering.
- Study of the state of the art techniques in the area of the systems engineering.
- Elaboration of a new approach for verification and validation for systems engineering that is based on an established synergy between program analysis, software engineering techniques and automatic verification.
- Elaboration of verification techniques for state-chart diagrams for UML/SysML modeling languages.
- Illustration of the verification techniques on a case study.
- Design and implementation of a verification and validation software framework that prototypes the proposed approach.

1.4 Structure

Here is the way the rest of the thesis is organized. Chapter 2 is dedicated to a presentation of the relevant standardization bodies, standards and initiatives in the area of systems engineering. Chapter 3 is devoted to a presentation of some techniques that are relevant to verification and validation in systems engineering. Control flow analysis, data flow analysis and slicing are briefly introduced. Moreover, a presentation of the state of the arts techniques for the verification and validation in systems engineering is given. Chapter 4 lays down the detail of the new proposed approach for verification and validation in systems engineering.

Chapter 5 illustrates the proposed techniques on state-chart diagrams in UML/SysML languages. Chapter 6 provides a case study while Chapter 7 contains some conclusions on this research together with a discussion of future work.

Chapter 2

Systems Engineering, Definition and Standards

System engineering is a discipline that develops and exploits structured, efficient approaches to analyze and solve complex engineering problems. It covers the entire life cycle of an engineering system in order to ensure that the customer's requirements are satisfied. Currently, systems engineers are using different documentation approaches to express the system requirements. They are also using many modeling techniques to give a complete design of a system. This diversity of techniques and approaches limits the communication and the exchange of information among system engineers. Thus, international standards are needed to produce an effective and synergic collaboration among system engineers. It will facilitate worldwide system engineering technologies compatibility and interoperability. Many organizations have been working on providing standard frameworks and modeling languages for system engineering. This section provides background information about some of these Standard Organizations such as Object Management Group (OMG), INternational Council On System Engineering (INCOSE) and the International Standard Organization (ISO). Also it will present some initiatives and modeling languages related to system engineering developed by these organizations.

2.1 Object Management Group

OMG [14] is an international association founded in 1989. OMG [14] provides standards for object-oriented applications to help reduce complexity, lower costs, and hasten the introduction of new software applications. The OMG [14] is accomplishing these goals through the introduction of a suite of specifications that will lead the industry towards interoperable, reusable, portable software components and data models. OMG [14] has established numerous widely used standards such as MDA and MOF to name a few significant ones related to system engineering. The following sections present these standards.

2.1.1 Model Driven Architecture

The Model-Driven Architecture (MDA) starts with the well-known idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform. It is an approach to using models in software development, including the writing of specifications and the actual developing of applications, which allow the functionality and behavior of the system to be separated from implementation details. This enables the application to be easily ported from one environment to another by first creating one or more Platform Independent Models (PIM) that are later translated into one or more Platform Specific Models (PSM). MDA is being developed to include a broad range of concepts (such as the separation of the model from the implementation) so that it can be applied to all types of software development projects including electronic commerce, financial services, health care, aerospace and transportation. MDA provides an approach, and enables tools, to:

- Specify a system independent of the platform that supports it.
- Specify platforms.
- Choose a particular platform for the system.

- Transform the system specification into one for a particular platform.

The three primary goals of MDA are portability, interoperability, and reusability through architectural separation of concerns. MDA is based on the four-layer [15] meta modeling architecture, and several OMG's complementary standards as shown in Figure 2.1. These standards are Meta-Object Facility (MOF), Unified Modeling Language (UML) and (XMI). The layers are (1) meta-meta model layer, (2) meta model layer, (3) model layer, and (4) instance layer.

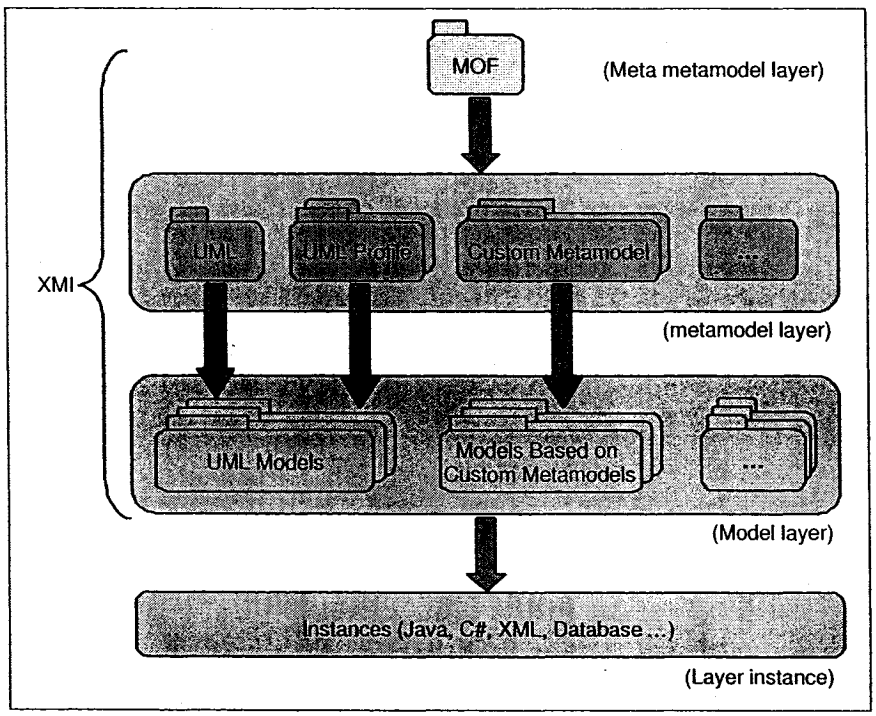

Figure 2.1: MDA Layers

The main objective of having four layers with a common meta-meta model architecture is to support multiple meta models and models, and to enable their extensibility and integration.

2.1.2 Meta Object Facility

The Meta Object Facility (MOF) [13] defines an abstract language and framework for specifying, constructing and managing technology neutral meta models. The MOF specification is intended to provide information on modeling capability. It is the OMG's standard for defining modeling languages and their interoperability. In this context, the MOF Model is referred to as a meta-meta model because it is being used to define meta models such as UML. The MOF provides a formal and clear semantics for each element of the UML meta model.

The UML and MOF are based on a conceptual layered meta model architecture, Figure 2.2, where elements in a given conceptual layer describe elements in the next layer down. For example:

- The MOF meta-meta model is the language used to define the UML meta model.
- The UML meta model is the language used to define UML models.
- A UML model is a language that defines aspects of a library system.

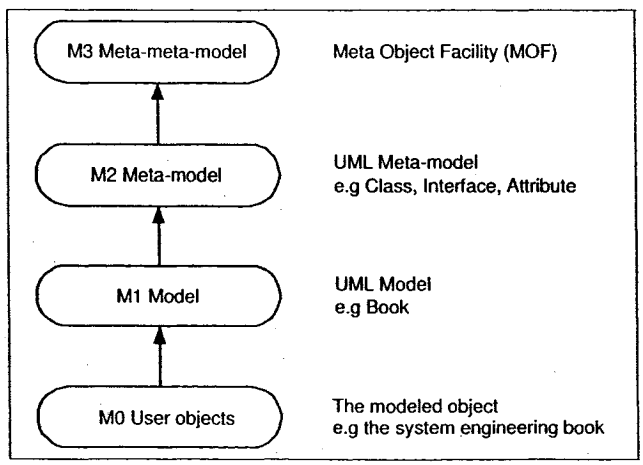

Figure 2.2: OMG 4-Layer Architecture

However, the UML modeling language was released before the MOF. Therefore, MOF and UML are very similar. Actually MOF is defined by a subset of the elements used in the UML core. In a nutshell, the four main modeling concepts are:

- Classes, which model MOF meta objects.
- Associations, which model binary relationships between meta objects.
- Data Types, which model other data (e.g., primitive types, external types, etc.).
- Packages, which modularize the models.

The MOF standard has also contributed to provide a solid foundation to the MDA architecture. The MOF formal meta models are used to define the concepts of Platform Independent Models (PIM) that can be mapped to Platform Specific Models (PSM). These two models are used to achieve the main goals of the MDA.

2.1.3 XML Metadata Interchange Format

The XML Metadata Interchange Format (XMI) [19] was proposed in response to an OMG Request For Proposal (RFP) on 1997-12-03 relating to a Stream-based Model Interchange. The RFP solicited proposals "for a transfer format specification for file export/import of models, and a transfer format specification for unique identification of the version of the MOF meta-meta model and any meta models referenced." Three initial submissions relative to the RFP were received, from: 1) Daimler-Benz Research and Technology and Recerca Informtica 2) Fujitsu/Softeam; and 3) a larger industry group led by DSTC, IBM, Oracle, Platinum Technology, and Unisys. The XMI proposal came as an initial submission from the third group. The first two submissions address the role of XML, but in neither case does XML constitute a central feature as in the XMI proposal.

In its initial proposal, the XMI specifies an open information interchange model that is intended to give developers working with object technology the ability to exchange programming data over the Internet in a standardized way, thus bringing consistency and compatibility to applications created in collaborative environments. By establishing an industry standard for storing and sharing object-programming information, development teams using various tools from multiple vendors can still collaborate on applications. The proposed standard will allow developers to leverage the web to exchange data between tools, applications, and repositories to create secure, distributed applications built in a team development environment. The XMI standard would combine the benefits of the web-based XML standard for defining, validating, and sharing document formats on the web with the benefits of the object-oriented Unified Modeling Language (UML). This is a specification of the OMG that provides application developers a common language for specifying, visualizing, constructing, and documenting distributed objects and business models. The main purpose of XMI is to enable easy interchange of meta data between modeling tools (based on the OMG UML) and between tools and meta data repositories (OMG MOF based) in distributed heterogeneous environments. XMI defines many of the important aspects involved in describing objects in XML:

- The representation of objects in terms of XML elements and attributes is the foundation of XMI.
- Since objects are typically interconnected, XMI includes standard mechanisms to link objects within the same file or across files.
- Object identity allows objects to be referenced from other objects with their IDs.
- The versioning of objects and their definitions is handled by XMI.
- XMI documents are validated using DTDs and Schemas.

Figure 2.3 illustrates an example of a XMI file.

```
    <xsd:attribute name="position" type="xsd:string" use="optional"/>
     <xsd:attribute name="addition" type="xsd:IDREFS" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>

<xsd:element name="Add" type="Add"/>

<xsd:complexType name="Replace">
 <xsd:complexContent>
  <xsd:extension base="Difference">
   <xsd:attribute name="position" type="xsd:string" use="optional"/>
   <xsd:attribute name="replacement" type="xsd:IDREFS" use="optional"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:element name="Replace" type="Replace"/>

<xsd:complexType name="Delete">
 <xsd:complexContent>
  <xsd:extension base="Difference"/>
 </xsd:complexContent>
</xsd:complexType>

<xsd:element name="Delete" type="Delete"/>
```

Figure 2.3: XMI File Example

We have used XMI as an input for creating semantic models. Therefore, every modeling design in different platforms that can be exported to XMI can be converted to the semantic model, and verification and validation can then be performed on these models. Figure 2.4 shows an XMI example.

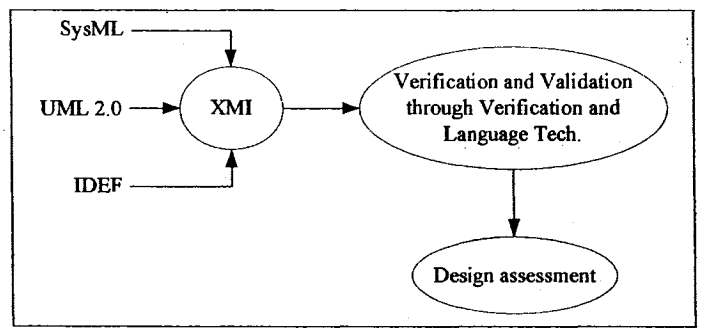

Figure 2.4: XMI Diagram

2.2 INternational COuncil on Systems Engineering

The INternational COuncil on Systems Engineering (INCOSE) [34] is an international professional society for systems engineers whose mission is to foster the definition, understanding, and practice of world class systems engineering in industry, academia, and government. INCOSE was formed in 1992 to develop, and enhance the interdisciplinary approach to enable the realization of successful systems. SysML initiative can be traced to a decision to pursue an extension of UML for systems engineering at the Model Driven Systems Design workgroup meeting at the International Council on Systems Engineering (INCOSE) in January 2001. This resulted in a collaborative effort between INCOSE and OMG.

The decision was made by the OMG to pursue UML for systems engineering. In March 2003, OMG issued a Request for Proposal (RFP) for a customized version of UML suitable for Systems Engineering. There was only one technology submission to the RFP, which was by the SysML group, proposing a Systems Modeling Language (SysML). The latter defines a general-purpose modeling language for systems engineering applications. It supports the specification, analysis, design, verification and validation of a broad range of complex systems that may include hardware, software, data, personnel, procedures, and facilities.

2.3 International Organization for Standardization

International Organization for Standardization (ISO) [35] is the world's largest developer of standards. Their standards are useful to industrial and business organizations of all types, to governments and other regulatory bodies, to conformity assessment professionals, to people in general in their roles as consumers and end users. ISO 10303 is one of the standards related to product data representation and exchange which is implemented by XML.

2.4 Unified Modeling Language

The Unified Modeling Language (UML) [37, 39] is a language that enables system developers to specify, visualize, and document models. These models are used to abstract the implementation details of software or systems. This abstraction enables developers to prepare the implementation of a system carefully before actually implementing it.

UML is the simplifying and the merging of three major notations. They are Grady Booch's methodology for describing a set of objects and their relationships, James Rumbaugh's Object-Modeling Technique (OMT), and Ivar Jacobson's approach, which includes a "use case" methodology. They were working to unify their methods at Rational Software and this work resulted in their first proposal in 1995.

OMG is a non-profit consortium that develops and maintains computer industry specifications. In 1996, they issued a request for proposal for a standard approach to object-oriented modeling. Booch, Rumbaugh, and Jacobson began working together with other developers and companies to create such a proposal for the OMG. In September 1997, they submitted their proposal, which was adopted unanimously two months later by the consortium. The OMG assumed responsibility for the maintenance of UML.

2.4.1 History

This section describes the history of the Unified Modeling Language. A summary of the history of UML is shown in Figure 2.5. Identifiable object-oriented modeling languages began to appear between the mid-1970s and the late 1980s. Their number increased from less than ten to more than fifty during the period between 1989 and 1994. However, none of these languages satisfied the oriented-object designers completely. The development of UML began in late 1994 when Grady Booch of Rational Software Corporation started his work on unifying the Booch and Object Modeling Technique (OMT) methods [23]. OMT, developed by Jim Rumbaugh, focused on the analysis of business and data intensive systems.

The Booch method, developed by Grady Booch, had particular strengths in design and implementation. These two methods were merged in the fall of 1995 and referred to the new language as the Unified Modeling Language version 0.8. At that time, Ivar Jacobson and his Objectory company joined Rational and this unification effort started by merging the Object-Oriented Software Engineering (OOSE) method with UML. OOSE is based around the use case concept that proved itself by achieving high levels of reuse by facilitating communication between projects and users [37].

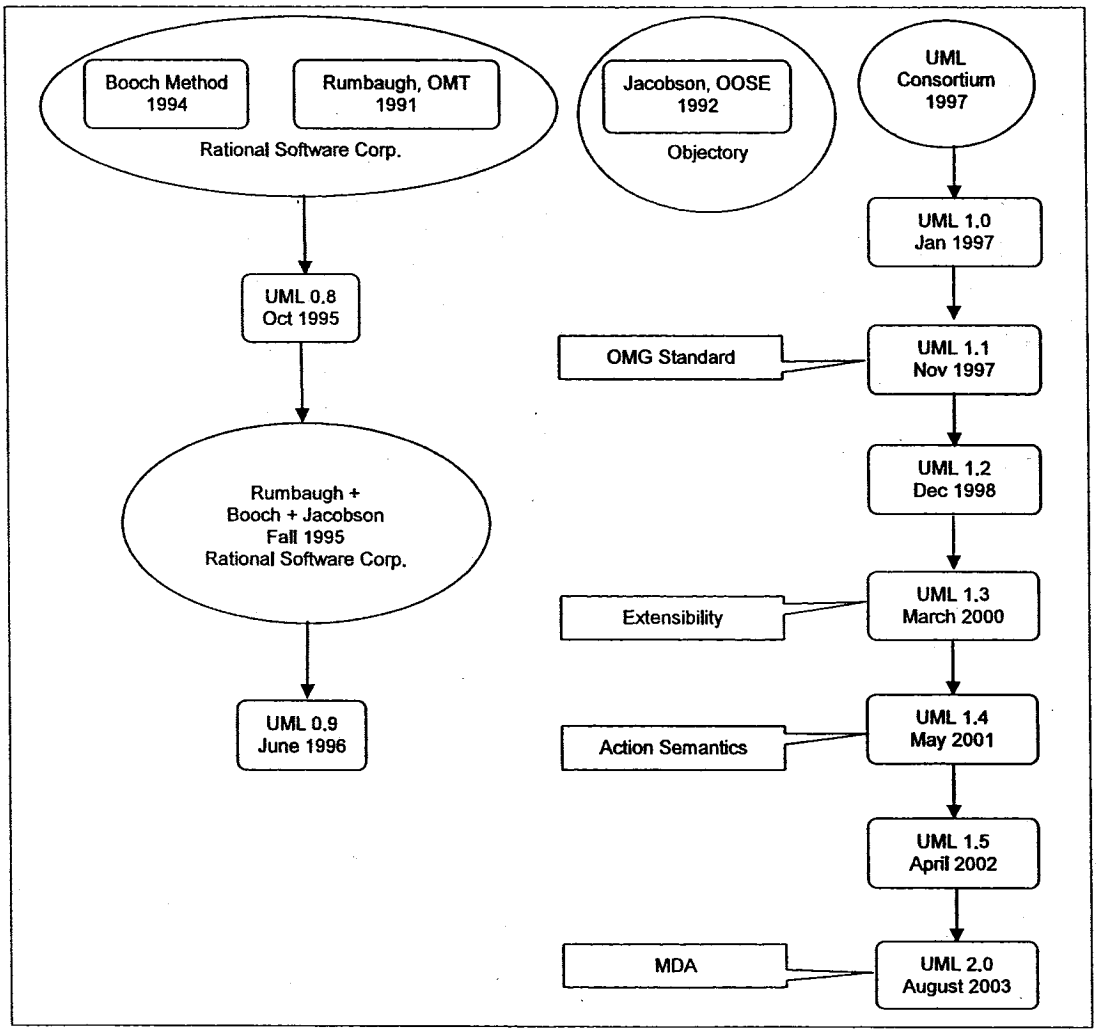

Figure 2.5: History of UML

Grady Booch, Jim Rumbaugh, and Ivar Jacobson were motivated to create a unified modeling language for three reasons. First, these methods were already evolving towards each other independently. Second, they brought a mature modeling language and some stability to the object-oriented marketplace by unifying the semantics and notation. Third, they expected that their collaboration would yield improvements in all three earlier methods, helping them to capture lessons learned and to address problems that none of their methods previously handled well. Grady Booch, Jim Rumbaugh, and Ivar Jacobson established four goals:

- Enable the modeling of systems (and not just software) using object-oriented concepts.
- Establish an explicit coupling to conceptual as well as executable artifacts.
- Address the issues of scale that is inherent to complex and mission-critical systems.
- Create a modeling language usable by both humans and machines.

The result of this collaboration led to the release of the UML 0.9 and 0.91 documents in June and October 1996. As the UML designers observed that additional focused attention was still required, they invited and received feedback from the general community. They tried to achieve the broader goal of an industry standard modeling language. In early 1995, Ivar Jacobson and Richard Soley decided to push harder to achieve standardization in the methods marketplace. During 1996, a Request for Proposal (RFP) issued by the OMG provided the catalyst for some organizations to join forces around producing a joint RFP response. Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML 1.0 definition. Those contributing most to the UML 1.0 definition included: Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys. This collaboration was submitted to the OMG in January 1997 as an initial RFP response.

