AN INSPECTION APPROACH FOR CONCEPTUAL MODELS OF THE MISSION SPACE IN A DOMAIN SPECIFIC NOTATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

Ö. ÖZGÜR TANRIÖVER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

SEPTEMBER 2008

Approval of the Graduate School of Informatics

_____

Prof. Dr. Nazife BAYKAL

Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Doctor of Philosophy.

_____

Prof. Dr. Yasemin YARDIMCI

Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy.

_____

Prof. Dr. Semih BİLGEN

Supervisor

Examining Committee Members

Assoc. Prof. Dr. Onur DEMİRÖRS (METU, II) _____

Prof. Dr. Semih BİLGEN (METU, EEE) _____

Assoc. Prof. Dr. Ali DOĞRU (METU, CENG) _____

Dr. Altan KOÇYİĞİT (METU, II) _____

Prof. Dr. Hayri SEVER (HU, CS) _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this wok.

Name, Last name: Ö. Özgür TANRIÖVER

Signature : _____

ABSTRACT

AN INSPECTION APPROACH
FOR CONCEPTUAL MODELS OF THE MISSION SPACE
IN A DOMAIN SPECIFIC NOTATION

Tanrıöver, Ö. Özgür
Ph.D., Department of Information Systems
Supervisor: Prof. Dr. Semih Bilgen

September 2008, 120 pages

An inspection approach is proposed for improving the quality of conceptual models developed in a domain specific notation. First, the process of identification of desirable properties of conceptual models in a domain specific notation is described. Intra- and inter-view properties are considered. Semantic properties are defined considering the conceptual modeling notation. A systematic inspection process is proposed for checking semantic properties of different types of diagrams and of the relations between these diagrams. This process is applied to two real mission space conceptual models. With the proposed inspection approach, it is possible to identify subtle semantic issues which are not identified by many of the contemporary UML CASE tools and other inspection methods.

Keywords: Conceptual model inspection, CMMS, Conceptual model quality improvement, UML model quality improvement.

ÖZ

ALANA ÖZGÜ BİR NOTASYONDA GÖREV UZAYI KAVRAMSAL MODELLERİ İNCELEME YAKLAŞIMI

Tanrıöver, Ö. Özgür
Doktora, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Prof. Dr. Semih Bilgen

Eylül 2008, 120 sayfa

Bu çalışmada, alana özgü bir notasyonda geliştirilmiş kavramsal modellerin doğrulanması için bir inceleme yaklaşımı önerildi. Önce, alana özgü bir notasyonda geliştirilmiş kavramsal modellerde doğruluk özelliklerinin belirlenmesi için kullanılacak bir çerçeve tanımlandı. Bu çerçeve kullanılarak, anlambilimsel doğruluk özellikleri kavramsal modelleme notasyonuna göre belirlendi. Görünüm içi ve aynı zamanda görünümler arası özellikler göz önüne alındı. Değişik diyagramların ve bu diyagramlar arasındaki ilişkilerin, ağırlıklı olarak anlambilimsel doğruluk özelliklerinin kontrol edilmesi için sistematik bir inceleme süreci önerildi. Bu süreç daha sonra iki gerçek görev uzayı kavramsal modeline uygulandı. Bu inceleme süreci ile güncel UML modelleme araçlarının ve diğer inceleme yöntemlerinin belirlemediği anlambilimsel mesele belirlenebildi.

Anahtar Kelimeler: Kavramsal model inceleme, CMMS, Kavramsal model kalite iyileştirme, UML modeli kalite iyileştirme.

ACKNOWLEDGMENTS

First of all, I would like to express my sincere gratitude to my supervisor Semih Bilgen for his guidance, insightful suggestions and comments throughout these years. I also appreciate deeply his never ending support and positive attitude.

I wish to express my sincere thanks to my committee members Onur Demirörs and Hayri Sever for their comments and suggestions over the last three years. I thank them for being always ready to reserve their valuable time to me.

Many thanks to my superiors at Banking Regulation Agency in the name of Ahmet Türkay Varlı, for tolerating me time for Phd. studies.

I would like to thank my friends, Alpay Karagöz and Utkan Eryılmaz. They have provided me with support whenever I needed it. Our discussions and their comments were inspiring and valuable.

I would like also to acknowledge the fruitful discussions, in Ankara in spring of 2007, with Dr. Dale Pace, formerly with the John Hopkins University, Applied Physics Laboratory.

Finally, I am grateful to my parents and especially my brother Oğuz, he gave me confidence, I knew that he was always there; ready to support me in any possible means.

Many thanks and apologies to others whom I may have inadvertently forgotten to mention.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

BOM : Base Object Model

BVUML : Behavior Verification for UML

C4ISR : Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance

CIM: Computation Independent Model

CM : Conceptual Model

CMMS : Conceptual Models of the Mission Space

DCMF : Defense Conceptual Modeling Framework

DL : Description Logics

DMSO : Defense Modeling and Simulation Office

DoD : The United States Department of Defense

DSN : Domain Specific Notation

EER: Enhanced Entity Relationships

FEDEP : Federation Development and Execution Process m

FOL: First Order Logic

GME : Generic Modeling Environment

GMVV&A: Generic Methodology for Verification, Validation and Accreditation

IEEE : Institute of Electrical and Electronic Engineers

ITOP : International Test Operations Procedure

KAMA : Short for conceptual modeling project developed for Turkish Armed Forces

MDA : Model Driven Architecture

MDSD: Model Driven Software Development

MODED: Methodology for Ontology Based Detection or Errors in UML Designs

MOF : Meta Object Facility

M&S : Modeling and Simulation

OCL : Object Constraint Language

OMG : Object Management Group

OMT : Object Modeling Technique

PBR : Perspective Based Reading

PIM : Platform Independent Models

PROMELA: Process Meta Language

PN : Petri Nets

REVVA : Common Validation, Verification and Accreditation Framework for Simulation

SEDEP : Synthetic Environment Development and Exploitation Process

SME : Subject Matter Expert

SQA : Software Quality Assurance

UML : The Unified Modeling Language

US DoD : U.S. Department of Defense

VIATRA : Visual Automated Model Transformation

VUML: Verify UML

V&V : Verification and Validation

VV&A : Verification, Validation and Accreditation

WFR : Well Formedness Rules

TURTLE : Timed UML and RT- Lotos Environment

XMI : Metadata Interchange

XML : eXtensible Markup Language

CHAPTER 1

INTRODUCTION

Conceptual models for simulations are limited representations of selected aspects of the real world. Unless the conceptual models for simulations are demonstrated to be correct, the predictions and explanations based on simulations will lead to inaccurate knowledge and decisions (MacKenzie, Shulmeyer & Yılmaz, 2002).

Recently, there has been a growing tendency to adopt UML for different modeling needs and domains. UML can also be adopted to be utilized for representing conceptual models for simulations as well. Although many advantages exist as discussed by Karagöz & Demirörs (2008), utilizing a notation derived from UML may lead to further problems for conceptual modeling because UML is a semi formal language (Kim & Carrington, 2000, Ober, 2004). In addition to completeness and correctness issues of translation of the problem frame to conceptual representation (validation), ambiguity inherent in UML, its support of multiple views and the extension mechanism may further increase incompleteness, inconsistencies, incorrectness and redundancies in models (verification). Hence, error reduction early in the simulation modeling lifecycle is needed. Conceptual model verification addresses these concerns.

1.1. The Context

In this section, we would like to present the context of this research. We will describe three important concepts on which the study is based. These are (i) simulation conceptual modeling, (ii) conceptual model representation, and (iii) verification and validation of conceptual models.

1.1.1 Simulation Conceptual Modeling

According to Merriam-Webster's Collegiate Dictionary (Merriam Webster, 2008) a conceptual model is "an abstract representation (model) of something generalized from particular instances (concept)". Conceptual models are used in very different scientific disciplines ranging from philosophy to software engineering. Each of these disciplines employs different modeling techniques, methodologies, tools and terminology for conceptual modeling. The part of reality for which a model is to be created is referred by the term “Object System” in software engineering. This is called “Universe of Discourse” in logic (Boman et al., 1997). From a knowledge engineering perspective conceptual models are the knowledge content of the knowledge base in terms of real-world entities and relations (Dieste, Juristo, Moreno, Pazos & Sierra, 2000).

It is agreed that a conceptual model is an abstract representation of a real world problem situation independent of the solution which includes entities, their actions and interactions, algorithms, assumptions and constraints. On the other hand, conceptual modeling is used for different phases of development; including requirements engineering (Insfran, Pastor & Wieringa, 2002), ontology modeling, design and data modeling (Lacy et al., 2001). For example, Taentzer (2003) argues that the purpose is to permit the modelers to perform a conceptual analysis before design choices and thinking of implementation concerns. Hence, the conceptual model forms the basis of the design model. From these various needs emerges the need for a flexible modeling notation for conceptual modeling which can be adopted for various purposes.

There is no consensus on the definition of the conceptual models within the simulation domain either. According to Pace (2000), the fundamental limitation is the absence of a complete and coherent theoretical foundation of simulation development that is widely accepted by M&S communities. Although there exists no consensus on the definition of a conceptual model, some common characteristics are identified by Karagöz & Demirörs (2008) based on a literature review. Following is the subset of these characteristics that have been also influential in our approach:

- The conceptual model is a simplified representation of the real world.
- The conceptual model is independent of the model code or software, while model design includes both the conceptual model and the design of the code.
- The conceptual model describes structural perspective, functional and behavioral capabilities.

- The conceptual model can be used for validation and verification activities.
- Conceptual modeling is iterative and repetitive, with the model being continually revised throughout the modeling study.

Figure 1 depicts different components of a conceptual model for simulations. A conceptual model for a simulation is the collection of information which consists of assumptions, algorithms, relationships and data. Mission space is concerned with representation and simulation space is concerned with simulation control.

Within the context of this thesis, we consider in particular conceptual models of the mission space (CMMS) used in the military simulation domain. US Defense Modeling and Simulation Office (DMSO) define CMMS as "simulation-implementation-independent functional descriptions of the real world processes, entities, and environment associated with a particular set of missions" (DMSO, 1997).

As our focus is the mission space, let us investigate mission space more closely. According to Pace (2000), the mission space includes simulation elements. A simulation element is the information describing concepts of entities as well as processes. Simulation elements may contain assumptions, algorithms, data, relationships which identify and describe that entities possible states, tasks, events, behavior, parameters, attributes etc. Furthermore, a simulation element may be a subsystem, an aircraft, a person, a group of people and also a process.

1.1.2 Conceptual Model Representation

As there are different usages and interpretation of a conceptual model, different modeling techniques have been used, for example UML, Object Modeling Technique (OMT) and Enhanced Entity Relationships (EER). The accuracy and precision of the conceptual model representation in these techniques depends. For example, The Unified Modeling Language (UML) and (OMT) provide a variety of concept development diagrams (i.e., use case, sequence, deployment, component, collaboration, class, state diagrams) to delineate different perspectives of a model.

During the last decade, among various modeling languages, UML (Unified Modeling Language) has become the de-facto standard for modeling software intensive systems. More recently, there has been a growing tendency to adopt UML for different modeling needs and domains. To respond to these needs, UML can be extended by means of either the Meta Object Facility (MOF, 2004) or the profiling mechanism. Allowing extension mechanisms with various representation capabilities and being a multi-purpose modeling language, UML seems to be the most promising modeling language for conceptual modeling as well.

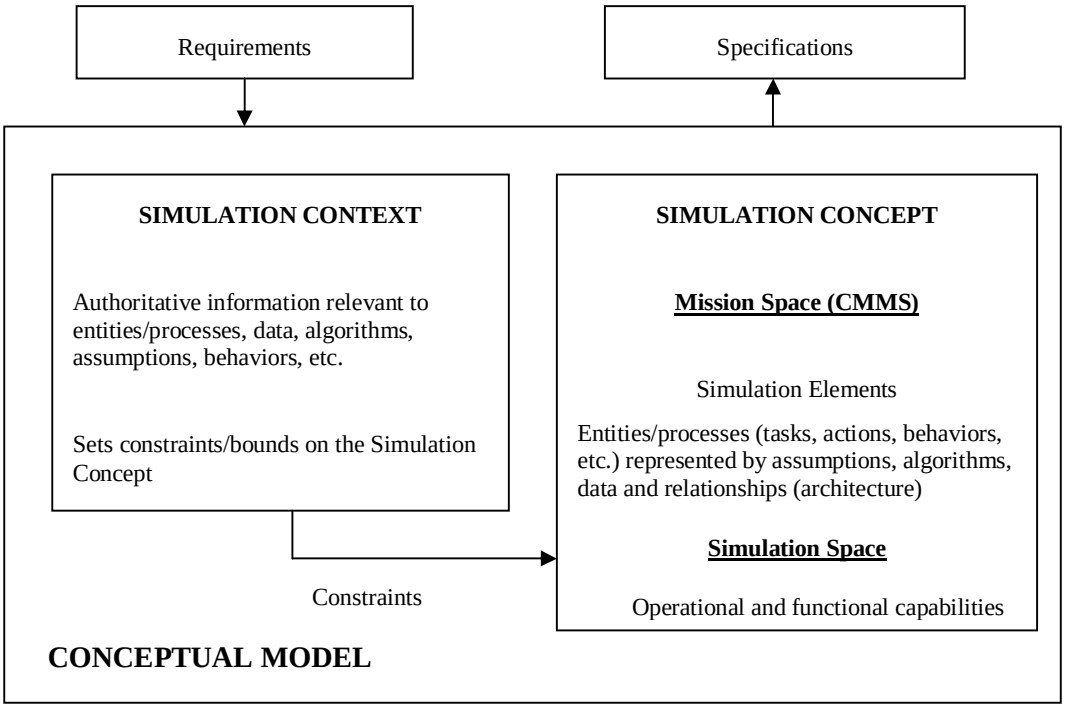


Figure 1: Components of a Conceptual Model for Simulations (DMSO, 1997)

However, the lack of a formal definition of a conceptual model makes it difficult to define a best set of UML views for representing it. To the best of our knowledge, in the military simulation domain, currently there are three approaches that support simulation conceptual modeling based on UML. The first one, "Syntactic Environment and Development and Exploitation Process" (SEDEP, 2007) is HLA (High Level Architecture) oriented. Two UML profiles have already been developed towards tool support (Lemmers & Jokipii, 2003). The second one, BOM (Base Object Model) (BOM, 2006) developed by SISO (Simulation Interoperability Standards Organization) is also HLA-oriented, hence platform specific. The third one is the KAMA notation (Karagöz & Demirörs, 2008) which is more CMMS (Conceptual Model for the Mission Space) oriented and platform independent.

In this study, we focus on a specific notation for conceptual model representation. This notation, KAMA (abbreviation of the Turkish words for Conceptual Modeling Tool) is based on the concepts of C4IRSMOS (Command, Control, Communications, Computer, Intelligence, Surveillance and Reconnaissance) domain (Karagöz & Demirörs, 2008). Seven types of diagrams are defined to represent both the structural and behavioral aspects of a conceptual model. As the conceptual modeling takes place during requirements analysis phase simple diagrams with only high level conceptual elements are defined to prevent the tendency to model the design issues. Hence KAMA models are simulation environment,

infrastructure and implementation independent.

In the definition of the KAMA notation UML's extension mechanisms were not used. Instead UML's meta-model and meta classes were reused when needs of the KAMA notation and associated constraints were not in conflict with UML meta model (Karagöz & Demirörs, 2008). Hence, KAMA reuses many parts of UML for defining itself, but do not comply with the UML meta model. However, KAMA is a MOF compliant notation.

On the other hand, KAMA notation has been developed taking into account various requirements presented in the literature, revised through experimental processes, shown to be fit for CMMS purposes and academically accepted (Karagöz & Demirörs, 2008).

1.1.3. Verification and Validation of Conceptual Model for Simulations

One striking observation one can make is that the published methodology or guidelines for simulation V&V community DMSO (2000a), FEDEP (2000), (ITOP, 2004) is that validation and accreditation of simulations is addressed in detail and extensively. However, the specific task of verification of conceptual models is not explicitly addressed in the V&V process models presented in these works.

This may be partly justified by the fact that during simulation software development projects, software quality assurance (SQA) is already performed for requirements, design and code verification. However, due to experimental intentions of simulations in general, in simulation software conceptual models are used to represent the real world to be simulated. Hence, in addition to information model represented by software system, a model of the simulation domain must be represented. Therefore, there exists a need for verification to assure that the conceptual models for simulations are represented as to respond the intended purpose of the simulation, in addition to verification of the model of software running the simulation. This distinction clears out why solely SQA is not enough.

On the other hand, as simulation projects may deal with wide range of experimental domain, experts (SME) with specific expertise of domain are used for validation of the conceptual models. Hence one may wonder; why is it necessary to conduct first a verification process? If after validation with SME, we reach to conclusion that the model is a sufficiently accurate representation of the domain of interest.

Firstly, it would be unwise to wait until the end of model development to find out that conceptual model does not address the requirements correctly and completely because of the represented models are incorrect and inconsistent themselves. So it would be wise to assure that models are at least internally correct and consistent before the validation process.

Secondly, verification employs techniques to identify illicit interpretations and undesired behavior. Hence verification identifies the issues validation may not and provides the foundation on which validation is based on. In this way during the validation SME can concentrate on the models fitness for the intended use and how well it represents the real world. This reasoning clears out why SME validation is not enough either.

1.2. The Problem

We see the research problem in two folds. These two aspects can be described as follows:

Firstly, since conceptual modeling is mostly related with the problem definition phase, any defect injected at this phase will cost too much effort and time sometimes even leading to unrecoverable situations. For this reason, (DMSO, 2001) and other urges the simulation conceptual model validation and verification efforts. Established international guides related to verification and validation (V&V) of simulations, such as, "DMSO – VV&A Recommended Practices Guide" (DMSO, 2000), "NATO - Verification, Validation, and Accreditation Federations" (NATO, 2007), "ITOP-International Test Operation Procedure for V&V" (ITOP, 2004) ITOP and "REVVA 2" (REVVA, 2005) may be used. Although, each one focuses on different aspects of V&V, simulation development needs and describe the V&V process; none of them provides any guidance or description specific to conceptual model verification.

Although validation of a conceptual model can be considered to provide enough credibility, a verification activity is needed to assure that conceptual models to be internally complete, consistent, coherent and correct before validation. Conceptual model verification should assure that the model does not include conflicting elements, entities, and processes. Redundant elements need to be avoided and all model components should be reachable to establish a coherent concept of the simulation. Hence, both structural and behavioral verification at the conceptual modeling stage are required to assure these requirements. In this way the validation activity will be dedicated to validation issues. Furthermore the conceptual model verification activity should especially identify semantic issues rather than purely syntactic defect.

Secondly, although many advantages exist such as discussed in (Karagöz & Demirörs, 2008), utilizing a notation derived from UML may lead to further problems for conceptual modeling because in the specification of UML (UML Superstructure, 2005) there is no systematic treatment of model correctness, consistency and completeness. Semantic properties are informally defined in the prose of the specification to give more flexibility and

expressive power to designs at different levels of abstraction, for different modeling methodologies or for different application domains. There is a lack of an agreed set of properties for quality UML models. Furthermore, when a derived notation is used, models may be required to conform to new properties. To eliminate unwanted interpretations, additional semantic properties should be conformed. These properties may stem from the modeled domain, the target domain (e.g., multiple inheritance is not allowed), and the modeling process domain (e.g., an activity diagram is required for each use case). What are the kinds of constraints/properties that assure sound interpretations? How to define constraints/properties for a domain specific notation? The study also examines these questions.

Although there exists many studies which are based on transformation of UML models to a formalism, generally, transformation approaches are partial. Furthermore, they suffer from the semantic correspondence, as shown in Section 2.3.3.2 complexity, scalability and most importantly the traceability problems. Besides, conceptual models are developed in sketchy manner early in the requirements elicitation phase, hence may be incomplete where formal techniques permits only predefined incompleteness especially when they are supported with tools. On the other hand, properties checked in UML inspection studies are related to syntax, static semantic or simple cross diagram dependency checks and the main artifact considered is not conceptual models but a software designs.

The problem addressed in this thesis can be stated as follows:

The effort of simulation model conceptualizations in domain specific notations such as KAMA is prone to incorrectness, incompleteness, and inconsistency and coherence problems. In addition to completeness and correctness of translation of the problem frame to conceptual representation (validation), ambiguity inherent in UML, its support of multiple views and the extension mechanism may further increase incompleteness, inconsistencies, incorrectness and redundancies in models (verification). Hence, error reduction techniques early in the simulation modeling lifecycle are needed.

Especially, semantic property checking is a major problem, as many syntactic errors can be eliminated through case tools. Conceptual models are in general not executable, therefore, it is not possible to verify and validate them using testing techniques, as one could do with a software system. Consequently we need to use alternative techniques. Techniques such as walkthroughs and inspections can be used rigorously for assuring conceptual model quality.

1.3. Our Approach

In order to tackle the problems defined in the previous section, this study presents a systematic, holistic and practical inspection process for verification of semantic properties to assure the quality of conceptual models in a domain specific notation. In our particular case the domain specific notation is KAMA.

In this study, an inspection approach is preferred to a formal approach due to various advantages: Firstly, informal techniques are easy to use and understand hence their application is straightforward. As checklists and guidelines are main tools in inspections, they can be performed without any training in mathematical software engineering. Inspections may be very effective if applied rigorously and with structure and they are relatively less costly and they can be used at any phase of the development process.

Furthermore, since conceptual models are used primarily as a means of communication, "Conceptual" implies human conceptualization, which inherently implies tractable abstraction levels and size. Hence, tool support is not crucial, but rather the verification results may also be used as a means to identify and resolve validation issues. It is more cost effective to integrate the verification tasks with the validation tasks which require human (in simulation domain subject matter expert interpretation) interpretation hence mostly a human activity.

The development of the inspection approach consisted of 3 major phases, which are, the property identification phase, process definition and conduct of inspections within case studies.

In the first phase of our study, we performed a literature review and investigated existing verification methods for UML design models as UML conceptual model verification is not addressed separately in the literature. Then we tried to identify the applicable properties to conceptual models in UML like notations. Based on this experience, we proposed a high level framework for identifying desirable properties of conceptual models in domain specific notations. This framework considers four categories of desirable properties: Syntactic, semantic, horizontal and vertical. Using this framework, properties proposed in the UML verification literature and meta model definition, we were able to identify desirable semantic properties for the KAMA notation.

We have also examined possible techniques that could be used for verifying UML conceptual models, which are explained in Sections 2.3 and 2.4. Based on our explorations for checking semantic properties, we have defined a practical inspection process composed

of simple verification tasks. We developed tasks for especially semantic properties as many syntactic errors can be eliminated through case tools. For checking semantic properties of structural views, we have developed deficiency patterns. By using a holistic approach, rather than a partial approach, we developed tasks for different type of diagrams and inter-diagram properties as well. The properties and related verification tasks are specified in natural language so that they could easily be understood and used by the inspectors and the domain experts.

In contrast to existing approaches, we develop an inspection process by integrating concepts derived from formal approaches. We identify semantic properties depending on the DSN and develop the inspection process for the identified properties. Tasks needed to be addressed to find semantic defects are precisely defined, thus we fulfill loosely defined steps in inspections with concrete content.

In order to identify improvement opportunities and observe the applicability of the approach, a multiple-case study involving two case studies was conducted. The first study has been performed as an exploratory case study. The second case study aimed to investigate the applicability and effectiveness of the inspection approach.

1.4. Organization of the Thesis

The thesis consists of five chapters and four appendices[1].

Chapter 1 introduces the context, defines the research problem, and outlines the solution approach.

Chapter 2 reviews related research on verification an validation of conceptual models for simulations, different kinds of desirable properties of UML-like notations, formal and informal approaches, and existing tools for verification of conceptual models.

Chapter 3 describes the proposed inspection approach in detail.

Chapter 4 is devoted to the research carried out for evaluating the proposed inspection approach. The research strategy, case study design, case study plans and findings of two case

[1] *The terms "domain specific notation" and "notation derived from UML" are interchangeably used throughout the theses. Both terms refer to a domain specific notation which substantially reuses UML or MOF in its definition. Also, the terms "conceptual model for the mission space" and "conceptual model" are interchangeably used after page 15 on. "Conceptual model for the mission space"is meant by both terms.*

studies are presented. The details of the inspections, their results and discussions are provided for each case study.

Chapter 5 presents the conclusions of the study and summarizes the contribution of this research. Possibilities for further investigation are also provided in this chapter.

After the bibliography, the four appendices which consist of (A) an overview of first order logic (FOL) as a formalism for UML class diagrams, (B) Petri Nets as a formalism for UML activity diagrams, (C) a detailed presentation of KAMA modeling notation and (D) the report on case study 2, respectively, are presented.

CHAPTER 2

RELATED RESEARCH

This chapter summarizes the literature and presents important concepts related to conceptual model verification. A multitude of different approaches have been proposed in the literature. These approaches will be presented in two main parts with emphasis on formal techniques. In general, the formal and informal approaches complement one another toward addressing V&V challenges.

In the first part of this chapter, the conceptual models and V&V will be described within the context of simulation system modeling. An assessment of applicability of various V&V techniques to KAMA notation will be presented.

In the second part, the related literature for identification of desirable properties of UML models will be presented.

In the third section of this chapter, two formal approaches; first order logic for structural views and Petri nets for behavioral views will be investigated in the search of a practical verification method for conceptual modeling in UML. The research work related to inspections will be shortly reviewed. Then a short assessment of formal verification work for UML will be presented.

In the last section, some of the tools described in the literature will be shortly reviewed. In the last part literature and basic concepts related to inspections will be given. The chapter ends with a brief critique of the existing work.

2.1. V&V of Conceptual Models for Simulations

Boehm (1984) describes software validation as a set of activities designed to guarantee that the right product is being built (from a user's perspective) and verification as activities that guarantee the product is being built correctly according to requirements specifications, design documentation, and process standards. Boehm's well-known maxim puts this as: "validation is building the right system whereas verification is building the system right."

In particular, verification is mostly static examination of the intermediate artifacts such as requirements, design, code, and test plans. Verification can be examined under two main titles:

Formal Verification: The application of mathematical techniques and mathematical argument whose purpose is to demonstrate the consistency between a program and its specification.

Informal Verification: The process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase.

From the point of view of military simulation, DMSO (2000) defines conceptual model validation and verification as "determining that the theories and assumptions underlying the conceptual model are correct and the representation of the validated requirements is reasonable and at the correct level of abstraction". DMSO adds two more requirements: 1) The conceptual model's structure, logic, mathematical and causal relations, and the processes need to be reasonably valid. 2) The conceptual model is also required to be internally complete, consistent and correct. Besides, in a seminal paper, Lindland et al. (1994) discuss a variety of conceptual model evaluation quality criteria such as completeness, consistency, modularity, implementation independence, maintainability, and generality.

To respond to these requirements a sound verification process should be conducted. We believe that such a process for simulation conceptual model verification should adhere to some agreed managerial level recommended practices. For identification of practices, we have investigated a set of important international standards.

Among these, for instance NATO (2007) focuses specifically on VV&A for federations which are developed according to the FEDEP (2000) (Federation Development and Execution Process). It considers VV&A activities are specified as an "overlay" process to FEDEP. On the other hand, the ITOP (2004) approach aims at supporting the capture, reuse and exchange of V&V information and provides structure for documenting V&V

information. Finally, The REVVA II (2005) methodology is intended to provide a generic VV&A framework. In spite of having different focuses, these standards share common concepts, but since standards are defined for V&VA altogether they do not explicitly differentiate verification. Furthermore the V&VA activity is focused on the executable simulations and not specifically on their underlying conceptual models.

According to all the mentioned standards, at the first stages of M&S development, based on the intended purpose of conceptual modeling, a detailed set of "Acceptability Criteria" (AC) be developed in such a manner that passing the AC implies fitness for the intended purpose. In principle, one may think that AC concept is equivalent to the functional and non functional requirements of the software requirements but in fact formulated directly based on the domain. As it can be seen in Figure 2, a vague intended purpose must have been already formulated, which is refined into a set of sub-purposes, which again must be decomposed to lowest AC from which, Verification Objective (VO) related to the simulation can be derived. Developing the V&V objective usually includes the decomposition into more easily assessable V&V tasks a shown in the Figure 2. Some of these tasks are related to verification some are to validation. In this hierarchy, our inspection approach is mostly related with verification tasks and objectives as delineated in the shaded area.

When a modeling formalism such as UML or KAMA is used, the AC and VO formulation for the conceptual models should also take into account the vast set of representational and abstraction capabilities of the modeling formalism. For example, if the purpose of the conceptual modeling is just to provide a generic repository for reuse than the set of criteria

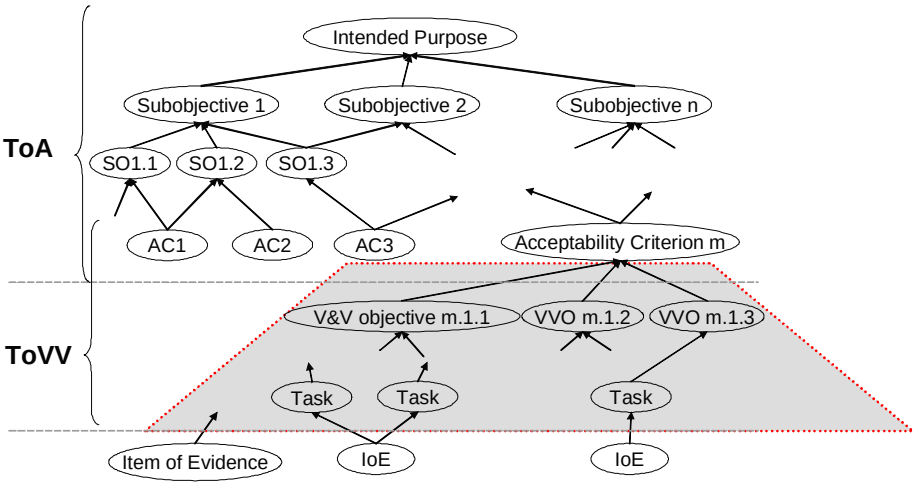


Figure 2 : Hierarchy of AC and V&V from (REVVA 2, 2005)

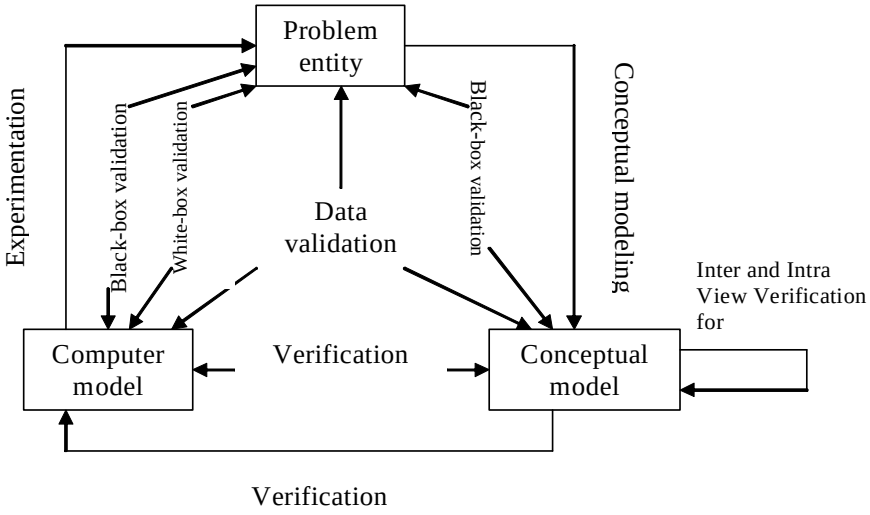


Figure 3: Conceptual Model Validation and Verification (DMSO, 2000)

will not focus to the executable models but rather to understandability, easy adaptation etc. On the other hand if conceptual model is to be used straight in FEDEP, runtime criteria should be defined also.

An overview of generally accepted simulation modeling, verification and validation process is shown in the Figure 3. The framework for simulation evaluation is formed by problem entity, conceptual model and computer model. The arrows represent the various technical processes that must be conducted to show a model is reliable.

There are some generally accepted principles for simulation V&V (Balci, 1998). The following principles also are taken into account in our CM inspection approach in this study.