In January 1997, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, and Softeam also submitted separate RFP responses to the OMG. These companies joined the UML partners to contribute their ideas. Together, the partners produced the revised UML 1.1 response. The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners. It was submitted to the OMG for their consideration and adopted in the fall of 1997 [37]. Then The standard has progressed through version 1.3 and on to version 1.4. The most recently adopted specification is version 1.4.1 in September 2002, which added Action Semantics. Although UML 1.x has enjoyed widespread acceptance, it has some shortcomings such as the lack of support for diagram interchange. Also, it is considered as being too complex and having an inadequate semantics definition. Finally, UML 1.x is not fully aligned with MOF. Thus, a major revision is required to address these problems [29]. For that reason, a formal RFP requirements has been issued that contains the following points:

- UML internals must have a more precise conceptual base for better MDA support.
- The user-level features must have new capabilities for large-scale software systems and consolidate the existing features.
- UML must support a constraint language.
- UML must have support for a standard for exchanging graphic information.

A new version of UML (UML 2.0) [29] has been specified to correct the shortcomings of UML 1.x.

2.4.2 Infrastructure and Superstructure

The UML infrastructure [15] refers to language's metamodel, which defines the semantics for how elements of a model get instantiated in another model. The core metamodel is used to define MOF [13], UML, Common Warehouse Metamodel (CWM) [12], and profiles. The

infrastructure is the top metamodel of the languages so they all derive from it. By deriving from the infrastructure, the languages are architecturally aligned. This permits the transfer of models from one to another. The infrastructure of UML is defined by the infrastructure library, which is a set of packages that satisfy the following requirements:

- It defines the core of a metalanguage that is to be reused to define a variety of metamodels, such as UML, MOF and CWM.
- It aligns the architectures of UML, MOF, and XMI so that model interchange is fully supported.
- It allows the customization of UML through profiles and the creation of new languages based on the same core as UML.

The infrastructure library is composed of the core package and the profiles package. The UML core package contains the complete metamodel particularly designed for high reusability. This allows other metamodels to benefit from the abstract syntax and semantics that have already been defined. Other metamodels at the same metalevel (e.g. UML) either import or specialize the specified metaclasses. The modeling languages UML, CWM, and MOF each depend on the common core. The idea is to allow UML and other MDA metamodels to reuse parts of the core package so that they can benefit from the abstract syntax and semantics that have already been defined.

To make the reuse easier, the core package is subdivided into four sub-packages:

- Primitive Types: Includes a few predefined primitive types (e.g. integer, boolean, and string) that are commonly used in metamodeling. They are designed specifically for UML and MOF. They define a small set of data types that are used to sepcify the core metamodel.
- Abstractions: Defines the common concepts (e.g. classifiers, behavioral elements, and generalizations) needed to build modeling elements and contains abstract metaclasses

that are intended to be further specialized or commonly reused by other metamodels. These metaclasses define the fundamental concepts that are common to most modeling languages.

- Basic package: Defines the common characterics of classifiers, classes, data types, and packages and refines the contents of the abstractions.
- Constructs: Contains concrete metaclasses (metaclasses that are not abstract) that lend themselves primarily to object-oriented modeling.

The UML metamodel is defined as one of the layers of a four-layer metamodeling architecture [15]. The first layer of the hierarchy is the meta-metamodeling layer. The purpose of this layer is to define the language for specifying a metamodel. It is more compact than the metamodel that it describes and can normally be reused to define several metamodels. The second layer is the metamodeling layer, which is an instance of the meta-metamodel. This means that every element of the metamodel is an instance of an element in the meta-metamodel. The primary purpose of the metamodel layer is to define a language for specifying models. They are more elaborate than the meta-metamodels of the first layer. Examples of metamodels in that layer include UML and CWM. The third layer is the model layer, which is an instance of the metamodel layer. The primary purpose is to allow users to model a wide variety of different types of systems, such as software or engineering systems. The fourth layer contains the run times instances of the model elements defined above.

One of the goals of UML 2.0 was to standardize and align the language with MDA. Creating the infrastructure was part of this goal. By making the infrastructure the top metamodel, other languages would also derive from it making it possible to create a language structure that facilitates reuse. Finally, the UML superstructure [17] defines the user level constructs required for UML 2.0. It is used to extend and customize the UML infrastructure to define the UML metamodel. The elements that make up the modeling notations of UML are defined in the superstructure. These are designed by adding or extending elements

defined in the infrastructure.

2.4.3 Profiles

The profile in UML allows the customizing of UML for a specific domain. The profiles in UML are in a package that contains mechanisms, which allow metaclasses from existing metamodels to be extended to adapt them for different purposes.

UML is defined using a metamodel, which is a model of a modeling language. The metamodel cannot be changed because it is too complicated. It would also be dangerous to modify the metamodel because that could make some of the UML applications, which rely on this metamodel, useless. Therefore, the profile is used to permit limited changes to the language. These changes do not affect the UML metamodel. UML includes three main extensibility mechanisms, constraints, stereotypes, and tagged values. Constraints are shown in Figure 2.6. They are textual statements expressed either in a formal language or in natural language. They are used to specify more details about components represented in the UML models. These details normally restrict how a component will be used. Constraints are normally written next to the UML element that is being restricted.

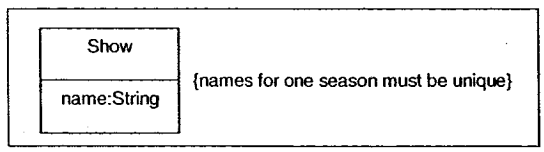

Figure 2.6: Example of a Constraint

The stereotype in UML is essentially a metaclass, which is a class whose instances are classes. These are the classes that are used to create metamodels. They are used to represent a variation of an existing model element (e.g. a different use) with the same form but a different intent. In the UML models, stereotypes are shown in between the "<<>>" symbols. Stereotypes extend the semantics but not the structure of the metamodel classes. Tagged values are a named piece of information attached to a component in models created

in UML. They represent additional modeling information beyond that information defined in the metamodel. Tagged values along with stereotypes are shown in Figure 2.7.

These three constructs permit many kinds of extensions to the UML metamodel without imposing modifications on it. Together, the constraints, stereotypes, and tagged values are refrered to as the profile

The classification of the UML 2.0 diagrams can be seen in Figure 2.8 that shows the taxonomy of the diagrams. The diagrams are divided into three categories, the structure diagrams, the behavior diagrams, and the interaction diagrams. The structure diagrams contain the class, component, composite structure, deployment, object, and package diagrams. The behavior diagrams contain the activity, use case, and state machine diagrams as well as all the interaction diagrams, which contain the communication, interaction overview, sequence, and timing diagrams.

There are a few changes in the UML 2.0 taxonomy compared with the previous versions. Some diagrams are new. They are the composite structure, the interaction overview, and the timing diagrams. Other diagrams have been modified from the previous versions. They are the activity, state machine, sequence, and communication diagrams.

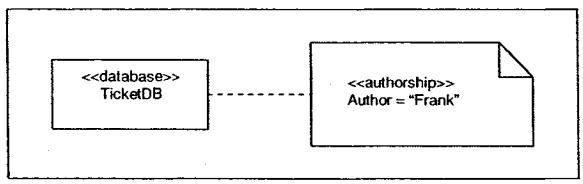

Figure 2.7: Example of a Stereotype and a Tagged Value

The structural diagrams show the static features of a model. Static features include classes and associations, objects and links, and collaborations. These static features provide the skeleton in which the dynamic elements of the model are executed.

The behavioral diagrams show the interactions among resources modeled in the structural diagrams and how they execute their capabilities. The interaction diagrams are a subset of the behavior diagrams. They emphasize on the interactions between objects.

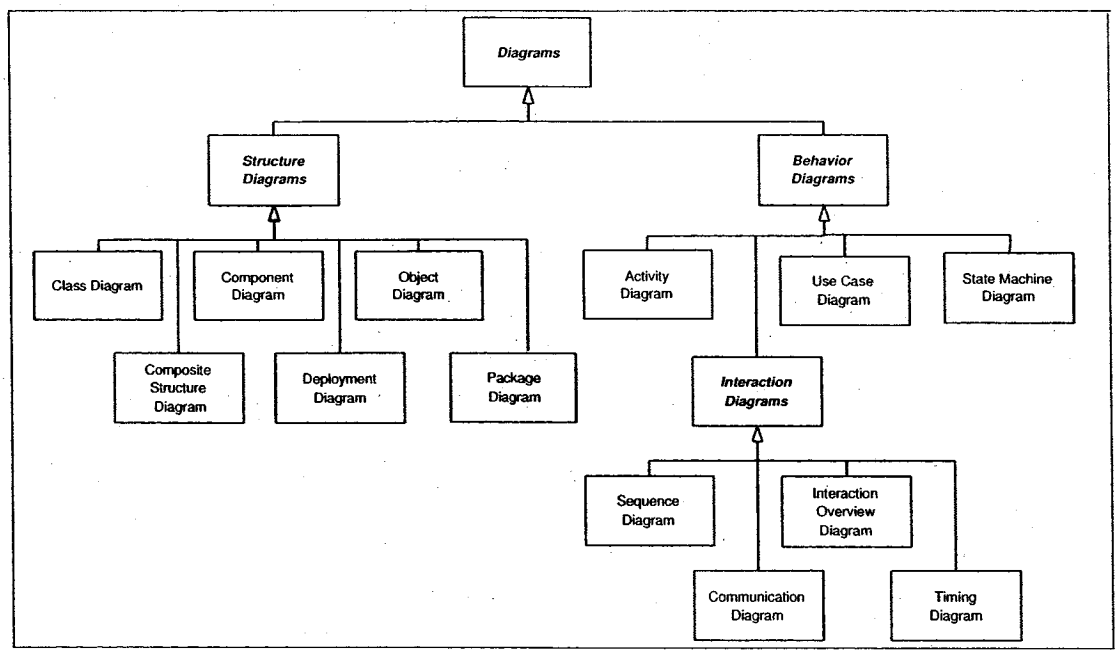

Figure 2.8: UML 2.0 Taxonomy

A state machine diagram shows the behavior of an entity. The state machines describe the significant states (conditions) that an entity can be in as a result of environmental stimuli or events that cause the entity to change. They describe the behavior of the entity and show the transitions between discrete states (conditions) the entity can exist in. Transitions can show the events that cause the changes. Guards determine if the state can change and the actions that occur. There may be a number of state machine diagrams that are needed to describe the entity and each diagram could focus on a different aspect of the entities' behavior. The purpose of the state machine diagram is to:

- Depict the various states that an object may be in.
- Show the transitions between the states of the objects.

The syntax and semantics of the state machine diagram contain a number of states. They also contain state transitions, which are events that cause objects to change into other

states. A state machine diagram contains an initial state node and a final state node, which show the beginning and the end of the execution of the diagram.

Figure 2.9 shows an example of state machine diagram. The example shows a complex state machine representing a phone system. The state machine is the part inside the inner rectangle. The rest of the diagram is the rest of the system, which starts at the empty circle and ends at the circle with an X inside. Once the system enters the state machine, it starts its execution at the small filled black circle and executes all the operations of the machine by following the arrows. This is similar to the activity diagram but in this case, the execution has some constraints. For example, if, after playing the dial tone for 15 seconds, the system is still at that state, it will advance to the timeout state.

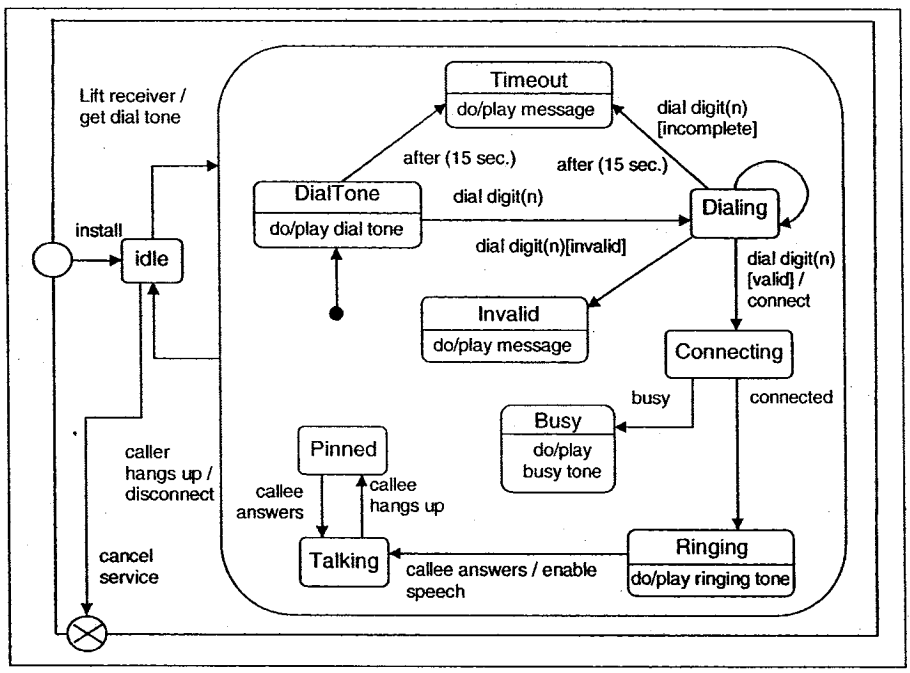

Figure 2.9: Example of a State Machine Diagram

2.5 System Modeling Language

The System Modeling Language (SysML) is a general-purpose system modeling language that supports the specification, analysis, design, verification, and validation of a broad range of complex systems. These systems may include hardware, software, data, personnel, procedures, and facilities.

At the moment, the SysML specification is not complete; Therefore, some elements in this section are not yet definitive.

SysML is a joint effort between the INCOSE and OMG. Companies that are involved in the partnership are: American Systems Corporation, ARTISAN Software Tools, BAE SYSTEMS, The Boeing Company, Ceira Technologies, Deere & Company, EADS Astrium GmbH, Eurostep Group AB, Gentleware AG, I-Logix, International Business Machines, International Council on Systems Engineering, Israel Aircraft Industries, Lockheed Martin Corporation, Motorola, National Aeronautics and Space Administration, National Institute of Standards and Technology, Northrop Grumman, oose.de Dienstleistungen fur innovative Informatik GmbH, PivotPoint Technology Corporation, Popkin Software, Project Technology, Raytheon Company, Structured Software Systems Limited, Telelogic AB, THALES, and Vitech Corporation.

The goal of designing SysML is to satisfy basic requirements of the systems engineering community as defined in the UML for systems engineers RFP [18]. Some constructs and diagrams are added to this UML subset as necessary to address other systems engineer's requirements to make SysML easier to learn, implement, and apply. SysML reuses UML (Figure 2.10) when it is required and when it is practical in a manner that minimizes changes to the underlying language. Therefore it would be easy to implement SysML tools for vendors who already support UML.

SysML is aligned with the ISO AP233 data interchange standard to support interopertability among engineering tools. Since it is a customization from UML, it also inherits

the XMI interchange from its predecessor.

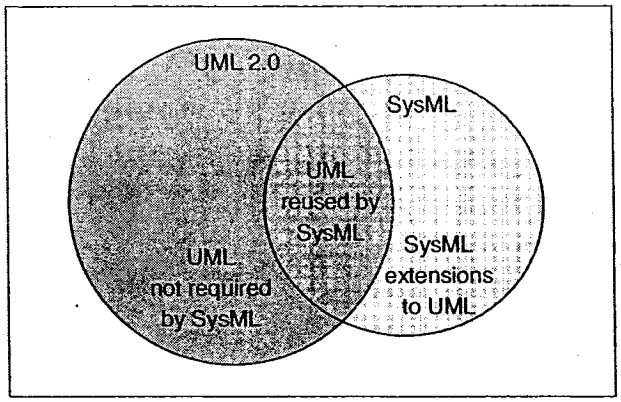

Figure 2.10: UML and SysML Compliance

2.5.1 History

Since 1997, UML has been widely used among the software engineering community. It is also being used in many other disciplines such as business processes and human resources. Many systems engineers use UML to model various kinds of systems such as real-time systems and hardware components. Some system engineers did not adopt UML and they decided to develop some alternatives to this modeling language. A common approach was to model additional systems engineering concepts in other modeling tools. This made it difficult to integrate the different viewpoints and achieve traceability. Therefore, OMG made the decision to pursue UML for systems engineering.

In March 2003, OMG issued a Request for Proposal (RFP) for a customized version of UML suitable for systems engineering [18]. The customization of UML for systems engineering is intended to support the modeling of a broad range of systems, which may include hardware, software, data, personnel, procedures, and facilities. There was only one technology submission to the RFP, which was made by the SysML group, proposing the Systems Modeling Language, SysML.

SysML reuses UML 2.0 diagrams to express system design models and some of the UML 2.0 diagrams were discarded such as the communication diagram and deployment diagram. Two new diagrams were also added. They are the parametric diagram and the requirement diagram. Figure 2.11 illustrates the complete SysML taxonomy of diagrams.

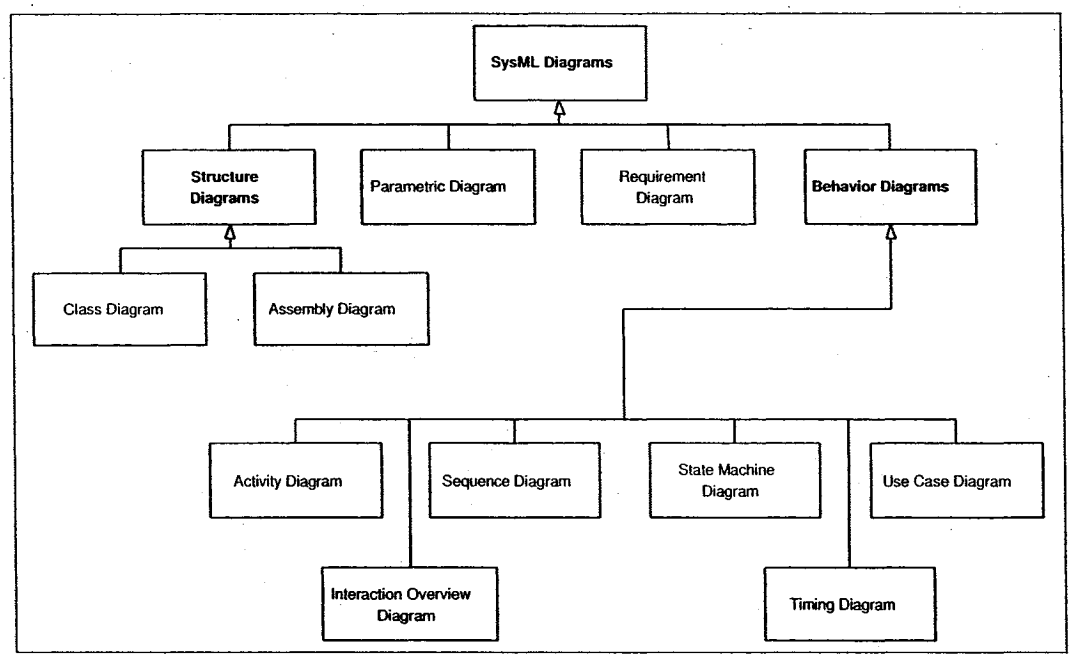

Figure 2.11: SysML Diagrams Taxonomy

2.5.2 Diagrams

The SysML partners have introduced two new diagrams to make systems modeling easier and more efficient. These are the parametric diagram and the requirement diagram. Four diagrams were modified: activity, assembly, class, and state machine. There are also four diagrams that have not been modified from UML 2.0: Interaction overview, sequence, use case, and timing.

Parametric Diagram

The parameter is a measurable factor, such as temperature or pressure, that defines a system and determines its behavior and varies in an experiment. This is why the SysML partners plan on adding the parametric diagram to their standard.

The parametric diagram was added to SysML to bring more compliance with other system modeling tools. It can be used to do simulations on models by modifying some values and observing changes to the whole system. The parametric diagram can also be used to perform various types of analysis. Changing a value in a parametric diagram and observing the consequences on the whole system, can identify potential problems. It shows what conditions could make the system unreliable.

The motivation and purpose of the parametric diagram is that many systems engineering tools offer ways to express mathematical relations among properties but UML 2.0 does not offer any way to express these relations. The parametric diagram defines a set of quantifiable characteristics and relations between them. These relations state how any verification on a value of one property impacts the value of other properties. This modeling of relations among properties is extremely valuable for analysis purposes. The result of the analysis on this diagram can be included in a verification or validation process to help assess performances and reliability.

Parametric syntax can be used in a parametric diagram but it can also be used in other diagrams. Assembly diagrams can easily be enhanced with parametric relations. Parametric Diagrams rely on other mathematical languages, such as MathML, to express equations relating to different properties.

Requirement Diagram

The requirement is a feature, a property, or a behavior that a system has to provide. Listing requirements is the very first step in the process of designing a system. It allows the designer to state explicitly what is expected from the future system. Requirements are also useful

in the verification and validation phase of the development because it is possible to verify if the system that has been built fulfills the requirements that were specified.

This new diagram produces a way to depict requirements and to relate them to other specification, design, or verification models. The requirements can be represented in graphical, table, or tree format. Requirements can be expressed and stored in different ways. A very common way to keep track of requirements is the usage of requirement management tools. While these tools are very powerful and can store a lot of information, they are also harder to integrate with other model elements. This is the strength of the requirement diagram; it allows to easily understand relations between requirements and their environment.

Chapter 3

Verification, Validation and Accreditation in Systems Engineering

3.1 Introduction

This chapter describes verification and validation techniques which are used in the thesis. We have described the modeling languages that we will use and the properties that can be verified and validated on the systems modeled by these languages. Now, we describe the methods that we will use to perform the verification and validation of those properties on the models. There are three main types of techniques that we introduce in this chapter. They are formal verification, program analysis, and software engineering. Formal verification includes methods such as model-checking and theorem proving. Program analysis includes flow analysis, type-based analysis as well as abstract interpretation. The software engineering techniques enable the verification and validation of coupling, cohesion, and object oriented metrics.

3.2 Formal Verification

Formal verification refers to the variety of scientific and engineering techniques for verifying and validating the correctness of systems. These techniques are often based on mathematical logic and can be used to perform verification and validation on a number of UML and SysML diagrams. For each technique that we discuss here, we describe what it is, how it operates, and we address the underlying advantages and disadvantages.

One of the goals of formal verification is to examine all the possible behaviors of a system to determine if it satisfies some properties or not. They can reveal inconsistencies, ambiguities, incompleteness as well as several other shortcomings in a system. Previous methods such as simulation and testing only permit the verification of some behaviors of systems. Those behaviors that have been tested or simulated may satisfy a property but those that have not been simulated and tested may not. Formal verification allows to verify properties by exhaustively checking all of the system states.

In this section, we concentrate on two formal verification techniques: Model-checking and theorem proving. Model-checking stands for the verification of a model by brute force enumeration. The model is usually expressed as a directed graph. Each node in the graph represents a state of the system and contains a set of properties. Model-checking verifies that each state of the graph satisfies the properties associated with the state. Theorem proving is a technique where both the system and its desired properties are expressed as formulas in a mathematical logic. This logic is given in a formal system, which defines a set of axioms and a set of inference rules. Theorem proving is the process of deriving a proof of a property from the axioms of the system. Proving properties is done in a series of derivations that involve instantiations of axioms and inference rules.

- **Model-checking**

  Model-checking [38] is a method for formally verifying finite-state (and sometimes even infinite-state) systems. Given a system model and some desired system properties,

model-checking explores all the states of the model to check whether the given system properties are satisfied by the model. Model-checking either proves that the properties are satisfied or generates counter examples that show that those properties are not satisfied.

The advantages of model-checking include automation and the generation of counter examples for those properties that are not satisfied. Automation means that performing model-checking is something that can be done automatically. Accordingly, the inner workings of the model-checking procedures are transparent to the system verified. Another advantage is that if the model-checking finds that a property is not satisfied then it can generate a counter example to show how and where the problem occurs. This might be of a great help to analyze the system design to meet the desired property in question.

The main disadvantage of model-checking is the state explosion problem. Since model-checking explores all the possible behaviors of the system, there can be many states to explore. If the number of possibilities become too large, model-checking might take too long to perform the verification. Many techniques have been developed to help alleviate the temporal and spatial needs of verification.

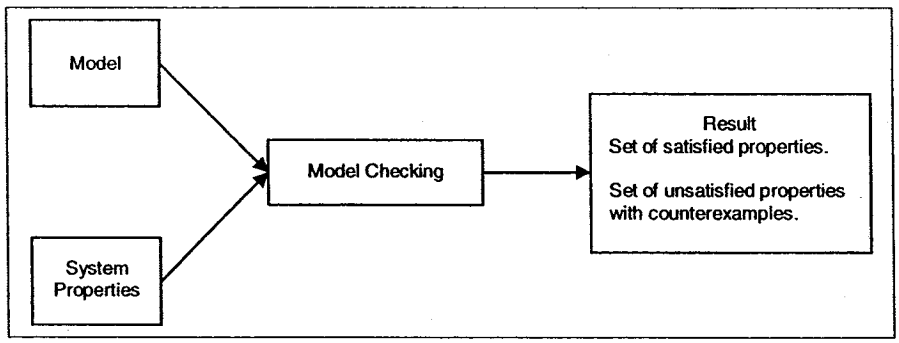

Figure 3.1: Formal Verification by Model-Checking

As shown in Figure 3.1, model-checking takes as input a model together with a set

of properties to be verified. Model-checking will produce as output two sets: the set of satisfied properties and the set of unsatisfied properties with counter examples. With the counter examples, it is possible to fix the model and then re-run again the model-checking procedure.

- **Theorem proving**

  Theorem proving [27] is a method for performing verification on formal specifications of system models. To use theorem proving, one applies a set of rules of inference to a specification in order to derive new properties of interest. Tools used for theorem proving consist of a collection of inference steps that can be used to reduce a proof's goal to simpler subgoals that can be discharged automatically by the primitive proofs of the tool. For theorem proving, a property and a model are used. The goal is to verify the given property. The theorem prover is either able to verify the property by completing the proof or find subgoals that give scenarios in which the property is violated. Theorem proving usually requires considerable technical expertise and understanding of the specification. It gives developers a lot of flexibility and control in doing the proof. An advantage of theorem proving is that it is not limited by the size of the model. Large systems that cannot be verified using model-checking, can still be verified by theorem proving since state space explosion is not anymore an issue. A disadvantage of theorem proving is that if one fails to complete the proof of a property, it does not mean that the property cannot be proved. It may just mean that the prover is not provided with enough information to complete the proof.

  Theorem proving works by using the proof inference technique. A developer validates the design of a system by proving a conjecture that describes how the system should work by using axioms that describe the system itself. As shown in Figure 3.2, the properties that the developed system must have are converted into a logical representation. The system (or the model of the system) is also converted into that particular

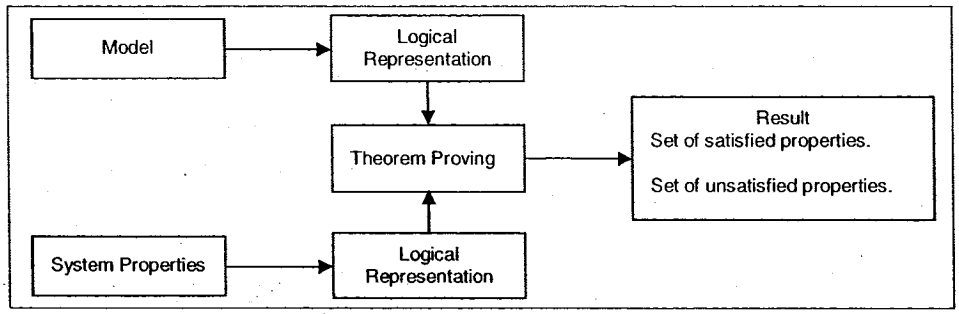

Figure 3.2: Theorem Proving Formal Verification

form. The results of the converted system are then used to show that the properties that the system must have are true or false. If the prover shows that the properties are true then the system has been validated.

3.3 Program Analysis

Program analysis [10] is the usage of automated techniques to explore program properties. Statically by analyzing a program, its behavior can be predicted without having to execute it. It allows the optimization of compilation, security analysis, and automated verification. There are different techniques for program analysis such as control flow analysis and data flow analysis. Control flow analysis aims at tracking the flow of execution within a given program. Control flow analysis is done by identifying "basic blocks" in the program execution. With these blocks, we can compute the execution flow and analyze it. Control flow analysis could be applied for verification and validation of behavioral design models to find portions of the system that are never used or that are repetitive. Data-flow analysis is a technique of automatically gathering information about the values calculated in a procedural computer program by applying rules to its control flow graph. The information gathered is often useful in optimizing the program. A simple way to perform data flow analysis of programs is to set up data flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes.

Data flow analysis techniques can be applied to behavioral diagrams to examine the use of resources. If some resource is declared in a structural diagram but never used in a behavioral diagram, this can denote as a problem.

3.4 Software Engineering Techniques

When designing a software system, different software engineering techniques play a big role in the process. A good software system offers components that are more robust, more maintainable, and more reusable. These components should be highly cohesive and loosely coupled. Different software metrics were developed to measure object oriented concepts [2] such as inheritance and encapsulation. In the following, we will cover coupling and cohesion. After that, we will talk about a metrics suite for object oriented systems.

- *Coupling* [8] is a measure of the interdependence among modules. Highly coupled modules interact in ways that make their separate modification difficult. Module coupling decreases the reusability of the coupled objects. It increases the chances of system corruption when changes are made to one or more of the coupled objects. Undesirable high levels of coupling occur when abstractions are poor and encapsulation inadequate. Good systems tend to have a low degree of coupling among objects. The minimum degree of coupling is obtained by making modules as independent as possible.

  Six levels of coupling were defined to measure the interdependence among the system modules. These levels were ordered according to their effects on system understandability, maintainability and reusability of the coupled modules. High level of coupling is considered when two modules are coupled in more than one way. No coupling occurs when modules are not dependent on others; this level of coupling is highly preferable. In the following we define the different levels of coupling from best to worst as follows:

  - *Independent Coupling*: No coupling exists between modules.

– *Data Coupling*: Two modules are data coupled if they pass data through scalar or array parameters. This type of coupling is depicted in Figure 3.3.

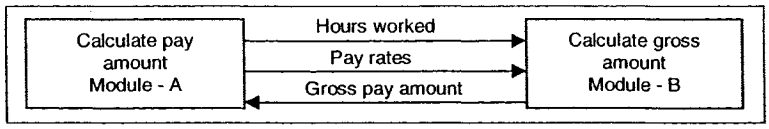

Figure 3.3: Data Coupling

– *Stamp Coupling*: Two modules are stamp coupled if they pass data through a parameter that is a record. Stamp coupling is perceived as worse than data coupling because any change to the record will affect all of the modules that refer to that record, even those modules that do not refer to the fields that are changed. Two modules are stamp coupled if they communicate via a passed data structure, which contains more information than necessary for the modules to perform their functions. Stamp coupling is illustrated in Figure 3.4.

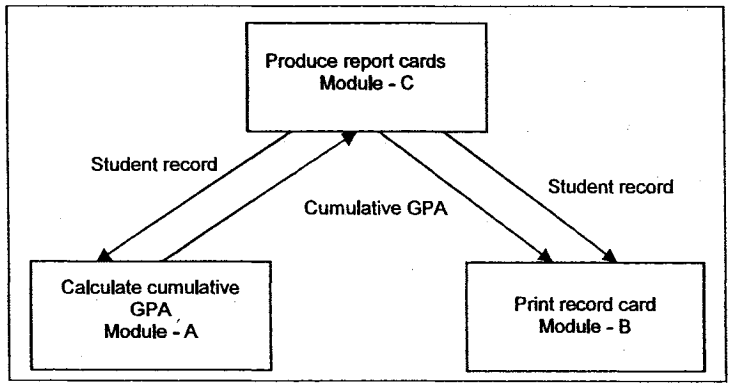

Figure 3.4: Stamp Coupling

Here we assume the "student record" contains a name, address, SIN, and information in addition to academic performance information.

– *Control Coupling*: Two modules are control coupled if one passes a flag value that is used to control the internal logic of the other. Control coupling is shown in Figure 3.5.

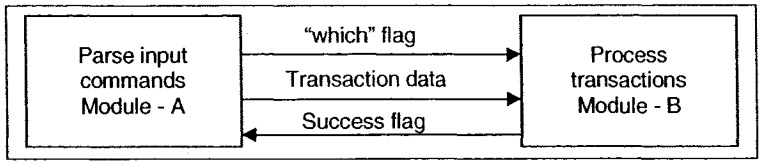

Figure 3.5: Control Coupling

– *External Coupling*: Two modules are externally coupled if they communicate through an external medium such as a file. Figure 3.6 shows external coupling.

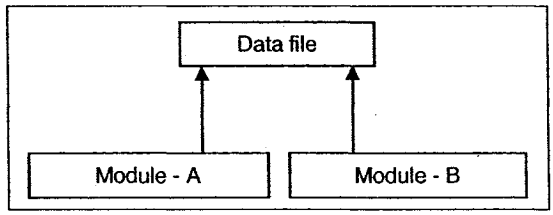

Figure 3.6: External Coupling

– *Common Coupling*: Two modules are common coupled if they refer to the same global data. Figure 3.7 shows common coupling.

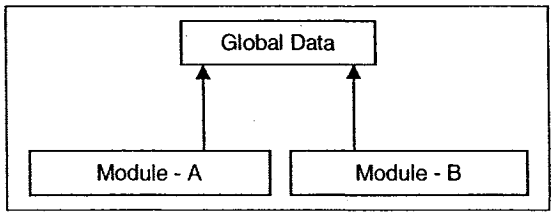

Figure 3.7: Common Coupling

– *Content Coupling*: Two modules are content coupled if they access and change each other's internal data state or procedural state. This type of coupling is considered the worst and is avoided in design.

- *Cohesion* [8] is a measure of the degree of how well the elements of an object work to achieve the same purpose. It measures how well-defined the purpose of an object is, and whether each part of the object contributes directly to achieve that purpose. High

cohesion means that an object has one well-defined purpose and every element in the object contributes directly to that purpose. On the contrary, low cohesion means that either the purpose is obscure or some of the elements do not contribute to the purpose. Low cohesion can occur due to the two mentioned reasons.

A module may exhibit any of seven levels of cohesion depending on how the activities within the module are related. Functional cohesion is the most desirable type of cohesion. Coincidental cohesion is the worst type and is always avoided. In the following, the seven types of cohesion are ordered from best to worst.

- *Functional Cohesion*: A functionally cohesive module contains elements that all contribute to the execution of one and only one problem-related task. A module is functionally cohesive if it can be described as a single coherent function. The goal of design is to achieve this type of cohesion. Modules of this type will be reusable. Figure 3.8 shows that each module has only one method.

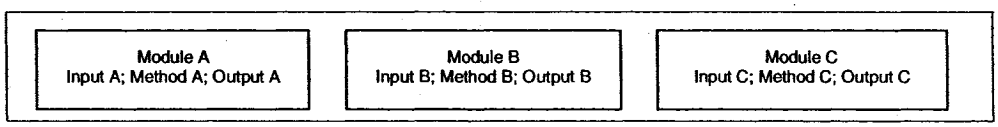

Figure 3.8: Functional Cohesion

- *Sequential Cohesion*: A sequentially cohesive module is one whose elements are involved in activities such that output data from one activity serves as input data to the next activity. A sequentially cohesive module is easily maintained. Indeed, it is almost as maintainable as a functionally cohesive module. The only real disadvantage of a sequentially cohesive module is that it is not so readily reusable in the same system, or in other systems, as is a functionally cohesive module. The reason is that it contains activities that will not in general be useful together. Sequential cohesion is shown in Figure 3.9.

- *Communicational Cohesion*: A communicational cohesive module is one whose

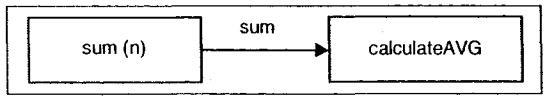

Figure 3.9: Sequential Cohesion

elements contribute to activities that use the same input or output data. Communicational cohesion occurs when a module performs operations related to a sequence of steps performed in the program. All the actions performed by the module are performed on the same data. Communicational cohesion is an improvement on procedural cohesion because all the operations are performed on the same data. Figure 3.10 shows communicational cohesion.

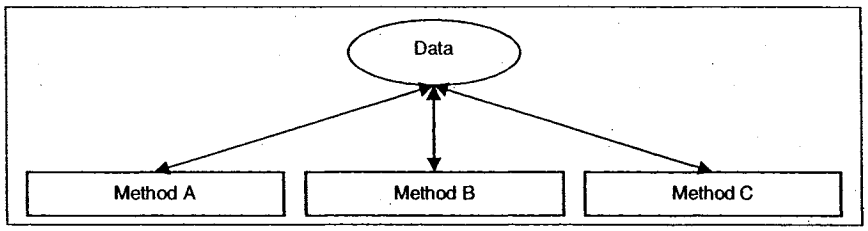

Figure 3.10: Communicational Cohesion

- *Procedural Cohesion*: As shown in Figure 3.11, a procedurally cohesive module is one whose elements are involved in different and possibly unrelated activities in which control flows from each activity to the next without sharing any data. As in sequential cohesion, tasks occur in sequence but they do not share data. Procedurally cohesive modules seem quite natural in their context. However, taken in isolation, they are extremely difficult to understand and seem incoherent. In order to understand such a module, the software engineer must know something about the program from which the module was taken. Such modules are rarely reusable.
- *Temporal Cohesion*: A temporally cohesive module is one whose elements are involved in activities that are related in time. A temporally cohesive module is

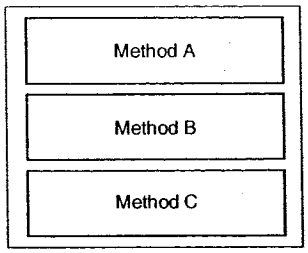

Figure 3.11: Procedural Cohesion

one which performs several sequential functions, which have a weak relationship (or no relationship) to one another but all of which must be performed in a short period of time.

An example of a temporally cohesive object is an "Initialization" object. Temporal cohesion can be removed by insuring that each function within the temporally cohesive object is reassigned to the proper object.

- *Logical Cohesion*: A logically cohesive module is one whose elements contribute to activities of the same general category in which the activity or activities to be executed are selected from outside the module by a flag for example. Figure 3.12 shows an example of logical cohesion.

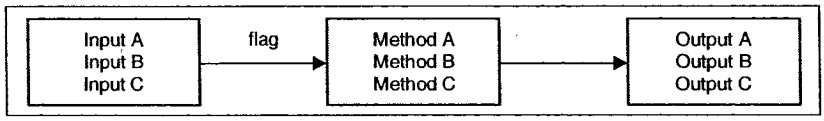

Figure 3.12: Logical Cohesion

- *Coincidental Cohesion*: A module is said to exhibit coincidental cohesion if there is little or no constructive relationship among the elements of the module. This could also be called a random module. It occurs when modules are not partitioned to represent a single abstraction, but are partitioned for some reason such as the number of lines of code. Coincidental cohesion can also show up in modules that do not represent a single, object oriented concept such as a math object. In

a multiple inheritance scheme, a "class" has been defined that only represents a collection of commonly occurring code rather than a single object oriented concept. There is no simple way to remove a coincidently cohesive module from a system. Some redesign has to be performed.

- *Object oriented metrics* can be used to measure the quality of the product before it is produced. This is important since the cost of reducing the risk of a flaw within a program before it is developed is much lower than the cost of fixing a returned product.

  Different metrics [36] suites were developed in the software engineering discipline. Metrics measure software qualities such as reusability and maintainability. Many of these metrics are proposed in 1994 by Chidamber and Kemerer [3]. This metric suite consists of six metrics that are crucial for object oriented development. The six metrics of this suite are:

  - Weighted Methods per Class (WMC)
  - Response For a Class (RFC)
  - Lack of Cohesion of Methods (LCOM)
  - Coupling Between Object Classes (CBO)
  - Depth of Inheritance Tree (DIT)
  - Number of Children (NOC)

3.5 Projects on Verification and Validation

In this section, we give the state of art on the work being done on design verification and validation of systems engineering models. For this purpose, we go through some examples of research efforts being conducted on design verification and validation. For each work, we will give its main objective, the used methods and the targeted diagrams of the modeling language.