1) V&V must be conducted at each phase of modeling,
2) The outcome of V&V should not be considered correct or incorrect,
3) Objectives of the phase should be taken into account,
4) V&V must be conducted by personnel other than the developer,
5) Exhaustive model testing is not possible,
6) V&V must be planned and documented,
7) Errors should be detected as early as possible,
8) Multiple views and interpretations of model must be identified and resolved properly,

9) Testing of each sub model does not guarantee integral model quality,

10) Simulation conceptual model validity does not guarantee the simulation results.

On the other hand, within the modeling and simulation literature, a variety of specific techniques for V&V have been suggested by authors such as Law & Kelton (1999). And Sargent (1994). Balci (1998) offers a collection of 77 verification, validation and testing techniques. These techniques, however, vary extensively - e.g., alpha testing, induction, cause and effect graphing, inference, predicate calculus, proof of correctness, and user interface testing. For example, in terms of verification, these techniques can be categorized as informal, static, dynamic, symbolic, constraint and formal (Balci, 1998). Appropriate techniques can be selected for particular projects, however many of the techniques are overlapping in coverage.

A pre-assessment of these techniques has been done for their applicability to KAMA notation. Applicable techniques are identified by considering properties such as relevancy, applicability, appropriate complexity and perceived risk. The result of this assessment can be seen in Figure 4. However this assessment only provides very high level information that has not been further detailed.

2.2. Desirable Properties for UML Models

In order to talk about verification of something, first rules or desirable properties should be defined. There is a lack of agreement and the kind of desired properties for quality UML models vary highly in the literature. Probably least researched area is about the definition of a set of properties or rules against which models should be checked. In the following paragraphs, we present different types of properties with examples from the literature.

Like any other language, UML has its syntax and semantics specified (UML Superstructure, 2005). The syntactic correctness rules or well-formedness rules of a UML model is specified either in the abstract syntax through meta-models or OCL constraints. For example, the properties such as “every class should have a unique name” or “an initial node in an activity diagram has at most one outgoing flow” are desired syntactic properties for a UML model.

Informal		Static		Dynamic		Formal	
Audit	1,2,5	Cause-Effect Graphing		Acceptence		Induction	
Face Validation	1,2,3,4,5,6,7	Control analysis	3	Testing		Inference	1,6
Inspections	1,2,3,4,5,6,7	Calling Structure	4	Alpha Testing	4,7	Logical Deduction	1,2,5,6
Reviews	1,2,3,4,5,6,7	Concurrent Process	3,4	Debugging	4,7	Inductive Assertions	
Turing Test		Control flow	4	Execution Testing	4,7	Lambda Calculus	
Walkthroughs	1,2,3,4,5,6,7	State Transition	4,7	-Monitoring	4,7	Predicate Calculus	
		Data Analysis		-Profiling	4,7	Predicate-Transformation	1,3,4,6
		Data Dependency		-Tracing	4		
		Data Flow		-White-Box	4	Proof of Correctness	1,3,6
		Fault/Failure Analysis		Testing	4	Petri Nets	4,7
		Semantic Analysis	1,3,5	-Branch	4		
		Interface Analysis		-Condition	4		
		Structural Analysis	1,2,3,4,5,6	- Data Flow	4		
		Symbolic Evaluation	1,6	-Loop	1,2,3		
		Syntax Analysis	1,3,4,6	-Path	,4,5,6		
		Traceability Assessment	1,3,6	Visualization/ Animation	1,2,3,4 ,5,6		

	KAMA VIEWS
No	VIEW TYPE
1	Entity- Ontology
2	Command Hierarchy
3	Mission Space
4	Task Flow
5	Organization Structure
6	Entity-Relations
7	Entity-State

Figure 4: An Assessment of V&V Techniques for Applicability to KAMA Conceptual Models

On the other hand some of the semantics for UML elements are described informally in natural language in the specification, however the specification is huge and there is not a systematic treatment of semantic properties. A simple example of semantic property is "All generalization hierarchies must be acyclic". Different approaches exist for detecting properties and will be discussed in the next sections.

We can distinguish between static properties and dynamic properties of a UML model (Sourrouille & Caplat, 2003). The static semantics of UML is formally described in terms of its meta-model and OCL constraints. A static inspection of a model can reveal static properties; however dynamic properties such as absence of deadlocks and livelocks cannot be completely verified until runtime.

The most interesting properties of the dynamic type in the literature (Berardi et al., 2005) are related with the semantics of class diagrams. The following are some examples of properties of this type:

- *Consistency of the class diagram.* A class diagram is *consistent*, if its classes can be populated without violating any of the constraints in the diagram.
- *Class Consistency.* A *class* is *consistent*, if it can be populated without violating any of the constraints in the class diagram.
- *Class and Relation Equivalence.* Two classes are *equivalent or redundent* if they denote the same set of instances whenever the constraints imposed by the class diagram are satisfied.
- *Class Subsumption:* A class C_1 is *subsumed by* a class C_2 if, whenever the constraints imposed by the class diagram are satisfied, the extension of C_1 is a subset of the extension of C_2. A generalization hierarchy can be used to reduce the complexity of the class diagram..

Similar properties are used by Queralt & Teniente (2006) however they add OCL constraints in class diagrams. In this work the following properties are defined and checked:

- *Satisfiability:* A class schema is *satisfiable* if there is a non-empty state of the information base in which all its integrity constraints are satisfied.
- *Liveliness of a Class or Association:* Even if a class schema is *satisfiable*, it may turn out that some class or association is empty in every valid state. Liveliness of

classes or associations determines if a certain class or association can have at least one instance.

- *Redundancy of a Constraint:* A constraint is *redundant* if integrity does not depend on it, that is, if the states it does not allow are already not allowed by the rest of constraints.

Semantic properties applicable for specific kinds of diagrams or a subset of diagram types exist. Csertan et al. (2002) for example, verify general properties defined for state diagrams. In this study, properties defined by Levenson (1995) are used. Among the defined properties are: “All variables must be initialized” and “All states must be reachable”.

On the other hand, in Zhao et al. (2004) for example deadlock, liveness, boundedness and reachability properties are verified by transforming UML model into Petri Nets. Other PN properties defined by Murata (1989) can be used for activity diagrams for example.

Engels et al. (2001) mention horizontal and vertical UML consistency properties. They also acknowledge that horizontal consistency properties are desired and may be a means to reduce contradictions that might exist due to the overlapping information residing in different views of the same model. An example of a property related to horizontal consistency is: "Each class with states must be represented with a state-chart diagram".

They also claim that, vertical consistency properties are used to reduce inconsistencies or contradictions that exist when applying UML to the different abstraction levels. An example for this type of property is: "The set of states of an object defined by a father class must be a subset of the set of states of an object of the child class”. Usually, horizontal consistency properties are explicitly modeled in the UML views and the meta model, whereas vertical consistency properties are implicit or expressed informally. Some research studies (Kurzniarz et. al, 2002; Kurzniarz et. al 2003; Straten et. al. 2005) have tried to formally define these kinds of properties.

There are also properties needed due to the use of distributed UML diagrams for a given part of the model. Tun & Bielkowicz (2003) claim that UML views (diagram types) are fragmented and the connection between them is implicit. For example a class view may be distributed among various diagrams and it is not easy to trace the connection between these diagrams. A model tree can be used for automatic checking or one integrated view should be constructed before conducting any kind of property checking.

In addition to these types of problems, some of the UML semantics is deliberately underspecified by OMG to allow adaptation for various needs. Different interpretations of UML models are possible as UML can be extended for domain specific modeling. As an example, there is no agreement on the proper way to inherit attributes with the same name (as attributes having the same name is allowed in different classes of the same static model) in multiple-inheritance. Relaxed interpretation inherent in UML complicates property checking, which usually requires language to be completely and precisely specified. When defining a new notation, UML's semantic aspects are constrained for domain needs, mostly using natural language.

On the other hand, Berenbach (2004) describes techniques for analyzing large UML models and proposes a set of heuristics for creating verifiable analysis and design models. Heuristics and processes for creating semantically correct models are presented for analysis and design phases. Some examples are:

- There will be at least one message on a defining sequence diagram with the same name as each included use case since a set of sequences diagrams are represented by a use case.
- Use an activity diagram to show all possible scenarios associated with a use case.
- Package dependencies should be based on content (model organization): a dependency between two packages should exist if and only if there is a dependency between artifacts belonging to these packages.

Ambler (2005) lists a collection of conventions and guidelines for creating effective UML diagrams and defines a set of rules for developing high-quality UML diagrams. In total 308 guidelines are given with descriptions and reasoning behind each of them. It is argued that, applying these guidelines will result in an increased model quality. However, inter-view properties are not considered at all. Some examples of properties are:

- Model a dependency when the relationship is transitory in a structural diagram.
- Role names should be indicated on recursive relationships.
- Each edge leaving a decision node should have a guard.

On the other hand, SD Metrics (2007) tool checks adherence to UML design rules. These rules span from well formedness rules of UML to object oriented heuristics collected from literature. Most of the rules are simple syntactic rules. Some examples are:

- The class is not used anywhere. (completeness)
- Use of multiple inheritance - class has more than one parent. (style)
- The control flow has no source or target node, or both. (correctness)

A perspective based reading method for UML design inspection, so called object oriented reading techniques, has been presented by Travassos et al., (2002). Examples of properties provided are:

- There must be an association on the class diagram between the two classes between which the message is sent. If not, an association is present in the sequence diagram, because of the message exchange, but not present in the class diagram.
- For the classes specified in the sequence diagram, the behaviors and attributes specified for them on the class diagram should make sense.

In Unhelkar (2005), quality properties within and among each diagram type have been described along with checklists for UML quality assurance. Although conceptual modeling (CIM - Computation Independent Model) is considered separately and verification and validation checklists in different categories such as aesthetics, syntax and semantic are provided, most of the checklist items are related to validation and completeness. Items related to verification are mostly syntax or simple cross diagram dependency checks. Some of the examples of the properties looked for in the models are:

- The notation for fork and join nodes should be correctly used to represent multithreads.
- All use cases should be numbered.
- The aggregations should represent a genuine “has” a relationship.
- Activities that are supposed to be parallel should be represented so.

The literature review shows that many of the previous works consider only certain types of properties and there is a lack of agreement on a set of desirable properties for quality UML models. Furthermore, the questions such as what are the types of desirable properties and “how can we identify these for domain specific notations” is not addressed.

2.3. Formal Techniques for UML Model Verification

Having reviewed the kind of properties that are desirable, we review in this section the formal approaches that can be used to identify these properties in UML conceptual models.

In the UML literature, property checking is related to work on the formalization and the verification of UML diagrams. This is due to the fact that problems in UML models are explained by a lack of formal semantics for the language.

Hence, the first category of work on property checking of UML models focuses on formalization of UML language itself (Kurniarz et al., 2002; Ober, 2004), (Kim and Carrington, 2000). However, UML includes many modeling notations and a formalization of all concepts is a huge if not controversial undertaking. Hence, most research in this line has aimed at formalization of a single diagram or the relationship of two diagrams (Kurniarz et al., 2002; Ober, 2004).

On the other hand, there are many studies that rely on the transformation of UML models to a formal language (Amalio & Polack, 2003). In transformation approaches, the formal language to be chosen and restricting the formal language to a subclass can ensure decidability in exchange of completeness. In general, transformation approaches suffer from the complexity, scalability and most importantly the traceability problem: to what extent can a model and the identified problem be traced back to the original UML model.

The research work on verification of UML models either emphasizes the structural perspective or behavioral perspective. The following section will summarize the research based on these two perspectives.

2.3.1 Approaches with Structural View Emphasis

In this section, first we will briefly review the related work mostly focusing on the structural perspective of UML models. Then, by using the formal definitions (Berardi , Calvanese & De Giacomo, 2005) through an example class model of missile control, we show that class diagram inconsistency type of deficiency can occur even in well formed class diagrams. The concepts presented in this section provided the inspiration of the structural deficiency patterns formulated in Chapter 3.

2.3.1.1 Related Work

There are many studies that concentrate on the development of rule-based systems where rules can be defined in a declarative language (Berenbach, 2004; Egyed, 2006; Sourrouille & Caplat, 2003). This is rooted to absence of an agreement on a common set of properties of

UML-based models. However, underlying rule based mechanisms of these approaches can not implement dynamic property checking which can only be verified by formal methods.

In an interesting paper which consider domain specific modeling along with constraint checking (Sourrouille & Caplat 2003), claims enhancements to the OCL is needed. It is also important to note that OCL can be used at the model level, to describe semantic model constraints, as well as to constrain the UML meta-model for domain specific modeling. They mention three reasons to do so: First, some constraints cannot be expressed in OCL since OCL is a query language and any declaration or change on the models can not be performed. This also leads to the second limitation of OCL which is its inability to express consistency restoration actions, which could be useful to implement automatic corrections. Finally, a claim against the current syntax of OCL is that it is hard to use especially for novice users without a modeling experience.

In another approach, inference engine *Sherlock* (Caplat, 2006) linked to a UML case tool is used. In this work, models are built using the UML modeling tool and adorned with tags and constraints. As a lighter alternative, they have chosen to describe the UML meta-model instead of first describing MOF directly using the inference engine language. The definition is based on notions such as concept, attribute, relationship, instance, task, and rule, which make it possible to implement quickly a basic meta-model. Next, models are expressed in terms of this meta-model. UML meta-model notions and generic rules are added (e.g., the value of the tag *constraintDefault* should belong to the set {*inconsistency*, *illegality, incompleteness,* etc}). Finally, the UML model is loaded and checked.

Dupey (2000) has proposed to generate Z formal specification with proof obligations from UML diagrams. This is done automatically with the RoZ tool. UML notations and formal annotations reside together: the class diagram provides the structure of Z formal skeleton while details are expressed in forms attached to the diagram. Either OCL or Z-Eves constraints are used. Then the Z-Eves theorem prover is used to validate a given diagram.

Marcano & Levy (2002) describe an approach for analysis and verification of UML/OCL models using B formal specifications. In the latter work, a systematic translation of UML class diagrams and OCL constraints of a system into a B formal specification is given. In this one, they propose to manipulate in parallel an UML/OCL model and its associated B formal specification. At first B specification is derived from the UML class diagrams. Then, the OCL constraints of the model are automatically translated into B expressions. Two types of constraints are taken into account: invariants specifying the static properties, and pre/post-conditions of operations specifying the dynamic properties. The objective is to enable the use

of automated proof tools available for B specifications in order to analyze and verify the UML/OCL model of a system.

Killand & Borretzen (2001) have proposed an ontology based method to improve design quality by ensuring consistency among the multiple design views. The "Methodology for Ontology-Based Detection of Errors in Software Design" (MODED) integrates multiple software design views into one common model to identify errors among those views. They apply "MODED" to detect inconsistencies in a software design, specified utilizing the UML, of the London ambulance dispatch system.

Andre, Romanczuk & Vasconcelos (2000) have presented a translation of UML class diagram into algebraic specification in order to check consistency. This approach aims at discovering particular kind of multiplicities inconsistency in a class diagram. The approach deals with all interesting concepts of UML class diagram: class, attribute, association, generalization, association constraints and heritance. The Larch prover discovers some of inconsistencies automatically; others require the intervention of the user.

2.3.1.2 A Formalism for UML Class Diagram: First order logic

According to Mota et al. (2004), First Order Logic (FOL) is quite suitable for representing UML class diagrams for consistency verification purposes because of the following reasons:

1) FOL is computationally universal as any problem with a computation situation can be described in it. Moreover, such descriptions can be reduced to Horn clauses for logic programs.

2) There are many efficient inference engines for handling first order expressions or logic programs.

3) FOL can be used to integrate other logical representation language and so we may perform a synthesis across UML models and formal verification notion.

Another very nice advantage of FOL is that recently XMI (meta data interchange XML) is being used for model exchange between enterprises. Any valid XML description may be associated to DOM (Domain object model). As DOM descriptions are easily mapped into first order expressions, all modern UML-based case tools which export XMI can be used for this purpose.

For the above mentioned reasons, it is a good idea to represent a UML class diagram in FOL. The example class diagram in Figure 5 is represented in FOL assertions in Figure 6. The formal definitions used for this translation can be found in Appendix A.

Figure 5 illustrates a syntactically correct (well-formed) inconsistent class diagram. Intuitively, "mobile launcher" and "fixed launcher" classes are disjoint i.e. they can not have common instances as imposed by the generalization relation. But formally, it is easy to show that by 12 and 13 this knowledge base becomes inconsistent.

It is in general possible to translate FOL statements to an input language of an inference engine such as Prolog to check incrementally the consistency conditions.

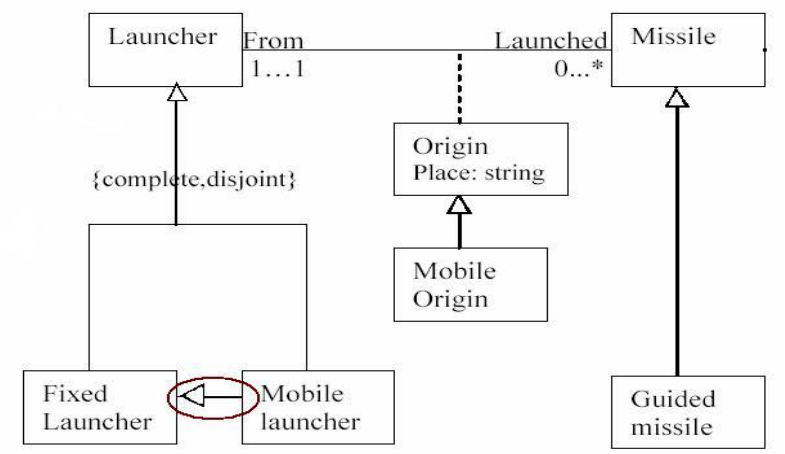


Figure 5: Semantically Incorrect UML Class Diagram Example

1) $\forall$ x,y. Launched(x,y) $\wedge$ Origin(x) $\rightarrow$ Missile (y)
2) $\forall$ x,y. From(x,y) $\wedge$ Origin(x) $\rightarrow$ Launcher (y)
3) $\forall$ x,y. Origin(x) $\wedge$ Place(x,y) $\rightarrow$ String(y)
4) $\forall$ x. Origin(x) $\rightarrow$ $\exists$ y. Launched (x,y)
5) $\forall$ x. Origin(x) $\rightarrow \exists$ y. From (x,y)
6) $\forall$ y. Missile(y) $\rightarrow$ ($1 \leq \#$ {x | Origin(x) $\wedge$ Launched(x,y)} ≤ 1)
7) $\forall$ x. Mobile Origin(x) $\rightarrow$ $\exists$ y. Launched(x,y)
8) $\forall$ x. Mobile Origin(x) $\rightarrow$ $\exists$ y. From(x,y)
9) $\forall$ x. Guided Missile (x) $\rightarrow$ Missile(x)
10) $\forall$ x. Mobile Origin(x) $\rightarrow$ Origin(x)
11) $\forall$ x. Mobile Launcher(x) $\rightarrow$ Launcher(x)
12) $\forall$ x. Mobile Launcher(x) $\rightarrow$ ┐Launcher(x)
13) $\forall$ x. Fixed Launcher(x) $\rightarrow$ Launcher(x)
14) $\forall$ x. Mobile Launcher(x) $\rightarrow$ Fixed Launcher(x)
15) $\forall$ x. Launcher(x) $\rightarrow$ Mobile Launcher(x) $\vee$ Fixed Launcher(x)

Figure 6: FOL Representation of UML Class Diagram shown in Figure 5.

However, in general the decision problem of validity in FOL is undecidable. In order to overcome the decidability problem of FOL, a fragment of FOL, called Description Logics is used for representing concepts and relationships. As an example, (Berardi , Calvanese & De Giacomo, 2005) and (Van Der Straeten, Mens, Simmonds & Jonkers, 2003) rely on the transformation into description logics. As opposed to first order logic, where satisfiability is known to be undecidable, subsets of description logics, which can be used for semantic consistency of only restricted subset of class diagrams, have decidable inference mechanisms. Even though (Berardi , Calvanese & De Giacomo, 2005) proved that class consistency in UML class diagrams for example is exptime-hard.

By exploiting the services of description logic inference engines for example (ICOM, 2000) various kinds of checks for properties can be performed. Among these are the properties already given in Section 2.2. For example, the *consistency of a class diagram* can be checked by checking the satisfiability of the corresponding description logic knowledge base. On the other hand, class consistency can be checked by checking satisfiability of the corresponding concept in the knowledge base representing the class diagram. Furthermore, *checking class equivalence* and subsumption amounts to check the equivalence of the corresponding concepts and subsumption.

2.3.2 Approaches with Behavioral View Emphasis

This category of work focuses on property checking in mostly behavioral diagrams such as activity, state-chart and sequence diagrams. For verification of behavioral properties, first suitable formal verification formalism (e.g. a Petri Net) has to be chosen capable of verifying the aspects associated to the property. For example, for the property of deadlock freedom, the formalism has to support the aspects of concurrency, communication and interaction of processes. A UML model must first be translated into such a specification language.

In the first part of this subsection, we review some of the related research under this category. After about a brief review of the relevant literature, we shall give some definitions related to Petri nets and workflow nets which are also used in developing the inspection tasks and referred in our inspection process in Chapter 3.

2.3.2.1 Related Approaches

There are various types of formalisms used in different researches that deal with verification of behavioral properties in activity and state-chart diagrams:

Eshuis & Wieringua, (2004) describe a tool for verification of workflow models specified in UML activity diagrams. The tool translates an activity diagram into an input format for a model checker. The tools is based on a formal semantics, techniques are used to reduce an infinite state space to a finite. With the model checker, any propositional property can be checked against the input model. If a requirement fails to hold, an error trace is returned by the model checker. They illustrate the whole approach with a few example verifications.

In Chang et al. (2005) for example, UML models are translated to PNs for analyzing the behavioral aspects. The goal is to use the configurable graphic interface and the mathematical analysis methods of the PN to verify the logic correctness of the flow control mechanism and then apply the standard modeling and implementation capabilities of UML, to transform the control specifications into desired computer codes with specified logic and configuration. The PN models were analyzed by efficient algorithms that solve recursively corresponding integer programming problems to discover structural errors in the models.

Considering the low usage of formal methods and the increasing acceptance of the UML in industry, (Apvrille et al. , 2004.), proposed a real-time UML profile named TURTLE, an acronym for Timed UML, and RT-Lotos Environment. Core characteristics of TURTLE are supported by a toolkit which includes a diagram editor, a RT-Lotos code generator and a result analyzer. The toolkit reuses validation tool offering debug-oriented simulation and exhaustive analysis. TTool hides RT-Lotos to the end-user and allows to directly check TURTLE modeling against logical errors and timing inconsistencies. TURTLE is compliant with UML 1.5.

There are criticisms directed to the transformation approaches: (Csertan, et. al., 2002) argues that ad-hoc transformation rules lack the necessary preciseness. Also, there is a need for a high-degree of flexibility due to the changing and extensible UML standard. They claim that their graph transformation framework (called VIATRA: Visual Automated model Transformations) for UML-based system verification has the following advantages, although they provide no empirical evidence:

- Both the UML dialect to be used by the modeler and the input notation of the target mathematical analysis tool are defined by their respective *meta-models.* This offers flexibility.
- Transformations can be defined in the form of a set of simple transformation rules correlating individual UML notational elements with the target mathematical notation. These transformation rules themselves can be designed visually in UML.

- The transformers are automatically derived from the rules by using the mathematically well-defined and widely used principle of graph transformations.

Apart from approaches using formal proof environments, algorithmic approaches exist. (Litvak, 2003) describes an algorithmic approach to a check consistency between UML Sequence and State diagrams. The algorithm also handles complex state diagrams, e.g. diagrams that include forks, joins, and concurrent composite states. They have implemented BVUML, a tool that assists in automating the validation process. BVUML implements the consistency check algorithm. Hybrid sequence and state diagrams are introduced to visualize the process; in these diagrams states are associated with the sequence diagrams.

As another alternative, (Damm & Harel, 2001) proposed 'Live Sequence Charts' (Damn & Harel, 2001) and look for inconsistency among state and sequence diagrams by means of first order logic. User selectively designate parts of a chart, or even the whole chart itself, as universal (live, or mandatory), thus specifying that messages have to be sent, conditions must become true, etc. Main extension deals with specifying "*liveness*", i.e. things that must occur. The designer may incrementally add liveness annotations as knowledge about the system evolves. As Live Sequence Charts allow the distinction between possible and necessary behavior both globally and locally, this makes it possible to specify forbidden scenarios, for example, and enables naturally specified structuring constructs such as sub charts, branching and iteration. In this way undesired behavior is assured not to occur.

Finally, recently, Gagnon et al. (2008) presented a framework supporting formal verification of concurrent UML models using the Maude language. The interesting part of this research is that they consider both static and dynamic features of concurrent object-oriented systems, because majority of research of this category are based on a single perspective. Specifically they focus on UML class, state and communication diagrams. The formal and object-oriented language Maude, based on rewriting logic, supports formal specification and programming of concurrent systems, as well as model checking. The major motivations of their work are: (1) translating concurrent UML diagrams into a Maude formal specification and (2) applying model checking to the generated specifications. While being a promising technique, this method also suffers from the problem of limited scope of applicability as discussed below in Section 2.3.3.

2.3.2.2. A Formalism for UML Activity Diagrams: Petri Nets

Activity diagrams are usually used for defining the flow of higher level events corresponding higher level of abstraction in the design process. An activity diagrams is used to describe

activities that are either within an object or between objects. In either case the main information represented is the control flow and concurrency.

In order to detect design errors and modeling issues of the behavioral system specification, it seems to be a good idea to convert activity diagrams to Petri Nets (PN) to verify correctness. Rather then designing a complete translation of the UML model it is convenient to restrict the translation to those aspects that contribute to the properties of interest. When verifying a property, that has to do with communication and interaction of activities or components, such as deadlock freedom, sub models of the UML model that have to do with the structure aspect such as class diagrams need not be formalized.

For example, Boccalette et al. (1999) have developed a set of rules to transform simple UML activity diagrams to Petri Nets. We are able to transform a given activity model to a Petri Net by applying these rules. An example of an activity diagram transformed to a Petri Net will be shown. Petri nets can be used to analyze behavior of activity diagrams. The formal definitions of Petri Nets, rules of translation of activity diagrams to Petri Nets can be found in Appendix B.

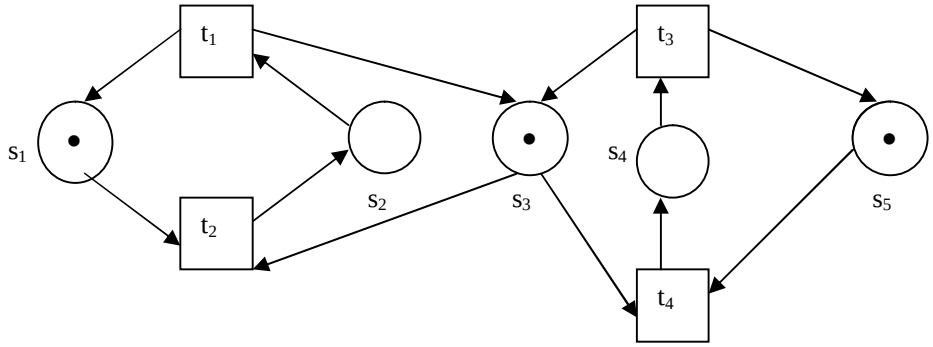


Figure 7: A Petri Net Example

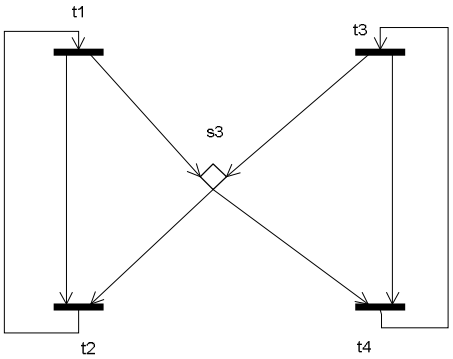


Figure 8: A Corresponding Activity net to the Petri Net in Figure 7.

In Figure 7, we present an example Petri net and its corresponding activity net in Figure 8. The transformation is achieved by means of applying the rules of presented in the previous section. Remark that places s1, s2, s4, s5 and all the tokens are lost in this transformation. This problem is known in the literature as the semantic correspondence problem. Not all the modeling constructs find their counterpart in the target domain. This is one of the drawbacks of approaches which transform the models to a formal notation and the result of the analysis back to the original notation.

2.3.2.3 Workflow Nets and Activity Diagrams

Nevertheless, Petri nets have proven to be a powerful system modeling and analysis means for computer hardware, software systems, manufacturing process and control systems, knowledge management systems, information processing systems, and communication networks (Chang et al., 2005). We can see activity diagrams and Petri nets have reciprocal features. The UML activity diagrams are user friendly while PNs require formal strictness. The UML describe systems effectively while PNs analyze systems strictly and formally. Lastly, PN models can be simulated.

A variant of Petri Net so called "workflow net" is very similar to UML activity diagrams. A workflow net is defined as a Petri net which models a workflow process definition in (Aalst 2002). With the translation schema in Appendix B, it is rather easy to translate an activity diagram into a workflow net also. A workflow net satisfies two requirements. First of all, a workflow net has one input place (*i*) and one output place (*o*). Secondly, in a workflow net there are no dangling tasks and/or conditions. Therefore, every transition (place) should be located on a path from place *i* to place *o*. Note that the definition of a workflow net is a syntactical definition; the requirements can be verified statically because they only relate to the structure of the Petri net.

On the other hand, *soundness property* relates to the dynamics of the workflow process definition. A workflow net is sound if the following properties are satisfied:

1) It is possible to terminate, i.e., it is possible to reach a state with at least one token in the output place *o*.
2) The moment a token appears in *o*, there are no tokens left behind in the workflow net. This means that there will be no dangling references.
3) There are no dead tasks, i.e., starting with a token in the input place *i*, it should be possible to execute an arbitrary task by following the appropriate route through the WFnet.

Soundness is the minimal property any workflow process definition should satisfy. Furthermore, in order to add hierarchy to a process model another property called *safeness* has been defined. A PN for a process is safe if the number of tokens in each place of the net can not be greater than "1".

Note that soundness implies the absence of livelocks and deadlocks. In other words, a process is sound if it's Petri net with a start and end place is live and bounded. Formal definitions of these properties can be found in Appendix B. The correctness of a defined process can be thus verified by using standard Petri Net analysis tools such as Woflan (2002). When a process is not sound, diagnostics can indicate why it is not. For a number of important subcategories including the so called free choice Petri-nets - liveness and boundedness can be shown in polynomial time.

2.3.3 Limitations of Formal Approaches for UML based Conceptual Model Verification

Generally formal techniques rely on formal mathematical reasoning, inference and proof of correctness. Firstly, while being very effective they are often very costly due to their complexity and sometimes due to the size of the model under consideration. (Garth et al., 2002).

On the other hand, researchers agree that to be used effectively for UML designs, there are some important problems to be resolved about the formal techniques. According to Mota for example, the following are some of still open problems (Mota et al., 2004):

1) Automatic property extraction from UML diagrams to help the modeler to choose the kind of property to prove (safety, reachability).
2) Integration of inference engines such as Prolog's
3) Better translation mechanism to help the trace of the error, states and actions relevant to that error
4) Presentation of reasoning results such as counter example analysis in a more tractable way.

Thirdly, in transformation approaches, the formal language to be chosen and restricting the formal language to a subclass can ensure decidability in exchange of completeness. As examples, (Berardi et al., 2005) and (Van Der Straeten, 2005) rely on the transformation into description logics. As opposed to first order logic, where satisfiability is known to be undecidable, subsets of description logics, which can be used for semantic consistency of only a restricted subset of class diagrams for example, have decidable inference mechanisms.

Fourth, mapping of UML diagrams into a formal notation may bring semantic correspondence problem. This may result in loss of the original UML model at the verification level, and the difficulty of mapping back the result of verification on to the original model. Once the models are transformed and analyzed in the target formalism, to what extent can the model and the identified problem be traced back to the original UML model. Note that this problem had been shown in Section 2.3.2.2.

Furthermore, many of the studies based on transformation to formal language are restricted to one or two types of diagram. Only certain dynamic aspects are analyzed with Petri Nets for example. Moreover, the formalism also restricts the type of properties to be checked.

On the other hand, most of the formal techniques assume at least a predefined completeness in models. However conceptual models, unlike design models are developed in a sketchy manner at the initial phase of the requirements elicitations and may be incomplete in various ways which is difficult to define in advance.

For the mentioned limitations, we have not used a formal approach but instead focused on identification of different type of desirable property and development of an inspection process. The proposed approach is presented in Chapter 3.