The increasing demand on having well designed systems emphasizes the interest on design verification and validation methods and tools. Applying verification and validation from the design phase of a system could help detecting and resolving problems earlier. This significantly reduces the subsequent cost of design, development, and testing. These systems could be designed using different modeling languages such as UML and SysML. Although verification and validation is an essential task, we still observe a lack of application of verification and validation on systems engineering.

It is worthy to say that, as much as we are aware, most of the projects on design verification and validation target only software engineering models and do not address systems engineering ones. On the other hand, the majority of these efforts addresses only UML [28] designed software systems and they do not cover all of the diagrams of this modeling language. Moreover, the projects that we reviewed are using formal verification and validation techniques such as model checking [31] and theorem proving [40]. None of these efforts tried to apply program analysis [20] (e.g. inter-procedural analysis) and software engineering (e.g. the use of cohesion and coupling algorithms) techniques. Finally, previous work does not provide precise feedback after verification and analysis; design assessments are generally ad hoc and do not present any analytical results to detail the problems of the design.

Several research institutions and industries are working on methods and tools for verification and validation of design models. Herein, we will briefly describe some of these projects.

3.5.1 Validation and Verification of Object Technology

This project, Validation and Verification of Object Technology [44], is being developed by the Information Security and Object Technology (ISOT) Research Group at the University of Victoria. It started in November 2001 and aims to use UML as a graphical Object-Oriented notation and the Prototype Verification System-Specification Language (PVS-SL) [25], a specification language integrated with support tools and theorem prover, as a formal

notation.

The main objective of the VVOT project is to define a framework and associated tools for the development of object-oriented software systems using formal methods. The VVOT project is divided into a mainstream project and a number of derived sub projects designated for specific aspects of object technology. The main project formalizes a subset of UML models such as state-chart diagram.

The resulting formal model works as an input for the sub projects for specific activities, applications, or architectures including security, testing, verification, and validation.

Precise UML Development Environment (PrUDE) is a subproject in VVOT that consists of developing a software verification and validation environment for UML models, providing support for model execution, model-checking, and proof-checking. PrUDE will use XMI [19] as an interface to the UML model.

Specification Based testing of OO Software Systems (SoftTest) is a subproject that targets the testing of OO techniques such as polymorphism, encapsulation, and inheritance. The objective of this subproject is to target the development of a framework for automatic test cases generation and test result evaluation for OO software systems, based on the project's formalized version of UML.

Formal Methods and Security Testing (SecTest) is a subproject that consists of extending the tools developed in the PrUDE project with a security properties editor and analyzer. The analyzer and properties will be combined with the tools developed in the SecTest project for security test cases generation and execution based on a formal secure model of the software.

The last subproject in VVOT is Formal Methods and Corba Security Models Design (Corba-S). The aim of this subproject is to diagnose how the implementation of Corba security models can be improved by involving formal methods in their design.

3.5.2 Model-Based Verification and Validation of Properties

This work [9] is done at the Faculty of Computer Science, Electrical Engineering, and Mathematics at the University of Paderborn in Germany. This work shows how verification and validation can be achieved for UML models. These UML models contain use-case diagrams and sequence diagrams to express a high level of abstraction. It also contains class diagrams, statecharts, and activity diagrams for a low level of abstraction. This project uses graph transformation techniques for automated translation of UML models into a language understood by a verification tool. The work chooses CSP [21] as a formal language and FDR [30] as the model checker for the verification and validation of UML models.

This work shows how model-based analysis of properties can be made applicable within an UML-based development process. Examples for such properties include deadlock freedom, timing consistency, and limited memory resources. The approach is to design a partial formalization of UML models such that existing verification techniques can be reused.

3.5.3 vUML

This project, Verify UML [11], which is developed by the Model Driven Engineering Project at the Center for Reliable Software Technology in Finland. It verifies UML models automatically where the behavior of the objects is described using UML State-charts diagrams. The project uses the PROMELA [32] language as a formal specification language for expressing UML models and the SPIN model checker [22] to perform the verification.

SPIN is used to detect the following error types in the UML model: deadlocks, livelocks, reaching a state marked as invalid, violating a constraint of an object, sending an event to a terminated object, and overrunning a message queue. If an error is found during the verification, the tool creates a UML sequence diagram that shows how to reproduce the error in the model. vUML has been specifically designed to verify concurrent and distributed models containing active objects.

3.5.4 OMEGA

The OMEGA project [24] is being developed by a group of European institutions such as VERIMAG, the Wiezmann Institute, and the University of Nijmegen as well as other companies, such as Telelogic, I-Logix, Rational, and Israeli Aircraft Industries. This project started in January 2002 and will end in December 2004.

The objective of this project is to define a development methodology in UML for embedded and real-time systems based on formal techniques. OMEGA selects a suitable subset of UML such as class and state diagrams, the Object Constraint Language (OCL) [16], and component descriptions that include provided and required interfaces. The project also selects use-case diagrams and live sequence charts (an extension of UML sequence diagram).

OMEGA presents a technique to perform model-checking on UML models based on a mapping of object oriented UML models into a framework of communicating extended timed automata.

The project previews an adaptation of existing validation tools for the validation of UML models by mappings from UML (XMI) into input formats of the existing tools by respecting the defined reference semantics by extensions of internal formalisms to cope with the expressive power of UML, improvement of existing validation methods, and development of compositional verification methods based on the components concept.

3.5.5 Socle

Socle project is sponsored by Defence R&D Canada - Valcartier. It started in January 2000 at Ecole Polytechnique de Montreal. The objective of the project is to express security constraints of a UML model directly in OCL and to verify its correctness. This project adopts an approach that includes security specification and correction at every step of design. It expresses UML and Java formal semantics in abstract state machines and expresses OCL formal semantics and temporal extensions in μ-calculus. It adapts model-checking techniques

to secure UML/μ-OCL models.

3.5.6 Verification and Validation of UML Dynamic Specifications

This project [43] is sponsored by the NASA Software IV&V Facility. It is being developed by the Lane Department of Computer Science and Electrical Engineering at West Virginia University. The problem addressed in the project is on the measurement and analysis of the real-time dynamic behavior of software specification and design artifacts for applications modeled in UML. This includes the verification of performance and timing behavior of real-time activities, complexity, and risk assessment. This project will develop an environment for verification of automated architectural-level risk assessment and for verification of performance modeling, and fault injection analysis.

3.5.7 Hugo/RT

Hugo/RT [33] is a project from the Computer Science Department at Ludwig-Maximilians University in Munchen, Germany with cooperation of LORIA/INRIA Nancy, France. It is a UML model translator for model checking and theorem proving. The main focus of the model-checking component of Hugo is to verify the consistency of UML state machines against specifications expressed as collaboration or sequence diagrams [26].

Hugo/RT implements a back end that translates a few UML models, namely state machines, collaborations, interactions, and OCL constraints to timed automata for the real-time model checker UPPAAL [7] and a back end to on-the-fly model checker SPIN [22]. It also translates this UML models in the system language used by the KIV theorem prover [26].

Some UML features are currently not handled correctly by Hugo/RT:

- Orthogonal regions must not show incoming or outgoing transitions.
- Internal transitions are not supported.

- Signal hierarchies are not supported.
- Synch states are not supported.
- Deep history states are not supported.
- Change events are not supported.

Chapter 4

A New Paradigm for Verification and Validation of Systems Engineering

4.1 Approach

The main trait of our proposed Verification and Validation paradigm lies in the harmonious synergy between three predefined well-established techniques that are: Formal-automatic verification (model-checking), software engineering techniques (metrics), and program analysis (static analysis). The foundation of our paradigm is sustained by three distinctive layers. First, model checking, as formal verification technique, can be fully automated and has been successfully used for the verification of real applications including digital circuits and controllers, communication protocols, etc. Second, we derive for each diagram a formal semantic model reflecting its characteristics and express the properties that the design must satisfy as temporal logic formulas. Then in the last layer, We apply model checking on the semantic model. A widely used model checker within the scientific community is SMV and SPIN. It has interesting features such as branching time logic for expressing properties (CTL). The model checker that we are currently using is SPIN and NuSMV [4] (a modified version of the original SMV). The NuSMV architecture can be used for verifying hardware and also

for verification in other engineering fields (e.g. software) in which systems can be modeled as an finite state machine (FSM).

4.2 System Aspects and System Properties

There are many systems engineering aspects including requirements, structure, concurrency, and performance. In the sequel, we briefly present those that we target in this work: (In the thesis we only address a few of them)

- *Requirements*: They are a description of what a system should do and are captured by requirement diagrams in SysML or using sequence and use case diagrams in UML 2.0.
- *Time*: It is captured by timing diagrams, which provide a visual representation of objects changing state and interacting over time.
- *Concurrency*: It identifies how activities, events, and processes are composed (sequence, branching, alternative, parallel, etc.). It could be specified using sequence, activity, state-chart and timing diagrams.
- *Performance*: It is the total effectiveness of the system. It makes reference to the timeliness aspects of how systems behave. This includes different types of quality of service characteristics such as latency and throughput. Timing and sequence diagrams depict performance aspects.
- *Structure*: It is shown in class and composite structure diagrams. The class diagram shows the relationships between different classes of the system. The composite structure diagram shows the internal structure of the building blocks of the system and how these blocks are interfacing with other components of the system.
- *Interface*: It identifies the shared boundaries of the different components of the system whereby the information is passed. This aspect is shown using class diagrams in UML

2.0 and SysML, composite structure diagrams in UML 2.0, and assembly diagrams in SysML.

- *Control*: It identifies the order in which actions, states, events, and/or processes are arranged. It is captured using state machine, activity, and sequence diagrams in UML 2.0 and SysML.

Verification and validation contribute to the design assessment by detecting the unsatisfied properties. Hence, system developers will know if the design is flawed and apply corrective measures. The following properties fall in the scope of our Verification and Validation approach:

- *Latency*: It is the measure of the temporal delay between the request for the execution of an operation and the reply to this request. Detecting latency contributes to verification and validation by analyzing the efficiency of the system.

- *Liveness*: It asserts that under certain conditions, a given event will occur. It is known as "something good will always happen". Liveness analysis consists of checking whether some important or crucial events may or may not eventually happen in the system.

- *Safety*: It means that nothing bad can occur with respect to the design of the system. In other words, it is a judgment of the acceptability of risk, which implies that no harm will occur under the specified conditions.

- *Deadlock*: It describes a state wherein a process is waiting for some event that will never happen. It could be waiting for a resource to be available before continuing its execution while another process is holding indefinitely this resource. In this situation, the system would not progress.

- *Livelock*: It is a situation where two or more processes continually change their states

in response to changes in the other process (or processes) without performing useful services. It is different from deadlock since the processes are progressing.

- *Precedence*: It specifies the order between events in the system with respect to time. Namely, events must not occur unless a specific event or a sequence of events were finished. If the ordering of events is not respected then verification and validation will help the developers review the ordering between events in their design.

- *Reachability*: It consists of checking whether a particular state is reachable in a design, starting from an entry point of the system. Unreachable states negatively impact the quality design since they denote dead entities in the system.

- *Complexity*: It designates the quality of being intricate and compounded. It measures the degree to which a system design is difficult to be understood and/or to be implemented.

- *Maintainability*: It measures the easiness and rapidity with which a system design and/or implementation can be changed for perfective, adaptive, corrective, and/or preventive reasons.

- *Reusability*: It measures the easiness and rapidity with which a part (or more) of a system design and/or implementation can be reused.

- *Coupling*: It measures how strongly system parts depend on each other. Generally, a loose coupling is sought in a high-quality design. Moreover, there is a strong correlation between coupling and other system quality attributes such as complexity, maintainability, reusability, etc.

- *Cohesion*: It refers to the degree to which system components are functionally related (internal "glue"). Generally, a strong cohesion is sought in a high-quality system design.

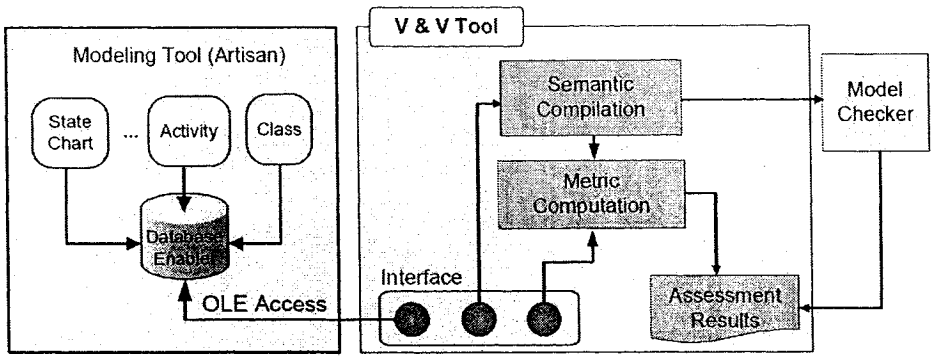

Figure 4.1: Architecture of the Framework

4.3 Verification and Validation Framework

Our Verification and Validation framework requires an underlying modeling tool wherefrom various models can be fetched and assessed. Our current choice is ARTiSAN Real-time Studio which is a modeling tool that supports UML and SysML designs. Additionally, it provides component-based development specifically for real-time systems [41, 42]. The current version of our framework is composed of three core components, as shown in Figure 4.1. First, we have the semantic compilation component responsible for deriving the semantic model of a specific diagram. It communicates with the model checker by providing the semantic model along with the properties to be verified. Second, we have the metric computation component that is used for applying metric algorithms. We have provided an interface that accesses the object repository of the modeling tool and retrieves the needed information about the diagrams. Finally, the assessment results component is devoted to the presentation of interpreted results. Should a specified property fail, the trace provided by the model-checker is analyzed and the relevant information is provided as feed-back. In the thesis we will consider verifying state-chart diagram.

The quality of an object oriented system depends on different attributes such as complexity, understandability, maintainability, stability, and others. According to the type of diagram, we have a class of metrics for structural diagrams and another for behavioral ones.

In the literature, many metrics were developed to measure the quality of software systems, especially for structural diagrams namely class and package. However, until now, they were not considered in the verification and validation of systems engineering designs. We have adopted a set of fifteen metrics including *Coupling Between Object classes* (CBO), *Depth of Inheritance Tree* (DIT), and *Instability*. CBO measures the interrelationship between objects. DIT measures the level of a class in the class inheritance hierarchy. The instability metric measures the rate of instability of a package. A package is unstable if it depends more on other packages than they depend on it.

Chapter 5

Verification and Validation of State-Chart Diagram

5.1 State-Chart Description

The State Machine package is a sub-package of the Behavioral Elements package [17]. It specifies a set of concepts that can be used for modeling discrete behaviors through finite state-transition systems representing a visual formalism for complex *reactive systems* (event driven systems which continuously react to external stimuli). Examples of such systems include communication networks, operating systems, embedded controllers for telephony, automobiles, trains and avionics systems, etc.

UML state-charts are used to model the behavior of event-driven active objects. An active object is a state machine object endowed with its own thread of execution and communicating with other active objects by exchanging event instances.

The most important characteristic of an active object is its opaque encapsulation shell, which strictly separates the internal structure of an active object from the external environment. The only objects capable of penetrating this shell, both from the outside to inside, are event instances.

The UML state machines (state-charts) model extends conventional state machines along three orthogonal dimensions *hierarchy*, *concurrency* and *communication* resulting in a compact visual notation. Hierarchy is the ability to cluster states into a super-state (an OR state), or refine an abstract state into more detailed states; concurrency denotes orthogonal subsystems that proceed (more or less) independently and is described by an AND composition of states; communication between concurrent components is via a broadcast mechanism.

In UML, the definition of the state-charts is similar to that of hierarchical state machines and as such, they support state nesting and behavioral inheritance.

State machines can be used to specify the behavior of various elements that are going to be modeled. For example, they can be used to model the behavior of individual entities like class instances.

The state machine is a specification that thoroughly describes all possible behaviors of some dynamic model. Behavior is modeled as a traversal of a graph of state nodes connected by one or more transition arcs. Transitions are triggered by the dispatching of a series of events. During the traversal, the state machine executes a series of actions [17].

The dynamics of the model requires that objects change values of their attributes but not all such changes are required to be considered in the state transitions. Only the attributes that are involved in the state transitions need to be considered. More specifically, if a particular attribute value is never considered in the decisions related to the state transition then that attribute can be safely filtered out from the state-chart of the object.

Furthermore, the diagram determines how objects of that class react to events and for each object state it specifies what action the object will perform when it receives an event. The same object may perform a different action for the same event depending on the object's state and the action's execution will typically cause a state change.

A state machine has only one top-level state, and a set of transitions. Moreover, each state machine owns its transitions and its top state. All remaining states are transitively

owned through the state hierarchy rooted in the top state. The internal transitions are owned by their containing states.

5.1.1 State

From the UML definition we know that a state models a certain situation where some (usually implicit) invariant condition holds. The invariant may represent a static situation such as in the case where an object is waiting for some external event to occur. However, it can also model dynamic conditions such as performing some activity (i.e., the model under consideration enters the state when the activity commences and leaves it as soon as the activity is completed).

- **Entry**

  An optional procedure that is executed whenever the state is entered regardless of the transition taken to reach the state. If defined, entry actions are always executed to completion prior to any internal activity or transitions performed within the state.

- **Exit**

  An optional procedure that is executed whenever the state is exited regardless of the transition taken out of the state. If defined, exit actions are always executed as the final step prior to leaving the state.

- **Activity**

  An optional activity that is executed while being in the state. The execution starts when the state is entered and it stops either by itself, or when the state is exited.

- **Internal Transition**

  A set of transitions that, if triggered, occur without exiting or entering this state. Thus, they do not cause a state change. This means that the entry or exit condition of

the State will not be invoked. These transitions can be taken even if the state machine is in one or more regions nested within this state.

- **State-Vertex**

  A state vertex is an abstraction of a node in a state-chart graph. In general, it can be the source or destination of any number of transitions with the following attributes:

  1. There are transitions associated with a state vertex:
     - outgoing: specify the transitions departing from the vertex
     - incoming: specify the transitions entering the vertex
  2. Container: The composite state that contains this state vertex

- **Deferrable Event List**

  Lists the events that are deferrable for this state.

5.1.2 Top State

Designates the top-level state that is the root of the state containment hierarchy. There is exactly one state in every state machine that is the top state.

5.1.3 Composite State

A composite state is a state that contains other state vertices (states, pseudo-states, etc.). There is an association between the composite and the contained vertices so that a state vertex can be a part of at most one composite state.

Any state enclosed within a composite state is called a sub-state of that composite state. If it is not contained by any other state then it is called a direct sub-state otherwise it is referred to as a transitively nested sub-state.

5.1.4 Composite Concurrent State

When the composite state is decomposed directly into two or more orthogonal conjunctive components called regions (usually associated with concurrent execution) then the composite state is concurrent. A region is a direct sub-state of a concurrent state.

5.1.5 Simple/Basic State

A simple or basic state is a state that does not have any other sub-states.

5.1.6 Pseudo-state

A pseudo-state is an abstraction for different types of transient vertices in the state machine graph. These are in turn typically used to connect multiple transitions into more complex state transitions paths.

- **Initial**

  Represents a default vertex that is the source for a single transition to the default state of a composite state. There can be at most one initial vertex in a composite state.

- **Final State**

  A special kind of state signifying that the enclosing composite state is completed. If the enclosing state is the top state, then it means that the entire state machine is completed. A final state cannot have any outgoing transitions.

- **Fork**

  Serves to split an incoming transition into two or more transitions that are terminating at orthogonal target vertices. The segments outgoing from a fork vertex should have no guards. By combining a transition entering a fork pseudo-state with a set of transitions exiting the fork pseudo-state, we get a compound transition that leads to a set of concurrent target states.

- **Join**

  Serves to merge several transitions emanating from source vertices in different orthogonal regions. Similarly, the transitions entering a join vertex cannot have guards.

- **Junction**

  Vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path (known as a merge).

  Conversely, they can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions. This realizes a static conditional branch.

- **Choice**

  When it is reached, the result is the dynamic evaluation of the guards of its outgoing transitions. This accomplishes a dynamic conditional branch. It allows splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run-to-completion step. If more than one of the guards evaluate to true, an arbitrary one is selected – this situation is not recommend as it would be nondeterministic. If none of the guards evaluates to true, then the model is considered malformed.