2.4. Tool Support for UML Model Verification

UML tools exists which can be used for property checking. However, many of them are based on syntax (Rational, 2004) and some of the well-formedness rules (WFR) of static semantics (Argo, 2002; OCLE, 2005; Poseidon, 2006). Basic consistency checks for UML can be done with case tools which are becoming more and more sophisticated (Egyed, 2006). In these standard case tools, properties of a behavioral nature such as the absence of deadlocks and livelocks can not be checked.

For example, Argo UML tool has already many well formedness rules implemented and a critics section shows a list of correction and improvement suggestions on modeling elements used. Figure 9 shows a screenshot of the tools interface. The ‘to do’ window shows a critic on a ‘fork’ element which has two input transitions. Furthermore, the tool performs these checks on the fly and categorizes them into three priorities.

However, many of them are based on well formedness rules of static semantics and heuristics of object oriented development specifically for producing JAVA code. As illustrated in Figure 9 although syntax problem of the ‘fork’ element has been identified by the tool, the possibility of the livelock on the “fork node” has not been identified.

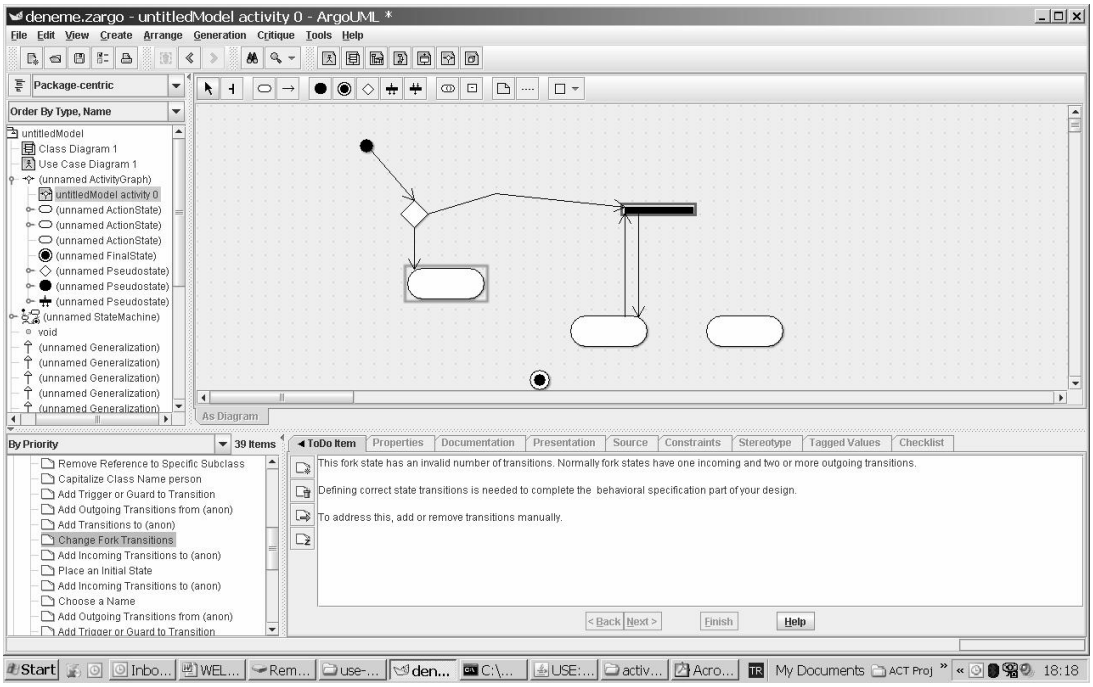


Figure 9: ARGO UML Case TOOL- Produced Critics on Activity Diagram Example

Furthermore, a couple of MDSD (Model Driven Software Development) tools have been developed to support meta-modeling. Graph Transformation Based Editors like DiaGen (De Lara et al. 2002), ATOM (Minas et al 2003) and Meta Edit (2007) which generate domain-specific editors from language specifications based on graph transformation.

In these environments, the abstract syntax of the language is given by a type graph which is very similar to a meta model. Language constraints restricting the set of valid diagrams, are modeled by restricting the set of editing commands, i.e. these editors are usually syntax-directed. An editor command is modeled as a graph rule being applied to the abstract syntax graph of the current diagram. Hence, graph transformation-based editors are usually purely syntax-directed, i.e. each editing operation yields a syntactically correct diagram. (Taentzer , 2006)

Meta Edit for example is a commercial case tool for domain specific software development. It allows users to define both basic rules and checking rules depending on the graph type. By the help of this tool, the developer can define the modeling elements, the types of graphs and the relations between modeling elements, so the meta-model rules are enforced while developing the model. These rules depend on the graph type and can vary between graphs. The model checker is a powerful tool for developing more complicated meta-model rules. So by using Meta Edit (2007) for modeling, various verification activities can easily be

performed and also injection of various kinds of defects can be prevented. Hence, when the domain rules are mostly static the tool may be helpful for verification

Other environments such as Open Architecture-ware (2007) and GME (2006) can be used to check properties related to syntax and simple consistency rules of the domain specific notation. Since there is a lack of agreement even on a common set of desirable properties of UML-based models themselves, identifying and defining desirable properties for domain specific notations is not trivial. For this reason, some studies have concentrated on the development of rule-based systems where rules can be defined flexibly in a declarative language (Wagner, Giese & Nickel, 2003), (Berenbach, 2004) to check compliance to static semantics. However, underlying rule based mechanisms of these approaches in general can implement semantic property checking.

Lilius & Paltor (1999) for example developed vUML, a tool for automatically verifying UML models. UML models are translated into the 'Process Meta Language' (PROMELA) language and model-checked with the SPIN model checker. The behavior of the objects is described using UML statechart diagrams. The user of the vUML tool neither needs the know how to use SPIN nor PROMELA. If the verification of the model fails, a counterexample described in UML sequence diagrams is generated. The vUML tool can check that a UML model is free of deadlocks and livelocks as well as that all the invariants are preserved. In general translation employed is not trivial.

Other tools exists and each one implements a particular kind of semantic property checking, (Statemate-Magnum), (Tabu), (Eishuis & Weringues, 2004), (Schinz et al., 2004) adopting a particular formalism. Hence, complexity and semantic correspondence problems remain to be tackled. On the other hand, in conventional case tools, semantic and behavioral properties such as the absence of deadlocks and livelocks can not be checked.

2.5. Inspections and Reviews for UML Model Verification

2.5.1 Software Inspections

Inspections and reviews are informal techniques used in software quality assurance. Fagan (1976), is one of the pioneers who have introduced software inspections. He defines an inspection as *"formal, efficient, and economical method of finding errors in design and code*". An error or a defect is defined as "*any condition that causes a malfunction or that precludes the attainment of expected or previously specified results*". It is argued that, inspections have evolved into one of the most cost-effective methods for early defect

detection and removal (Laitenberger & DeBaud, 2000). Gilb claims that inspections can lead to the detection and correction of anywhere between 50 and 90 percent of defects (Gilb & Graham, 1993).

There have been many researchers attempting to improve the performance of inspections. For example phased inspections proposed by Knight and Myers (1993), divide the normal inspection into several smaller phases. These phases can be carried out by one or more inspectors. Each phase focuses on one specific type of defect (compared to inspections, which look for all types of defect in one big inspection). On the other hand, Active Design Reviews (Parnas & Weiss, 1985) for example were created to ensure complete coverage of design documents.

Each new study proposed to improve the inspection process, by changing the characteristics of the phases or by defining different roles in the inspection organization. However the inspection phases of Fagan's original description i.e. preparation, inspection, and rework and follow-up have remained (Laitenberger & DeBaud, 2000). Among these three phases individual defect detection phase (preparation phase) is proved to be very crucial (Johnson & Tjahjono, 1998). Our inspection approach is aligned with these basic phases.

2.5.2 Defect Detection Methods

During defect detection activity, inspectors read the software document to determine whether quality requirements, such as correctness, consistency, testability, or maintainability, have been fulfilled.

The defect detection and defect collection activities can be performed either by inspectors individually or in a group meeting. Since recent empirical findings reveal that the synergy effect of inspection meetings is rather low (Johnson & Tjahjono, 1998), defect detection should be considered as an individual rather than a group activity. Basili (1996) also claims that the main focus of inspection should be defect detection activity.

The defect detection activity can be categorized in three basic classes. The most widely used defect detection method is ad-hoc review. Ad-hoc review provides no explicit support to the inspectors. The inspectors have to decide how to proceed, or what specifically to look for, during the reading activity. Hence, the results of the review activity in terms of potential defects or issues are fully dependent on inspectors experience and expertise.

Checklist-based reading on the other hand, (Gilb & Graham, 1993) provides some guidance to the inspector. However, checklists are mainly in the form of yes or no questions.

Although a checklist provides some guidance about what to look for in a review, it does not describe how to perform the required checks.

Thirdly, in response to lack of effectiveness in the use of ad-hoc and checklist methods Porter et al. (1995) developed a scenario based method to offer more procedural support. A scenario describes how to find the required information in a software document. Different scenario based method have been developed for detecting defects, each one creates scenarios differently. For example, the method developed by Porter, is called defect based reading as scenarios are derived from defect types with a set of guiding questions. The inspection method developed in this thesis can be also categorized as a defect based review method.

2.5.3 UML Model Inspections

According to our knowledge, few studies have been done in the area of inspection of UML models. Three notable studies in this are will be summarized.

Travassos et. al. (2002) describes a family of software reading techniques for the purpose of defect detection of high-level object-oriented designs represented using UML diagrams. This method is a type of perspective based reading for UML design inspection and can be considered as following the line of techniques discussed by Basili et al. (1996). The "Object-Oriented Reading Techniques" consist of 7 different techniques that support the reading of different design diagrams. This method is composed of two basic phases. In the horizontal reading phase, UML design artifacts such as class, sequence and state chart diagrams are verified for mainly inter-diagram consistency. In the vertical reading, design artifacts are compared with requirements artifacts such as use case description for design validation. Hence most of the properties checked in these studies are related to validation and the main artifact considered is a software design rather than a conceptual model.

Laitenberger et al. (2000) presented an experiment that was carried out to investigate the effectiveness of perspective based reading (PBR) for UML design documents in comparison to checklists. The results of the experiment showed that PBR scenarios help improve inspectors understanding of the inspection artifacts. It was found to reduce the cost of defects in the meeting phase in comparison to checklists based reading.

An important book on UML quality assurance Unhelkar (2005) describes quality properties within and among each diagram type along with checklists for UML quality assurance. The foundation for quality properties are set by the discussion on the nature and creation of UML models. This is followed by a demonstration of how to apply verification and

validation checks to these models with three perspectives: syntactical correctness, semantic meaningfulness, and aesthetic symmetry. The quality assurance is carried out within three distinct yet related modeling spaces:

- Model of problem space (Computation independent model in MDA terms)
- Model of solution space (Platform independent model)
- Model of background space (Platform specific model)

Verification and validation checks are also organized according to these three modeling spaces, making it easier for the inspectors to focus on the appropriate diagrams and quality checks corresponding to their modeling space.

Although in Unhelkar (2005) conceptual modeling (CIM - Computation Independent Model) is considered separately and verification and validation checklists in different categories such as aesthetics, syntax and semantic are provided, most of the check list items are related to validation and completeness. Items related to verification are mostly syntax, static semantic or simple cross diagram dependency checks.

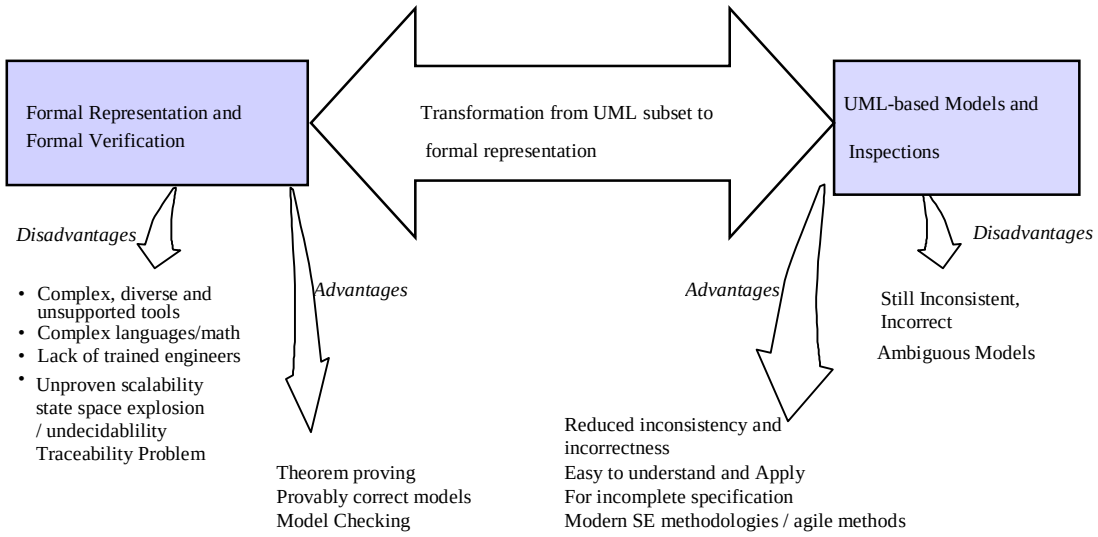


Figure 10: Comparison of Inspections to Formal Verification for CM in UML

2.6 The Need for a Systematic Inspection Method

The literature that has been reviewed in this chapter shows that in general, firstly transformation approaches are partial. Secondly, they suffer from the semantic correspondence, (as shown in Section 2.3.3.2) complexity, scalability and most importantly the traceability problems. Furthermore, conceptual model are developed in sketchy manner early in the requirements elicitation phase, hence may be incomplete where formal techniques permits only predefined incompleteness especially when they are supported with tools.

On the other hand, there is a lack of an agreed set of properties for quality UML models. Properties checked in UML inspection studies are related to syntax, static semantic or simple cross diagram dependency checks and the main artifact considered is a software design rather than a conceptual model. When a domain specific notation is used verification is particularly important as discussed in Section 1.2. However approaches, methods and techniques do not address this need systematically.

Figure 10 summarizes the motivation advantages and disadvantages of both the formal approaches and informal approaches for CM verification. In this study, an inspection approach is preferred to a formal approach due to various advantages: Firstly, informal techniques are easy to use and understand. Their application is straightforward. As checklists and guidelines are the main sources, they can be performed without any training in mathematical software engineering. Inspections may be very effective if applied rigorously and with structure and they are relatively less costly and they can be used at any phase of the development process.

Since conceptual models are used primarily as a means of communication, "Conceptual" implies human conceptualization, which inherently implies tractable abstraction levels and size. Hence, tool support is not crucial, but rather the verification results may also be used as a means to identify and resolve validation issues. It is more cost effective to integrate the verification tasks with the validation tasks which require human (in simulation domain subject matter expert interpretation) interpretation hence mostly a human activity.

We believe that a systematic and holistic approach, rather than using formalisms, may provide significant practical results. In this study we identify and formulate desirable properties and accordingly define an appropriate inspection process to improve the quality of conceptual models in a notation derived from UML.

CHAPTER 3

THE INSPECTION APPROACH FOR CONCEPTUAL MODELS IN A DOMAIN SPECIFIC NOTATION

As discussed in the previous section, there is need for a systematic, holistic and practical approach for conceptual model quality assurance. The motivating reasons for our approach can be summarized as the following:

In general, transformation approaches are partial. Furthermore they suffer from the complexity, scalability and most importantly the traceability problem: to what extent can a model and the identified problem be traced back to the original UML model.

Most of the properties checked in UML inspection studies related to semantic are validation issues. More over, the main artifact considered is a software design rather than a conceptual model. Items related to verification are mostly syntax, inter-diagram dependency checks. Finally, none of the previous works on inspections provides any direction how to identify desirable properties such as class consistency, refinement consistency and behavioral properties for conceptual models, in UML based notations.

The work presented in this thesis takes a systematic and holistic but a less formal approach. The differences from the above mentioned works can be stated as the following: Although some of the studies related to property checking mention briefly the need for property identification and consider various types of desirable properties for UML models, they are not founded on a property identification framework that considers domain specific notations (DSN). This framework is presented also in Tanriover & Bilgen (2007). Secondly, in this study, based on ontological interpretation for the structural view of conceptual models, we developed structural deficiency patterns which have not been proposed before. Thirdly, unlike Travassos et al. (2002) and Unhelkar (2005), we focused mostly on semantic

properties and developed verification tasks rather than validation tasks. Finally, our main artifact is conceptual models rather than software design models.

This chapter presents the inspection approach proposed for improving the quality of conceptual models developed in a domain specific notation. First, the process of identification of desirable properties of conceptual models for domain specific notation is described. Intra- and inter-view properties are considered designated. Based on this, semantic properties are defined considering the conceptual modeling notation. Then, a systematic inspection process is proposed for checking mostly semantic properties for different type of diagrams and for relations between these diagrams.

3.1. A Framework for Identifying Properties for a DSN

When adopting UML for a given domain, relevant concepts and their interrelations must be described by means of a meta model. As shown in Figure 11, meta-model definition includes the concrete syntax, abstract syntax and static semantics of the derived notation. For graphical notations like UML, the concrete syntax contains boxes and arrows, abstract syntax contains constructs such as classes (nodes), attributes, associations (relations) and so on, and relationships between these notation elements (Stahl & Völter, 2006).

A model is often represented with several views (diagrams) and each view may be composed of multiple diagrams. Diagrams are composed of permissible modeling element type instances (model elements) which may be refined by a set of other model elements. Modeling element types can be generalized into two fundamental sub-types; concepts and relations. Relations are generally of sub-typed to generalization or association so on. These instances of modeling element types are used to compose a conceptual model obeying the rules specified by abstract syntax and static semantics.

For UML based notions, various type of properties discussed in Section 2.2 can be classified in four broad categories. These are syntactic, semantic, horizontal and vertical properties. In the following paragraphs, we describe each of these to provide a framework for identification of properties for a specific UML based notation for conceptual modeling.

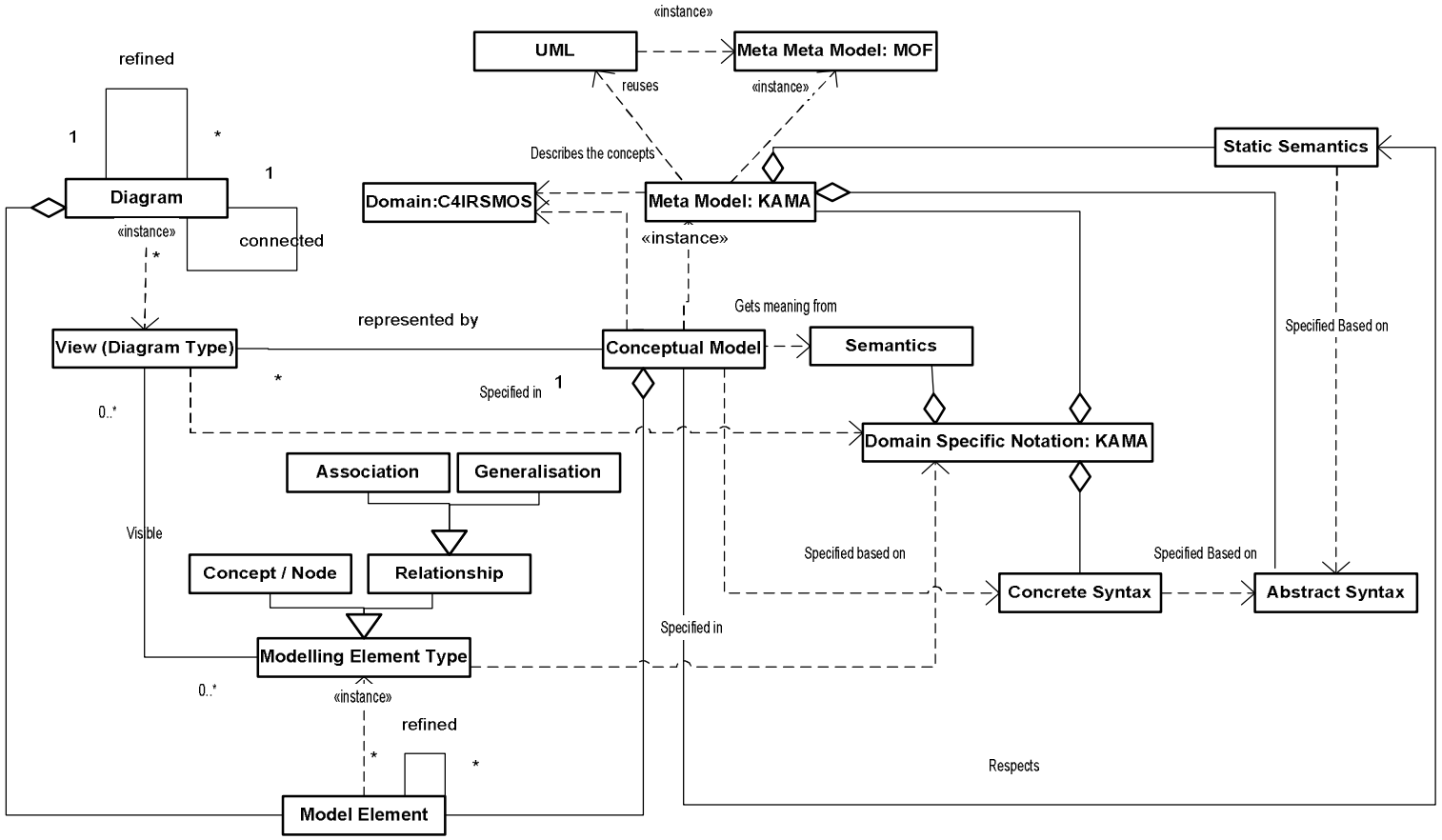


Figure 11: The Framework for Identifying Desirable Properties of Conceptual Models in a DSN.

Syntactic Properties

The UML's meta-model (*abstract syntax*), *concrete syntax* and *WFR* (*well-formedness rules* in OCL) of the static semantics, establish the properties for well-formedness of a model (UML Superstructure, 2005). For example, the properties such that "every class should have a unique name" or "an activity an initial node in an activity diagram has at most one outgoing flow" are desired syntactic properties for a UML based model.

In general, syntactic correctness is usually a prerequisite to any further analysis of desired properties. Fortunately, some of the contemporary case tools for UML are capable of performing many of the syntactic and well formedness checks (Rational, 2004), (Argo, 2002), (Poseidon, 2006). However, a derived notation may refine or change the concrete syntax, abstract syntax and the static semantics. For this purpose a couple of MDSD (Model Driven Software Development) tools have been developed to support meta-modeling and syntax directed editing. For example, Meta Edit (2007), Open Arhitectureware (2007) and GME (2006) can be used to check syntax and static semantics of models in the derived modeling notation. Hence, our main interest in this approach is semantic properties of conceptual models.

Semantic Properties

In general, even if a UML model is syntactically well, it may not be *semantically* well (Berardi et al., 2005). Examples had been shown in Section 2.3.1 and 2.3.2. Although well-formedness of a model can usually be checked by a static inspection, semantic aspects such as the conflicting constraints in a class diagram and such as absence of deadlocks and livelocks in activity diagrams cannot be completely verified until run time. The problem here is that UML is not an executable language.

Some of the semantics for UML elements are described informally in natural language in the specification, however the specification is huge and there is not a systematic treatment of semantic properties, such as class consistency, inheritance consistency, consistency between views and desirable semantic properties (such as deadlock, liveness etc.) for views used to model behavior such as state chart diagrams and activity diagrams. A simple example of semantic property is "All generalization hierarchies must be acyclic". This property is expressed in (UML, 2005) as OCL constraint, because it is relatively easy to express in a formal language. Note that defining and expressing a property is one thing, detecting whether it holds, is another, which may require different means or algorithms. For finding cyclic inheritance a tree traversal algorithm may do the task. It is generally agreed that

because of the huge variety of modeling constructs provided, a single general technique is not enough.

In addition, some of the UML semantics is deliberately underspecified by OMG to allow adaptation for various needs. Different interpretations of UML models are possible. As an example, there is no agreement on the proper way to inherit attributes with the same name (as attributes having the same name is allowed in different classes of the same static model) in multiple-inheritance. Relaxed interpretation inherent in UML complicates property checking, which usually requires language to be completely and precisely specified. When defining a derived notation, UML's semantic aspect can be tailored for the domain and described using natural language such as specified in (KAMA, 2006) and (Karagöz & Demirörs, 2008). Hence, the interpretations of the models and semantic properties in the new modeling notation may vary from UML's.

On the other hand, UML is historically rooted in a collection of different graphical modeling languages which have been studied for property checking. For example; for cardinality constraint consistency checks, EER (Enhanced Entity Relationship) modeling notation may be considered to have similar semantics to class diagrams (neglecting operations). Similarly, properties of Petri Nets may be adopted for control flow checks of task-flow diagrams. Thus, various types of properties can be adopted for certain parts of UML conceptual models.

Horizontal and Vertical Properties

Although inter-diagram consistency properties are not explicitly specified in UML, they are implicitly present because meta-classes appear in the meta-model of more than one diagram type (UML Superstructure, 2005). For example, a message in a sequence diagram has to match an operation in a class diagram. In the following paragraphs, we describe requirements to be taken into account in formulating inter-diagram properties for a domain specific notation. Horizontal and vertical properties are two main aspects that need to be considered.

Horizontal properties define the consistency of the common meta classes used in different views of the notation. Firstly, horizontal properties may vary depending on the meta-model of the derived UML notation. Secondly, the modeling methodology for the modeling domain organizes the views according to a development process and order in the process may also establish implicit relationships between the views or modeling elements (Sourrouille & Caplat, 2003).

Table 1: Process for Identification of Horizontal and Vertical Wellness Properties

1. For each view identify the set of modeling elements allowed to be utilized in the view.
2. Identify modeling elements that occur in more than one diagram type as overlapping modeling elements, constituting the set D. 2.1 Based on these modeling elements, define or identify horizontal dependency property between views sharing the modeling element d ϵ D 2.2 Decide on the direction of dependency based on the development processes.
3. Identify all refined modeling elements constituting the set R, for each r in R 3.1 Identify modeling elements refined in a different type of view constituting the set R1. Define or identify a vertical refinement property using the elements of both views. e.g. Missions in mission space diagrams are refined by tasks in task-flow diagrams in KAMA. 3.2 Identify modeling elements refined with the same type of view, constituting the set R2. Define or identify a vertical refinement property for this element. e.g.: A structured task-flow node refined by an other task-flow diagram in KAMA.
4. If a modeling element is in R1 and also in R2 then define or identify a refinement property for the views of this element. e.g.: If a class A is in both in R1 and R2 since a class can be represented by state chart and also be refined by other sub classes in a class hierarchy *say* with class B then the corresponding state views of the super and sub classes, should be equivalent after removing the transitions (events) and states that are specific to the specialized class.
5. If an identified modeling element is in R and D then define or identify a refinement relation for the views of D and the views of R in question considering the modeling elements in a relationship with the element (both refined element and refining elements). e.g.: If a class (object) is represented in a sequence diagram, hence in D and also a class (object) is shown both with a state diagram hence in R the "sequence of messages" in sequence diagram coming in and out of the class (object) (i.e.: in relation with the class) must be a subset of event sequences on the state diagrams.

Vertical properties, on the other hand, refer to consistent refinement of modeling views. Refinement is a transformation that takes a model from an abstract level to a more concrete or detailed level. A specific form of refinement is hierarchical refinement which establishes a relationship between 1 model element to n model elements as shown in Figure 1. Structured activity nodes in activity views and super states in state charts are natural points or facilities for hierarchical refinement. Hence, for these views we can talk about the top level state or activity.

Secondly, a more general form of refinement can be defined between n elements to m elements. In this case we can talk about the abstraction levels for diagrams. For ensuring a consistent refinement of the model, these refinement properties should be defined, which again has not been explicitly defined in the UML specification (UML, 2005). An example for this type of property is: "The set of states of an object defined by a father class must be a subset of the set of states of an object of the child class". Therefore, the refinement properties should be extracted for the domain modeling notation.

Thirdly, as presented in Figure 1, the same view may be represented on more than one diagram, even at the same abstraction level. Connectors in task-flow diagrams constitute an example for this. Lastly, two diagrams of the same type are not necessarily related by a refinement relation. They may be at the same level of abstraction, but represent different aspects of the system. For example, "two state diagrams can represent object states in two different scenarios in the context of a use case".

Based on these requirements and observations, we have formulated the process presented in Table 1 to identify horizontal and vertical properties which had not been explicitly specified before. Note, that the 4'th and 5'th steps are optional, depending on whether the consistent refinement of views is considered important for the intended use of the conceptual model.

Considering the concepts in Figure 11, their relationships, information provided in KAMA notation specification and the process defined in Table 1, we developed the inspection process in two fundamental phases. The first phase is the property identification phase in which, depending on the modeling notation, one flexibly identifies and formulates applicable desirable properties. The second phase is the development of related verification tasks for intra-diagram and inter-diagram inspections. The actual conduct of the inspection is performed based on this process. The following subsections present these two main phases.

3.2. Property Identification of KAMA Notation

In accordance with the framework presented in the previous section, KAMA notation defines concepts specific to the domain of interest and elements in the meta model are mapped to the concepts of C4IRSMOS (Command, Control, Communications, Computer, Intelligence, Surveillance and Reconnaissance) domain (Karagöz & Demirörs, 2008). The diagrams as well as the model elements serve to the specific needs of the C4IRSMOS domain. 7 types of diagrams are used to represent both the structural and behavioral aspects of a conceptual model. The diagram examples are in Appendix C.

As the conceptual modeling is performed during requirements analysis phase simple diagrams with only high level domain elements are defined to prevent the tendency to model the design issues. Hence KAMA models are simulation environment, infrastructure and implementation independent, i.e. a conceptual model developed in KAMA may be realized by various simulations which may be developed in various development environments/languages.

On the other hand, Karagöz & Demirörs (2008) explains that; since two most important views in KAMA, mission space and task-flow views included relationships (such as achieves, quantifiedBy, produces, inputTo) and model elements (workproduct, role, measure, objective) does not comply with the UML meta model and they could not exist in a valid UML model, it was impossible to represent all of the required concepts using UML profiles. Hence, UML meta-model elements were reused where possible. That is, when needs of the KAMA notation and associated constraints were not in conflict with UML meta model. Hence, KAMA reuses many parts of UML for defining itself, but do not comply to the UML meta model. However, KAMA is a MOF compliant notation (Karagöz & Demirörs, 2008).

For defining the semantics, natural language and OCL constraints were used. However, as explained in the above paragraph only a small number of additional static semantic properties are defined, because simply UML were reused mostly. Only a small number of OCL constraints in addition to UML's were initially defined. And almost all of them were simple syntactic and only a few dependency properties that the models in diagram should adhere.

Table 2: KAMA vs UML: Basic Syntactic Differences

Type	KAMA View	Similar UML 2.0	KAMA 1.0 Concepts	UML 2.0 Concepts	KAMA Relationships	UML 2.0 Relationships
Structural	Entity ontology	Class Diagram	Entity, KAMA Capability, KAMA Attribute	Class, Attribute, Operation	inheritance, part/whole	inheritance, composition
	Command hierarchy	Class Diagram	Actor	Actor	superior, subordinate	directed association, directed association
	Organization structure	Class Diagram	Role, Actor	Actor, Actor	own	association
	Entity relationships	Class Diagram	Entity, Work product, KAMA Capability, KAMA Attribute	Class, Class, Attribute, Operation	part/whole, association, generalization	composition, association, generalization
	Mission space	Use Case Diagram	Mission, Actor, Role, Work product, Objective, Measure, Entity	Use case, Actor, Actor, Class, Class, Class, Class	responsible, extends, includes, achieve	association, extends, includes, association
Behavioral	Task-flow	Activity Diagram	Task, Actor, Role, Decision node, Fork node, Synchronization node, Initial task, Final task, Objective, Measure, Work product	Activity, Transition, Actor, Actor, Decision node, Fork node, Synchronization node, Initial node, Final node, Class, Class, Class,	control flow, inputTo, produces, responsible, realize, achieve, quantifiedBy	control flow, directed association, directed association, association, association, directed association,
	Entity-state	State Chart Diagrams	State, Trigger, Event, Initial-state, Final-state,	Entity, State, Trigger, Event, Initial-state, Final-state,	transition	transition

3.2.1. Syntactic Property Identification Phase

There are three purposes of this phase. The first purpose is to familiarize the inspector with the UML knowledge to the extended modeling notation. The second purpose of this phase is to identify the diagram types, syntactic and static semantic differences of the derived notation from the UML. And the last one is to decide on the inspection strategy; the sequence and the scope of inspection process can be tailored.

For syntactic property identification of KAMA notation, we referred to the conceptual modeling notation specification documentation (KAMA, 2006; Karagöz & Demirörs, 2008) and UML 2.0 specification (UML Superstructure, 2005). Initially, 7 types of diagrams are used to represent both the structural and behavioral aspects of a conceptual model. We have identified that the structural diagrams were ontology diagrams, command hierarchy diagrams, entity-relations diagrams tailoring UML language elements defined for class diagrams. We have also identified that the behavioral diagrams were task-flow diagrams adopting language elements from activity diagrams and entity-state diagrams adopting the elements from state-chart diagrams. Table 2 shows a summery of the diagrams types, concepts and relation types defined in KAMA notation and syntactic differences with UML 2.0. KAMA diagram examples and concrete syntax of the KAMA notation can be seen in Appendix C.

After retrieving the required information as above, concrete syntax differences are identified from Karagöz & Demirörs (2008) to check for correct concrete syntax in diagrams. Secondly, by quickly skimming example of diagram types and, rules specified by (Ambler, 2005; Briand et al., 2003; Killand & Borretten, 2005; Ohnishi, 2002; O'Sullivan, 2003; SD Metrics; Unhelkar, 2005). (UML property references) and some well known UML books (e.g. Fowler (2000), syntactic, simple static semantic and consistency properties for given diagram types and used modeling elements, previously unknown, are identified. As there are slight variations from UML's syntax, most of UML's WFR were reused for defining desirable syntactic properties for conceptual models in KAMA specification.

However, we have quickly identified that KAMA notation adds the 'workProduct' and 'Measure' modeling element types to the task-flow diagrams which differs from the UML's activity diagrams. We realized that control flow semantics may have changed accordingly, which has been later taken into account in task-flow inspection phase.

3.2.2. Identify Intra-diagram Semantic Properties

Based the framework in Section 3.1, we have investigated each diagram type by referring to the notation description (Karagöz and Demirörs, 2008) and UML property references. We identified semantic properties for structural and behavioral views. A list of the identified properties can be found in properties are presented in section 3.2.4.

The properties identified for structural view can be checked with the help of structural deficiency patterns. The structural deficiency patterns have been developed to provide a guidance to check kind of properties such as class diagram consistency, class or relation liveness and class or relation equivalence defined in Section 2.3.1. On the other hand, behavioral verification tasks for task-flow diagrams are formulated to check for properties such as liveness, deadlock, reachability and boundedness was defined for Petri nets in Section 2.3.2. of this theses. The formulated inspection process is presented in the Section 3.3. But before presenting the inspection process, in the following subsections, we describe identification process of structural, behavioral, inter- and intra-diagram properties.

A. Identify Structural (Class Like) Diagram Properties

As already stated, we are willing to identify possible deficiencies regarding the semantics of structural perspective. For the reasons explained in Section 2.6, we have developed deficiency patterns to be used in inspections, instead of using a formal approach. Although the patterns can not address all of the deficiencies, we believe that inspector will be only guided and even can identify semantic deficiencies which have not been explicitly defined as patterns before hand. The following basic assumptions and observations have led us in developing deficiency patterns to be used in structural view checks:

We adopted a pattern based approach which stemmed from the observation that there are in fact two fundamental relations for structural modeling and the rest of the relations are variants of these. These fundamental relations are *association* and *generalization* as shown in Figure 11. For example, aggregation and composition are stronger forms of association. Dependency relation is an association further specialized to abstraction, substitution, usage and realization relations in UML. Usage relation is further stereotyped to <<call>>, <<create>>, <<instantiate>> and <<send>> relationship types and abstraction is stereotyped into <<derive>>, <<refine>> and <<trace>>. For example, patterns with generalization or association relations may be also applicable to derived relations. Note that for example <<include>> stereotype is derived from aggregation relation and <<extend>> stereotype is derived from generalization relation. Similar patterns can be developed for use case

diagrams. Hence, we believe that patterns developed for general type of relations can be used to easily develop patterns for more specific type of relations such as relations used in DSNs.

Secondly, patterns are formulated assuming an ontological interpretation of conceptual models, for the reasons argued by Gemino & Wand (2004) and as defined in Everman & Wand (2005) rather than classical object oriented interpretation. The complete definition of ontological interpretations of UML models can be found in Everman & Wand (2005). For example, although it is possible to interpret generalization differently as discussed in the MOF standard (MOF 2.0, 2004) and in (Gitzel, 2006), for example, we assume that generalization shall be interpreted as deep instantiation and is transitive. Hence, constraints in the higher level of the ontological hierarchy should hold in the lower levels. Other examples of differences between ontology vs object oriented interpretation can be also found in Everman & Wand (2005)

Thirdly, inconsistencies may occur because of the possibility of representing a view of the model by spanning the view over multiple diagrams. When the same model element is used in more than one diagram of the same view at the same abstraction level, contradictions and redundancies may be introduced and remain undetected. However, to our knowledge most of contemporary case tools allow the same modeling element to appear in more than one diagram. We observed that transitivity and asymmetry property of derived relations in the domain specific notation may cause redundancy or contradiction related to this fact.. That is, if we model symmetry by using an asymmetric relation (e.g. A → B and B → A) this may be indication of a contradiction, and if we explicitly assert a relation that is already implied, (e.g. A → B → C and also A → C) this will result in a redundancy. This situation may remain undetected if these diagrams considered separately. Hence, during inspection, this possibility should be taken into account. The structural diagrams with a reoccurring common model element should be unified into a single diagram and then applying the verification tasks to this unified diagram.

Fourthly, most of the contradictions emerge from the patterns when one class participates in more than one relation and/or relationship type and/or constraint. So we have tried to formulate patterns to illustrate possible deficiencies when different types of relations are used. Obviously, we recognize that there may be plenty of ways in which models may contain deficiencies and by means of such patterns we do not aim for a complete check of static views. We only provide the basic deficiency patterns which can be confronted in practice and direct the inspector to the kind of structural deficiencies we would like to find in the structural models.

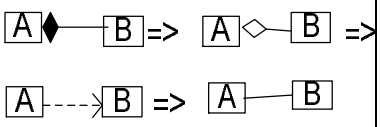
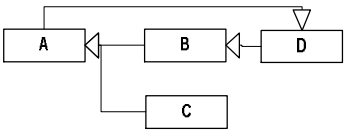
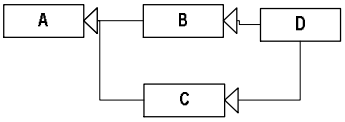
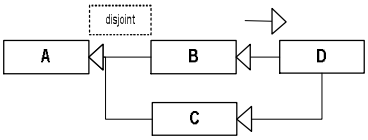
	***1.1 Strength of association kind of relations:*** We observed that there is a partial order between strength of association kind of relations. The re-occurrence of a weaker kind of association between two classes can be considered as a redundancy to be validated with the SME. For example; given that A is composed of B, A is associated to B may be signaled as a redundancy warning to be validated with the SME.
	***1.2 Circular transitive relation****:* This deficiency is the well known UML wff-rule which corresponds to the circular inheritance problem. This occurs if a class transitively inherits from itself, in this pattern, with the generalization relation between A and D and D and A. However, in addition to circular inheritance, we identify any circularity formed by any transitive relations (such as aggregation, composition and dependency) as semantic issue to be validated with the SME.
	***1.3 Lattices-multiple inheritances:*** Although permissible in UML and allowed in C++ like object-oriented interpretations, handling of this pattern depends on how the model is interpreted in the target domain. For instance, this pattern is not allowed for the JAVA. Remark that, any relation which is a subtype of generalization relation (such as <extend> and <include> relations) can form this pattern. Also, the aggregation, composition and dependency forms this pattern, this should be identified as a semantic issue. Hence, lattices and multiple inheritance should be validated with the SME.
	***1.4 Disjoint inheritance:*** This pattern is based on the possible (overlapping, disjoint, complete, incomplete) constraints that can be applied to a set of generalization relations. This example shows two cases of contradictions. The first one is formed since A and C are disjoint, they can not have same instances but on the other hand, since D is inherited from both B and C, there should be at least some common instances. So this forms a contradiction. The second one is formed by inheriting B from D rather than D from B. In this case, since D is a type of C and C is a type of A, B becomes a type of A by transitivity. However, by disjointness constraint B cannot have instances that C has, hence this again forms a contradiction. Such situations should be validated with the SME.

Figure 12. Patterns Developed Based on Strength of Relations, Generalization and Transitivity

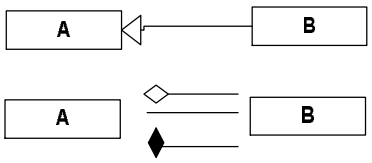
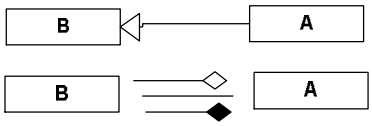
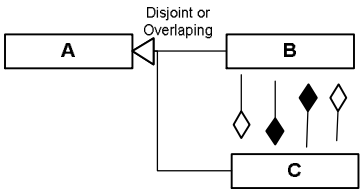
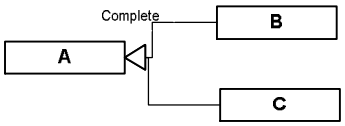
	***1.5 Generalization with aggregation:*** This pattern is based on the observation that utilization of different types of relations may be a source of redundancy or contradiction. In this case if we have a generalization relation between A and B and if we define an aggregation or a composition from A to B, this forms a very rare pattern. E.g. a chicken-and-egg kind of ontology. So, this should be identified as a redundancy warning to be validated by the SME.
	***1.6 Generalization with opposite aggregation:*** This pattern is similar to <generalization with aggregation> pattern. However, in this case, generalization relation is B to A, and if we define an aggregation or a composition from A to B, this leads to a possible contradiction. E.g. "a chicken is a bird and birds are composed of chicken" kind of ontology. Although, sometimes this may be valid, this should be validated by the SME.
	***1.7 Disjoint or overlaping with aggregation:*** This pattern is based on disjointness and overlapping constraints which can be defined on a set of generalization relations. There are two main cases. The first case occurs when classes B and C are disjoint, but there is a composition or aggregation relation between them. This forms a possible contradiction because this is equivalent to saying that B and C have no common instances but B is composed of C. Remark that if B is composed of C and only C, this pattern will result in a contradiction. If there were other classes that B is composed of and which are not inherited from A, this pattern would not cause a contradiction. The second case occurs when class B and C are overlapping, that is they have common instances and there is a composition relation between B and C. Hence, overlapping constraint becomes a redundancy. The first case should be validated with the SME.
	***1.8 Hidden abstract class:*** This pattern is developed based on completeness constraint which can be defined on a set of generalization relations. Since an *abstract class* can never be instantiated; it should not be shown on any of the views which show instance level elements.

Figure 12. (continued)

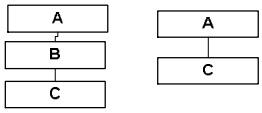
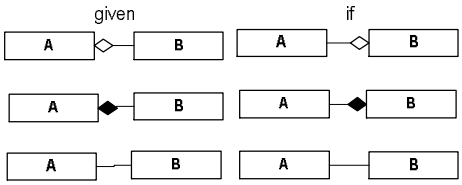
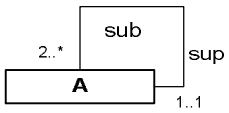
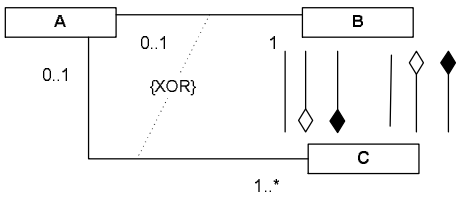
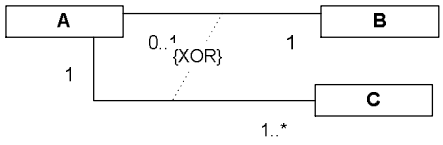
	***2.1 Redundancy by transitivity:*** This pattern is based on the transitivity property of dependency, composition, aggregation and generalization relations. In fact, many of the relations in UML can be considered in this category since they are a subtype of generalization relation. In this case since a class is transitively related to another one, there is no need to specify a direct relation between two. Remark that the strength of relation concept can be used to identify transitive redundancies by considering all weaker forms of relations.
	***2.2 Contradiction by asymmetry:*** This pattern is based on the asymmetry property of directed relations, such as composition, aggregation and generalization. For classes A and B, all the classes of the inheritance hierarchy of A an B should be checked for identification of this pattern. In this case, the utilization of a relation with direction between classes in both directions may be signaled as a possible contradiction and should be validated with the SME.
	***2.3 Recursive association multiplicity:*** This pattern is based on possible contradictions when recursive relations are used. Since the lowest multiplicity of one end is greater than the others' highest multiplicity, this forms a contradiction.
	***2.4 Association constraint (XOR) with association:*** This pattern is developed considering the possible constraints that can be used on a set of association relations. These are XOR, NAND, or similar constraints. If we define any association kind of relation between B and C then we immediately fall in a contradiction because B and C can never exist at the same time because of XOR constraint. Hence, these situations should be identified as a contradiction.
	***2.5 Association constraint (XOR) with multiplicity:*** This pattern is similar to the previous one. In this case, since the relation between A and B and A and C can not exist at the same time, the lower multiplicity of both of the relation at the side of A should be zero. Otherwise, XOR constraint can not be possibly satisfied in the run time. So if the lowest multiplicity is greater than 0, this causes a contradiction.

Figure 13. Patterns Developed Based on Asymmetry and Deep Inheritance

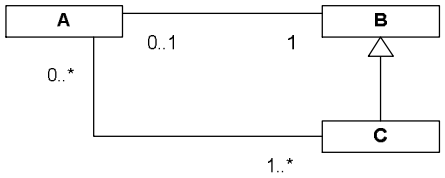
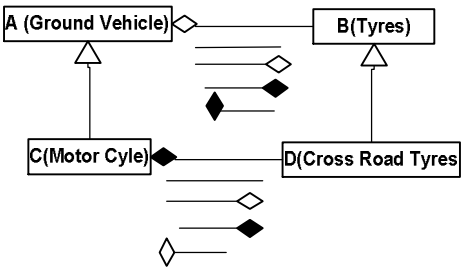
	***2.6 Inherited constraints I:*** This pattern is developed based on the concept of deep inheritance. In this case, we assume that the multiplicities should be inherited from the base class. Hence, the multiplicity range of the base class should be wider than that of the inherited class. If we encounter this kind of structure, we check the lowest and highest multiplicities of B and C. If l(Bc) =< l(Cb) and h(Bc) => h(Cb) and l(Ab) =< l(Ac) and h(Ab) => h(Ac) <> TRUE than multiplicities of B and C form a contradiction.
	***2.7 Inherited constraints II:*** This pattern is similar to "inherited constraint I". This pattern is included because it can be confronted in practice. If we encounter this kind of structure, we should check the lower and highest multiplicities of A and C and B and D. If l(Ba) =< l(Dc) and h(Ba) => h(Dc) and l(Ab) =< l(Cd) and h(Ab) => h(Ac) <> TRUE, multiplicities of B and D or A and C forms a CONTRADICTION. On the other hand, remark that given A is composed of B, if we have D is composed of C than we identify this as a contradiction because of contradiction by asymmetry pattern.

Figure 13. (continued)

Lastly, in practice, conceptual models are developed in a sketchy manner, at a high level of abstraction early in the simulation development life cycle. Hence, only basic modeling constructs such as classes and various relationships are used in the models at this phase. Furthermore, usually domain specific notations allow only a limited number of types of relations and model elements in each diagram type. For this reason, a limited number of deficiency patterns can be helpful.

On the other hand, since the models are not very complex, it is not difficult to identify the deficiency patterns in models. Based on the above assumptions and observations, we have formulated the patterns in Figures 12 and 13 Figure 12 shows patterns mostly based on generalization relations and Figure 13 shows patterns mostly based on association relations.

B. Identify Behavioral Diagram Properties

Types of behavioral diagrams in KAMA notation are entity state diagrams derived from state chart diagrams and task-flow diagrams derived from activity diagrams. However, only semantic properties for task-flow diagrams are identified because entity state diagrams do not differ or add new modeling elements to state-chart diagrams. However, task-flow diagram differs from the abstract syntax of activity diagrams. By analyzing the KAMA meta-model (Karagöz & Demirörs, 2008), we have identified the main differences of KAMA from UML's abstract syntax for the activity perspective. KAMA only adopts basic level activities, the object flows perspective is omitted and resource type of modeling element such as an input entity, output entity is used as input or output of a task node. This actually changes the control flow of UML activity diagrams.

Furthermore, as the UML specification refers to Petri Net semantics, we have decided to reuse the properties which are formally defined for Petri Nets (Murata, 1989). As it has been shown in Section 2.3.2, it is quite straight forward to translate an activity diagram to a Petri Net., because they are syntactically very similar. Note that Petri net properties are investigated in Section 2.3.2 of this thesis. For example, for control flow semantic checks, the soundness property defined by (Aalst, 2002) for workflow nets (a variant of Petri Net) was useful. According to Aalst (2002) soundness is composed of three properties:

1) it is possible to terminate, i.e., it is possible to reach a state with at least one token in the output place *o* and the moment a token appears in *o*,

2) there are no tokens left behind in the workflow net,

3) there are no dead tasks, i.e., starting with a token in the input place *i*, it should be possible to execute an arbitrary task by following the appropriate route through the WFnet.

As already stated in Section 2.3.2, soundness implies the absence of deadlocks and livelocks. From the analysis perspective as presented in Appendix B, if a workflow net is live and bounded it is sound. For checking the three properties that makes up the soundness property for KAMA task-flow diagrams, we have formulated the inspection tasks presented in Section 3.3.

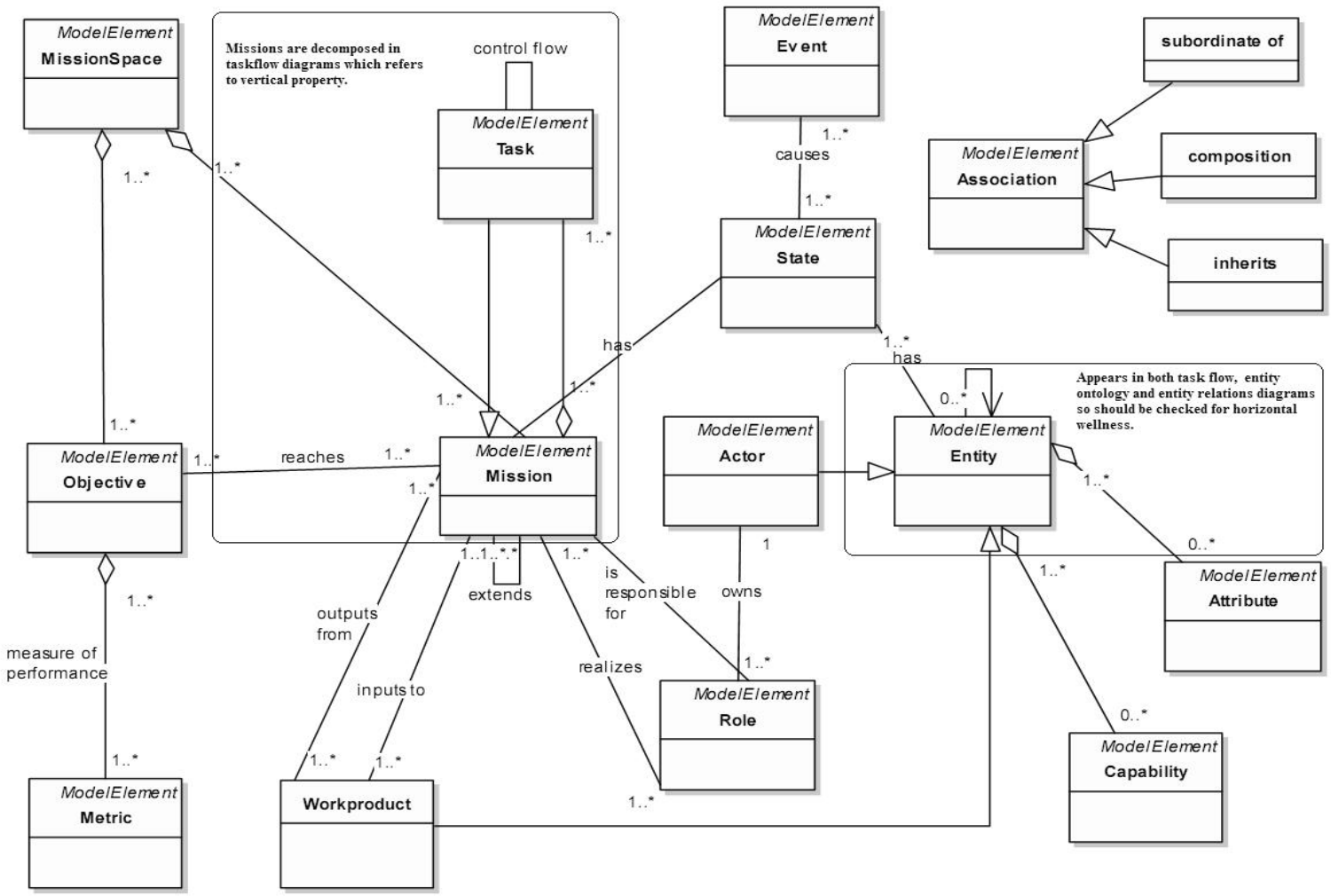


Figure :**14**. The Meta Model Defining KAMA's High Level Abstract Syntax from Karagöz & Demirörs (2007)

3.2.3. Identify Inter-diagram Properties

After identifying intra diagram properties for each diagram type, one should consider inter diagram properties. In Figure 14, the meta model for defining the abstract syntax of the KAMA notation is presented. With the help of abstract syntax, the notation definition in (Karagöz & Demirörs, 2008), Table 2 and using the process in Table 1 horizontal and vertical properties for KAMA are identified. Table 3 shows some of these properties. In KAMA, unlike to UML, use cases (missions) are refined in (activity) task-flow diagrams rather than in sequence diagrams, for which a property for this dependency has been defined. Missions are at the top of the structural hierarchy and refined by root task-flow diagrams. Root task-flow diagrams are further refined to main task-flow diagrams and so forth. A similar hierarchical decomposition is also defined for ontology diagrams.

Furthermore, for example, in KAMA two state diagrams can represent same entity's states in two the context of a two different mission as defined by the last relation in Table 3. So, we defined a property such that each entity can be represented with more than one state diagrams.

Another example is that, for each <input> and <output> entity in task-flow diagrams, a corresponding entity has to be present in ontology diagrams. Note, we derived properties for mostly used modeling elements and views. The identified properties, in fact, should be validated by the domain experts, if they are really required. Also the range of properties should be defined taking into account the final purpose of the conceptual models as discussed in Section 2.1.

Table 3. Inter-view Dependency Property Examples

Elements in Views	Inter-view Dependency Property	Origin of the Property
M = set of missions *T = set of tasks* *E = set of entities* *S = set of states*	$Mt \subseteq M \times T$ *< m,t > ∈ Mt meaning t* ***refines*** *m* $Es \subseteq M \times E \times S$ *<m,e,s> ∈ Es meaning an* ***entity e*** *can be in* ***state*** *s* ***in context*** *of mission m*	*Refinement property identified from the derived notation i.e. KAMA* *Horizontal property identified partially from derived notation and partially inherent in UML*

Note that, conceptual modeling process definition and conceptual modeling notations meta model may have an effect on the inspection process. For example, KAMA modeling methodology recommends the task-flow diagrams to be central in the definition of conceptual models and enforces a structural refinement strategy. Thus the vertical inspection tasks can be achieved more effectively by first top down check of refinement rather than a bottom up check.

3.2.4. Semantic Properties Identified for KAMA Conceptual Models

The identification phase of semantic wellness properties for KAMA models produced the set of properties, for which an inspection process is developed. The set of desirable semantic properties for KAMA conceptual models can be summarized as intra-view semantic properties and inter-view semantic properties.

Intra-view properties for structural perspective are: Class consistency, multiplicity consistency, relation and class liveness, consistency of inherited constraints, lack of transitive cycles, lack of redundant relations and coherence of inter-association constraints.

Intra-view properties for behavioral perspective are: Liveness of tasks, deadlock freedom in task-flows, lack of dangling tasks in task-flows, completeness and disjointness of guard conditions.

Inter-view properties are: Mission vs task-flow dependency, ontology vs sub-ontology dependency, task-flow vs. sub task-flow dependency, mission and task-flow refinement consistency, refinement consistency of entities in task-flow and entity ontology views, consistency of actor in mission space and organization views and consistency of attributes in entity state and entity ontology views.

The inspection tasks developed for checking the above properties are presented in the next section. Note that the application of the inspection process, in general, can not identify all the violation to the full set of identified properties above. Properties of structural perspective for example, can be partly checked by the help of deficiency patterns presented in section 2.2. For a complete assurance formal approaches should be used as discussed earlier in section 2.3.

3.3. Inspection Process

3.3.1. Intra-diagram Inspection

The purpose of intra-diagram inspection is to find the deficiencies in each of the structural diagrams and behavioral diagram. At the start of each diagram inspection, trivial syntactic errors such as name clashes in class diagrams, merge nodes with multiple outgoing flows, utilization of undefined modeling elements in conceptual modeling notation guide are checked. In this phase conceptual modeling notation specification (Karagöz & Demirörs, 2008) and (UML Superstructure, 2005) superstructure specification may serve as a reference to the inspector. Then, semantic checks for diagrams are conducted, as presented in the following paragraphs.

Structural diagrams inspection phase: In this phase, local contradictions and redundancies in diagrams derived from UML class diagrams are checked. In KAMA notation structural diagrams have been identified as entity ontology, command hierarchy and entity-relation diagrams. The inspector is presented with the deficiency patterns and their descriptions for him to familiarize with the kind of defects he will be looking for. Table 4 summarizes the inspection tasks for structural diagrams.

Table 4: Structural Diagram Inspection Phase Tasks

1. Identify syntactical properties such as omissions, missing attributes and name clashes, based on the syntactic properties.
2. Look for deficiency patterns in the class model, based on formulations defined in figure 3, 4. 2.1 Look for a match with each pattern for a contradiction or a redundancy. Consider the transitive closure of the relations for pattern matching. 2.2 Depending on the matched pattern validate the issue with the SME.
3. Identify and try to instantiate localized complex structures (structures with central classes participating in more than one relation and/or relationship type) not considered in task 2 by using the semantics of the modeling elements forming the structure. Validate the issue with the SME.

Table 5: Mission Space Diagram Inspection Phase Tasks

1. Identify syntactic errors such as duplicate names, dangling missions without actors.
2. Identify local patterns similar to structural patterns 1.2, 1.3, 2.1 and 2.2 to be validated with the SME.
3. Check the <inclusion> and <extends> relations for semantically correct usage. 3.1 Trace and check the relation to the refining task-flow diagram of the use case to make sure they are properly used.

Mission space diagram inspection phase: In this phase, use case like diagrams, in the case of KAMA conceptual modeling notation, the mission space diagrams are verified. The tasks in Table 5 are used for the mission space diagram inspection phase:

Table 6: Task-flow Diagram Inspection Phase Tasks

1. Check for syntactic errors such as dangling nodes, initial nodes with more than one outgoing transitions.
2. Identify decision nodes 2.1. Check if all flows outgoing from the decision nodes have guards 1.2. Check the constraints on the guards to make sure that they do not overlap (overlapping such as constraint on one guard is x>=0 and on the other x=<0) 1.3 Check if the guards define a complete set (such as x=>0 and x<0) 2.1.2. Identify overlapping and incomplete conditions.
3. Identify fork nodes 3.1 Check if the fork node has only one entrance, if not make sure that a task-flow is not missed before the flow is joined. 3.2. Check if all the flows from the fork node are joined by a (same) join node (non-structurally joined nodes or fork nodes may indicate concurrency problems) 3.2.1. If not, run the localized flows (flows coming out of the fork node) with UML's activity diagram (Petri-Nets like) control flow semantics 3.2.2. Identify livelocks and their causes.

Table 6: (continued)

4. Identify join nodes 4.1 Check if join nodes have only one exit transitions. 4.2 If not, it is possible that the join node is placed too early; there is possibility that there is still a need for a parallel flow. 4.3. Trace incoming transitions of the join nodes to make sure that all may eventually be activated. 4.4. If not, identify causes of deadlock.
5. If the task-flow is complex (includes more than one fork node or composite decision nodes) trace each flow from the start to end. 5.1. Make sure that every task may execute. 5.2. Identify dead tasks.
6. Trace the flows reaching the final nodes 6.1. Make sure that they do not originate from a fork node. 6.2. If they do, there is a possibility that some activities will terminate abruptly, try to identify such activities.
7. Identify loops by tracing through transitions. 7.1. Run the localized loop with UML's activity diagram (Petri Nets like) control flow semantics. 7.2. Identify livelocks and their causes.
8. Identify activities with <input> and <output> entities (An entity may be attached to a task according to the definition of KAMA notation). 8.1. Make sure that if tasks use outputs of one another, they also follow the implied sequence in the control flow because a produced entity may be an input for another task, causing the task to never start or to prevent parallel flow. 8.2. Identify deadlocks and redundancy.

Task-flow diagram inspection phase: The purpose of this phase is to verify the diagrams derived from UML activity diagrams i.e. KAMA task-flow diagrams. The activities in Table 6 are defined for the task-flow diagram inspection phase.

Table 7: Inter-diagram Inspection Tasks

1. Trace missions and check if they are modeled in task-flow diagrams and vice a versa.
2. Compare ontology diagrams with corresponding sub-ontology diagrams and make sure that there is only one sub-ontology diagram for an entity in the upper ontology diagram.
3. Identify further decomposed tasks in task-flow diagrams, make sure there is only one sub-task-flow diagram for a super task-flow node.
4. Identify <inputs>, <outputs>, <actor> in non-leaf task-flow diagrams 4.1 Trace <inputs>, <outputs>,<actor> in the next lower task-flow diagram 4.2 Ensure that there is at least one <input> and/or <output> and/or <actor> attached to the next lower task-flow and identify missing <inputs> and/or <outputs> and/or <actor> for the next lower task-flow diagram.
5. Identify <inputs>, <outputs>, <actor> entities in leaf task-flow diagrams 5.1 Trace <inputs> ,<outputs>, <actor> entities in the task-flow in the upper task-flow diagram 5.2 Check if there is at least one <input> or <output>, <actor> attached to the upper task-flow and identify missing <inputs> or <outputs> or <actor> in the leaf task-flow.
6. Identify extended missions, 6.1 Compare task-flow diagrams of the mission with task-flow diagram of the extended mission: the extended task-flow diagram should be reachable by only extracting model elements from extending diagram.
7. Check each <input> and <output> entity in task-flow diagrams, a corresponding entity has to exist in ontology diagrams.
8. Check all the actors in mission space diagrams are defined in organization diagrams.
9. Check if variables used in state chart diagrams are defined as attributes of corresponding entity.
10. Check if operations used as transitions in state diagrams are defined in the corresponding entity diagram.

3.3.2. Inter-diagram Inspection

In this phase the inter-diagram properties are verified. Using, the diagram definition of conceptual modeling notation (KAMA, 2006) its meta-model (Karagöz & Demirörs, 2008) and properties produced by the inter-diagram property identification process, we developed inspection tasks presented in Table 7. Note that presented tasks are not exhaustive; it may be augmented with newly identified properties.

3.3.3. Issue Classification

In order to be able to take corrective actions as the issues are identified, it is useful to delineate a categorization of issues. Issues may arise from syntactic or semantic wellness requirements. Furthermore, an identified issue may be a result of non conformance to the concrete syntax or abstract syntax. On the other hand, an identified issue may be related to static semantics defined by well formedness rules of UML or to dynamic semantics as illustrated by the conflicting constraints in class diagrams.

Some issues can be identified as incorrectness, so that the model is considered to have an error (for example, syntactic errors such as a transition with no target state). Some others can be identified as incompleteness or redundancy which does not mean an error in the model, but may be signaled as a warning.