  This situation can be avoided by using one outgoing transition with the predefined *else* guard. Choice vertices should be distinguished from static branch points that are based on junction points.

- **Shallow history**

  Notation that represents the most recent active sub-state of its containing state (but

not the sub-states of that sub-state). A composite state can have at most one shallow history vertex. A transition coming into the shallow history vertex of a state is equivalent to a transition coming into the most recent active sub-state of that state. Moreover, a transition may originate from the history connector to the initial shallow history state. This transition is taken in case the composite state had never been active before.

- **Deep History**

  Notation that represents the most recent active configuration of the composite state that directly contains this pseudo-state; that is, the state configuration that was active when the composite state was last exited. A composite state can have at most one deep history vertex. A transition may originate from the history connector to the default deep history state. This transition is taken in case the composite state had never been active before.

5.1.7 Submachine-state

A submachine state is a syntactical convenience that provides reuse and modularity features. It implies a macro-like expansion by another state machine and is semantically equivalent to a composite state. The state machine that is inserted is called the *referenced state machine* while the state machine that contains the submachine state is called the *containing state machine*.

The same state machine may be referenced more than once in the context of a single containing state machine. Thus, a submachine state can be considered a call to a state machine subroutine with one or more entry and exit points specified by some stub states.

5.1.8 Stub-state

Stub-states can appear within submachine states and represent an actual sub-vertex contained within the referenced state machine. It can serve as a source or destination for transitions that connect a state vertex in the containing state machine with a sub-vertex in the referenced state machine.

5.1.9 Synch-state

A synch-state is a vertex used for synchronizing the concurrent regions of a state machine. It is different from a state in the sense that it is not mapped to a boolean value (active or not active), but an integer specifying the maximal count of the synch-state. The count is the difference between the number of times the incoming and outgoing transitions of the synch-state are fired. A synch-state is used in conjunction with forks and joins to ensure that one region leaves a particular state or states before another region can enter a particular state or states.

5.1.10 Hierarchical State Decomposition

The hierarchical state decomposition allows for sharing or reuse of the behavior related to a state by its sub-states.

5.1.11 Events and Guards

- **Event**

  An event is a specification of a type of an observable occurrence. For simplified modeling purposes, the occurrence that generates an event is assumed to take place at an instant in time with no duration.

- **Deferrable Event**

An event that can be deferred if present in the corresponding list of deferrable events of a state.

- **Guard**

  A guard is a boolean expression that is attached to a transition as a fine-grained control over its firing. The guard is evaluated when an event is dispatched by the state machine. If the guard is satisfied (true) then the transition is enabled, otherwise, it is disabled.

- **Guard Evaluation**

  Guards are required to be pure expressions without side effects (Guard expressions with side effects are considered malformed).

5.1.12 Transitions

A transition is a directed relationship between a source state vertex and a target state vertex. It takes the state machine from one state configuration to another, representing the complete response of the state machine to a particular event. A transition has associated trigger and guard. Trigger specifies the event that fires the transition. There can be at most one trigger per transition. Guard is a boolean predicate that provides a fine-grained control over the firing of the transition. It must be true for the transition to be fired and evaluated at the time the event is dispatched. There can be at most one guard per transition.

- **Compound Transition**

  A special type of transition that represents a path made of one or more transitions, originating from a set of states and targeting a set of states.

- **High-level (Group) Transitions**

Transitions originating from the boundary of composite states (high-level or group transitions). The result consists in the exiting of all the sub-states of the composite state.

- **Internal Transitions**

  Executes without exiting or re-entering its corresponding state.

5.2 State-Chart Semantics

Generally, state machines have three key required components. First, an event queue which holds incoming event instances until they are dispatched. second, an event dispatcher mechanism that selects and de-queues event instances from the event queue for processing and the last one is an event processor which processes dispatched events.

5.2.1 States

- **Active States**

  A state can be either active or inactive during execution. A state becomes active when it is entered as a result of some transition, and becomes inactive when it is exited as a result of a transition. A state can be exited and entered as a result of the same transition (self transition).

- **State Entry and Exit**

  Whenever a state is entered, its entry action is executed before any other action. Conversely, whenever a state is exited, it executes its exit action as the final step prior to leaving the state. If defined, the activity associated with a state is forked concurrently at the instant when the entry action of the state is completed. Upon exit, the activity is terminated before the exit action is executed.

- **Activity in State**

  Represented by the execution of a sequence of actions, that occurs while the state machine is in the state. The activity starts executing upon entering the state, following the entry action. If the activity completes while the state is still active, it raises a completion event. In case where there is an outgoing completion transition, the state will be exited. If the state is exited as a result of the firing of an outgoing transition before the completion of the activity, the activity is aborted prior to its completion.

- **Deferred Events**

  A state may specify a set of event types that may be deferred in that state. An event that does not trigger any transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state. Instead, it remains in the event queue while another non-deferred message is dispatched instead. This situation persists until a state is reached where either the event is no longer deferred or where the event triggers a transition.

- **Composite State**

  1. Active state configurations

     For composite states, the term current state may designate more than one active state. In a hierarchical state machine more than one state can be active at one moment. If the state machine is in a simple state that is contained in a composite state, then all the composite states that either directly or transitively contain the simple state are also active. Furthermore, since some of the composite states in this hierarchy may be concurrent, the current active state is actually represented by a tree of states starting with the single top state at the root down to individual simple states at the leaves. Such a state tree represents a state configuration.

The following invariants always apply to state configurations (except during transition execution):

- If a composite state is active and not concurrent, exactly one of its sub-states is active.
- If the composite state is active and concurrent, all of its regions (sub-states) are active.

2. Entering a non-concurrent composite state

   Upon entering a composite state, there are several cases:

   - Default entry: An incoming transition that terminates on the outside edge of the composite state. In this case, the default initial transition is taken. The presence of a guard on the transition should be always true (may be needed for informal reasons). The entry action of the state is executed before the action associated with the initial transition.
   - Explicit entry: If the transition goes to a sub-state of the composite state, then that sub-state becomes active and its entry code is executed after the execution of the entry code of the composite state. This rule applies recursively if the transition terminates on a transitively nested sub-state.
   - Shallow history entry: If the transition terminates on a shallow history pseudo-state, the active sub-state becomes the most recently active sub-state prior to this entry, unless the most recently active sub-state is the final state or if this is the first entry into this state. In the latter two cases, the default history state is entered. This is the sub-state that is target of the transition originating from the history pseudo-state (if no such transition is specified, the situation is illegal and its handling is not defined). If the active sub-state determined by history is a composite state, then it proceeds with its default entry.

- Deep history entry: Same as for shallow history except that the rule is applied recursively to all levels in the active state configuration.
- Entering a concurrent composite state: Whenever a concurrent composite state is entered, each one of its concurrent sub-states (regions) is also entered, either by default or explicitly. If the transition terminates on the edge of the composite state, then all the regions are entered using default entry. If the transition explicitly enters one or more regions (in case of a fork), these regions are entered explicitly and the others by default.
- Exiting a non-concurrent state: When exiting from a composite state, the active sub-state is exited recursively. This means that the exit actions are executed in sequence starting with the innermost active state in the current state configuration.
- Exiting a concurrent state: When exiting from a concurrent state, each of its regions is exited. After that, the exit actions of the regions are executed.
- Deferred events in composite states: An event that is deferred in a composite state is automatically deferred in all directly or transitively nested substates.

- **Final State**

  When the final state is active, its containing composite state is completed, which means that it satisfies the completion condition. If the containing state is the top state, the entire state machine terminates, implying the termination of the entity associated with the state machine.

- **Submachine-state**

  A submachine state is a convenience that does not introduce any additional dynamic semantics. It is semantically equivalent to a composite state and may have entry and exit actions, internal transitions, and activities.

5.2.2 Events

Events are generated as a result of some action either within the system or in the environment surrounding the system. An event is then conveyed to one or more targets. The means by which events are transported to their destination depend on the type of action, the target, the properties of the communication medium, and other factors. For simplified modeling purposes the propagation is considered to be instantaneous. An event is received when it is placed on the event queue of its target. An event is dispatched when it is de-queued from the event queue and delivered to the state machine for processing. At this point, it is referred to as the current event. Finally, it is consumed when the event processing is completed. A consumed event is no longer available for processing. Again, for simplified modeling it is required that the intervals between event reception, event dispatching, and consumption is zero-time. Any parameter values associated with the current event are available to all actions directly caused by that event (transition actions, entry actions, etc.).

5.2.3 Transitions

- **Compound and High Level Transitions**

    1. Compound transitions: A compound transition is a derived semantic concept. It represents a semantically complete path made of one or more transitions, originating from a set of states (as opposed to pseudo-state) and targeting a set of states.

    2. High-level/group transitions: Transitions originating from the boundary of composite states are called high-level or group transitions. If triggered, they result in exiting of all the sub-states of the composite state executing their exit actions starting with the innermost states in the active state configuration. In terms of execution semantics, a high-level transition does not add specialized semantics, but rather reflects the semantics of composite state exit.

3. The transition execution semantics:
   - A compound transition is an acyclically unbroken chain of transitions joined via join, junction, choice, or fork pseudo-states that define a path from a set of source states (possibly a single state) to a set of destination states, (possibly a single state). For self-transitions, the same state acts as both the source and the destination set. A (simple) transition connecting two states is therefore a special common case of a compound transition.
   - The tail of a compound transition may have multiple transitions originating from a set of mutually orthogonal concurrent regions that are joined by a join point.
   - The head of a compound transition may have multiple transitions originating from a fork pseudo-state targeted to a set of mutually orthogonal concurrent regions.
   - In a compound transition multiple outgoing transitions may emanate from a common junction point. In that case, only one of the outgoing transition whose guard is true is taken. If multiple transitions have guards that are true, a transition from this set is chosen. The algorithm for selecting such a transition is not specified. In this case, the guards are evaluated before the compound transition is taken.
   - In a compound transition where multiple outgoing transitions emanate from a common choice point, the outgoing transition whose guard is true at the time the choice point is reached, will be taken. If multiple transitions have guards that are true, one transition from this set is chosen. The algorithm for selecting this transition is not specified. If no guards are true after the choice point has been reached, the model is malformed.
4. Internal transitions: An internal transition executes without exiting or re-entering

the state in which it is defined. This is true even if the state machine is in a nested state within this state.

- **Completion Transitions and Completion Events**

  A completion transition is a transition without an explicit trigger, although it may have a guard defined. When all transition and entry actions and activities in the currently active state are completed, a completion event instance is generated. This event is the implicit trigger for a completion transition.

  The completion event is dispatched before any other queued events and has no associated parameters. For instance, a completion transition emanating from a concurrent composite state will be taken automatically as soon as all the concurrent regions have reached their final state.

  If multiple completion transitions are defined for a state, then they should have mutually exclusive guard conditions.

- **Enabled Transitions**

  A transition is enabled if the following conditions are met:

  - All of its source states are in the active state configuration.
  - The trigger of the transition is satisfied by the current event (an event satisfies a trigger if it matches the event specified by the trigger).
  - If there exists at least one full path from the source state configuration to either the target state configuration or to a dynamic choice point in which all guard conditions are true (transitions with no guards can be treated as if their guards are always true).

  Since more than one transition may be enabled by the same event, being enabled is a necessary but not sufficient condition for the firing of a transition.

5.2.4 Guards

1. In a simple transition with a guard, the guard is evaluated before the transition is triggered.

2. In compound transitions involving multiple guards, all guards are evaluated before a transition is triggered, unless there are choice points along one or more of the paths. The order in which the guards are evaluated is not defined.

3. If there are choice points in a compound transition, only guards that precede the choice point are evaluated according to the above rule. Guards downstream of a choice point are evaluated if and when the choice point is reached (using the same rule as above).

4. Guards should not include expressions causing side effects. Models that violate this are considered malformed.

5.2.5 Transition Execution Sequence

1. Every transition, except for internal transitions, causes exiting of a source state, and entering of the target state. These two states, which may be composite, are designated as the main source and the main target of a transition.

2. The least common ancestor (LCA) state of a transition is the lowest composite state that contains all the explicit source states and explicit target states of the compound transition. In case of junction segments, only the states related to the dynamically selected path are considered explicit targets (bypassed branches are not considered).

3. If the LCA is not a concurrent state, the main source is a direct substate of the least common ancestor that contains the explicit source states, and the main target is a substate of the least common ancestor that contains the explicit target states. In case where the LCA is a concurrent state, the main source and main target are the

concurrent state itself. The reason is that if a concurrent region is exited, it forces the entire concurrent state exit.

Once a transition is enabled and is selected to fire, the following steps are carried out in order:

- The main source state is properly exited.
- Actions are executed in sequence following their linear order along the segments of the transition: The closer the action to the source state is, the earlier it is executed.
- If a choice point is encountered, the guards following that choice point are evaluated dynamically and a path whose guards are true is selected. Entry and exit actions are executed while states are entered or exited.
- The main target state is properly entered

5.2.6 State Machine

- **Event Processing - Run-to-completion Step**

  Events are dispatched and processed by the state machine, one at a time as they are coming out of the queue. The order of de-queuing may employ some priority-based schemes. If such schemes are present then events may be treated in an out of order manner. In order to simplify the model, such schemes will not be considered in the implementation.

  The run-to-completion processing means that an event can only be de-queued and dispatched if the processing of the previous current event is fully completed. Run-to-completion may be implemented in various ways. It may be realized by using a monitor or by an event-loop running in its own concurrent thread, and reading events from the queue.

1. Event precessing:

   - The semantics of event processing is based on the run-to-completion assumption, interpreted as run-to-completion processing. The processing of a single event by a state machine is known as a run-to-completion step. Before starting a run-to-completion step, a state machine is in a stable state configuration with all actions (but not necessarily activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event will never be processed while the state machine is in some intermediate and inconsistent situation. The run-to-completion step is the passage between two state configurations of the state machine.

2. Run-to-completion step

   - The run-to-completion assumption simplifies the transition function of the state machine, since concurrency conflicts are avoided during the processing of event, allowing the state machine to safely complete its run-to-completion step. When an event instance is dispatched, it may result in one or more transitions being enabled for firing. If no transition is enabled and the event is not in the deferred event list of the current state configuration, the event is discarded and the run-to-completion step is being completed.

In the presence of concurrent regions it is possible to fire multiple transitions as a result of the same event - as many as one transition in each concurrent region in the current state configuration. The order in which selected transitions fire is not defined.

In order to determine which of the enabled transitions actually fire (in the case where one or more transitions are enabled), the state machine selects a subset and fires them as follows:

- Each orthogonal region in the active state configuration that is not decomposed into concurrent regions (i.e., "bottom-level" region) can fire at most one transition

as a result of the current event. When all orthogonal regions have finished executing the transition, the current event is fully consumed, and the run-to-completion step is completed.

- During a transition, a number of actions may be executed. If these actions are synchronous, then the transition step is not completed until the invoked objects complete their own run-to-completion steps.
- An event can arrive at a state machine that is blocked in the middle of a run-to-completion step from some other object within the same thread, in a circular fashion. This event can be treated by orthogonal components of the state machine that are not frozen along transitions.

- **Run-to-completion and Concurrency**

  It is possible to define the state machine semantics by allowing the run-to-completion steps to be applied concurrently to the orthogonal regions of a composite state rather than to the whole state machine. Though this approach would allow the event serialization constraint to be relaxed, such semantics may prove to be quite subtle and difficult to implement. Therefore, the dynamic semantics are based on the premise that a single run-to-completion step applies to the entire state machine and includes the concurrent steps taken by concurrent regions in the active state configuration.

- **Conflicting Transitions**

  It was already noted that it is possible for more than one transition to be enabled within a state machine. If that happens, then such transitions may be in conflict with each other. In the case of two transitions originating from the same state, triggered by the same event, but with different guards, if that event occurs and both guard conditions are true, then only one transition will fire. In other words, in case of conflicting transitions, only one of them will fire in a single run-to-completion step.

Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty. Only transitions that occur in mutually orthogonal regions may be fired simultaneously. This constraint guarantees that the new active state configuration resulting from executing the set of transitions is well formed.

An internal transition in a state conflicts only with transitions that cause an exit from that state.

- **Firing Priorities**

  In situations where there are conflicting transitions, the selection of which transitions will fire is based in part on an implicit priority. These priorities resolve some transition conflicts, but not all of them. The priorities of conflicting transitions are based on their relative position in the state hierarchy. By definition, a transition originating from a sub-state has higher priority than a conflicting transition originating from any of its containing states.

  The priority of a transition is defined based on its source state. The priority of joined transitions is based on the priority of the transition with the most transitively nested source state.

- **Transition Selection Algorithm**

  The set of transitions that will fire is a maximal set of transitions that satisfies the following conditions:

  1. All transitions in the set are enabled.
  2. There are no conflicting transitions within the set.
  3. There is no transition outside the set that has higher priority than a transition in the set (enabled transitions with highest priorities are in the set while conflicting transitions with lower priorities are left out).

This can be implemented by a greedy selection algorithm, with a straightforward traversal of the active state configuration. States in the active state configuration are traversed starting with the innermost nested simple states and working outwards toward the top state. For each state at a given level, all originating transitions are evaluated to determine if they are enabled. This traversal guarantees that the priority principle is not violated. The only non-trivial issue is resolving transition conflicts across orthogonal states on all levels. This is resolved by terminating the search in each orthogonal state once a transition inside any one of its components is fired.

- **Synch-states**

  Synch states provide a means of synchronizing the execution of two concurrent regions. Specifically, a synch state has incoming transitions from a fork (or forks) in one region, the source region, and outgoing transitions to a join (or joins) in another, the target region. These forks and joins are called synchronization forks and joins.

  The synch state itself is contained by the least common ancestor of the two regions being synchronized. The synchronized regions do not need to be siblings in state decomposition, but they must have a common ancestor state.

  When the source region reaches a synchronization fork, the target states of that fork become active, including the synch state. Activation of the synch state is an indication that the source region has completed some activity. This region can continue performing its remaining activities in parallel. When the target region reaches the corresponding synchronization join, it is prevented from continuing unless all the states leading into the synchronization join are active, including the synch states.

  A synch state may have multiple incoming and outgoing transitions, used for multiple synchronization points in each region. Alternatively, it may have single incoming and outgoing transitions where the incoming transition is fired multiple times before the outgoing one is fired. To support these applications, synch states keep count of the

difference between the number of times their incoming and outgoing transitions are fired. When an incoming transition is fired, the count is incremented by one, unless its value is equal to the value defined in the bound attribute. In that case, the count is not incremented. When an outgoing transition is fired, the count is decremented by one. An outgoing transition may fire only if the count is greater than zero, which prevents the count from becoming negative. The count is automatically set to zero when its container state is exited.

The bound attribute is for limiting the number of times outgoing transitions fire from a synch state. For a state, to realize the equivalent of a binary semaphore, the bound should be set to one. In this case multiple incoming transitions may fire before the outgoing transition does, whereupon the outgoing transition can only fire once.

- **Stub-states**

  Stub states are pseudo-states signifying either entry points to or exit points from a submachine. Since a submachine is encapsulated and represented as a submachine state, multi-level (*deep*) transitions may logically connect states in separate state machines. This is facilitated by stub-state, representing real states in a referenced machine to/from transitions in the referencing incoming/outgoing machine. Stub-states therefore, can only be defined within a submachine state, and are the only potential sub-vertices of a submachine state.

5.2.7 UML State-Charts and Extended Hierarchical Automata

In this section we show that UML state-charts and Extended Hierarchical Automata (EHA) share enough characteristics so that if we set certain restriction on the state-charts we can construct the equivalent EHA [6]. The restrictions are needed in order to help the formalization that is used in the translation process and are related to "object orientation", like

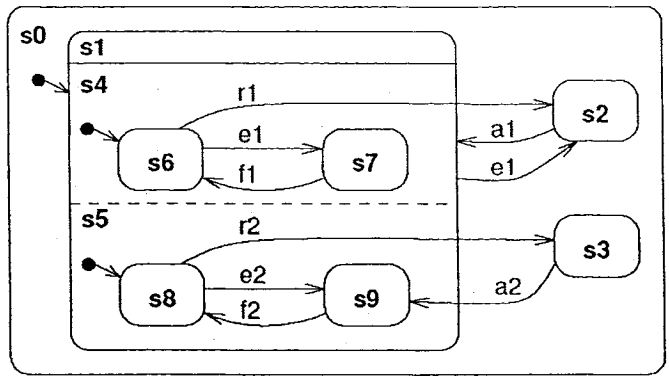

Figure 5.1: Example of State-Chart

inheritance, sub-behavior etc. Moreover, history, action and activity states are not considered as well as the entry and exit actions of states. Apart from the "object orientation" which requires further research, the above simplifications do not have any strong impact on the semantics at a conceptual level.

Both the UML state-charts and the Extended hierarchical automata share a number of key concepts such as:

1. State Refinement function
2. Transition relation:
   - Triggered by a combination of:
     - Event.
     - Boolean guard.
   - Sequence of associated actions.
   - Connect a source and a target state.

This makes it possible to construct an equivalent EHA for a given UML state-chart. In order to show how a state-chart and an EHA are equivalent, we will refer to the example in Figure 5.1 [6]. The state refinement is as follows:

- s_0 is refined into an automaton consisting of three states s_1, s_2 and s_3.
- s_1 is further refined into two states s_4 and s_5.
- s_4 and s_5 are each refined to a single automaton.
- s_0, s_1, s_4 and s_5 are composite states
- s_1 is a concurrent state.

5.2.8 State Transitions

System states are modeled by *configurations* which are sets of states. Below is a number of configurations for our state-chart example:

- $\{s_1, s_6, s_8\}$
- $\{s_1, s_6, s_9\}$
- $\{s_1, s_7, s_8\}$
- $\{s_1, s_6, s_9\}$
- $\{s_1, s_7, s_8\}$
- $\{s_1, s_7, s_9\}$
- $\{s_2\}$
- $\{s_3\}$

The state transitions are labelled by trigger events and guards and they connect source and target states in such a way that if the source state is in the current configuration and a trigger is offered and the guard is satisfied then source state is left, the actions are executed and a new configuration containing the target state is entered.

Consider that the current configuration is $\{s_2\}$ and a_1 is the event offered. In this case, the state s_2 is left and s_1 is entered. Moreover, if a particular sub-state is the target (s_1 being composite), then it should be designated otherwise the default one would be chosen. In the example we have the default (initial) states of s_4 and s_5, namely s_6 and s_8.

- **Inter-level and Compound Transitions**

  Inter-level and compound transitions may have more than one target in order to point to all relevant states. Compound transitions may also have more than one source state meaning that all such states must be in the current configuration for the transition to be enabled.

  For our state-chart, if the configuration is $\{s_3\}$ and $\{a_2\}$ is offered the resulting configuration will be $\{s_1, s_6, s_9\}$ where $\{s_9\}$ is inter-level. Other inter-level transitions are from $\{s_6\}$ to $\{s_2\}$ and $\{s_8\}$ to $\{s_3\}$. Firing from $\{s_6\}$ requires a configuration containing $\{s_6\}$ regardless of the state where $\{s_5\}$ resides bringing the system to a configuration that contains $\{s_2\}$. The same consideration apply when firing from $\{s_8\}$.

- **Event Concurrency**

  If more than one event is presented at a given moment, a dispatcher is assumed to enqueue the events. When an event is processed, if more than one transition is enabled at one point there might be a conflict. This might happen when the intersection of the sets of states left by the transition is not void. This situation can be handled using priorities.

- **Transition Priorities**

  Higher priority is given to the transition for which the source state is a subset of the source of the other transition. More specifically for our state-chart example we have the following:

- Provided that the configuration contains both s_1 and s_6 and the event dispatched is e_1 then the transition from s_6 to s_7 will fire since it has higher priority than the one to s_2.
- If the conflict is not solved by the priorities then the transition is considered non-deterministic and the model is likely not well formed.

5.3 State-Chart Verification and Validation

5.3.1 Objectives

Derive a method to automatically generate a formal model and corresponding properties for a given state-chart in order to check whether it complies to some general behavior constrains. Additionally, in many cases there are other specific (particular) requirements that accompany the state-chart. If that is the case, then the model must also be checked against the derived properties thereof.

5.3.2 Procedure

From the foregoing paragraphs we have seen how a state-chart diagram can be translated to an equivalent Extended Hierarchical Automaton whose operational semantics are defined as a Kripke structure. Thus, a formal model can be constructed accordingly for a given model checker. For the property specification, depending on the model checker, either LTL or CTL like properties can be specified for both the general and the particular constrains of the model. For the SPIN model checker in particular, the LTL subset of CTL can be used.

5.3.3 Translation of UML State-Charts to Extended Hierarchical Automata

The translation maps a UML state-chart to an extended hierarchical automaton $H = (F, E, \rho)$ by defining the set of sequential automata F, the composition function ρ and the set of events E [6].

Set of Sequential Automata

For each automaton $A \in F$, $A = (\sigma_A, s^0_A, \lambda_A, \delta_A)$ the attributes thereof can be derived based on the rules described in following paragraphs.

States

The States of the state-chart are uniquely mapped to states of sequential automata.

- **Root automaton** H. If the (composite) top state s_0 of the state-chart is concurrent then it is mapped to the single (initial) state of a degenerate root automaton H. Otherwise the direct sub-states of the top state are mapped to states σ_H of the root automaton H.

- **Sub−automata** in $\mathcal{A}$ H. Each non−concurrent composite sub-state s of the state-chart defines the states of a unique sequential automaton A_s, as direct sub-states of s are mapped to states of σ_{A_s}. Note that regions (direct sub-states of a concurrent composite state) are not mapped to any state in the extended hierarchical automaton.

- **Initial state.** The initial state s^0_A of an automaton A is the state that corresponds to the state of the state-chart marked by an initial pseudo-state.

Transitions

In order to define the mapping of the transitions, we need the following definitions. A transition of the state-chart is characterized by its least common ancestor (LCA) state,

which is the lowest level non-concurrent state that contains all the source states and target states. The main source main target of a transition is the direct substate of its LCA that contains the sources targets. According to the above rules, main sources and main targets are always transformed to states of the same automaton.

Each transition τ in the state-chart is mapped to a unique transition t of the extended hierarchical automaton as follows. The source SRC_t (target TGT_t) of t is the state that corresponds to the main source (main target) of τ. This means that a compound or interlevel transition of the state-chart is mapped to a transition of the automaton containing the states corresponding to its main source and main target (this automaton is a sub−automaton of the state representing the LCA). The original source and target states will be included in the label of the transition in the form of source restriction and target determinator, as described below.

Transition labels. The label of a transition t is of the form (SR_t, EV_t, G_t, AC_t, TD_t). SR_t and TD_t are generated using the source(s) and target(s) of τ, while the EV_t, G_t and AC_t of t are inherited from τ:

- **Source restriction**. If the set of states that corresponds to the source(s) of τ is the same as SRC_t, then SR_t must be empty, otherwise it is such a set of source(s).

- **Target determinator**. TD_t is the normalized set of states that corresponds to the target(s) of τ. Normalizing means computing the maximal set of orthogonal basic states that are sub-states of the states entered by τ explicitly or by default. In this way, TD_t explicitly contains all the states which have to be entered when the transition is fired, while some of these states are not explicitly pointed to by τ. The following is a sketch of a normalization algorithm which visits the states reached by (segments of) τ, starting from its main target:

  1. If a basic state is reached then it is added to TD_t and recursion stops.

2. If a composite state is reached at its boundary then the algorithm is applied recursively to its initial sub-state, or to the initial sub-state of each of its regions.
3. If a non−concurrent composite state is reached and its boundary is crossed then the algorithm is applied recursively to its direct sub-state where the transition continues (note that branch segments are not considered in the current formalism).
4. If a concurrent composite state is reached and its boundary is crossed then the algorithm is applied recursively to:
   - The direct sub-state(s) of those regions where the transition continues (i)
   - The initial sub-states of the other regions (ii)

Trigger Events

In UML state-charts, each transition (including compound transitions) can have at most one trigger event, since join, fork and branch segments can not have a trigger. Accordingly, EV_t is exactly the trigger event of τ.

Guards

Since fork and joint segments have no guards, each transition may have a single guard (note that branch segments are not considered in the formalism). Accordingly, G_t is exactly the guard of τ.

Actions

AC_t is exactly the sequence of actions of τ.

Composition Function

The composition function ρ is determined by the sub-state relationships of composite states. If a composite state s is non−concurrent and it is not a region then its direct sub-states

form the states of A_s, a sub−automaton of s, where $\{A_s\} = (\rho\ s)$. If a composite state s is concurrent then every one of its regions forms a sub−automaton of s, in such a way that this automaton contains the direct sub-states of the region.

Set of Events

The set of events E is defined as the union of two (not necessarily distinct) sets, the set of events used in the state-chart as triggers of the transitions and the set of events generated by actions. In open systems, the set of events generated by the environment (external stimuli) is also included.

5.3.4 Model Checking of State-Charts by Translating to SPIN

SPIN is a model checker for the verification of asynchronous processes. SPIN uses Linear Temporal Logic (LTL), for specifying the correctness properties. Furthermore, SPIN uses an on-the-fly explicit state model checking rather than the symbolic method employed by other model checkers (e.g. SMV) and uses a number of optimization techniques to reduce the size of the state space, including partial order reduction.

The input language of SPIN is PROMELA, a simple program like notation for specifying process interactions. Process interactions can be specified with primitives like rendezvous or asynchronous message passing through buffered channels, shared variables or a combination of these. SPIN has been used mainly in the verification of the communication protocols and distributed systems but is certainly not restricted to this specific area. We will see in the subsequent paragraphs how SPIN can be used to model check state-charts [5].

5.3.5 Linear-Time Temporal Logic Properties

Linear-Time Temporal Logic (LTL) has been recognized as the leading technique for the specification of temporal rules. It is the de facto standard specification formalism used by

the formal verification community, and a large body of knowledge exists in the literature regarding its use and theoretical properties. In the following, we will discuss about LTL modalities and flavors.

In LTL, the modalities F (Future) and G (Globally) are commonly written as "$\Diamond$" and respectively "$\Box$", e.g., $\Diamond\Box p$ stands for FG p, "eventually globally p". The other operators include X - next, U - until, W - weak.

There are two general flavors of LTL properties that are commonly used:

1. **Safety**: it states that *something bad should never happen.* A safety property is used to rule out bad (catastrophic) behaviors of the system. e.g:
   - Invariants:
     - "x is always less than N" ($\Box(\neg reactor_{temp} > 1000)$)
     - "A missile is never launched unless the launch sequence is successfully completed" ($\Box\ \neg(launch\ \&\ \neg launch_{seq})$)
     - "A train will not cross a train crossing with the gates in the up position" ($\Box\ \neg(cross \wedge gates = up)$)
   - Deadlock freedom: "the system never reaches a state where no moves are possible"
   - Mutual exclusion: "the system never reaches a state where two processes are in the critical section"
2. **Liveness**: states that *something good will eventually happen.* A liveness property is meant to ensure that the system does what is meant to do. e.g.:
   - Termination: "the system eventually terminates" ($\Box(start \rightarrow \Diamond terminate)$)
   - Response properties: "if action X occurs then eventually action Y will occur"
     - If the launch sequence is successfully completed then the missile will be launched ($\Box(launch_{seq} \rightarrow \Diamond launch)$)

– If the train is approaching the crossing, the gates will go down ($\Box$ ($approaching \rightarrow \Diamond gates = down$))

The LTL properties can be further categorized according to the following patterns:

- Absence: p is false (e.g. $\Box(\neg p)$).
- Existence: p becomes true (e.g. $\Diamond(p)$).
- Bounded existence: p occurs at most n times (e.g. ($\neg p$ W (p W ($\neg p$ W (p W $\Box\neg p$)))) for $n = 2$).
- Universality: p is true (e.g. $\Box(p)$).
- Precedence: q precedes p (e.g. $\neg p$ W q).
- Response: q responds to p (e.g. $\Box(p \rightarrow \Diamond q)$.

5.3.6 LTL Model Checking using SPIN

SPIN uses the automata theoretic approach in its model checking engine of the LTL specifications. The inputs to the SPIN model checker represent a description of a concurrent system in PROMELA and its correctness properties are expressed in LTL. The PROMELA description consists in a user-defined process templates (using *proctype* definitions) and at least one process instantiation (using the run command).

The PROMELA (Process or Protocol Meta Language) language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered).

SPIN translates each process template into a finite automaton. The global behavior of the concurrent system is obtained by computing the asynchronous interleaving product of the automata corresponding to each process. To perform verification, SPIN also converts the correctness claim in LTL to a Büchi automaton and computes the synchronous product of the

claim and the automaton for the global behavior. If the language accepted by the resulting Büchi automaton is empty the original claim does not hold on the original system. SPIN actually uses the negation of the correctness claim as the input, so a non-empty intersection gives counter-examples to the correctness claim. A Büchi automaton accepts a system run if and only if it forces the automaton to pass through one or more of its accepting states infinitely often. Such accepting behaviors of a Büchi automaton are called acceptance cycles. To prove that no execution of the system satisfies the negated correctness claim, it suffices to prove that the synchronous product of the system and the Büchi automaton representing the negated claim has no acceptance cycles. SPIN does the computation of automata for concurrent components, their asynchronous product representing the global system. The Büchi automaton for the correctness claim is checked "on-the-fly" by using a nested depth-first search algorithm.

Moreover, PROMELA allows message type definitions using *mtype* statement to declare symbolic values from the specific values to be used in message passing. Message channels are used to model the transfer of data from one process to another. They are declared either locally or globally using the *chan* statement with the size of the channel in square brackets and a list of message types in braces. The *proctype* statement declares a process with parameters, but it does not run them. Such a process is instantiated by a run operation, which can also specify actual parameters. Alternatively, the active modifier can be used to make an instance of the *proctype* to be active in the initial system state. For message passing syntax, PROMELA uses *ch*!*expr* to send the value of expression *expr* to the channel *ch*, and *ch*?*msg* to receive the message. The message is retrieved from the head of the channel, and stored in the variable *msg*. The channels pass messages in first-in-first-out order [1].

The basic control flow constructs in PROMELA are case selection using *if...fi*, and repetition using *do...od* constructs, which use the syntax of guarded commands. However, the semantics of the selection and repetition statements in PROMELA are different from other guarded command languages. First, the communication can be either CSP (Communicating

Sequential Processes) style rendezvous or asynchronous. Moreover, the statements are not aborted when all guards are false but they block, providing the required synchronization. In PROMELA there is no difference between conditions and statements; the execution of every statement is conditional on its executability. Statements are either executable or blocked (FALSE). The executability is the basic means of synchronization. A process can wait for an event to happen. PROMELA accepts two different statement separators: an arrow "→" and the semicolon ";". The two statement separators are equivalent. The arrow is sometimes used as an informal way to indicate a causal relation between two statements.

The *timeout* statement models a special condition that allows a process to abort waiting for a condition that may never become true. It provides an escape from a deadlocked or hang state. The timeout condition becomes true only when no other statements within the distributed system is executable.

A system described in PROMELA can be automatically analyzed for correctness violations. The following types of violations are typical:

- *Assertions*: The statement *assert*(*exp*) statement has no effect if the boolean condition *exp* holds. If the condition does not necessarily hold, i.e., there is an execution sequence in which the condition is violated, the statement will produce an error report during verifications with SPIN.

- *End-states*: Valid end-states are those system states in which every process instance and the *init* process has either reached the end of its defining program body or is blocked at a statement that is labelled with a label that starts with the prefix end. All other states are invalid end-states, signifying deadlocks. During verification an error is reported if there is an execution that terminates in an invalid end-state.

- *Progress states*: A progress state is any system state in which some process instance is at a statement with a label that starts with the prefix progress. A non-progress cycle is detected by the verifier if there is an execution that does not visit a progress state

infinitely often. Non-progress cycles indicate the possibility of starvation or lock-out.

- *Temporal claims*: LTL formulae can be used to express general *safety* and *liveness* properties. SPIN compiles an LTL formulae into a *never claim*, the negation of the correctness property. A *never claim* statement is a special type of process that, if present, is instantiated once. It is equivalent to a Büchi automaton representing the negated property, and is used to detect behaviors that are considered undesirable or illegal.

For instance, the LTL property: $\Box(pUq)$ states that it is always guaranteed that p remains true at least until q becomes true. Similarly $\Box(\Diamond p)$ states that at any point in an execution it is guaranteed that eventually p will become true at least once more.

The automata generated by SPIN for the above two formula are shown below in algorithms 1 and 2, written in the syntax of PROMELA. Both automata contain one non-accepting state (the initial state of the Büchi automaton, to, and one accepting state named accept here).

Algorithm 1 SPIN compiles for "$\Box(pUq)$"

```
never {
to:
if  then
  :: (p) → goto to
  :: (q) → goto accept
end if
accept:
if  then
  :: ((p)||(q)) → goto to
end if
}
```

When checking for state properties, such as assertions, the verifier reports an error if there is an execution that ends in a state in which the *never claim* has terminated. i.e., has reached the end of its body. When checking for acceptance cycles, the verifier reports an error if there is an execution that visits infinitely often an acceptance state. Thus, a temporal claim

Algorithm 2 SPIN compiles for "$\Box\Diamond p$"

```
never {
to:
if  then
  :: (true) → goto to
  :: (p) → goto accept
end if
accept:
if  then
  :: (true) → goto to
end if
}
```

can detect illegal infinite (hence cyclic) behavior by labeling some statements in the never claim with an acceptance label. In such situations the *never claim* is said to be matched. In the absence of acceptance labels, no cyclic behavior can be matched by a temporal claim. Moreover, to check a cyclic temporal claim, acceptance labels should only occur within the claim and nowhere else in the PROMELA system. A *never claim* is intended to monitor every execution step in the rest of the system for illegal behavior and for this reason it executes in lock-step with the other processes (synchronous product). Such illegal behavior is detected if the *never claim* matches along a computation.

5.3.7 Implementing State-Charts in SPIN using Extended Hierarchical Automata

The use of EHAs is motivated by the need for a structural operational semantics definition for state-charts which is difficult in the presence of inter-level transitions. The EHA model uses transitions between states at the same level by lifting inter-level transitions to the uppermost states that are exited and entered, with annotations on the transitions to describe the actual source and target, providing the necessary operational semantics.

System states of an EHA H are modeled by configurations. A configuration is a set of states of the component sequential automata of H, with every sequential automaton

contributing at most one state to a configuration. The root automaton is part of every configuration and when a non-basic state takes part in a configuration, each of its direct sub-automata must contribute to the configuration. The initial configuration is derived from the initial states of the set of sequential automata in a top-down manner starting from the root automaton [5, 1].

The translation from state-charts to SPIN is based on the formal operational semantics presented in the previous section. We reiterate the three principal rules:

1. *Progress Rule*: This rule is applied to a sequential automaton A if one of its states s is in configuration C and if one of the outgoing transitions is enabled and taken non-deterministically. The transition label determines the effect of the transition: the target state s' and the target determinator states are entered.

2. *Composition Rule*: This rule applies to an automaton A that has one of its states s in the configuration C but all outgoing transitions are disabled. If the state s is refined to a set of automata $\{A_1, ..., A_n\}$, the rule delegates the step to the sub-automata by collecting the results of the steps performed by the A_i.

3. *Stuttering Rule*: Applies to a basic state s in the configuration C with none of its outgoing transitions enabled. The effect is to remain in state s without generating any events.

The example in Figure 5.2 represents a state-chart model for a TV set [6]. The top level state TV is an OR (composite) state, whose sub-states are WORKING and WAITING. WORKING is an AND (composite concurrent) state whose orthogonal sub-states are IMAGE and SOUND, each of which is in its turn an OR state. The top level default state is WAITING. The transition labels t0 through t8 are used in order to help the understanding of the translation process (they are not part of the state-chart syntax). The transition with trigger *out* from the AND state WORKING to DISCONNECTED is an inter-level transition, with its source and target at different levels of the state hierarchy.