At the end of each inspection task a recurrent task for classification of identified issue is performed. Definitions in (Linland et al., 1994) and (MOF 2.0, 2004) can be used to develop the issue type classification schema. These categories must be defined and delineated according to the verification and validation needs of the specific conceptual modeling environment. In Figure 18, we have developed a schema to be used in the inspection process for KAMA notation. Each deficiency is categorized first by type such as redundancy, contradiction, (deadlock, live-lock, dead-tasks, dangling references etc.) or incompleteness. Then, in accordance with the classification schema, it is classified by type of property such

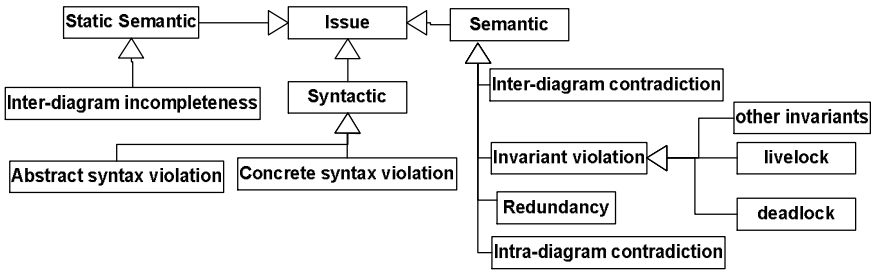


Figure 15. Classification of Issues

as syntactic, semantic, inter-diagram or intra-diagram. This information about issues is helpful in determining corrective actions later.

The proposed inspection process concludes with the classification of issues phase. The next chapter will present the multiple case study performed in developing and validating this inspection approach.

CHAPTER 4

APPLICATION OF THE INSPECTION APPROACH

This chapter describes our case studies to evaluate the application of the inspection approach. Two case studies were conducted to explore the applicability of the approach in real life settings. These case studies were performed to discover improvement opportunities for the inspection approach and its applicability. Section 4.1 explains the research strategy followed and Section 4.2 describes the case study design including the research questions. Sections 4.3 and 4.4 explain the conduct of the case studies.

4.1. Case Study Research

Emprical research techniques are applied in many disciplines (Yin, 2003). One of the emprical research methods is case study research. Yin describes case study as a research type which investigates a contemporary phenomenon within its real-life context. The case study research enables to learn about the state of the art, and generate theories from practice. According to Yin, it also allows understanding nature and complexity of processes, by answering “how” and “why” questions”. Yin (2003) argues that if the phenomenon and the context cannot be separated and it is not possible to stabilize certain number of variables a case study research should be preferred to other research methods.

There are not many real life conceptual models in our disposition and there is no directly comparable inspection method to ours as other inspection methods do not aim to check the kind of semantic properties as we aim and their subject of verification is software designs rather than conceptual models in a DSN, The ad-hoc review method can be seen as the only

comparable method to ours. However, note that, in the second case study which is presented in section 4.4, as we have applied the inspection process to an CM already gone through ad hoc reviews we happened to have provided a result based comparison of our method to ad-hoc review. Hence, instead of experiments, we have decided that case study research would be appropriate to investigate the applicability of the inspection approach. On the other hand, multiple case studies have been performed in order to improve the validity of the conclusions.

4.2. Research Design

We have conducted two case studies to validate our proposed inspection based approach for conceptual model verification. The first case study is performed with a conceptual model developed by two experts in a laboratory setting. A conceptual model developed by a software development company is used in the second case study. The two cases aimed to address different research questions.

The first case study has been mainly conducted for exploration purposes. We aimed to improve and refine the process we had initially formulated by applying it to a real conceptual. By a side product, the case study results were also used to improve the definition of KAMA notation. The model used in the second case study had gone through classical verification and validation reviews before the inspection was applied. Hence, the second case study aimed to validate the effectiveness of the inspection process.

We had identified the following research questions for the research:

- What are the desirable properties of conceptual models in KAMA notation?
- How can we detect the compliance of models to these properties?
- Is the inspection approach effective for detecting semantic defects in conceptual models of the mission space?

The first case study helped to respond to the first and second questions. In order to respond to the first question, we examined the UML model verification literature, existing approaches are experimented. Then we explored a typical conceptual model during the case study and inspired by the properties in the literature and by exploring the semantics of the graphical modeling notations in general, we identified desirable properties for KAMA conceptual models. The set of desirable semantic properties for KAMA notation is summarized in section 3.2.4.

For the second question, with the experimentations with methods in the literature, we had identified the drawbacks of the formal approaches. So, instead of following a top down approach, we have conducted a bottom up approach, starting from the modeling language definitions and actual models, to define an inspection process. During the first case study, we gradually defined the desirable properties and tasks for verification. By means of the case study we have shown that an inspection process could be defined and performed on the fly depending on the conceptual modeling notation and models at hand in accordance with this framework. On the other hand within the case study, the applied inspection took around 24 man hours and detected 85 issues, we concluded that the inspection approach is applicable with reasonable effort and worked out quite well to detect defects in conceptual models in domain specific notations. After the first case study, in the mean time we have focused on the semantic property checking and improved the inspection process accordingly.

The second case study focused mainly on the third question, which aimed to evaluate the effectiveness of the final approach with a conceptual model developed in a real life setting. In the second case study, the conceptual model under consideration had gone through a verification and validation process which was conducted by experts. These experts had experience with both UML language and the domain. During this process various syntactic and semantic 150 issues had been already identified and corrected. Even though the conceptual model was corrected previously, we were able to identify more than 30 semantic non trivial issues by conducting our inspection approach which had not been identified in previous reviews. Hence, this case study helped us to conclude that our inspection is effective in detecting semantic defects which may be also used as indications of validation issues. Note that the domain expert can help to resolve these issues.

4.3. Case Study 1

In this section, we describe first of the two applications of the proposed inspection process, to a KAMA conceptual model. The first case study was an exploratory study. We have started with a high level inspection process definition. We have preceded a bottom up fashion, starting from the modeling language definitions and actual models, to refine and improve the inspection process. We refer the reader to technical report (Tanriover and Bilgen, 2008) for details of the findings of this case study.

Table 8: Conceptual Model 1 Metrics

Diagram type	Number	Model Element Type	Number
Mission space	1	Missions	5
Task-flow	40	Activities	94
Entity ontology	5	Entities	68
Command Hierarchy	1	Command and Control Units	12

4.3.1. General Setting

General setting for the case study can be summarized as follows: Two modeling experts both having experience with UML modeling and KAMA notation had developed a conceptual model for a typical mission scenario. The conceptual model consists of one mission space diagram, one command hierarchy diagram, 5 ontology diagrams, and 40 task-flow diagrams

at varying levels of structural decomposition with different levels of complexity. The mission space consisted of 5 missions each of which was described with detailed task definitions. The task view consisted of 94 activities. Table 8 gives an idea about the size and the scope of the model.

The model was in its early stage of the CM development process (at the first iteration of three review stages) and was developed in a sketchy manner. For example, the classes did not include methods and accordingly did not include state chart diagrams. Hence during the inspection, only some of the inspection tasks were performed. In the model under consideration, for example, semantic wellness checks regarding cardinalities for any of the structure diagrams were not necessary because cardinalities were not used. Similarly, the consideration of state chart diagram related wellness properties were also left out of the scope of the inspection, because state chart views were not yet developed.

4.3.2. Case Study Organization

The first case study has been conducted right after the initial version of the approach was defined. The study aimed to test if considerable number and type of defects can be detected and to identify improvement possibilities for the initial version of the approach. As a side product, improvements to the definition of KAMA notation have been identified.

During the case study there were three roles responsible for activities:

Modelers: Responsible for developing the conceptual model using the KAMA notation.

Inspector: Responsible for performing the verification by inspecting the conceptual model developed.

Experts: Responsible for performing the defect approval and resolution in the inspection meeting.

The initial inspection process used was a preliminary version of the process defined in Section 3.3 of this thesis. There were three important differences. Firstly, the initial version of the inspection method included an initial task to check for concrete syntax in diagrams. Secondly, structural deficiency patterns were not defined. Thirdly, four of the properties in the inter-diagram inspection task were not identified.

The conceptual model inspection process was to be conducted in 2 main phases. The defect detection and reporting phase was to be conducted by an inspector. Review of the conceptual model has been already performed informally during conceptual model development phases by the two modelers. Our inspection process was performed at the end. This phase took 24 hours. After defect detection phase, an inspection meeting for validating the defects detected was planned. The inspector, modeler, two experts participated to the meeting. The conduct of this meeting took 6 hours.

The expected outputs for the case study were corrected conceptual model and the verification report. Main sources of evidence and data of case study were defect detection documentation and inspection meeting minutes.

4.3.3. Conduct of the Case Study 1

The defect detection phase of the inspection was conducted by one inspector who was experienced in object oriented software analysis and design, petri nets verification. The verification activity was carried out following the method that was defined in Section 3.3. The KAMA modeling tool, used for developing the model; was also used as the medium of inspection during the case study.

We have conducted the 3-phase inspection approach for verifying the conceptual models. The approach began with a pre-inspection phase where the inspector identified the diagram types used in the conceptual model, identified unfamiliar types of notation elements and relations used in different types of diagrams, and determined the inspection strategy. The inspector then performed intra-diagram and inter-diagram inspections based on activities defined in Section 3.3.

A. Intra-diagram Inspection

Structural diagram inspection phase: Applying the initial version of the inspection tasks in table 4, we identified only seven non-trivial defects because the allowed relationship types in structural diagrams were limited in KAMA notation and the model belonged to an early modeling phase. As an example, a redundancy on the command hierarchy diagram in figure 15 was identified. In command hierarchy diagrams sub/superior relation is a transitive relation derived from generalization meta-class of UML. When considering the pattern between "Brigade Assessment Center", "Division Command Center" and "National Command Center", we have identified that the sub/super relation between "National Command Center" and "Brigade Assessment Center" forms a semantic redundancy, by "redundancy by transitivity pattern".

Mission space diagram inspection phase: Applying the initial version of the tasks in Table 5, we identified 10 issues. One of the examples of the issues identified was missing extended and included missions. Since KAMA notation enforces structural decomposition from missions to task-flow, by comparing the task-flow diagrams with the mission space diagrams, we have identified 3 activities not included in the mission space diagrams.

Task-flow diagram inspection phase: As the result of performing the activities in Table 6, 23 issues were identified. Below are some examples of non trivial issues identified in the task-flow diagrams in Figure 17:

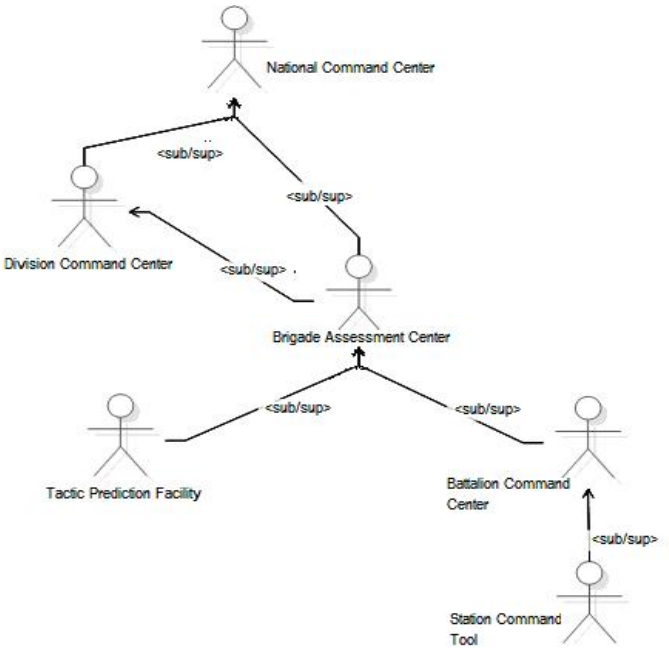


Figure 16. KAMA Command Hierarchy Diagram with Redundancy

In “Develop Pointer Information” workflow, since initial fork node may only be activated if all the preconditions are satisfied i.e. all the input places filled with tokens and assuming that task can not read transition guard values from the task context, initial fork node will never be activated. Hence the task-flow shall immediately fall in a deadlock. This issue has been identified as a behavioral semantic incorrectness based on UML’s control flow semantics.

In “Watch Mission Region” workflow, since the entity “Identification/recognition data” is an input to the task “Locate Allied Forces” only after being produced by “Search the Region” task, the fork node has no effect on the flow. This has been identified as a redundancy due to control flow semantics of UML and KAMA.

In “Develop Pointer Information” workflow; since the flows coming out of the fork node terminates with a decision node without a merge node; either of the activities in the diagram may terminate abruptly leaving dangling references. Although the usage of this pattern may be intentional or non-intentional, we have identified this issue as dynamic semantic deficiency.

B. Inter-diagram Inspection

The initial version of inter-diagram inspection tasks in Table 7 were performed and several issues were identified. For instance, during our check ontology diagrams vs. the set of task-flow diagrams, we identified 9 entities used in task-flow diagrams but not defined in ontology diagrams. As the result of the vertical property checks, we identified 29 issues. According to an identified refinement property for KAMA, a sub task-flow should show main task-flow entities in higher or at least equivalent level of detail. As an example, consider the models in Figure 17. “Develop Communication Information” task-flow is a sub task-flow of “Develop Pointer Information”. However, although the output entity “Communication Intelligence Data” exists in “Develop Pointer Information” main task-flow, associated or refining entities are not shown at all in “Develop Communication Information” sub task-flow. So this has been identified as incompleteness.

4.3.4. Discussion and Findings of Case Study 1

The first case study, was an exploratory study focusing on the first and the second case study research questions. We had the opportunity to explore what kind of properties are desired in a conceptual model and how can we proceed to detect defects. For these purposes, by exploring conceptual models, by adopting properties from the formal verification literature and by exploring the possible semantics for graphical notations, we were able to identify an initial set of desirable properties for KAMA conceptual models. Based on this experience,

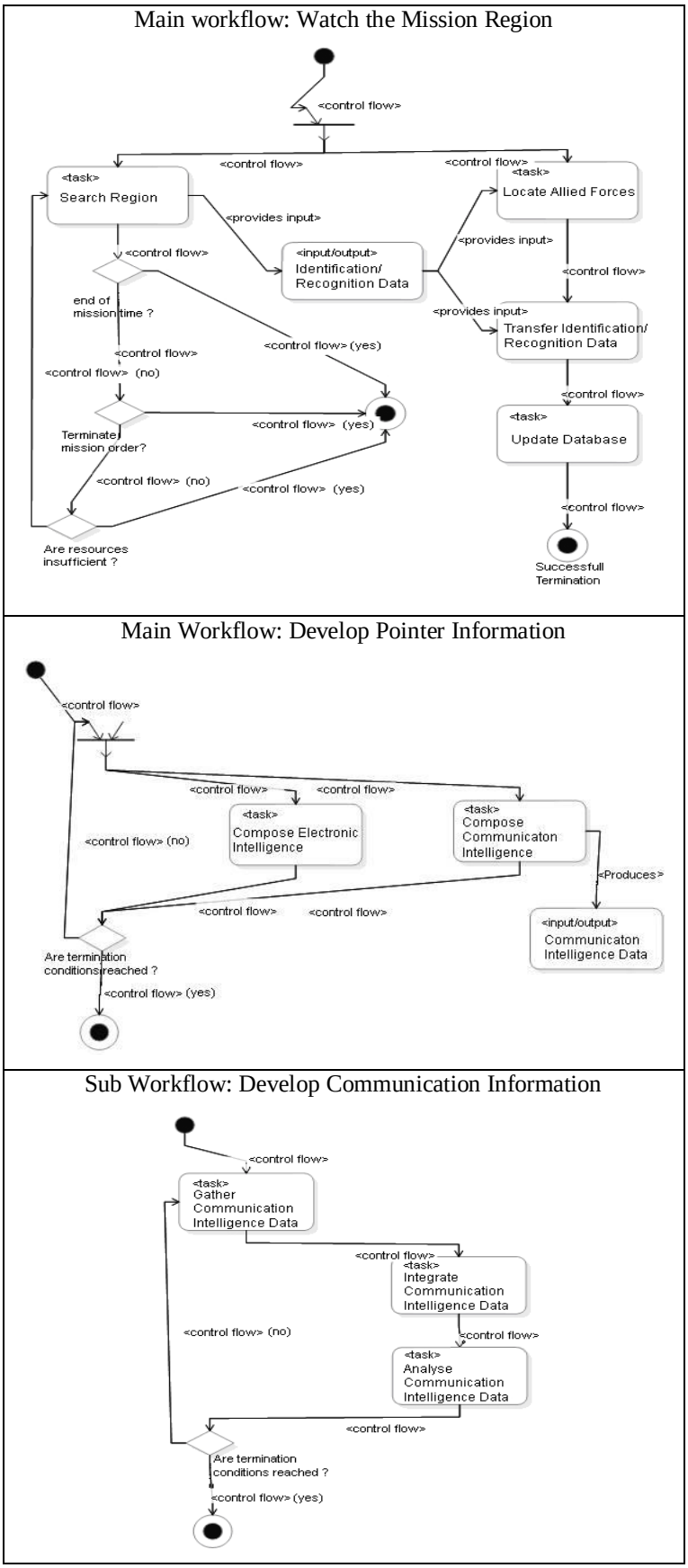


Figure 17. Defective KAMA Task-flow Diagram Examples

we have formulated a systematic property identification framework. Afterwards, by using the developed framework, we have further improved this set. These properties were summarized in section 3.2.4.

For the second research question, we have explored the verification methods in the literature and concluded that for our purposes formal approaches had drawbacks. So instead, as we defined desirable properties in accordance with property identification framework presented in section 3.1, we have gradually defined inspection tasks for verification. In fact, we were able to define the inspection process on the fly, based on the conceptual modeling notation and models at our dispositions.

The inspection performed as part of the case resulted with 85 issues identified. The issues found are documented in a technical report (Tanriover and Bilgen, 2008There were 85 issues defined as a result of this inspection. 10 of them were identified as major, 7 of them as moderate and 68 of them as minor. Major issues consisted of the following type of deficiencies; 6 semantic deadlocks in the task-flow diagrams, 3 improper usage of fork nodes which may result in dangling references. 29 moderate issues included incompleteness dues to inter-diagram refinement and dependency checks and improvement suggestions for the notation such as the need for a different symbol for tasks that have sub-tasks.

Expert opinion was used for validation of the issues identified in the inspection. For this purpose an inspection meeting was done after the defect detection phase. The inspector, modeler, two experts participated to the meeting. The two experts agreed that, although some of the 85 of issues signaled minor problems and some of them were not definitive defects, 39 of the identified issues included behavioral defects and are agreed to be subtle and not easy detect in ad-hoc reviews. 17 of these issues were agreed to be definitive defects and 22 issues were identified as incompleteness. It is also agreed that, these could be also categorized as defects upon the intended purpose of the conceptual modeling and CM modeling process used. As a result, we concluded that the inspections, defined based on the framework and the process presented in chapter 3, can be used to detect the desirable properties for CM's.

On the other hand, the application of the inspection to the model took around 24 man hours and 85 issues were detected, Hence, we concluded that the inspection approach is applicable with reasonable effort and worked out quite well to detect defects in conceptual model in KAMA notation. After the first case study, in the mean time we have focused on the semantic property checking and improved the inspection process accordingly. The final version of the inspection process is presented in section 3.3

The study also revealed that; semantic errors cannot be detected merely by the constraints defined in language definition in (Karagöz & Demirörs, 2008) hence KAMA tool is not sufficient and behavioral diagrams are more prone to errors than the structural diagrams.

Note that, the issue classification phase of the inspection process has been performed only in the first case study and was useful for orienting the issue validation in the inspection meeting.

4.3.5. Improvements Done After the Case Study 1

Since the initial inspection process used was a preliminary version of the process defined in Section 3.3 of this thesis, there were two important differences with the final version. The first one was in the scope of structural inspection tasks. The structural deficiency patterns were not defined. In the initial version, instead of using deficiency patterns the inspector was trying to find conflicting constraints, checking class consistency and checking cardinality constraint conflicts in an ad-hoc manner. We have realized that in this way the deficiency identification effectiveness was purely dependent on the inspector's verification experience. Even with the same inspector, since any guidance was not available the same kind of defects was not identified consistently, even though they existed in the model's different diagrams. Based on these observations, we have developed the set of deficiency patterns listed in Section 3.2.2 to guide the inspector in structural deficiency check.

For validation of the structural deficiency patterns identified in the defect detection phase, 3 individual meetings with 3 different experts was done. During these meetings the experts are presented with structural deficiency examples and they agreed that these issues pointed to redundancies and contradictions in the structural perspective of conceptual models. They also agreed that these type of redundancies and contradictions are not easy to detect and deficiency patterns could help the inspectors to detect these type of issues.

The second improvement to the initial version was made in the scope of inter-diagram properties. In the initial version of the inspection we had identified horizontal and vertical properties either based on dependency concept or refinement concept alone in an ad-hoc manner. After the case study, we have developed the inter-diagram property identification process described in Section 3.1. This process is formulated in such a manner that it can be used to identify interview properties for a given domain specific notation. The subtle semantic inter-diagram properties can be identified. We have used the process defined systematically to identify inter-diagram properties for KAMA models. Last three of the

properties in the inter-diagram inspection tasks were added to the inter-diagram inspection phase.

There were also many improvement suggestions for the notation definition. The process of identification of the desirable properties helped also to reveal underspecified aspects of the KAMA notation. These suggestions were included in the notation definition.

4.4. Case Study 2

The second case study focused on the second and third questions mentioned in Section 4.2, which aimed to evaluate the effectiveness of the approach with a conceptual model developed in a real life setting. After the first case study, in the mean time, we had focused on the semantic property checking and improved the inspection process accordingly. Hence there had been changes, refinements and improvements on the initial inspection process for semantic property checking. The purpose of the case study was to explore and evaluate the effectiveness of the final approach.

4.4.1. General Setting

The conceptual model in case study 2 was developed using Enterprise Architect v6.5 case tool. Apart from concrete syntax, by the help of the profiling mechanism the modeling elements were extended for the UML 2.1 modeling elements. The tool was used in such a manner that it provided a user interface to define all KAMA diagram types and related modeling elements.

The mission space view included 70 missions represented in 21 mission space diagrams. The topmost mission space diagram included 7 missions and in order to decompose the model mission space diagrams were developed for these highest level missions. Task-flow view is represented with 397 tasks in 45 task-flow diagrams. There are a total of 95 entities that were grouped in 15 logical packages and represented in 16 entity ontology diagrams. 25 entity state diagrams that represent the behavior of complex entities were developed.

Before the inspection was performed, the model in case study 2 had already been reviewed over 2 days by one expert. Also, a review meeting with the participation of 6 members of the development team was held. Later on, the conceptual model was subjected to a walkthrough that took five days. 4 engineers from the conceptual model development team and 3 from the acquirer organization joined the meetings in this third phase. There were 150 issues identified during these meetings. The issues identified were related with task-flow diagrams, incompleteness regarding entities, additional attributes and capabilities to the entities,

definition of roles and actors. Our inspection-based verification was applied after all these three review activities were realized.

4.4.2. Case Study Organization

This section presents the organisation for the second case study for validating the inspection approach. The objective is to explore applicability and effectiveness of the inspection approach for detecting defects in conceptual models of the mission space.

Expert: Responsible for validating the semantic issues during face to face meetings.

Inspector: Responsible for performing the verification activities.

Domain Expert: Responsible for validating and resolving semantic issues in the conceptual model.

The conceptual model was to be verified with the process presented in chapter 3 by an inspector who has UML modeling and software verification experience. Because we were unable to arrange meetings with the SME, we have used an Expert for validating the semantic issues which required SME validation.

The inspection process defined in Section 3.3 of this thesis will be used as a basis for the case study. The expected outputs for the case study are corrected conceptual model and the verification report. Main sources of evidence and data of case study are documentation, participant observation and meeting minutes.

The modeling tool used for developing the model is the EA 6.5 (Enterprise Architect, 2006) tool is also used as the medium used during the inspection.

4.4.3. Conduct of the Case Study 2

The inspection process defined in Section 3.3 of this thesis was used as a basis for the case study. The details of the findings and discussion can be found in Appendix D of this thesis.

Before each intra-diagram inspection, the validation function of Enterprise Architect v6.5 was executed on each diagram with wff and syntactic rules checks. The tool's standard validation function which included syntactic, wff and other checks signaled no errors. Then, the verification tasks proposed in Section 3.3 was performed. Note that, the issue classification phase of the inspection process was not performed in this case study.

During the inspection the model tree browser is used that helped the inspector to manage the browsing (which may sometimes be rather complex) needed for inter-diagram verification tasks. The execution of inspection tasks has been tailored for the conceptual model. We have

used a perspective oriented inspection strategy for effectiveness reasons. For this, we have conducted the inter-diagram inspection tasks not as a standalone activity but rather decided to perform the inter-diagram task related to each of diagram type just after finishing the intra-diagram inspection for that diagram. The order of execution of the verification tasks was as follows:

1) Entity – ontology diagram has been verified with class like diagram inspection tasks, inter-diagram task 2, task 9 and task 10 has been performed in the former order.

2) Organization diagrams have been inspected with class like diagram inspection tasks.

3) On mission space diagrams, mission space diagram inspection tasks, inter-diagram task 1, task 8 and task 6 have been performed. Task 6 was not conducted since hierarchical refinement is used with structured activity nodes for specialization of missions. The leaf extending missions was modeled with in activity diagrams, with structured activity nodes. By definition, the property searched in task 6 is satisfied.

4) Task-flow diagrams have been inspected with the inspection tasks for task-flow diagrams and only inter-diagram inspection task 3, task 4 and task 5. In the conceptual model of the second case study, the tasks are associated to entities. Hence, during task 4, we have checked the refinement relation such that only entities composing or specializing the upper entity in the ontology can be associated to sub tasks of the structured task node. That is the assigned entity is decomposed to sub entities and assigned to tasks in the refining sub task-flow diagram, in accordance with the generalization or composition hierarchy defined in the entity ontology view. During the inspection, the facility of the EA 6.5 tool to view the class hierarchy tree is used to obtain all the lower level entities transitively based on both aggregation and generalization relations. Note that only first sub level has been checked for each diagram, the deeper levels of activities are not checked to avoid duplicate checks. Because the lower level activities are verified with the same inspection tasks. In this way only one sub level of refinement check for each activity diagram will cover the whole model.

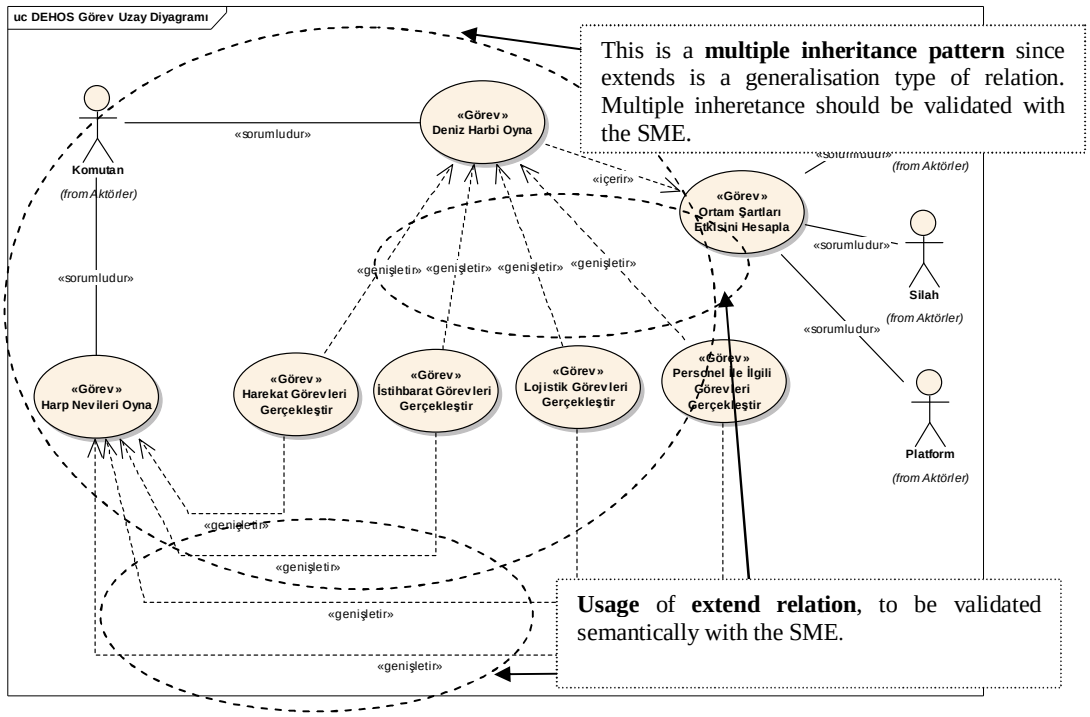


Figure 18. A violation of the multiple inheritance pattern.

There have been a few defects which were detected based on structural deficiency patterns. As an example, in Figure 18 shows a violation of the multiple inheritance pattern. Since extends is a generalization type of relation. Multiple inheritances should be validated with the SME. Note that, as the definition of the patterns is not formally given and use case patterns are not explicitly defined, this pattern could be identified as a structural issue by the inspector. This example shows that with the guidance of patterns initially defined using natural language, the inspectors are able identify similar deficiencies which are not previously formally specified.

Another example is the violation of generalization with aggregation patterns which can be seen in figure 18. This pattern is signaled as warning, and can be considered as an indication of a validation issue.

We have identified many issues with task-flow diagrams. Figure 19 for example provides an example. The “yiyecek su miktarı” decision node may cause the activity to go in to a deadlock when it evaluates to true. As a second defect “yakıt miktarı” decision node causes the task-flow to deadlock

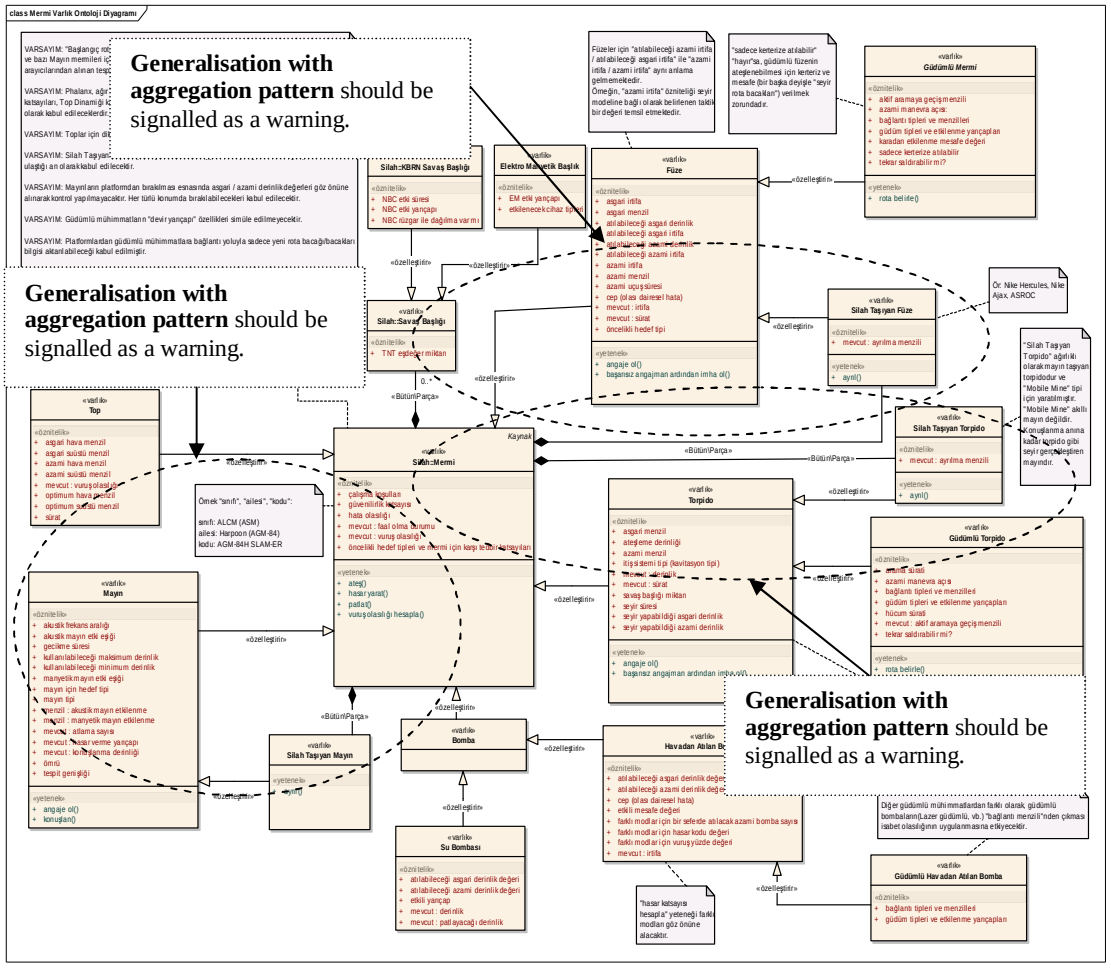


Figure 19. A violation of generalization with aggregation pattern

For example, with inter-diagram inspection task 4, we have checked the refinement relation such that only entities composing or specializing the upper entity in the ontology can be associated to sub tasks of a structured task node. That is the associated entity is decomposed to sub entities and assigned to tasks in the refining sub task-flow diagram, in respecting the generalization or composition hierarchy defined in the entity ontology view. In the case study, the facility of the EA 6.5 tool to view the class hierarchy tree is used to obtain all the lower level entities transitively based on both aggregation and generalization relations. By applying this process a refinement defect is identified in Figure 21. In this case although the “bindirme yap” task-flow is properly refined into a sub task-flow, its associated entity “platform” is not properly refined because when we check the entity hierarchy the “Komutan” entity is neither a subtype nor a part of platform entity.

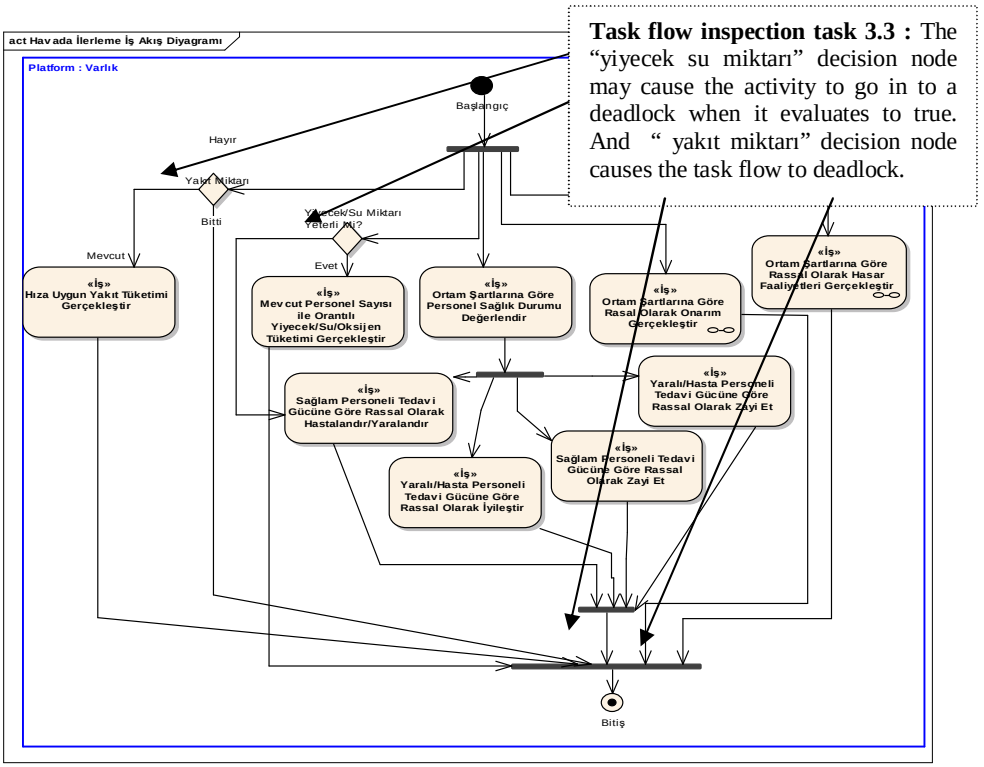


Figure 20. Task-flow defect identified with task-flow inspection task 3.3

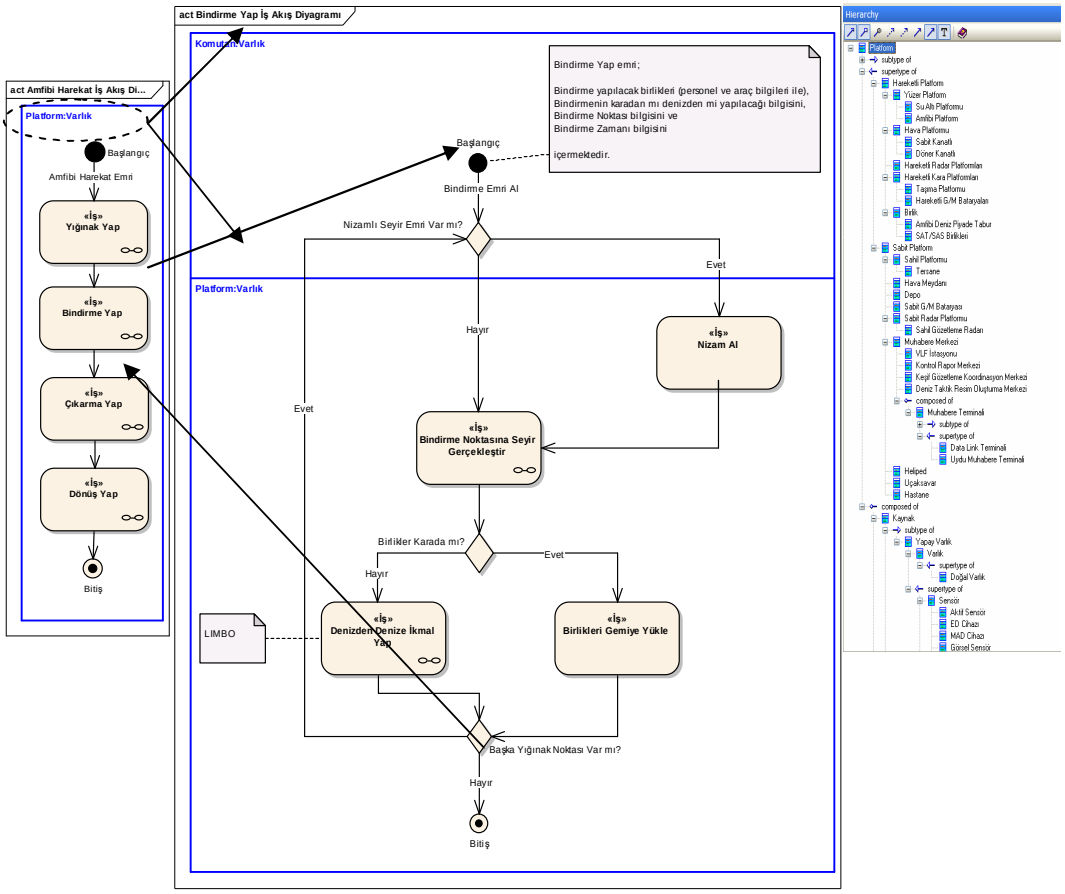


Figure 21.Task-flow defect identified with inter- diagram inspection task 4

Table 9: Metrics Collected During the Case study 2.

CONCEPTUAL MODEL	*Mission Space Diagrams*	*Entity Ontology Diagrams*	*Task-flow Diagrams*	*Entity State Diagrams*	*Organization Diagrams*
Total # of important model elements (concepts)	70 missions	97 entities	397 tasks	174 states	8 actors
# of diagrams in the model	21	16	45	25	1
# of diagrams inspected	21	16	20	-	1
Inspection tasks	Structural diagram + mission space diagram verification tasks + inter-diagram task no: 1.	Structural diagram verification tasks + inter-diagram task 2,	Task-flow diagram verification tasks + inter-diagram task 3, task no: 4, task no:6, task no:7.	Inter-diagrams task no: 9, task no:10.	Structural diagram verification tasks
Limitations	-	-	Inter-diagram task no:6 is performed for 15 activity diagrams.	Tasks are performed for 5 state diagrams only.	-
# of defects detected	19	5	27	7	-
Time per diagram : *Detection time + recording time*	3-10 minutes	3-10 minutes	3-25 minutes	5 – 30 minutes	3 minutes
Total time: ***Detection time + recording time***	3 man / hrs	4 man / hrs	10 man/ hrs	3 man /hrs	3 min.
	TOTAL # SEMANTIC ISSUES DETECTED: 56, more than 30 non-trivial				
	TOTAL TIME SPENT ON INSPECTION is 20 hrs. + 8 hrs for inspection preparation. *Inspection preparation*: Studying and understanding the inspection process, browsing through the model for initial familiarization, time spent due to limitations caused by differences of the usage of notation, adopting the inspection process for the model and the capabilities of the case tool: 8 hrs.				

4.4.4. Findings of the Case Study 2

The second case study focused on the third question mentioned in Sec. 4.2, which aimed to evaluate the effectiveness of the inspection approach with a conceptual model developed in a real life setting. After the first case study, in the mean time we had focused on the semantic property checking and improved the inspection process accordingly.

In order to respond the third question, we applied the inspection process to a conceptual model that had gone through extensive reviews (3 reviews). As already explained in section 4.2 the only comparable method to ours is ad-hoc reviews. The conceptual model under consideration had gone through ad-hoc verification and validation review and inspections conducted by experts. During this process 150 semantic and syntactic issues had already been identified and corrected. These experts had experience with both UML language and the domain. Even though the conceptual model was corrected and accepted to be valid, we were able to identify more than 30 semantic issues by conducting our inspection process which had not been identified in previous reviews. Note that, some of these issues are to be used as indications of validation issues to be resolved by the domain expert.

One important observation in the case study was “the model tree browser” of the EA 6.5 tool proved to be very helpful for inter-diagram verification tasks. For the specific model of the case study, during the pre-inspection phase, we were able to tailor the order of execution of inspection tasks to be more effective. We have conducted the inter-diagram inspection tasks not as a standalone activity but rather decided to perform the inter-diagram task related to each of diagram type just after finishing the intra-diagram inspection for that diagram. Instead once a perspective is inspected, all the related tasks to that perspective is performed .This slight adaptation of the process has improved the inspection effectiveness, because in this way the inspector does not have to consider the same diagram twice for inter-view and intra-view tasks.

Table 9 shows metrics collected during the case study. Based on the collected metrics, we can conclude that given the effort spent, type of defects detected and defect rate our inspection is effective in detecting semantic defects because none of these issues had been identified by the 3 stage V&V process applied beforehand.

CHAPTER 5

CONCLUSIONS

Simulation model conceptualization in domain specific notations is prone to incorrectness, incompleteness, and inconsistency and coherence problems. In addition to completeness and correctness of translation of the problem frame to conceptual representation (validation), ambiguity inherent in semi formal domain specific notations and the support for multiple views and may further increases incompleteness, inconsistencies, incorrectness and redundancies in models (verification). Furthermore, since conceptual modeling is mostly related with the problem definition phase, any defect injected at this phase will cost too much effort and time sometimes even leading to unrecoverable situations. Especially, semantic property checking is a major problem, as many syntactic errors can be eliminated through case tools. Since conceptual models are in general not executable, therefore, it is not possible to verify and validate them with testing techniques..

Hence, for error reduction at conceptual modeling phase a systematic, holistic and practical approach is needed. However, as discussed in Chapter 2 of this thesis, related methods and approaches fall short to completely respond to this need.

In this research, in order to respond to this need, we have developed an inspection approach.

This chapter summarizes the contributions of this research, derives conclusions from the study and suggests future research directions based on the findings.

5.1 Contributions

The main contribution of this research is the development a systematic, holistic and practical inspection process for verification of semantic properties to assure the quality of conceptual

models in a notation derived from UML. Instead of taking a top down approach and tailor available verification methods, we have proceeded in a bottom up fashion, starting from the modeling language definitions and actual models. We have shown that such an inspection process could be developed and customized depending on the notation and conceptual modeling process. In our particular case the domain specific notation was KAMA. Through the case studies, we have also shown that a systematic and holistic inspection approach, rather than using formalisms, can also provide significant practical results as presented in the previous chapter.

In order to develop the inspection approach we had to tackle the related research problems. There are three contributions of this study that address these problems.

Firstly, we have proposed a process for identifying desirable properties of conceptual models in domain specific notations. This process is based on four categories of desirable properties: Syntactic, semantic, horizontal and vertical. Using this framework and properties proposed in the UML verification literature and meta model definition, we were able to identify desirable semantic properties for the KAMA notation. As a side product, some of the properties identified has been used to improve formality of the definition of the KAMA notation as well. Although some of the studies related to property checking mention briefly the need for property identification and consider various types of desirable properties for UML models, they are not founded on a property identification framework that considers domain specific notations.

Secondly, for checking mostly semantic desirable properties, we have defined a practical inspection process composed of simple verification tasks. We developed tasks for semantic properties as many syntactic errors can be eliminated through case tools. By using a holistic approach, rather than a partial approach, we developed tasks for different type of diagrams and inter-diagram properties as well. These verification tasks are formulated in natural language in such a manner that an inspector can manually perform a set of tasks to identify most of the semantic defects. None of the works in the inspection literature provide any guidance on how to check semantic properties such as class consistency, refinement consistency and properties in behavioral diagrams in domain specific notations. Especially, through the case studies we have observed that, with the defined tasks for detecting properties derived from Petri nets, we were able detect many deadlocks, livelocks and dangling tasks. This study also showed us that, without using any verification formalism, the application of the inspection process to models of a typical mission scenario, in fact revealed defects many of which were non-trivial.

In contrast to existing approaches, we developed an inspection process by integrating concepts derived from formal approaches. We precisely defined which tasks need to be addressed in an inspection to find semantic defects, thus we fulfill loosely defined steps in inspections with concrete content. By using inspection to detect defects, we prevent the drawbacks of the formal approaches such as the complexity and the traceability problems.

Thirdly, based on ontological interpretation for the structural view of conceptual models, we developed structural deficiency patterns which have not been proposed before. We were able to use these patterns as a means to detect typical structural semantic issues. These issues signals validation issues, hence should be resolved in validation with the subject matter expert.

In summary, the work presented in this thesis takes a systematic and holistic but a less formal approach. The differences from the works in the literature can be stated as follows: Although some of the studies related to property checking mention briefly the need for property identification and consider various types of desirable properties for UML models, they are not founded on a property identification framework that considers domain specific notations. Secondly, in this study, based on ontological interpretation for the structural view of conceptual models, we developed structural deficiency patterns which have not been proposed before. Thirdly, unlike Travassos et al. (2002) and Unhelkar (2005), we focused on semantic properties and developed verification tasks rather than validation tasks. Finally, our main artifact is conceptual models rather than software design models.

5.2 Limitations and Future Work

By using a holistic approach, rather than a partial approach, we developed tasks for different type of diagrams and inter-diagram properties as well. However, tdeishe set of properties can be augmented, depending on the intended purpose of the conceptual models, hence the acceptability criteria. We believe that the set of properties are adequate for conceptual models developed for a reuse library. However, if the models will be used directly in FEDEP for instance, the properties derived from platform requirements should also be considered in the customized inspection process.

With the approach presented, we recognize that many of the subtle issues especially in structural diagrams may not be detected, since we only provide a set of common defect patterns. On the other hand, for behavioral diagram checks, we only guide the inspector by means of inspection tasks which facilitate detecting defects in relation to desirable properties. Hence, we do not aim at a complete verification of the model.

This approach provides only a guidance to handle the complexity of the challenge of verification of properties of models developed in notations derived from UML. When the number and complexity of diagrams participating in refinement or dependency relations increases, manual inspection of inter-diagram properties becomes difficult and time consuming. Hence, to what extent the approach is applicable to large scale complex models is still an open issue. However, as conceptual models are incrementally developed, applying the proposed inspection process in each iteration may help remove defects and result in increased model quality. Also, other drawbacks which may be attributed to the informality of the approach obviously still remain such as its high dependence on the expertise of the inspector.

There are two levels at which desired properties can be defined when a domain specific notation is used: They can be defined on the meta-model level and on the model level. In our approach properties defined on the meta-model level are considered. The meta-model-level properties provide only general checks. Model-level properties are related to the domain of interest and modeler in course of developing the conceptual model has to define these properties. In order to accomplish a full verification of the conceptual models, properties should be defined and applied at the model level as well.

The lack of comparison with other review methods in terms of type of issues can be considered as a limitation about the validity of the inspection approach. We have not conducted such experiments. However, as already stated in section 4.2, because of the differences of the objective (we focus on semantic defects) and object (we consider CMs) of other review methods and ours, other review methods may not be directly comparable to ours. The only comparable review method is ad-hoc review and in the second case study, we happen to provide implicitly, a limited comparison of results.

Also the identification process of structural deficiency patterns may be criticized and validity of the patterns can be questioned. We have developed these patterns based on our observations, defect types identified in our studies as discussed in detail in Section 3.3. The validity of defined patterns is partly achieved by expert opinion and by our successful applications within the course of this research. However, although we call them patterns, we have not proven, empirically these deficiency examples to occur repeatedly in empirically meaningful number of CMs. Along with this limitation, we recognize also that, there can be other useful deficiency patterns that are not identified by us but could be identified if empirically meaningful number of CMs are investigated for mostly done structural errors. However, due to also operational reasons, we did not have access to many CMs to conduct

such an identification process.

Another limitation of the inspection approach is the lack of emphasis on managerial and organizational dimensions. Although, as described in Section 2.5 the findings in recent research shows that the emphasis in inspections should be put to technical (defect detection) dimension, Laitenberg & DeBaud (2000) argues that an inspection method should also define organizational and managerial dimension. In these dimensions, the issues such as, how to plan the inspection, the optimal team size: the size of the model vs the size of the team, how to organize meetings: roles in the meeting conduct of meetings, the reporting the inspection should be addressed. Although in the case studies we have defined main roles and inputs and outputs, we have not detailed the inspection organization.

Furthermore, the tool support dimension is not adressed. Tool dimension describes how inspections can be supported with tools. Although we explored various tools as discussed in section 2.4, we have not investigated how they can support each verification task within our inspection approach. We observed that a single tool will not be enough to support the verification tasks proposed in this study but rather a set of tools should be identified. In general, environments such as(Meta Edit, Open Architectureware and GME can be used to check properties related to syntax and simple consistency rules of the domain specific notation.

On the other hand tools with good OCL support such as OCLE (2006), Poseidon (2005) can be also used. Since, we have identified a set of desirable properties in natural language for KAMA models in this research, these tools now can be used for especially checking interview dependencies. However, as already discussed in Section 2.1 it is not practical to use OCL for more complex semantic properties such as deadlocks. For task-flow inspections, Petri net analysis tools may be helpful if the view is too complex and critical. We also foresee that for structural view verification tasks ontology analysis tools such (Compatangelo & Meise, 2002) as EER- conceptual tool may be very helpful.

However, we believe that unless really needed the usage of a mixture of tools will not be effective. Because conceptual models are used primarily as a means of communication, "Conceptual" implies human conceptualization, which inherently implies tractable abstraction levels and size". Hence, tool support is not crucial, but rather the verification results must also be used as a means to identify and resolve validation issues. It is more cost effective to integrate the verification tasks with the validation tasks which require human (in simulation domain subject matter expert interpretation) interpretation hence mostly a human

activity. Nevertheless, we are planning to further investigate existing tools to support each of the verification task.

A sub research direction may be to develop an algorithm to automatically detect deficiency patterns in structural views. For this purpose an improved version of either the algorithm in Alleno & Porres (2005) and Xing & Stroulia (2007) can be developed. Another possibility is development of a graph isomorphism algorithm. However, it is known that to check weather two graphs are isomorphic has exponential worst time complexity. But in practice there exists efficient algorithm for around 100 of vertices. So in principle the graph isomorphism algorithm can be used for deficiency pattern checking in structural views of KAMA conceptual models.

Last perspective missing in the current form of our methodology is the consideration of the risk perspective. To make sure that the simulation model is fit for purpose in a cost effective way, V&V activities have to be focused on the most important aspects of the simulation conceptual model. MS community had for long time acknowledged the need for a risk based V&V process and the concepts has found grounds both in REVVAI/II and GMVVA (Generic Methodology for VV&A). The paper by Brade & Köster (2001) presents an example to risk based V&V. They define V&V levels or credibility levels that are related to the criticality of the user's simulation based decision. They claim that, this type of goal-driven V&V to achieve the desired V&V level promises to increase efficiency and effectiveness of M&S V&V significantly.

Some hesitation may arise about semantic desirable properties, if they are syntactic or indeed semantic. In a seminal paper Rapaport (1995) claims that syntactic definition is essential and semantic definition is a syntactic definition itself. Hence, syntactic definition may be claimed to include semantics. However, we believe that this discussion is beyond the scope of this work.

REFERENCES

Aalst, W. (2002). *Workflow Management Systems - Models, Methods, Tools*. Cambridge, Mass: MIT Press.

Allen, M. & Porres, I. (2005). *Version Control of Software Models, in Advances in UML and XML-Based Software Evolution.* London: Idea Group Publishing.

Amalio, N. & Polack, F. (2003). Comparison of Formalisation Approaches of UML Class Constructs. *In Z and Object-Z, In International Conference of Z and B Users (ZB 2003), Lecture Notes in Computer Science 2561*. Springer-Verlag.

Ambler, W. S. (2005). *The elements of UML 2.0 style*. USA: Cambridge University Press.

Andre, P. , Romanczuk, A. , Royer, J.C. & Vasconcelos, I. (2000). Checking the consistency of UML Class Diagrams Using Larch Prover. In T. Clarck (Ed.), *Proceedings of the third rigorous object-oriented methods workshop, BCS and WIC.*

Apvrille, L. , Courtiat, J.P., Lohr, C. & De Saqui-Sannes, P. (2004). TURTLE : A Real-Time UML Profile Supported by a Formal Validation Toolkit. *IEEE Transactions on Software Engineering, 30 (7),* 473–487.

Argo (2002). An open source UML case tool. Retrieved January, 1996, from http://argouml.tigris.org/

Balci, O. (1998). Verification, Validation, and Accreditation. *In Winter Simulation Conference proceedings.* Washington, D.C.: ACM.

Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sorumgard, S. & Zelkowitz, M. V. (1996). The Empirical Investigation of Perspective-Based Reading. *Empirical Software Engineering Journal, 2(1),* 133-164.

Berardi, D., Calvanese, D., & De Giacomo, G. (2005). Reasoning on UML class diagrams. *Artificial Intelligence, 168,* 70-118.

Berenbach, B. (2004). Evoluation of large, complex, UML analysis and design models. *Proceedings of 26th International Conference on Software Engineering,* ICSE.

Boehm, B. W. (1984). Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, vol. 1, no. 1, pp. 75-88.

BOM (2006). Base Object Model. Retrieved at October, 2007, from http://www.boms.info/

Boman, M., Bubenko, J. A., Johannesson, P., and Wangler, B. (1997). *Conceptual Modelling*. London: Prentice Hall.

Briand, L., Labiche, Y., & O'Sullivan, L. (2003). Impact Analysis and Change Management of UML Models. Technical Report SCE-03-01. *In Proceedings of IEEE International Conference on Software Maintenance (ICSM)*, Carleton University.

Caplat, G. (2006). "Sherlock Environment". Retrieved April, 2006, from http://servif5.insa-lyon.fr/chercheurs/gcaplat/

Chang, L. P., Jong – Li, D., Lin – Yi, P., & Muder, J. (2005). Management and Control of Information Flow in CIM Systems Using UML and Petri Nets. *International Journal of Computer Integrated Manufacturing, 18,* 2 - 3.

Compatangelo, E. & Meisel, H. (2002). Intelligent support to knowledge management: conceptual analysis of EER schemas and ontologies. In *Internal report, Dept. of Computing Science*, UK: University of Aberdeen. Retrieved October, 2007 from http://www.csd.abdn.ac.uk/research/conceptool/

Csertan, G. Huszerl,I. Majzik,Z. P. & Patar, A. (2002). VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. *In Proceedings of the 17th International Conference on Automated Software Engineering*. Edinburgh, UK: IEEE.

Damm, W. & Harel, D. (2001). Breathing life into message sequence charts. *Formal Methods in System Design, 19,* 45-80.

DMSO (Defense Modeling and Simulation Office) (1997). *Conceptual Models of the Mission Space (CMMS) Technical Framework, USD/A&T-DMSO-CMMS-0002 Revision 0.2.1*.

DMSO (Defense Modeling and Simulation Office) (2000a). *Verification, Validation and Accreditation (VV&A) Recommended Practices Guide.* Retrieved December, 2007 from http://vva.dmso.mil/

DMSO (Defense Modeling and Simulation Office) (2000b). *Conceptual Model Development and Validation*. Retrieved December, 2007 from www.msiac.dmso.mil/vva/Special_Topics/Conceptual/conceptual-pr.PDF

De Lara, J., & Vangheluwe, H. (2002). AToM3: A Tool for Multi-Formalism Modelling and Meta-modelling. *In Proc. FASE'02, Springer LNCS 2306*. Retrieved April, 2006, http://atom3.cs.mcgill.ca

Dupey, S., Ledru, Y. & Chabre-Peccoud, M (2000). An Overview of RoZ : A Tool for Integrating UML and Z Specifications. *In 12'th Conference on Advanced information Systems Engineering - CAiSE'2000*, volume 1789, *Lecture Notes in Computer Science*, Stockholm, Sweden: Springer-Verlag.

Egyed, A. (2006). Instant consistency checking for UML. *In Proceedings of the 28th International Conference on Software Engineering (ICSE)*, Shangai, China.

Eishuis, R. & Weringua, R. (2004). Tool support for verifying activity diagrams. *IEEE Transactions on Software Engineering, 30 (7),* 437-447.

Engels, G. , Küster J., Heckel, R., Groenewegen, L. (2001) A methodology for specifying and analyzing consistency of object-oriented behavioral models, ACM SIGSOFT Software Engineering Notes, v.26 n.5, Sept. 2001.

Enterprise Architect 6.5 (2006). *UML case tool*. Retrieved October, 2007 from http://www.sparxsystems.com.au/

Everman, J. E, & Wand, Y. (2005). Ontology based object oriented domain modeling: fundamental concepts. *Requirements Engineering, 10,* 146–160.

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development, *IBM Systems Journal, 15(3),* pp. 182-211.

FEDEP (2000). *Recommended Practice for High Level Architecture (HLA), Federation Development and Execution Process (FEDEP)*, IEEE 1516.3.

Fowler, M. (2000). *UML Distilled*. - 2nd ed., Upper Saddle River, NJ: Addison-Wesley.

Gagnon, P., Mokhati, F, & Mourad, M (2008). Applying Model Checking to Concurrent UML Models, *Journal of Object Technology*, 7 (1), 59-84, retrieved February, 2008 from http://www.jot.fm/issues/issue_2008_01/article1/

Gemino, A. & Wand, Y. (2004). A framework for evaluation of conceptual modeling techniques. *Requirements Engineering, 9,* 248–260.

Gilb, T. & Graham, D. (1993). *Software Inspection*. Mass.: Addison-Wesley Publishing Company.

Gitzel, R. (2006). *MDSD using a Meta-model-Based Extension Mechanism for UML*. Munich : Peter Lange Publishing.

GME (Generic modeling environment) (2006). Retrieved November, 2006, from http://www.isis.vanderbilt.edu/projects/gme/

GMVV&A (2007). *General Methodology for Verification and Validation and Accreditation*. Retrieved October, 2007, from http://www.sisostds.org/

ICOM (2000). *A prototype design tool for Intelligent Conceptual Modelling.* Retrieved December, 2007, http://www.cs.man.ac.uk/˜franconi/icom/

Insfran, E., Pastor, O. & Wieringa, R. (2002). Requirements engineering-based conceptual modelling. *Requirements Engineering, 27(2),* 61–72.

ITOP (International Test Operations Procedure) (2004). *General Procedure for Modeling and Simulation Verification and Validation Information Exchange*. ITOP 1-1-002, WGE 7.2.

Johnson, P. & Tjahjono, D. (1998). Does Every Inspection Really Need a Meeting. *Journal of Empirical Software Engineering, 3(1),* 9-35.

KAMA (2006). *A conceptual modeling tool for the mission space*. Progress Report II, General Staff Presidency, Turkish Armed Forces.

Karagöz, A. & Demirörs, O. (2007). Developing Conceptual Models of the Mission Space (CMMS) – A Meta-model Based Approach. *In Proceedings of Simulation Interoperability Workshop (SIW)*, SISO.

Karagöz , A. & Demirörs, O. (2008). *A Conceptual Modeling Notation.* Unpublished doctoral dissertation, Middle East Technical University, Ankara.

Killand, T. & Borretzen, J. (2001). UML Consistency Checking. *In Research Report SIF8094*, Trondheim: Norway, Institute for Datateknikk OG Informasjonsvitenskap.

Kim, S. & Carrington, D. (2000). A formal mapping between UML models and Object-Z specification and B. Volume.1878, *Lecture Notes in Computer Science,* Berlin: Springer-Verlag.

Knight, J. C. & Myers, E. A. (1993). An Improved Inspection Technique, *Communications of the ACM, 36(11)*, 51-61.

Kremer-Davidson, S. & Shaham-Gafni, Y. (2005). *UML 2.0 Model Consistency – The Rule of Explicit and Implicit Usage Dependencies*, Consistency Problems in UML-Based Software Development, Lecture Notes in Computer Science, Berlin: Springer-Verlag.

Kuzniarz, L., Reggio, G., Sourrouille, J. L., & Huzar, Z. (2002). Research Report 2002:06. Workshop on Consistency Problems in UML-based software development, *In Workshop at the UML 2002 Conference*, Ronneby : Blekinge Institute of Technology. Retrieved November, 2006, from http://www.ipd.bth.se/consistencyUML/

Kuzniarz L., Huzar Z., Reggio G., Sourrouille J. L., Staron M. (2003). Research Report 2003. Workshop on Consistency Problems in UML-based software development II, Workshop Materials, Retrieved November, 2006, from http://www.ipd.bth.se/consistencyUML/

Lacy, L. W., Randolph, W., Harris, B., Youngblood, S., Sheehan, J., Might, R., & Metz, M. (2001). Developing a consensus perspective on conceptual models for simulation systems. Paper OlS-SIW-074, *In Proceedings of the Simulation Interoperability Workshop*.

Laitenberger, O. & DeBaud, J., M. (2000a). An Encompassing Life-Cycle Centric Survey of Software Inspection, *Journal of Systems and Software, 50(1),* 5-31.

Laitenberger, O., C., Atkinson, M. S., & El Em, K. (2000b) . An experimental comparison of reading techniques for defect detection in UML design documents. *Journal of Systems and Software, 53,* 183–204.

Law, A.M. & Kelton, W.D. (1999). *Simulation Modeling and Analysis.* 3rd Edition. New York: McGraw- Hill.

Lemmers, A. & Jokipii, M. (2003). SEST: SE Specifications Tool-set. *In Proceedings of Fall Simulation Interoperability Workshop*, SISO.

Leveson, N.G (1995). *Safeware: System Safety and Computers.* Addison-Wesley.

Lilius, J. & Paltor, I. P. (1999). vUML: A tool for verifying UML models. *In Technical report 272*, Turku, Finland: Turku Centre for Computer Science (TUCS).

Lindland, O.I., Sindre, G., & Sølvberg, A. (1994). Understanding quality in conceptual modeling. *IEEE Software, 11*, 2 ,42-49.

Litvak, B., S. Tyszberowicz, & A. Yehudia (2003). Behavioral consistency validation of UML diagrams. *In Proceedings of the 1st International Conference on Software Engineering and Formal Methods*, Brisbane, Australia: IEEE.

MacKenzie, G. R., Shulmeyer, G. G. & Yilmaz L., (2002), Verification Technology Potential with Different Modeling and Simulation Development and Implementation Paradigms, Invited Paper, *Foundations for V&V in the 21'st century Workshop (Foundations '02)*.

Marcano, R. & Levy, N. (2002). Using B formal specifications for analysis and verification of UML/OCL models, *In Workshop on consistency problems in UML-based software development, 5th international conference on the Unified Modeling Language*. Dresden, Germany: publisher.

Merriam Webster (2008). Online Dictionary, retrieved August, 2008, from http://www.merriam-webster.com/dictionary/concepts

Meta Edit (2007). *A case tool for domain specific software development*. Retrieved January, 2007, from http://www.metacase.com

Minas, M. (2002). Specifying Graph-like Diagrams with DiaGen, *in Electronic Notes in Theoretical Computer Science*, *72*, 2, Elsevier.