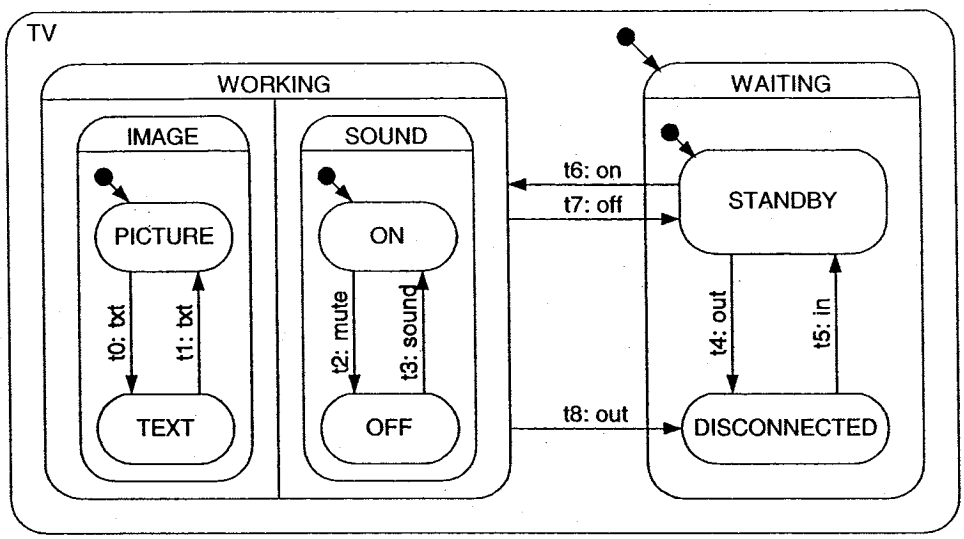

Figure 5.2: State-Chart Example Modeling a TV

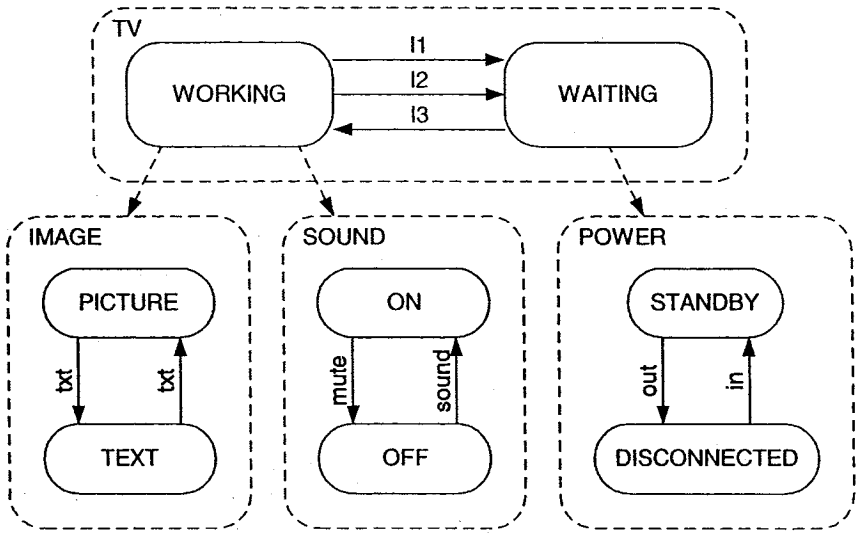

Figure 5.3: EHA Equivalent of the State-Chart TV Model

Transition translation table			
Label	Source restriction	Guard	Target Determinator
l_1	$\emptyset$	off	{STANDBY}
l_2	$\emptyset$	out	{DISCONNECTED}
l_3	{STANDBY}	on	{WORKING,PICTURE,ON}

Table 5.1: Transition labels for the EHA TV model

The EHA corresponding to the state-chart in 5.2 is shown in Figure 5.3. The original state-chart is transformed to sequential automata TV, IMAGE, SOUND and POWER, depicted by dashed rounded boxes. We can see that the state WORKING of sequential automaton TV is refined into the set of sequential automata {IMAGE,SOUND}, denoting their parallel composition. The state WAITING is refined into the singleton set {POWER}. The inter-level transitions labeled t_6, t_7 and t_8 in Figure 5.2 are replaced by transitions labeled l_3, l_2 and l_1, respectively (see the Transition translation Table 5.1). Note that in contrast to state-charts, a transition in an EHA always resides within one sequential automaton. The transition labeled l_3 is enabled if WAITING and STANDBY are active and the event *on* is present. The effect upon taking the transition is that the states WORKING, PICTURE and ON are entered.

Given an EHA and its operational semantics in terms of a Kripke structure k, the translation maps the EHA to a PROMELA model P [1] as follows:

1. Events are treated as uninterpreted symbols and represented as constants (integer values).

2. Since the UML semantics of state-charts do not specify the semantics of queues for storing events directed to an object, the specifier is free to choose among a set, a multi-set and a FIFO queue, as the representation. The choice can be an input parameter for the translator.

3. Sets and multi-sets are represented by their characteristic function (n one-bit variables) and multiplicity functions (n integer variables), respectively. A FIFO queue is directly

mapped to a PROMELA channel whose length LT is specified by the designer.

4. An individual state is modeled by a single bit variable. A configuration corresponds to those states whose bits are set (the corresponding value for setting a bit is 1 as usually).

5. The steps of the Kripke structure corresponding to the EHA are generated by the PROMELA process called STEP, which has the following four phases:

   - Selection of an event from the environment;
   - Identification of all the candidate transitions for firing; this includes identification of enabled transitions and resolution of conflicts based on transition priority;
   - Selection of those transitions among the candidate ones that will be fired; this includes selection among concurrent (orthogonal sub-states) and choice among nondeterministic alternatives;
   - Actual firing of the selected transitions, including identification of the resulting configuration and generation of new events.

6. The STEP process includes a loop to generate successive steps of the EHA. The atomicity of each step is guaranteed by using the atomic directive in PROMELA. This implies that the only values available for verification are the ones obtained at the end of each cycle.

7. The PROMELA code generated for selecting an event from the input queue (in case the queue is represented by a set) uses the selection command:

---

if then
:: $Qe_1 \rightarrow Ev = e_1;\ Qe_1 = 0$
...
:: $Qe_n \rightarrow Ev = e_n;\ Qe_n = 0$
end if

Here Qe_i is the bit representing the presence of event e_i. In case a multi-set representation is used, the guard is $Qe_i \geq 1$ and the action is Qe_i. If a channel is used, the input command $Q?Ev$ is used.

8. In order to identify the candidate transitions for firing, a boolean variable $Cand_i$ corresponding to the transition t_i in the EHA is used. An assignment to $Cand_i$ corresponds to the implementation of the progress rule.

9. The actual transition selection phase involves resolution of conflicts and selection of one transition among the candidates. This is done by nondeterministically assigning 1 to the bit variable Sel_i if $Cand_i$ holds. The code for such an assignment is generated recursively following the tree structure of the EHA.

10. The actual firing of the selected transition t_i involves setting the bit variables for the states that are entered and resetting (the corresponding value for resetting a bit is 0 as usual) the variables for states that are left (exited). Additionally, all the generated events have to be stored in the input queue.

An optimized version of the PROMELA code generated for the aforementioned TV model by this approach is shown below in algorithms 3 to 5. The translation reflects the UML semantics of transition priority and concurrent execution. As an example, several properties are specified for the model. One in the form of a *never claim* that captures the fact that the system should not have a configuration where both the STANDBY and the PICTURE states are active (similarly, additional such properties can easily be specified for other invalid configurations) and the remaining properties are specified in the form of assertions capturing some of the desired system behavior, namely the fact that while in the WORKING state, the configuration should also contain either (PICTURE or TEXT) and (ON or OFF) states for the first assertion and for the second one the same considerations apply where the fact that a configuration containing the WAITING state implies that either STANDBY or DISCONNECTED states are also part of that configuration. (algorithm 6)

Algorithm 3 Definitions

```
/* events */
#define txt 1
#define mute 2
#define sound 3
#define on 4
#define off 5
#define out 6
#define in 7

/* states */
bit WORKING,PICTURE,TEXT,ON,OFF,WAITING,STANDBY,DISCONNECTED

/* set of events in the current environment */
bit Q_ txt, Q_ mute, Q_ sound, Q_ on, Q_ off, Q_out, Q_ in;

/* selected event */
int Ev;

/* whether transition t_i is a candidate for firing */
#define Cand_ 0 (WORKING & PICTURE & (Ev == txt))
#define Cand_ 1 (WORKING & TEXT & (Ev == txt))
#define Cand_ 2 (WORKING & ON & (Ev == mute))
#define Cand_ 3 (WORKING & OFF & (Ev == sound))
#define Cand_ 4 (WAITING & STANDBY & (Ev == out))
#define Cand_ 5 (WAITING & DISCONNECTED & (Ev ==in))
#define Cand_ 6 (WAITING & STANDBY & (Ev == on))&
! (WAITING & STANDBY & (Ev == out))
#define Cand_ 7 (WORKING & (Ev == off)) &
! (WORKING & PICTURE & (Ev == txt)) &
! (WORKING & TEXT & (Ev == txt)) &
! (WORKING & ON & (Ev == mute)) &
! (WORKING & OFF & (Ev == sound))
#define Cand_ 8 (WORKING & (Ev == out)) &
! (WORKING & PICTURE & (Ev == txt)) &
! (WORKING & TEXT & (Ev == txt)) &
! (WORKING & ON & (Ev == mute)) &
! (WORKING & OFF & (Ev == sound))
```

Algorithm 4 procedure atomic procType STEP()

```
if  then
  Q_txt → Ev = txt; Q_txt = 0;
  Q_mute → Ev = mute;Q_mute = 0;
  Q_sound → Ev = sound; Q_sound = 0;
  Q_out → Ev = out; Q_out = 0;
  Q_in → Ev = in; Q_in = 0;
  Q_on → Ev = on; Q_on = 0;
  Q_off → Ev = off; Q_off = 0;
end if

if  then
  Cand_0 → PICTURE = 0; TEXT = 1;
  Cand_1 → TEXT = 0; PICTURE = 1;
  Cand_2 → ON = 0; OFF = 1;
  Cand_3 → OFF = 0; ON = 1;
  Cand_4 → STANDBY = 0; DISCONNECTED = 1;
  Cand_5 → DISCONNECTED = 0; STANDBY = 1;
  Cand_6 → WAITING = 0; STANDBY = 0; WORKING = 1; PICTURE = 1; ON = 1;
  Cand_7 → WORKING = 0; PICTURE = 0; TEXT = 0; ON = 0; OFF = 0; WAITING
  = 1; STANDBY = 1;
  Cand_8 → WORKING = 0; PICTURE = 0; TEXT = 0; ON = 0; OFF = 0; WAITING
  = 1; DISCONNECTED = 1;
else
  skip
end if
```

Algorithm 5 procedure atomic procType init()

```
/* initial configuration */
WAITING = 1; STANDBY = 1; DISCONNECTED = 0; WORKING = 0;
PICTURE = 0; TEXT = 0; ON = 0; OFF = 0;
run STEP();
never {
to:
if  then
  (true) → goto to
  (STANDBY & PICTURE) → goto accept
end if
accept:
if  then
  (true) → goto to
end if
```

Algorithm 6 procedure atomic procType monitor()

assert($\Box(WORKING \rightarrow (PICTURE \parallel TEXT)\ \&\ (ON \parallel OFF))$)
assert($\Box(WAITING \rightarrow (STANDBY \parallel DISCONNECTED))$)

5.3.8 Automated State and Transition Verification

In the foregoing paragraphs we have seen how we can derive a formal model from a given UML state-chart and the tool (SPIN) that can be used in order to apply model checking verification to the formal model of the state-chart. Several property categories were also presented. The next step will consist in extending this framework such way as to be able to automatically generate the properties from the UML state-chart specification.

For the general case, starting with the basics, we have to check that the state-chart has only one top state that it is so designed that no other state can have a transition to it. Moreover, the top state should be composed of more than one sub-state that can be any of basic, composite or (composite) concurrent states. Furthermore, a top state should always be composite.

A basic state should have no sub-states whereas a composite state should have more than one sub-state and a concurrent state should have only composite sub-states. A composite state can have at most one initial state whereas a composite concurrent state can have at most one initial state for each of its regions.

A final state is a special kind of basic state signifying that the enclosing composite state is completed. No further transition should be possible from a final state. If the enclosing state is the top state it means that the entire state machine has completed.

The transitions from one state to another are relating pairs of source and destination states. Each state save the top one should have a source state and each state save the final ones should have a destination state.

All states should be reachable. If a state is considered reachable by several paths, then such paths should be possible. In other words, if for a given state there are several states that share a relationship of the type source/target with that state such that it is the target

in more than one case, then the dynamics of the model should allow that state to be reached by all the paths leading to it.

In the following paragraph we enumerate the constrains that can be used to derive the general properties that the model should respect.

5.3.9 Rules

- **Top state**

  1. A top state is always a composite.
  2. A top state cannot have any containing states.
  3. The top state cannot be the source of a transition.

- **Composite state**

  1. A composite state can have at most one initial vertex.
  2. A composite state can have at most one deep history vertex.
  3. A composite state can have at most one shallow history vertex.
  4. There have to be at least two composite sub-states in a concurrent composite state.
  5. A concurrent state can only have composite states as sub-states.
  6. The sub-states of a composite state are part of only that composite state.

- **Final state**

  A final state cannot have any outgoing transitions.

- **Pseudo-state**

  1. An initial vertex can have at most one outgoing transition and no incoming transitions.

2. History vertices can have at most one outgoing transition.
3. A join vertex must have at least two incoming transitions and exactly one outgoing transition.
4. All transitions incoming a join vertex must originate in different regions of a concurrent state.
5. A fork vertex must have at least two outgoing transitions and exactly one incoming transition.
6. All transitions outgoing a fork vertex must target states in different regions of a concurrent state.
7. A junction vertex must have at least one incoming and one outgoing transition.
8. A choice vertex must have at least one incoming and one outgoing transition.

- **Synch state**

  1. The value of the bound attribute must be a positive integer
  2. All incoming transitions to a synch state must come from the same region and all outgoing transitions from a synch state must go to the same region.

- **Submachine state**

  Only stub-states are allowed as the sub-states of a submachine state. Submachine states are never concurrent.

- **Hierarchical state decomposition**

  Designed to allow sharing (reuse) of behavior. The sub-states (nested states) need only define the differences from the super-states (surrounding states). A sub-state can easily reuse the common behavior from its super-state(s) by simply ignoring commonly handled events, which are then automatically handled by higher level states. In this manner, the sub-states can share all aspects of behavior with their super-states.

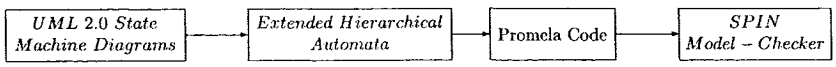

Figure 5.4: Model Transformation

- **Guards**

  A guard must be a pure boolean expression and should not have side effects (like for example changing the value of some attributes while being evaluated).

- **Transitions**

  1. A fork segment should not have guards or triggers.
  2. A fork segment should always target a state.
  3. A join segment should not have guards or triggers.
  4. A join segment should always originate from a state.
  5. Transitions outgoing pseudo-states may not have a trigger.
  6. An initial transition at the topmost level has no trigger.

5.3.10 Translation of UML State-Charts to Extended Hierarchical Automata

The translation maps a UML state-chart to an extended hierarchical automaton (Figure 5.4) $H = (F; E; \rho)$ by defining the set of sequential automata F, the composition function ρ and the set of events E. Each automaton $A \in F$ is defined by a set of states, the initial state, a set of transitions, and a set of transition labels. States of the state-chart are uniquely mapped to states of sequential automata. The initial state s_0 of an automaton A is the state that corresponds to the state of the state-chart marked by an initial pseudostate. Each transition τ in the state-chart is mapped to a unique transition t of the extended hierarchical automaton. The label of a transition t is of the form $(SR_t; EV_t; G_t; AC_t; TD_t)$. SR_t and TD_t

are generated using the source(s) and target(s) of τ , while the EV_t, G_t and AC_t of t are inherited from τ. The translation follows certain rules that we will explain shortly.

Defining the states of the Sequential Automata

The solution consist of a top-down walk through the hierarchical state machine. Given the (composite) top state s_0 of the state-chart, the direct substates of the top state are mapped to states σ_H of an automaton H called the root automaton.

The direct substates of each non-concurrent composite state s are mapped to the states of a unique sequential automaton A_s.

Regions (direct substates of a concurrent composite state) are not mapped to any state in the extended hierarchical automaton. However, the substates of each region are mapped to a new sequential automaton A_s.

Defining the Set of Transitions of the Sequential Automata

Each transition τ in the state-chart is mapped to a unique transition t of the extended hierarchical automaton. The corresponding EHA source and target is determined by the least common ancestor (LCA) state, which is the lowest level nonconcurrent state that contains all the source states and target states in the state-chart. The main source/target of a EHA transition is the direct substate of the LCA that contains the respective sources/targets. Hence, the main sources/targets are always transformed to states of the same automaton such that the compound or interlevel transitions are mapped to a transitions of the automaton containing the states corresponding to its main source and main target. The original source and target states will be included in the label of the transition in the form of source restriction and target determinator.

The label of a transition t is of the form $(SR_t; EV_t; G_t; AC_t; TD_t)$. SR_t and TD_t are the source restriction respectively the target determinator. Both are generated using the source(s) and target(s) of τ , while the EV_t, G_t and AC_t of t are inherited from τ

If the set of states that correspond to the source(s) of τ is the same as SRC_t, then SR_t must be empty, otherwise it is such a set of source(s).

The maximal set of orthogonal basic states that are substates of the states targeted by τ explicitly or by default represents the TD_t.

From the foregoing, we can summarize the main steps required in order to convert a given state-chart to the corresponding EHA:

- A label is assigned for each state such that it uniquely identifies its corresponding state in the hierarchy. This label is useful when computing the LCA.
- The states of the state-chart are mapped to states of individual sequential automata as described above.
- For each state in the EHA, we assign a (possibly empty) list of sub-automata that reflects its refinement. Obviously, the sub-automata lists are so constructed as to preclude any circular reference.
- Each of the state-chart transition is mapped to a transition in one of the sequential automata along with the related information concerning the source restriction and the target determinator as described above. (algorithms 7 to 11)

5.3.11 Promela Code Generation Algorithm

Once the translation of a given state-chart to the corresponding EHA is completed we should have the following data structures available:

- The list of states
- The list of events
- The list of transitions (augmented with the source restriction and target determinator where is the case)

Note that the number of states of the EHA can be smaller than the number of states of the original state-charts as the regions of the concurrent states are not represented in the EHA as states.

The Promela code generation procedure consists in several stages. At each stage some code blocks are generated as follows:

1. iterate the list of events
   - define a different constant (integer value) for each event
   - define the set of events (a boolean variable for each event)
2. iterate the list of states
   - define a single bit variable for each state (these variables represent the configuration of the EHA)
3. define an event holding variable (integer)
4. define the transition candidates as boolean expressions corresponding to the conjunction of the following terms:
   - the expected firing event (compared to the event holding variable)
   - the source state and if it is the case, the source restriction.
   - the absence of any other higher priority transition condition if the source state is composite (amounts to negating any nested firing condition in the source state)
   - the boolean guard (if present)
5. generate a *STEP* process that contains an unconditional block (equivalent to while loop) that contains the following:
   - an *atomic* block (where the input events are being intercepted) consisting in:

- an *if* statement that nondeterministically generates one of the events (assigns 1 to the corresponding boolean variable)
- an *if* statement that tests for event occurrences and sets the event holding variable accordingly to the intercepted event and consumes the event by setting the value thereof to 0.
- subsequent nested *if* statements where the transition candidates are matched against the event holding variable in conjunction with the current configuration such that when a match is found, the corresponding new configuration is set (if the action related to the transition candidate requires some events to be generated then these events should be present in the next event set and their corresponding variable should be set to 1) otherwise just skip the *if* statement. If all the *if* statements are skipped the result it that the EHA is stuttering. The *if* statements should be constructed following the EHA structure. This means that starting with the root automaton we have one *if* statement where all the transition candidates of the root automaton are tested with an additional last test consisting in the disjunction of all other remaining transition below the root automaton (test if at least one of the sub-automata transition candidates evaluates to true). Under this last condition, for each of the immediate sub-automata (direct children) below the root automaton we need to add subsequent nested if statements constructed the same way as in the case of the root automaton (using the procedure recursively) until we add *if* statements for all of the automata.