MOF 2.0 (2004). *Meta Object Facility core specification*. Retrieved December, 2005, from http://www.omg.org

Mota, E., Clarke, M., Groce A., Oliveira, W, Falcão, M. and Kanda, J. (2004). Veri Agent: An Approach to Integrating UML and Formal Verification Tools. *Electronic Notes in Theoretical Computer Science*, *95*, 111-129.

Murata, T. (1989). Petri Nets: Properties, analysis and applications. *Proc. IEEE, vol. 77.*

NATO (2007). Verification, Validation, and Accreditation of Federations, Retrieved November, 2007, from http://www.rta.nato.int/search.asp#MSG-019

Ober, I. (2004). *Harmonizing design languages with object-oriented extensions and an executable semantics*. Unpublished doctoral dissertation. Institute National Polytechnique de Toulouse, Toulouse, France.

OCLE, OCL Environment, LCI Team (2005). Computer Science Research Laboratory, Babes Boyls University, Romania, Retrieved December, 2006 from http://lci.cs.ubbcluj.ro/ocle/index.htm

Ohnishi, A. (2002). Management and Verification of the Consistency among UML models. In *Proc. WS15 Workshop on Knowledge-Based Object-Oriented Software Engineering* (KBOOSE), LNCS, Malaga, Spain: Springer.

Open Architectureware (2007) A platform for model driven development. Retrieved at October, 2007, from http://www.openarchitectureware.org/

Oscar, D., Juristo, N., Moreno, A. M., Pazos, j. & Almudena S. (2000). Conceptual Modeling in Software Engineering and Knowledge Engineering: Concepts, Techniques and Trends. *Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing Company.

Queralt, A. & Teniente, E. (2006). Reasoning on UML Class Diagrams with OCL Constraints, *Conceptual Modeling - ER* , LNCS, Berlin:Springer-Verlag.

Pace, D.K. (2000). Simulation Conceptual Model Development. In *Proceedings of the Spring Simulation Interoperability Workshop*. Retrieved November, 2005 from www.sisostds.org

Parnas, D. L. & Weiss, D. M. (1985). Active Design Reviews: Principles and Practice. In proceedings of *8th International Conference on Software Engineering*, (pp. 132-136), London, UK: IEEE Computer Society.

Porter, A. A. , Votta, L. G. & Basili V. R. (1995). Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment, *IEEE Transactions on Software Engineering, 21(6)*, 563-575.

Poseidon (2006). *UML Case Tool*. Retrieved October, 2006 from http://www.gentleware.com/

Rapaport, William J. (1995), Understanding Understanding: Syntactic Semantics and Computational Cognition. In James E. Tomberlin, ed., *AI, Connectionism, and Philosophical Psychology*, Philosophical Perspectives, Vol. 9, Atascadero, CA: Ridgeview, pp. 49-88; reprinted in Andy Clark and Josefa Toribio (1998), eds., *Language and Meaning in Cognitive Science: Cognitive Issues and Semantic Theory*, Artificial Intelligence and Cognitive Science: Conceptual Issues, Vol. 4, Hamden, CT: Garland.

Rational (2004). *Rational case tool.* Retrieved October, 2006 from http://www-306.ibm.com/software/rational/

REVVA 2 (2005). *VV&A Process Specification (PROSPEC) User's Manual*, v1.3. Retrieved October, 2007 from http://www.revva.eu/

Sargent, R.G. (1994). Verification and Validation of Simulation Models, in Winter Simulation Conference, Piscataway, NJ, USA: IEEE press.

Schinz,I., Toben, T., Mrugalla, C., & Westphal, B. (2004). The Rhapsody UML Verification Environment. *In procedings of Second International Conference on Software Engineering and Formal Methods (SEFM)*, 174-183, Beijing, China: IEEE.

SD Metrics.(2007) *List of object oriented design rules.* Retrieved December, 2007, from http://www.sdmetrics.com/LoR.html#LoR.

SEDEP (2007). *Euclid RTP 11.13.* Retrieved at December, 2007 from http://www.euclid1113.com/.

Sourrouille,J. L., & Caplat, G. (2003). A pragmatic view on consistency checking of UML models. In Kuzniarz L., Huzar Z., Reggio G., Sourrouille J. L., Staron M. (Eds.). Workshop on Consistency Problems in UML-based software development II, Workshop Materials, Research Report, 43-50.

Stahl, T. & Völter, M. (2006). Model-Driven Software Development- Technology, Engineering, Management. West Sussex, England, John Wiley and Sons Ltd.,.

Statemate-Magnum (2007). *A case tool for UML verification.* Retrieved April, 2007 from http://www.ilogix.com/products/magnum/index.cfm.

Tabu, Tool for the Active Behavior of UML (2004). Retrieved April, 2007 from http://www.cs.iastate.edu/~leavens/SAVCBS/2004/posters/Beato-Solorzano-Cuesta.pdf,.

Taentzer, G. (2003). AGG: A Graph Transformation Environment for Modeling and Validation of Software, *In Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, USA.

Tanriover, O. & Bilgen, S., (2007). An Inspection Approach for Conceptual Models for the Mission Space Developed in Domain Specific Notations of Uml, In *Proceedings of the Fall Simulation Interoperability Workshop,* Orlando, USA: SISO.

Tanriover, O. & Bilgen, S., (2008). Inspection Report of Conceptual Models Develeoped in Kama-Notation: Two Case Studies, Technical Report, Informatics Institute , METU/II-TR-2008-1.

Travassos, G.H., Shull, F., Carver, J. & Basili, V. R. (2002).Reading Techniques for OO Design Inspections. University of Maryland Technical Report, April(OORT V.3). Retrieved at December, 2007 from http://www.cs.umd.edu/Library/CS-TR-4353/CS-TR-4353.pdf

Tun, T.,& Bielkowicz, P. (2003). A Critical Assessment of UML using an Evaluation Framework. the CAiSE/IFIP8.1, *In International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'03, (*pp. 29-37), Velden, Austuria: IEEE.

UML Superstructure (2005). *Unified Modeling Language 2.0 Superstructure Specification. Object Management Group* august. Documentformal/05-07-04, retrieved December, 2005, from <http://www.omg.org/uml/>

Unhelkar, B. (2005).*Verification and Validation for quality of UML 2.0 Models.* Addison Wesley.

Van Der Straeten, R. (2005). *Inconsistency Management in Model Driven Engineering.* Doctoral dissertation, Vrije Universiteit, Brussel, Belgium.

Wagner, R., Giese, H., & Nickel, U. A. (2003). A plug-in for flexible and incremental consistency management.In Kuzniarz L., Huzar Z., Reggio G., Sourrouille J. L., Staron M (Eds.): *Workshop on Consistency Problems in UML-based software development II, Workshop Materials, Research Report.*

Woflan (2002), A. *Petri Nets analysis tool,* Retreived March, 2006, from http://is.tm.tue.nl/research/woflan.html

Xing, Z. & Stroulia, E. (2007). Differencing logical UML models. *Automated Software Engg.* 14 (2), 215-259.

Zhao,Y. Fan,X. Bai,Y. Vang,H. C., & Ding, W. (2004). Towards formal verification of UML diagrams based on graph transformation. *In Proceedings of the International Conference on E-Commerce Technology for Dynamic E-Business*. Beijing: IEEE.

APPENDICES

APPENDIX A

A formalism for UML class diagrams: First order logic

FOL can be used as a formalism for exploring the structural perspective of conceptual models. In the following, we formally define most general and used concepts and relations for structural perspective of conceptual modeling. These formal definitions can be used to convert, a class diagram to a set of FOL assertions.

Formal definition of a class

A class is a set of concepts or objects with at least some common properties. A class consists of a name, attributes, and operations. Let us denote a class by C, an attribute by a, its type by T, then the following assertion holds:

$$\forall (x,y)\ (C(x) \wedge a(x,y)) \rightarrow T(y)$$

where for every instance of class C, there exists an attribute y which is associated to x; then y is of type T.

An operation over a class is a function such that;

$$f(P_1,\ldots,P_m) = R$$

where $P_1, \ldots, P_m$ are the types of the m parameters, R is the type of the result single or complex value.

It is clear that this definition is not same with the definition of the function in mathematics. This only represents the signature of the function. The actual function definition can be expressed in terms of pre-conditions, post-conditions and invariants by OCL appended to the diagram. According to above

definition, a predicate corresponding to a function must have m+2 arguments. Given the form of the predicate, the following must hold:

1 - $\forall$ x, $P_1,\ldots,P_m$, r. $f(x,P_1,\ldots,P_m,r) \rightarrow P_1(P_1) \wedge \ldots \wedge P_m(P_m)$

where x is the name of the function, $P_1,...,P_m$ are parameters, r is the result.

2 - $\forall$ $x,P_1,...,P_m,r,r'$. $f(x,P_1,\ldots,P_m,r) \wedge f(x,P_1,\ldots,P_m,r') \rightarrow r = r'$

since f is a function.

3 - $\forall$ $,P_1,\ldots,P_m,r$. $C(x) \wedge f(x,P_1,\ldots,P_m,r) \rightarrow R(x)$

where type of the result is R, depending on the class and parameters.

Formal definition of Associations

An association is a relation between the instances of two or more classes.

Given two classes C_1 & C_2 cardinalities specifies the number of objects that can participate to the binary relation defined by the classes. Sometimes an association class may be needed to describe properties of associations themselves. Role may be attributed to classes to specify the roles they play within the associations.

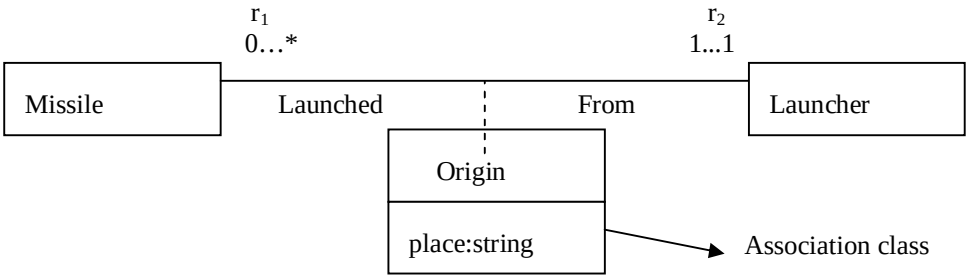


Figure 1: Association Example

An association A can be formally defined by the following assertion; (without an association class)

1 - $\forall$ $x_1,x_2,\ldots,x_n$. $A(x_1,\ldots,x_n) \rightarrow C_1(x_1) \wedge,\ldots, \wedge C_n(x_n)$ (from the definition of relation since association is a relation)

An association A with a related association class is defined by a predicate A and n predicates for each role name as the following;

2 - $\forall$ x,y. $A(x) \wedge r_1(x,y) \rightarrow C_1(y)$

$\forall$ x,y. $A(x) \wedge r_2(x,y) \rightarrow C_2(y)$

Such that predicate A(x) denotes each element of Association Class, r_1,r_2 relates these elements to objects of classes C_1 & C_2 respectively. That is; if an object of the association class A share role 1 with another object (or instance), this object (instance) must belong to Class C1.

3 - $\forall$ x. A(x) $\rightarrow$ $\exists$ y. r_i(x,y) for i = 1,…,n

such that there exists at least one y, for every element of class A, the role r_i holds. That is every element of the association class A participates at least once to one of the roles.

4 - $\forall$ x,y,y′ . A(x) $\wedge$ r_i(x,y) $\wedge$ r_i(x,y′) $\rightarrow$ y = y′ for i = 1,...,n

such that there is at most one element of A playing the same role.

5 - $\forall$ y_1,...,y_n,x,x′. A(x) $\wedge$ A(x′) $\bigwedge_{i=1}^{n}$ (r_i(x,y_i) $\wedge$ r_i (x,y_i)) $\rightarrow$ x = x′

For cardinality constraints of binary association without association class the following formulas must hold;

$\forall$ x_1. C_1(x) $\rightarrow$ ($n_\ell \leq$ # {y | A(x,y)} $\leq n_u$)

$\forall$ y_1. C_2(y) $\rightarrow$ ($m_\ell \leq$ # {x | A(x,y)} $\leq m_u$)

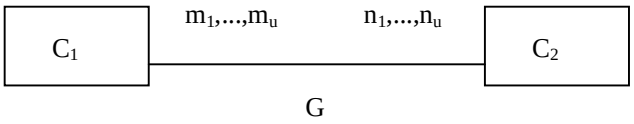


Figure 2: An association without an association class

Formal definition of Aggregation

Aggregation between two classes C_1 & C_2 exist if a set of instances of one class is contained in the other. Formally;

$$\forall \text{ x,y } G(x,y) \rightarrow C_1(x) \wedge C_2(y)$$

Formal definition of Generalization & Specialization

A generalization relationship exists between two classes C_1&C_2 if the instances of one of the classes are also instances of the other class. Formally;

$$\forall \text{ x.} C_1(x) \rightarrow C_2(x)$$

where *disjointnes***s** can be formally captured by;

$$\forall \text{ x.} C_1(x) \rightarrow \neg C_2(x) \wedge \ldots \wedge \neg C_n(x)$$

where *covering constraint* is formally captured by;

$$\forall \text{ x .} C(x) \rightarrow C_1(x) \vee \ \ldots \ \vee C_n(x)$$

APPENDIX B - A Formalism for UML Activity Diagrams : Petri Nets

To describe how and what kind of properties can be analyed for activity diagrams by means of Petri Nets, we give some formal definitions

A PN is a five-tuple PN = (N, M_o)= (P,T,I^+, I^-,M_o) where;

1. P= {p_1, p_2, ... ,p_m} denotes a finite set of places

2. T={t_1, t_2, ..., t_n} denotes a finite set of transitions.

3. $P \cup T = \phi$

4. I^- is the input incidence function defined on P x T (a set of input places for to activate a set of transitions)

5. I^+ is the output incidence function defined on T x P (a set of output places of a transition)

6. $\forall_p \in P, \exists_t \in T: I^-(p,t) \neq 0 \cup I^+(p,t) \neq 0$ and

$\forall_t \in T, \exists_p \in P: I^-(p,t) \neq 0 \cup I^+(p,t) \neq 0$

7. M_0 is a set of token values defined on P and is called the initial marking.

8. Transition t_j is enabled and ready to fire if $M(p) \geq I^-(p,t_j)$, $\forall p \in \bullet t_j$ (where $\bullet t_j$ represents set of all input places of t_j)

9. An enabled transition can be fired according to the occurrence of actual event on the transition.

10. After firing the transition, $I^-(p,t_j)$ tokens are removed from each of the associated input places $I^+(p,t_j)$ tokens are added into each of its corresponding output places.

A PN can be also represented by its incidence matrix C such that:

$$C_{m \times n} = -C^- + C^+ = -\begin{pmatrix} C^-_{11} & C^-_{12} & \dots & C^-_{1n} \\ . & . & \dots & . \\ . & . & \dots & . \\ C^-_{m1} & C^-_{m2} & \dots & C^-_{mn} \end{pmatrix} + \begin{pmatrix} C^+_{11} & C^+_{12} & \dots & C^+_{1n} \\ . & . & \dots & . \\ . & . & \dots & . \\ C^+_{m1} & C^+_{m2} & \dots & C^+_{mn} \end{pmatrix}$$

where

$C^-_{ij} = I^-(p_i, t_j)$ and $C^+_{ij} = I^+(p_i, t_j)$

From an initial marking m_0 to a next marking m by firing a set of enabled transition is shown as follows:

$\overrightarrow{M} = \overrightarrow{m_o} + C.\vec{y}$ or $m = m_o + C.y$

where $\vec{y}$ is a transition firing vector.

Analysis with Petri Nets

There are some important properties which are defined and analyzed in the PN literature (Murata 1989). We would like to describe a number of important properties such as liveness, boundedness and reachabilty.

Reachablity: The equation $m_0 + C.y = m$ presented in the previous section is called the marking equation. If the equation $m_0 + C.y = m$ has a solution for Y in net N than a marking m is reachable from m_0.

Place Invariants: A place invariant of a Net N is a vector $\dot{i}$, $\dot{i} \neq 0$ satisfying the following equality:

A place invariant of a Net N is a vector $\dot{i}$, $\dot{i} \neq 0$ satisfying the following equivalent equalities:

a) $\sum_{s \in \bullet T} \vec{i}_s = \sum_{s \in T \bullet} \vec{i}_s \ (\forall t \in N)$

b) $\vec{i} \ . \ \vec{t} = 0 \ (\forall t \in N)$

where N is a Petri Net, $s \in .t$ satisfies $m(s) \geq w(s,t)$ and $s \in t.$ satisfies $m(s) + w(t,s) \leq k(s)$ where w is weight function by default w = 1.

An example:

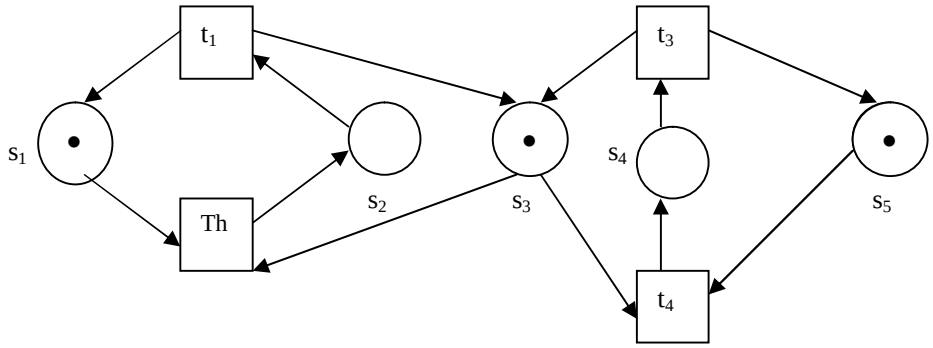


Figure 1: A Petri net and a corresponding activity net

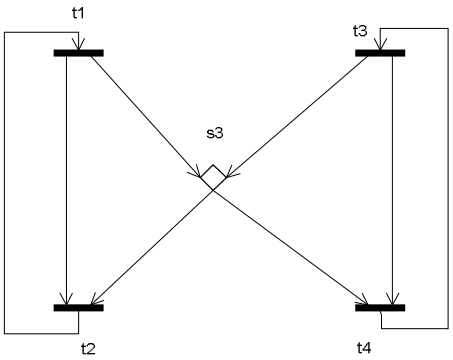


Figure 1 (cont.): A Petri net and a corresponding activity net

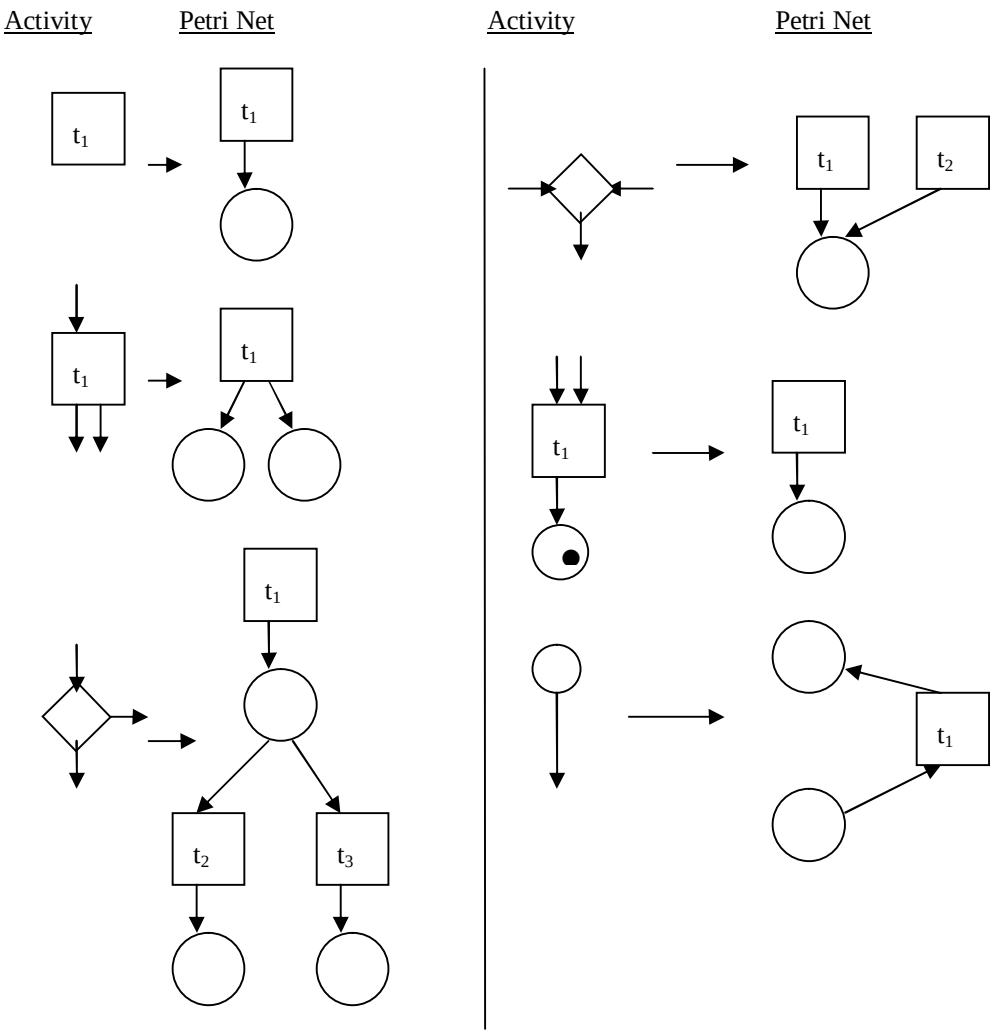


Figure 2 : A Set of Transformation Rules of Activity Diagrams to Petri Net

In order to detect design errors and modeling issues it may be a good idea to convert activity diagrams to Petri Nets (PN) to verify correctness of the behavioral system specification. Rather then designing a complete translation of the UML model it is convenient to restrict the translation to those aspects that contribute to the properties of interest. For example, [Boccalette et al., 1999)] have developed a set of

rules to transform simple UML activity diagrams to Petri Nets. Rules are shown in the following Figure 2. In Figure 1, we present an example Petri Net and its corresponding activity net. The transformation is achieved by means of applying the rules of proposed in Figure 2. We are able to transform a given activity model to a Petri Net by applying these rules. An example of an activity diagram transformed to a Petri Net will be shown. Petri nets can be used to analyze behavior of activity diagrams.

Remark that places s1, s2, s4, s5 and all the tokens are lost in this transformation. This problem is known in the literature as the semantic correspondence problem. Not all the modeling constructs find their counterpart in the target domain. This is one of the drawbacks of approaches which transform the models to a formal notation and the result of the analysis back to the original notation.

Possible States	**S_1**	**S_2**	**S_3**	**S_4**	**S_5**
1	1	0	1	0	1
2	0	1	0	0	1
3	1	0	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1

Remark that in the Petri Net in Figure 1, number of tokens in $\{s_2,s_3,s_4\}$ is not changed. Furthermore, neither in $\{s_1,s_2\}$, $\{s_4,s_5\}$

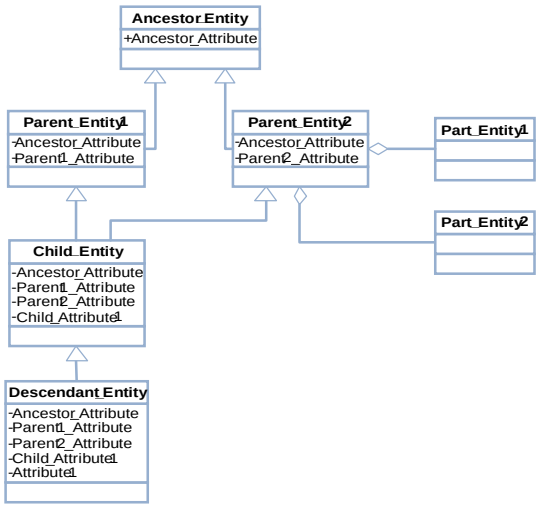
Infact, according to 2.b we may show that $\vec{i}$ = (0,1,1,1,0) is a place invariant:

$$(0,1,1,1,0)\ .\ \begin{array}{c} \\ S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & t_3 & t_4 \end{array} \\ \begin{bmatrix} -1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ -1 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \end{array} = (0)$$

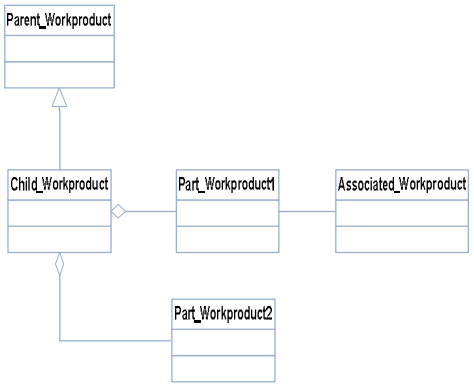
Liveness: If each place invariant each $\vec{i}$ without negative entries (any component of $\vec{i}$) i.e. $\vec{i}\ .\overrightarrow{m_o} > 0$ then a PN is called a live marked PN. A PN is structurally live if there exists an initial marking and firing sequence $\vec{y}$ such that every transition can be fired infinitely.

Boundedness: Each marked PN with a place invariant $\vec{i}$ satisfying $\vec{i} > 0$ for each place s the net is bounded where i_s a component of vector $\vec{i}$ at the place s. A PN is structurally bounded if there exist no initial marking such that the marked net is unbounded.

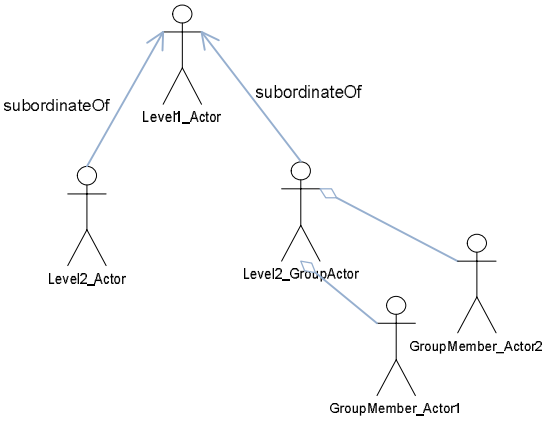
APPENDIX C - Kama Diagram Types: Structural Perspective (From Karagöz, 2008)



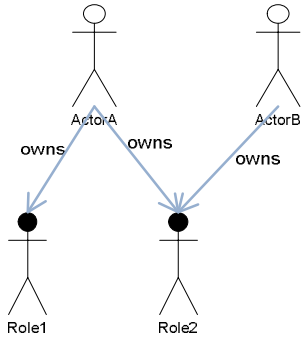
Entity Ontology Diagram



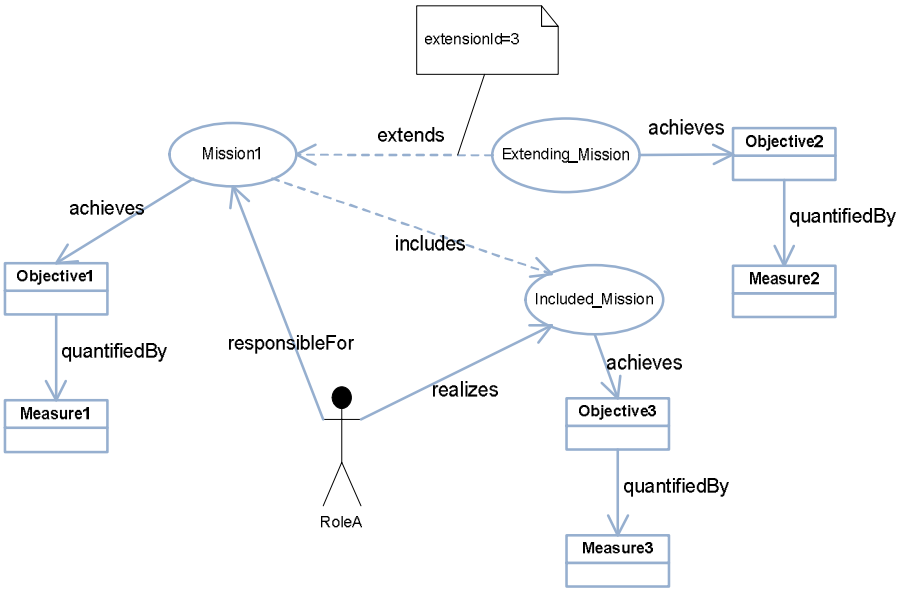
Entity Relations Diagram



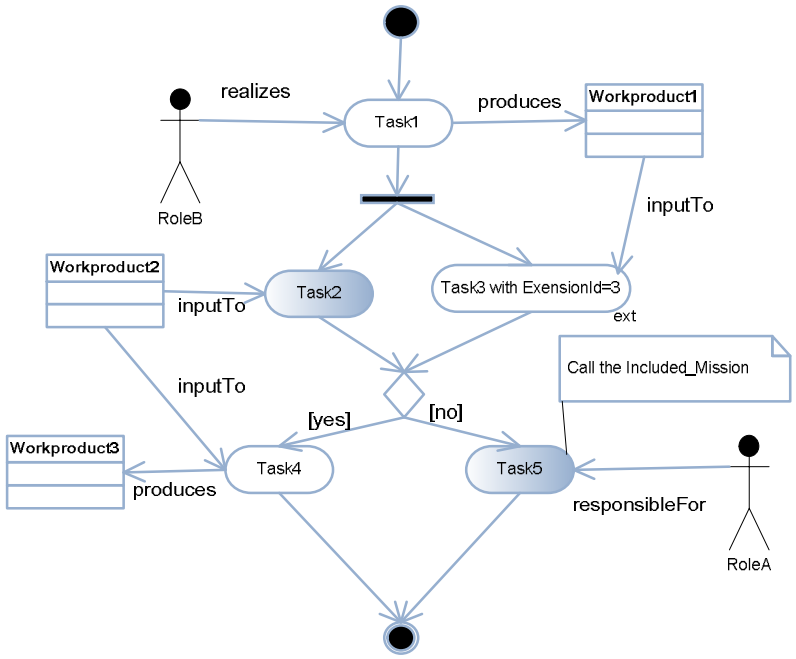
Command Hierarchy Diagram



Organization Structure Diagram



Mission Space Diagram



Task-flow Diagram

APPENDIX D - Report On Case Study 2

1. The Conceptual Model

The conceptual model considered in Case Study 2 was developed using Enterprise Architect v6.5 case tool. Apart from concrete syntax, by the help of the profiling mechanism the modeling elements were extended for the UML 2.1 modeling elements. The tool was used in such a manner that it provided a user interface to define all KAMA diagram types and related modeling elements.

The mission space view included 70 missions represented in 21 mission space diagrams. The topmost mission space diagram included 7 missions and in order to decompose the model mission space diagrams were developed for these highest level missions. Task-flow view is represented with 397 tasks in 45 task-flow diagrams. There are a total of 95 entities that were grouped in 15 logical packages and represented in 16 entity ontology diagrams. 25 entity state diagrams that represent the behavior of complex entities were developed.

Before the inspection described below, the conceptual model was already reviewed over 2 days by one expert. Also, a review meeting with the participation of 6 members of the development team was held and lasted 4 hours. Later on, the conceptual model was subjected to a walkthrough that took five days. 4 engineers from the conceptual model development team and 3 from the acquirer organization joined the meetings in this third phase. There were 150 issues identified during these meetings. The issues identified were related with task-flow diagrams, assumptions and constraints about the mission space, additional attributes and capabilities to the entities, definition of roles and actors. Our inspection-based verification was applied after all these three phases were realized.

2. Information about the Applied Inspection Process

Before each intra-diagram inspection, the validation function of Enterprise Architect v6.5 was executed on each diagram with wff and syntactic rules checks. The tool's standard validation function which included syntactic, wff and other checks signaled no errors. Then, the verification tasks proposed by Tanrıöver and Bilgen (2008) have been performed.