6. generate the *init* process where the initial configuration is set and the *STEP* process is spawned.

7. produce the output promela code by concatenating all the text block generated

For our previous TV example we can detail the outlined procedure as follows:

From the data structures we have the initial hierarchical structure of the state-chart:

TV (Composite)

WORKING (Concurrent)

IMAGE (Composite)

PICTURE (Basic)

TEXT (Basic)

SOUND (Composite)

ON (Basic)

OFF (Basic)

WAITING (Composite)

STANDBY (Basic)

DISCONNECT (Basic)

Also the corresponding EHA sequential automata is:

TV (Sequential root automaton)

WORKING

WAITING

IMAGE (Sequential sub-automaton / refinement of the WORKING state)

PICTURE

TEXT

SOUND (Sequential sub-automaton / refinement of the WORKING state)

ON

OFF

POWER (Sequential sub-automaton / refinement of the WAITING state)

STANDBY

DISCONNECTED

Events are {txt,mute,sound,on,off,in,out}.

States are listed below:

Event	Source	Target	Restriction	Dependent targets
off	WORKING	WAITING	-	STANDBY
out	WORKING	WAITING	-	DISCONNECTED
on	WAITING	WORKING	STANDBY	PICTURE / ON
txt	PICTURE	TEXT	-	-
txt	TEXT	PICTURE	-	-
mute	ON	OFF	-	-
sound	OFF	ON	-	-
out	STANDBY	DISCONNECTED	-	-
in	DISCONNECTED	STANDBY	-	-

Table 5.2: Transition list table

{WORKING,WAITING,PICTURE,TEXT,ON,OFF,STANDBY,DISCONNECTED}.

The list of transitions as defined by the transition list table 5.2 and also the complete source of our solution are depicted in 7 to 11. The code generation stages are:

1. Enumerate the list of events and generate the following text blocks.
2. generate the event holding text block.
3. Enumerate the list of states and generate the following text blocks.
4. Enumerate the list of transitions and generate the following text blocks.
5. generate the STEP process text blocks.
6. generate the init process text blocks.
7. Concatenate all the text blocks in the following order.

Algorithm 7 ObtentionOfSatesSequentialAutomata

```
1: Create a new automaton (root automaton) A[0]
2: if Top-state is composite then
3:    Add All substates of Top-state to A[0]
4:    Mark the state of the state-chart marked as initial (default) to be the initial state of
      A[0]
5:    Add all substates of Top-state to an unreviewed queue of states
6: end if
7: i = 1
8: repeat
9:    Pick the head of the unreviewed queue of states called S
10:   if state S is composite then
11:      if substates of S are concurrent then
12:         Add all substates of S to the tail of unreviewed queue of states
13:         Add substates to the list of concurrent states.
14:         Remove S from unreviewed queue of states
15:      else
16:         Create one new automaton A[i]
17:         Add All substates of S to automaton A[i]
18:         if S is in the list of concurrent states then
19:            Add A[i] to the set of sub-automata of the closest non-concurrent ancestor of
               S obtained by refinement function
20:         else
21:            Add A[i] to the set of sub-automata of S obtained by refinement function
22:         end if
23:         Mark the state of the state-chart marked as initial (default) to be the initial state
            of A[i]
24:         Add all substates of S to the tail of unreviewed queue of states
25:         Remove S from unreviewed queue of states
26:         i = i + 1
27:      end if
28:   else
29:      Remove S from unreviewed queue of states
30:   end if
31: until no more states in unreviewed set of states
```

Algorithm 8 ObtentionOfTransitionsOfAutomata

```
1: i = 1
2: for Each state S in the state-chart such that S is not in the list of concurrent states do
3:    Add All outgoing transitions to the unreviewed set of transitions
4:    repeat
5:       Pick up a transition τ from unreviewed set of transitions
6:       LCAState = LCA(τ.Source,τ.Target)
7:       if τ.Source.Parent ≠ τ.Target.Parent then
8:          SRCt[i] = τ.Source.Ancestor that is included in LCAState.substates
9:          TGTt[i] = τ.Target.Ancestor that is included in LCAState.substates
10:         if LCAState = τ.Source.Parent then
11:            SRt[i]= ∅
12:         else
13:            SRt[i]= τ.Source
14:         end if
15:         EVt[i] = τ.event
16:         Act[i] = τ.action
17:      else
18:         SRCt[i] = τ.Source
19:         TGTt[i] = τ.Target
20:         SRt[i]= ∅
21:      end if
22:      Determine(τ.Target, TDt[i])
23:      Remove τ from the unreviewed set of transitions
24:      i = i + 1
25:   until unreviewed set of transitions is empty
26: end for
```

Algorithm 9 Determine(currentState,TargetDeterminator)

```
1: if currentState is basic then
2:    TDt[i] = TDt[i] ∪ currentState
3:    Return
4: else
5:    if currentState is composed of concurrent substates then
6:       for Each substate S of currentState do
7:          Determine(S)
8:       end for
9:    else
10:      Determine(initial state of currentState)
11:   end if
12: end if
```

Algorithm 10 LCALabel(A,label)

```
1:
2: for Each state S in A do
3:    S.label = label + "." + index
4:    if S.refin ≠ ∅ then
5:       for Each state A′ in S.refin  do
6:          LCALabel(A′, S.label + "." + index)
7:       end for
8:    end if
9: end for
```

Algorithm 11 LCAFind(A,B)

```
1: {A and B are nodes that are already labeled.}
2: {The LCALabel algorithm is supposed to have been previously executed.}
3: index = 1 {This is the first index of the node label}
4: LCA = emptyLabel
5: while A.Label.Length ≥ index and B.Label.length ≥ index do
6:    if A.Label[index] == B.Label[index]  then
7:       LCA = LCA + A.Label[index]
8:       index = index + 1
9:    else
10:      Return LCA
11:   end if
12: end while
13: Return LCA
```

Chapter 6

Case Study

In this chapter, we are giving some insights about the implementation of the verification and validation of state-chart diagram. We are also presenting our contribution related to the branching points in this diagram.

We have implemented a module that takes a UML state-chart diagram and converts it to a format accepted by the model checker *SPIN* [4], which is the unified model checker in our framework. Our approach for the UML state-chart diagrams consists in converting them into Extended Hierarchical Automata (EHA).

The translation maps a UML state-chart to an EHA $H = (F,E,\rho)$ by defining the set of sequential automata F, the composition function ρ, and the set of events E. Each automaton $A \in F$ is defined as a tuple composed of a set of states, the initial state, a set of transitions, and a set of transition labels. States of the state-chart are uniquely mapped

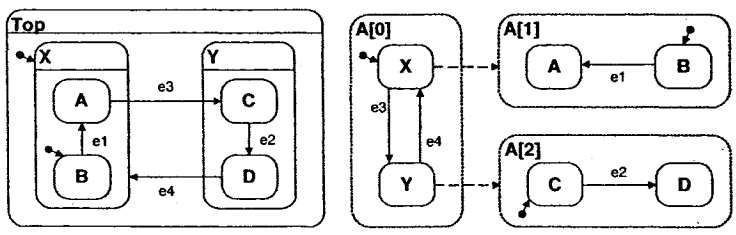

Figure 6.1: State-chart Example (left) and the Corresponding EHA (right)

to states of sequential automata. The initial state s_0 of an automaton A is the state that corresponds to the state of the state-chart, which is marked by an initial pseudo state. Each transition t in the state-chart is mapped to a unique transition t' of the EHA. The label of a transition t' is of the form ($SR_{t'}$,$EV_{t'}$,$GU_{t'}$,$AC_{t'}$,$TD_{t'}$). $SR_{t'}$ refers to the Source Restriction and $TD_{t'}$ to the Target Determinator of t'. $SR_{t'}$ and $TD_{t'}$ are generated using the source(s) and target(s) of t. $EV_{t'}$,$GU_{t'}$ and $AC_{t'}$ are event, guard and action of t' inherited from t. The corresponding EHA source and target is determined by the Least Common Ancestor (LCA) state, which is the lowest level nonconcurrent state that contains all the source and target states in the state-chart. The main source/target of a EHA transition is the direct substate of the LCA that contains the respective sources/targets. Figure 6.1 depicts a state-chart example and the corresponding EHA. Accordingly, we have for example LCA(A,B) = X and LCA(A,C) = *Top*.

From the foregoing, we can summarize the main steps required for the translation:

- The states of the state-chart are mapped to states of individual sequential automata as described above.
- A label (similar to a table of contents numbering e.g. 1.2.4) is assigned for each state in order to uniquely identify it in the hierarchy. This label is useful when computing the LCA.
- For each state in the EHA, we assign a (possibly empty) list of sub-automata that reflects its refinement. Obviously, the sub-automata lists are so constructed as to preclude any circular reference.
- Each of the state-chart transition is mapped to a transition in one of the sequential automata along with the related information concerning the source restriction and the target determinator as described above.

Once the translation of a given state-chart to the corresponding EHA is completed, the following data structures are available: The list of states, the list of events, and the list

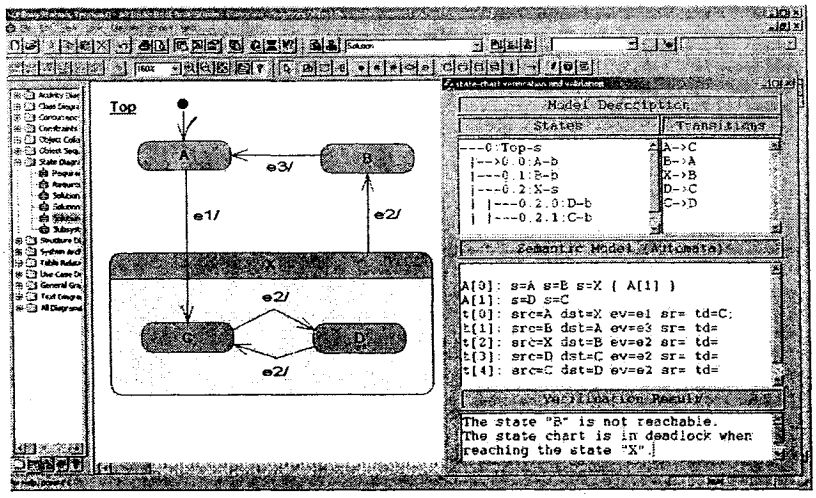

Figure 6.2: Snapshot of State-chart Verification

of transitions (augmented with the source restriction and the target determinator where is the case). The number of states of the EHA can be smaller than the number of states of the original state-chart since the regions of the concurrent states are not represented in the EHA as states.

The *SPIN* code generation procedure consists in several stages wherein different code blocks are generated. Subsequently, we define different constants for each event. An event holding variable would be assigned the corresponding value for the current event that is being supplied to the EHA. Furthermore, from the list of states, we define a single bit variable for each state. For every transition, we define a corresponding candidate as a boolean expression over the event holding variable, the source state, the source restriction, and the guard. However, in order to cope with the priorities, we must add to each transition candidate terms that test higher priority transition candidates if any. Figure 6.2 outlines a snapshot of the state-chart verification, showing a flawed design example that has deadlock when reaching the state "X" while state "B " is unreachable.

We added support for conditional branching where we have one incoming transition and several outgoing transition segments called branches. Branches are labeled with guards

that determine which one is to be actually taken. Only one of the branches can be taken even if the guard of more than one of the branches holds (in that case, one of the branches is arbitrarily chosen). If the special branch labeled with a guard "else" is present and all the other branches have guards evaluated to false, this special branch is taken.

We replace each transition to a branching point with several transitions corresponding to the individual branches (including the else branch if present) such that each new transition is connecting the initial source state to the corresponding branch destination state by the same event as the initial transition but having the guard of the corresponding branch.

The guard "else" of the special branch denotes the negation of the conjunction of all the other branch guards. If the initial transition has itself a guard then the guard of each transition is given by the conjunction of the initial state guard and the branch guard. Moreover, whenever we have nested branching (a branch is entering another branching point), it is possible to convert the nested branch points to a single branch point by constructing the guards for the final branches using conjunction expressions over the linked (dependent) branch guards.

Chapter 7

Conclusion

In this thesis we proposed a new unified paradigm for verification and validation of the UML state-chart diagram in the context of systems engineering design. Increased complexity in emerging systems requires the existence of techniques to automate and verify their design. Formal methods are useful in building automatic tools that support the design, verification, and validation of systems. Assessing the correctness of the system's design (especially in the early stages) will eliminate a huge cost for the next phases of the system's development. The overall purpose of the thesis is to present precise, end to end, procedures that can be used to produce verification and validation assessments on state-chart diagram. To do so, we presented semantic models, conversion algorithms, verification technology, and properties for this diagram.

The distinctive feature of our approach is an established synergy between three major approaches, which are model-checking, program analysis, and software engineering techniques. The language considered here is UML 2.0, which is the most prominent systems engineering languages. We introduced and detailed several algorithms that can be used in order to support a fully automated model checking procedure for a given state-chart diagram. Even though our approach may be considered less rich than the others, we can safely say

that with respect to identifying potential deadlocks or unreachable states, it provides an adequate verification methodology basis. To validate the proposed approach, we designed and implemented the algorithms into an integrated and automated environment that is presently capable of assessing state machine design models. Furthermore, we are confident that an extension for other diagrams would have a significant benefit due to using such a unified approach.

As future work, we intend to further develop the current implementation of our verification and validation environment by extending it to cover a broader range within the behavioral subset of UML/SysML diagrams. Moreover, we plan to undertake some real-life case studies in order to construct a more compelling case for our paradigm.

Bibliography

[1] First Official Release of Hugo/RT. http://www.pst.ifi.lmu.de/projekte/hugo Access Date: Oct 2006.

[2] Information Society Technologies, Development of Real-Time Embedded Systems (OMEGA). http://www-omega.imag.fr/ Access Date: Oct 2006.

[3] International Council on System Engineering. http://www.incose.org Access Date: Oct 2006.

[4] International Standard Organization. http://www.iso.org Access Date: Oct 2006.

[5] OMG-UML Importance. http://www.omg.org/news/pr97/umlprimer.html Access Date: Oct 2006.

[6] Unified Modeling Language. http://www.uml.org/ Access Date: Oct 2006.

[7] Verification, Validation, and Object Technology. http://www.isot.ece.uvic.ca/vvot.html Access Date: Oct 2006.

[8] Purandar Bhaduri and S. Ramesh. Model Checking of Statechart Models. Survey and Research Directions. *ArXiv Computer Science e-prints*, 2004. http://adsabs.harvard.edu/cgi-bin/nph-bib_query?bibcode=2004cs.....7038B&db_key=PRE Access Date: Oct 2006.

[9] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994. `http://citeseer.ist.psu.edu/476909.html` Access Date: Oct 2006.

[10] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994. `http://dx.doi.org/10.1109/32.295895` Access Date: Oct 2006.

[11] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A New Symbolic Model Verifier. In *Computer Aided Verification*, pages 495–499, 1999. `http://citeseer.ist.psu.edu/cimatti99nusmv.html` Access Date: Oct 2006.

[12] I. Majzik D. Latella and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. *Kluwer Academic Publishers, Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, 1999.

[13] Alexandre David, Gerd Behrmann, Kim G. Larsen, and Wang Yi. A Tool Architecture for the Next Generation of Uppaal. Technical report, 2002. `http://citeseer.ist.psu.edu/article/david03tool.html` Access Date: Oct 2006.

[14] J. Eder, G. Kappel, and M. Schrefl. *Coupling and Cohesion in Object-Oriented Systems.* 1992. `http://citeseer.ist.psu.edu/eder92coupling.html` Access Date: Oct 2006.

[15] Gregor Engels, Jochen M. Kuster, Reiko Heckel, and Marc Lohmann. Model-Based Verification and Validation of Properties. In Roswitha Bardohl and Hartmut Ehrig, editors, *Electronic Notes in Theoretical Computer Science*, volume 82. Elsevier, 2003.

[16] Object Management Group. Common Warehouse Metamodel (CWM) Specification, 2001. `http://www.omg.org/docs/ad/01-02-01.pdf` Access Date: Oct 2006.

[17] Object Management Group. Meta-Object Facility (MOF) Specification, 2002. http://www.omg.org/docs/formal/02-04-03.pdf Access Date: Oct 2006.

[18] Object Management Group. UML 2.0 Infrastructure Specification. 2002. http://www.omg.org/docs/ptc/03-09-15.pdf Access Date: Oct 2006.

[19] Object Management Group. UML 2.0 Infrastructure Specification, 2002. http://www.omg.org/docs/ptc/03-09-15.pdf Access Date: Oct 2006.

[20] Object Management Group. UML 2.0 OCL Specification, 2003. http://www.omg.org/docs/ptc/03-10-14.pdf Access Date: Oct 2006.

[21] Object Management Group. UML 2.0 Superstructure Specification, 2003. http://www.omg.org/docs/ptc/03-08-02.pdf Access Date: Oct 2006.

[22] Object Management Group. UML for Systems Engineering. 2003. http://syseng.omg.org/UML_for_SE_RFP.htm Access Date: Oct 2006.

[23] Object Management Group. XML Metadata Interchange (XMI) Specification, 2003. http://www.omg.org/docs/formal/03-05-02.pdf Access Date: Oct 2006.

[24] Esther Guerra and Juan de Lara. A Framework for the Verification of UML Models. Examples Using Petri Nets. pages 325–334, 2003. http://dblp.uni-trier.de Access Date: Oct 2006.

[25] Michael Hind and Anthony Pioli. Traveling Through Dakota: Experiences with an Object-Oriented Program Analysis System. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, page 49. IEEE Computer Society, 2000.

[26] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. http://doi.acm.org/10.1145/359576.359585 Access Date: Oct 2006.

[27] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[28] Ivar Jacobson. *Object-Oriented Software Engineering.* ACM Press, New York, NY, USA, 1992.

[29] A. Knapp, S. Merz, and C. Rauh. Model Checking Timed UML State Machines and Collaborations. Technical report, Institut fur Informatik, Ludwig-Maximilians-Universitat Munchen and Institut fur Informatik, Technische Universitat Munchen, 2002. `http://citeseer.ist.psu.edu/knapp02model.html` Access Date: Oct 2006.

[30] D. Kroening. Application Specific Higher Order Logic Theorem Proving, 2002.

[31] Diego Latella, István Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart diagrams using the spin model-checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.

[32] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. 1999. Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS).

[33] Johan Lilius and Ivan Porres Paltor. vUML: a Tool for Verifying UML Models. (TUCS-TR-272), 8 1999. `http://citeseer.ist.psu.edu/lilius99vuml.html` Access Date: Oct 2006.

[34] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual.* Springer-Verlag London, 1997.

[35] Stephan Merz. Model Checking: A Tutorial Overview. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, 2001.

[36] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing Statecharts in Promela/Spin, 1997. http://citeseer.ist.psu.edu/holzmann98implementing.html Access Date: Oct 2006.

[37] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[38] Octavian Patrascoiu. Object Oriented Metrics. In *Proceedings of the International Symposium on System Theory (ISST-00)*, Craiova, Romania, 2000. http://www.cs.kent.ac.uk/pubs/2000/1692 Access Date: Oct 2006.

[39] Tom Pender. *UML Bible.* Wiley, New York, NY, USA, 2003.

[40] Marco Roveri. PSL Sugar: Formal Specification Language, 2004. http://sra.itc.it/people/roveri/courses/afm/Sugar.pdf Access Date: Oct 2006.

[41] J. Rushby and D.W.J. Stringer-Calvert. A Less Elementary Tutorial for the PVS Specification and Verification System. Technical Report SRI-CSL-95-10, Menlo Park, CA, 1995. http://citeseer.ist.psu.edu/250070.html Access Date: Oct 2006.

[42] Johann Schumann. Automated Theorem Proving in High-Quality Software Design. In *Intellectics and Computational Logic*, pages 295–312, 2000.

[43] ARTiSAN Software. ARTiSAN Real-time Modeler, 2002. http://www.microprocess.com/agls/documents/ARTISAN/FichesProduits/Model_4.pdf Access Date: Oct 2006.

[44] ARTiSAN Software. ARTiSAN Real-time Studio, 2002. http://www.artisansw.com/pdflibrary/Rts_5.0_datasheet.pdf Access Date: Oct 2006.