During the inspection the model tree browser is used that helped the inspector to manage the browsing (which may sometimes be rather complex) needed for inter-diagram verification tasks. The execution of inspection tasks has been tailored for the conceptual model. We have used a perspective oriented inspection strategy for effectiveness reasons. For this, we have conducted the inter-diagram inspection tasks not as a standalone activity but rather decided to perform the inter-diagram task related to each of diagram type just after finishing the intra-diagram inspection for that diagram. The order of execution of the verification tasks was as follows:

1) Entity – ontology diagram has been verified with class like diagram inspection tasks, inter-diagram task 2, task 9 and task 10 has been performed in the former order.
2) Organization diagrams have been inspected with class like diagram inspection task.
3) On mission space diagrams, mission space diagram inspection tasks, inter-diagram task 1, task 8 and task 6 have been performed. Task 6 was not conducted since hierarchical refinement is used with structured activity nodes for specialization of missions. The leaf extending missions was modeled with in activity diagrams, with structured activity nodes. By definition the property searched in task 6 is satisfied.
4) Task-flow diagrams have been inspected with the inspection tasks for task-flow diagrams and inter-diagram inspection task 3, task 4 and task 5. In "Second" model the tasks are associated to entities. During task 4, we have checked the refinement relation such that only entities composing or specializing the upper entity in the ontology can be associated to sub tasks of the structured task node. That is the assigned entity is decomposed to sub entities and assigned to tasks in the refining sub task-flow diagram, in accordance with the generalization or composition hierarchy defined in the entity ontology view. In the case study, the facility of the EA 6.5 tool to view the class hierarchy tree is used to obtain all the lower level entities transitively based on both aggregation and generalization relations. Note that only first sub level has been checked for each diagram, the deeper levels of activities are not checked to avoid duplicate checks. This because the lower level activities are verified with the same inspection tasks. In this way only one sub level of refinement check for each activity diagram will cover the whole model.

Metrics Collected During the Case study

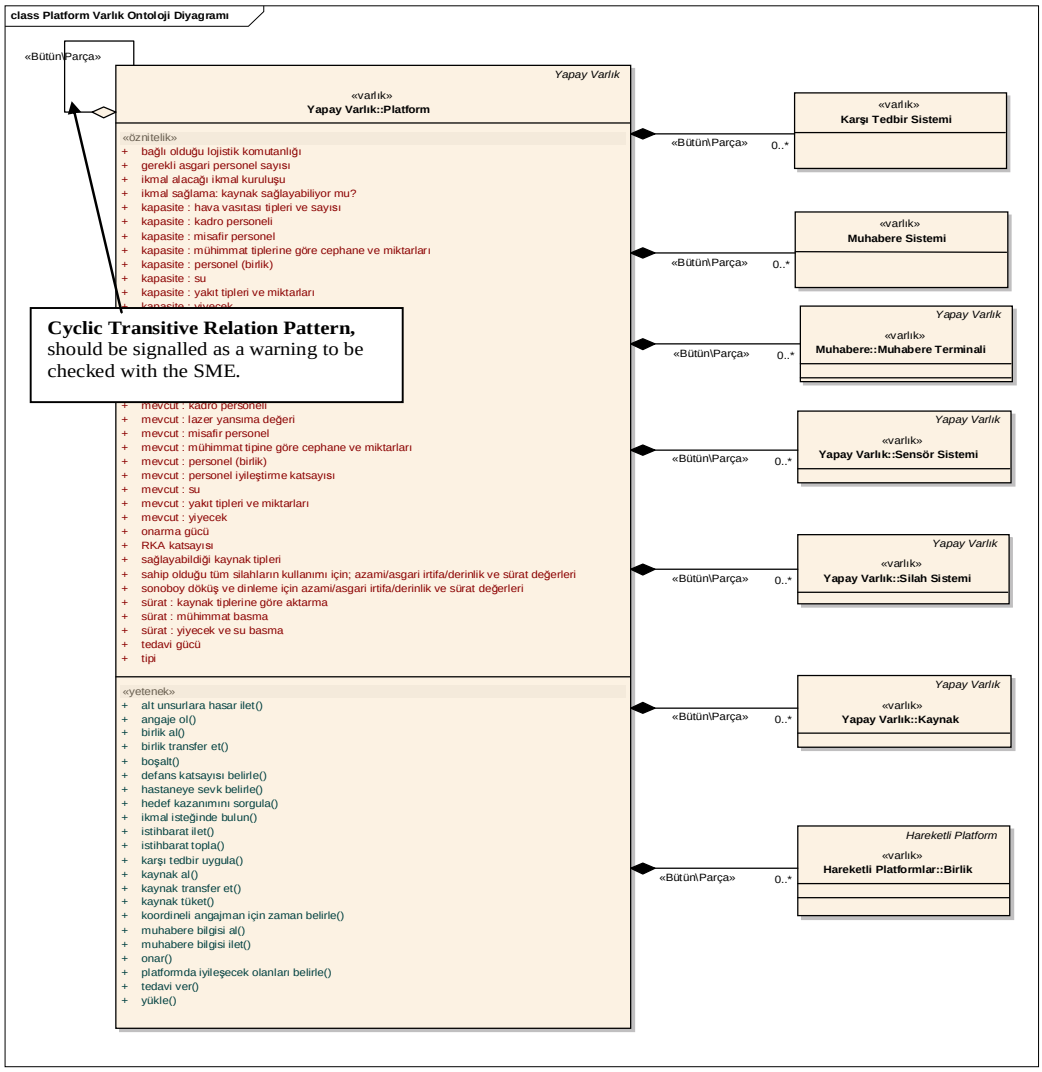
"SECOND" MODEL	***Mission Space Diagrams***	***Entity Ontology Diagrams***	***Task-flow diagrams***	***Entity State Diagrams***	***Organization Diagrams***
Total # of important model elements (concepts)	70 missions	97 entities	397 tasks	174 states	8 actors
# of diagrams in the model	21	16	45	25	1
# of diagrams inspected	21	16	20	-	1
Inspection tasks	Structural diagram + mission space diagram verification tasks + inter-diagram task no: 1.	Structural diagram verification tasks + inter-diagram task 2,	Task-flow diagram verification tasks + inter-diagram task 3, task no: 4, task no:6, task no:7.	Interdiagrams task no: 9, task no:10.	Structural diagram verification tasks
Limitations	-	-	Interdiagram task no:6 is performed for 15 activity diagrams.	Tasks are performed for 5 state diagrams only.	-
# of defects detected	19	5	27	7	-
Time per diagram : Detection time + recording time	3-10 minutes	3-10 minutes	3-25 minutes	5 – 30 minutes	3 minutes
Total time: Detection time + recording time	3 man / hrs	4 man / hrs	10 man/ hrs	3 man /hrs	3 min.

TOTAL # SEMANTIC ISSUES DETECTED: total: 56, non-trivial: more than 30.

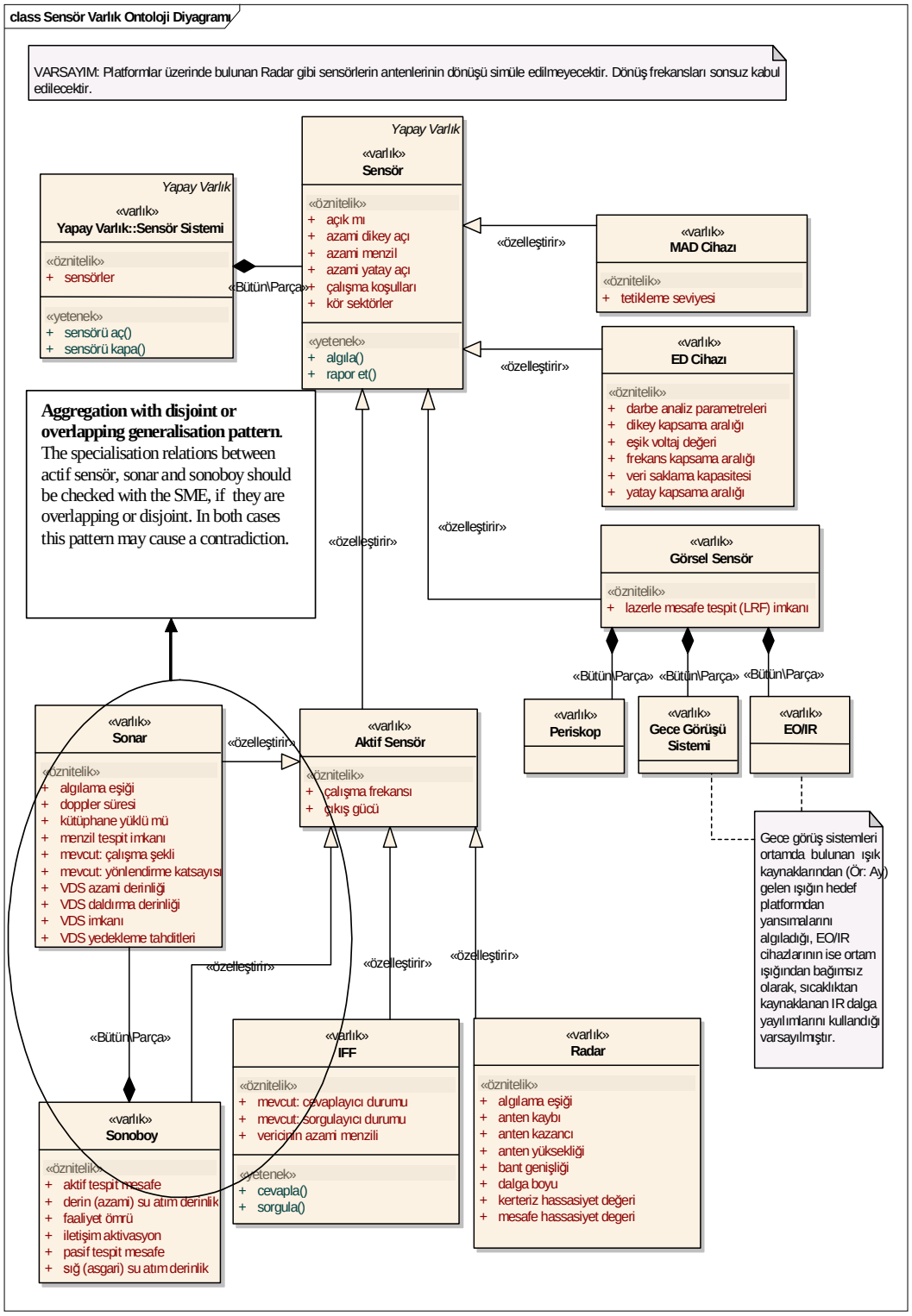
TOTAL TIME SPENT ON INSPECTION is 20 hrs. + 8 hrs per person, for inspection preparation.

Inspection preparation: Studying and understanding the inspection process, browsing through the model for initial familiarization, time spent due to limitations caused by differences of the usage of notation, adopting the inspection process for the model and the capabilities of the case tool: 8 hrs.

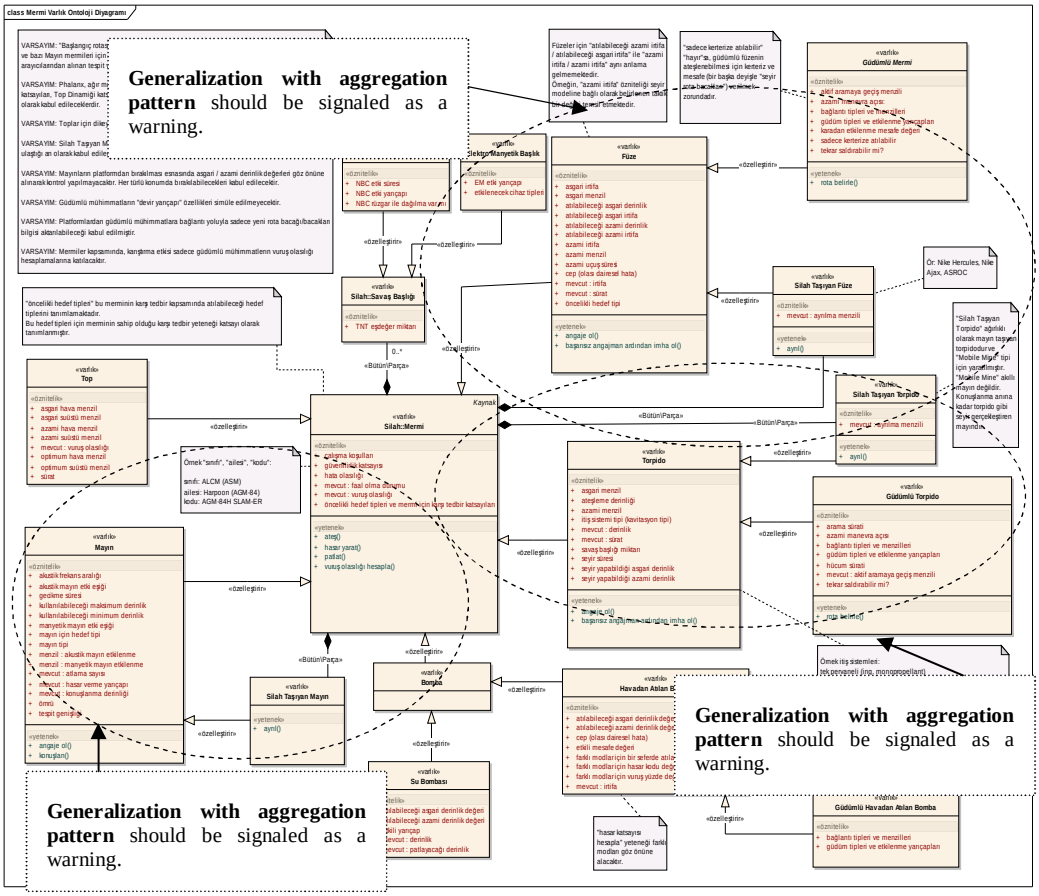
3. Inspection Findings:



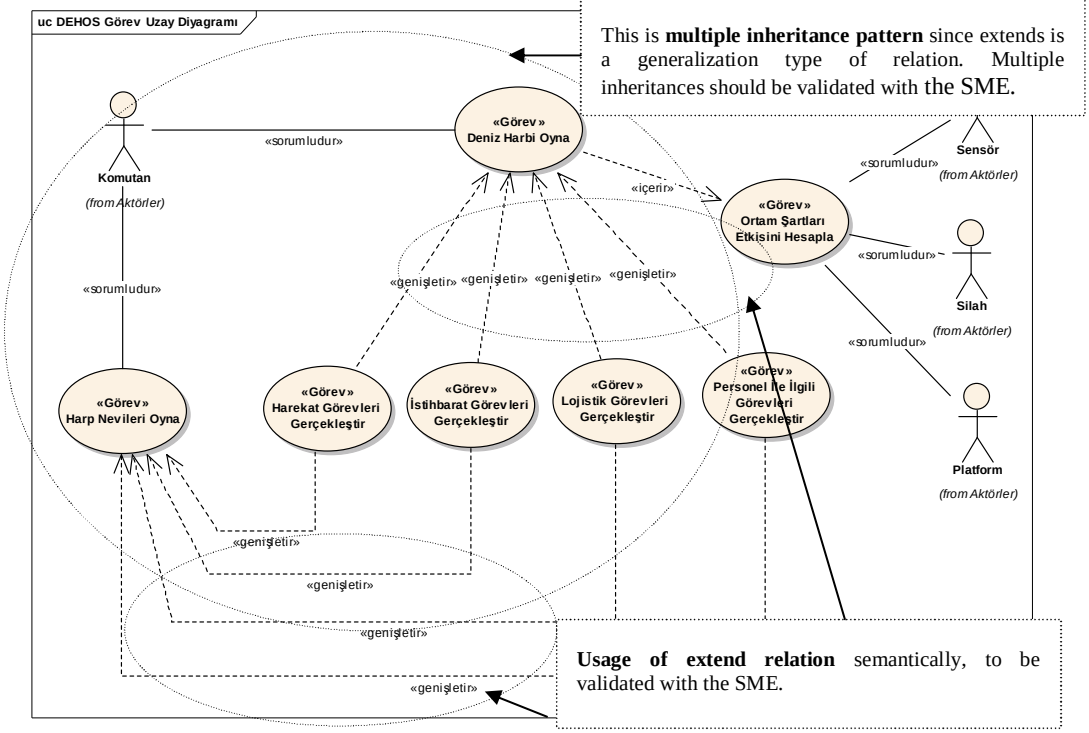
Intradiagram inspection time: 2 MINUTE



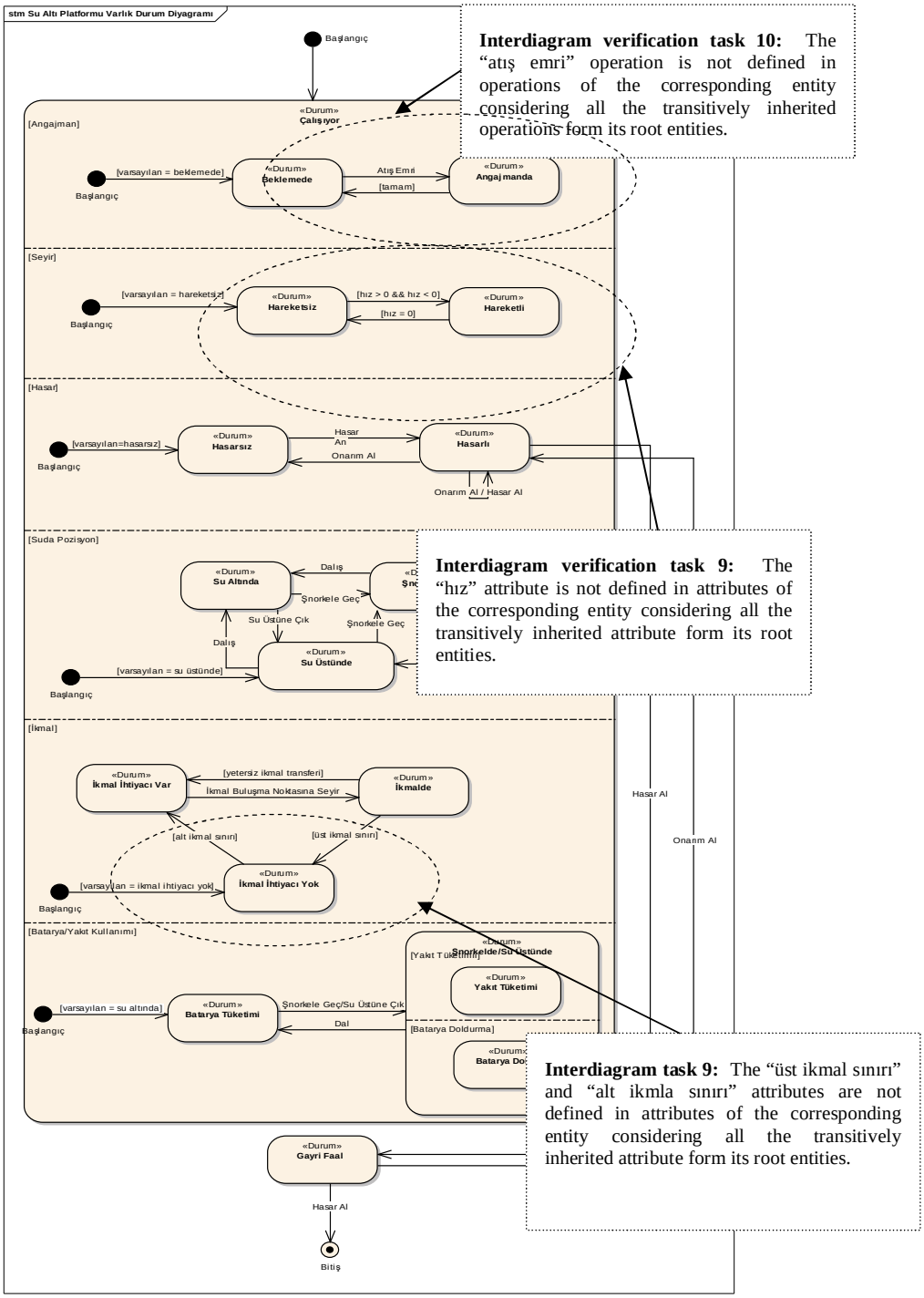
Intradiagram inspection time: 3-5 MINUTES



Intradiagram inspection time: 3-5 MINUTES

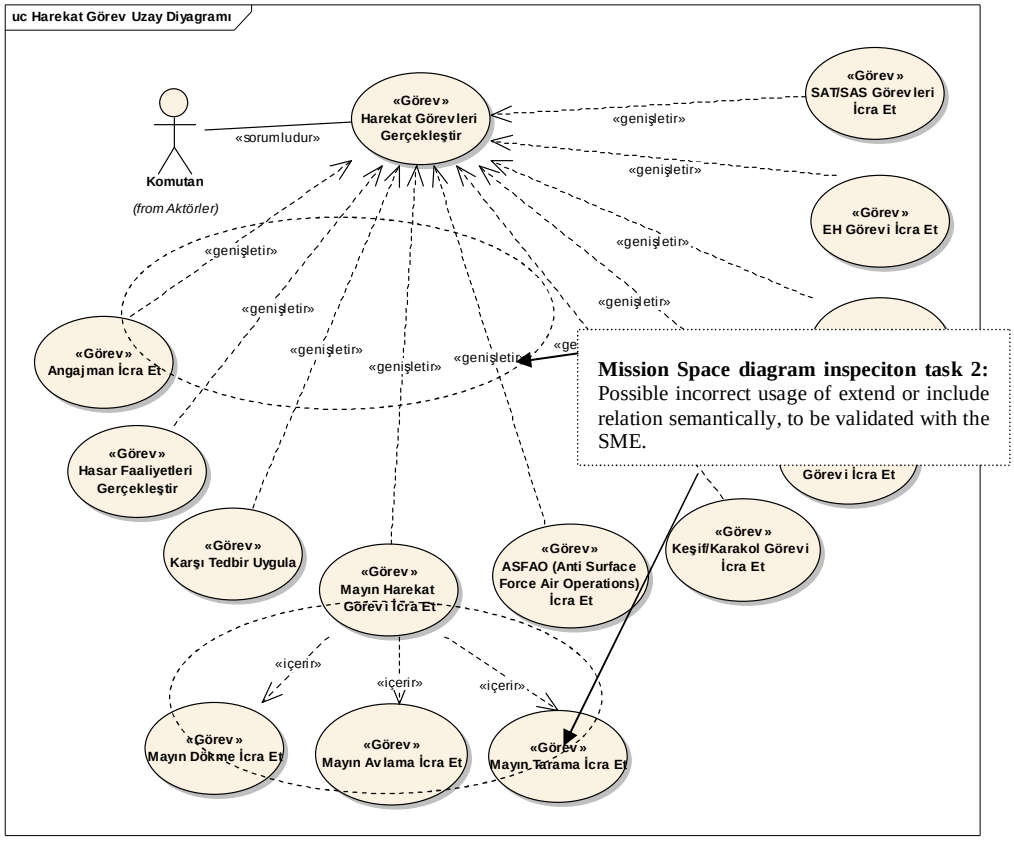


Intradiagram inspection time: 3-5 MINUTES



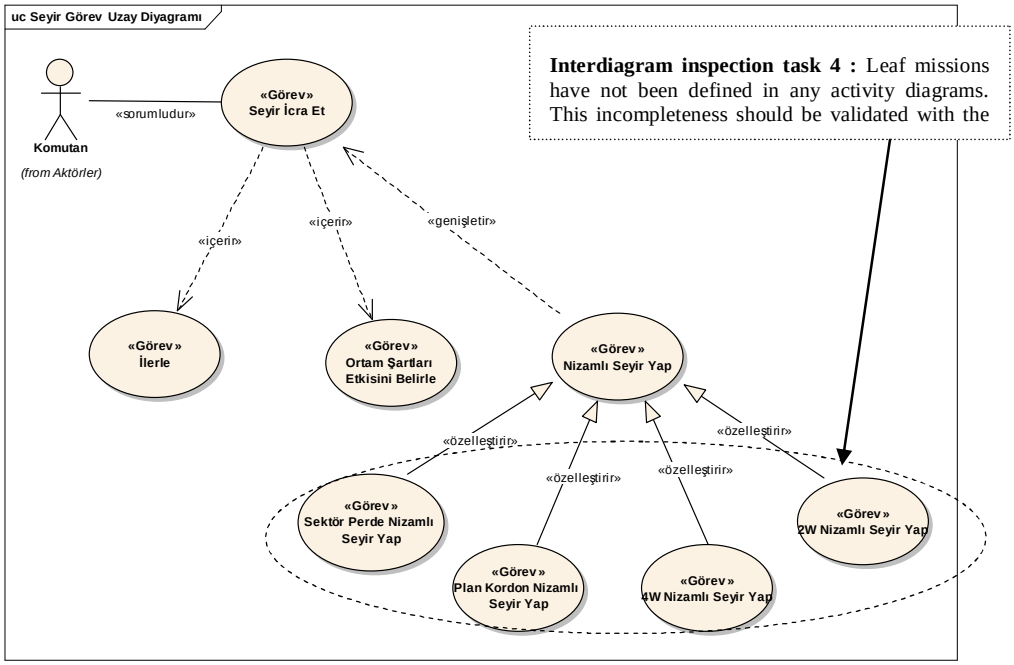
Note: There are 25 entity state diagrams in the model. For each of the diagrams 10-25 minutes is spent for performing the task 9 and task 10 depending on the number of variables and transition actions in the state chart diagram and the depth of inheritance hierarchy of the entity to find all the inherited attributes from upper level entities. However since the same kind of finding will be identified, we did not check all of the statechart diagrams. On the other hand the intra-diagram verification of state chart diagrams is left completely out of scope of this study.

Identification time: 10-25 MINUTES

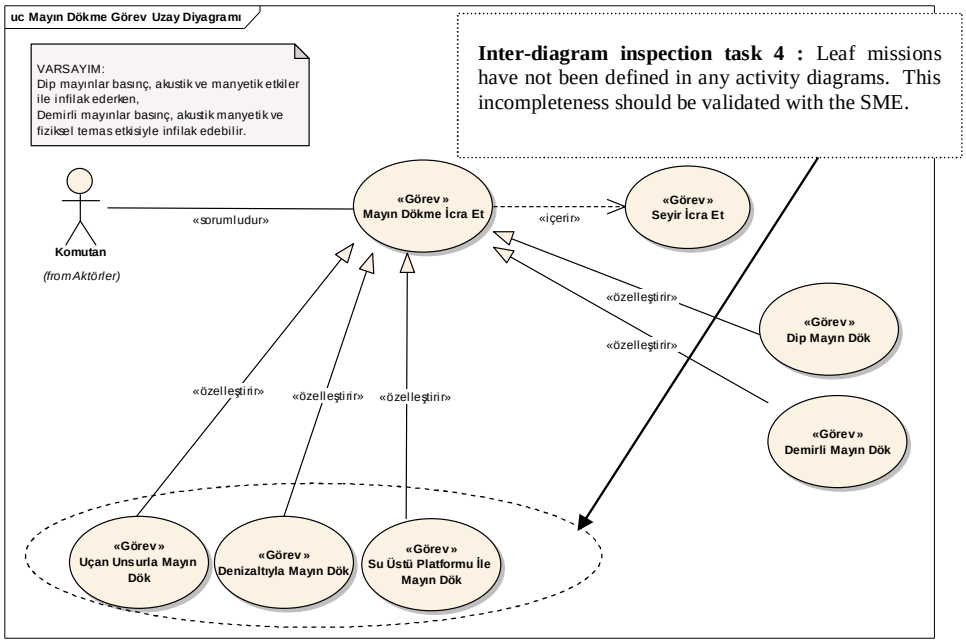


Intradiagram inspection time: 2 MINUTES

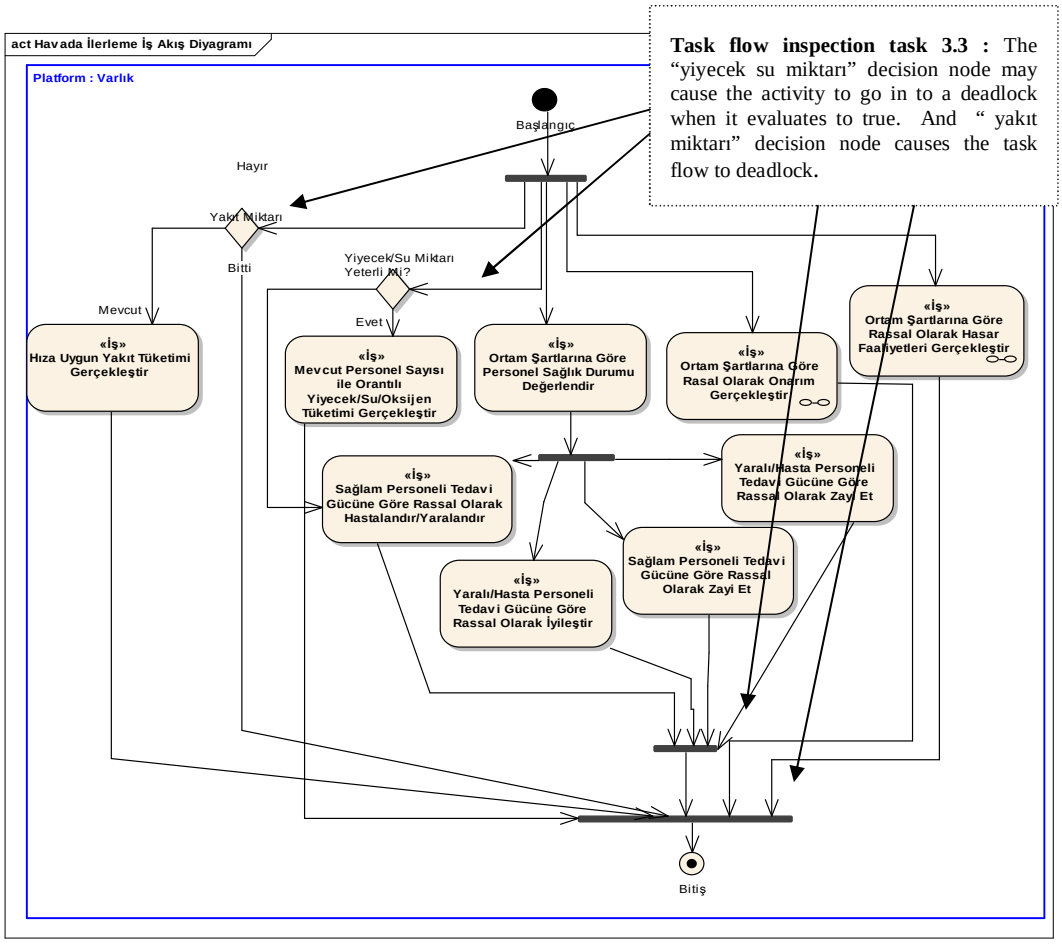
Note: There are 21 mission space diagrams in the model. It takes about 1-3 minutes to check for intra-diagram properties of mission space diagrams. The same kind of finding for 9 mission space diagrams.



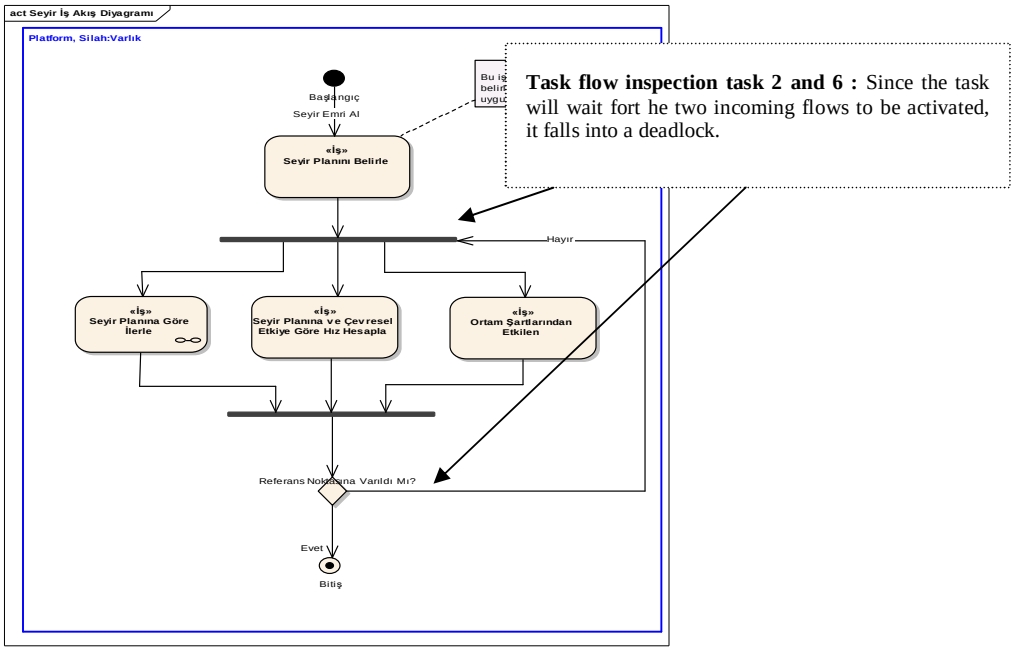
Intradiagram inspection time: 1-3 minutes



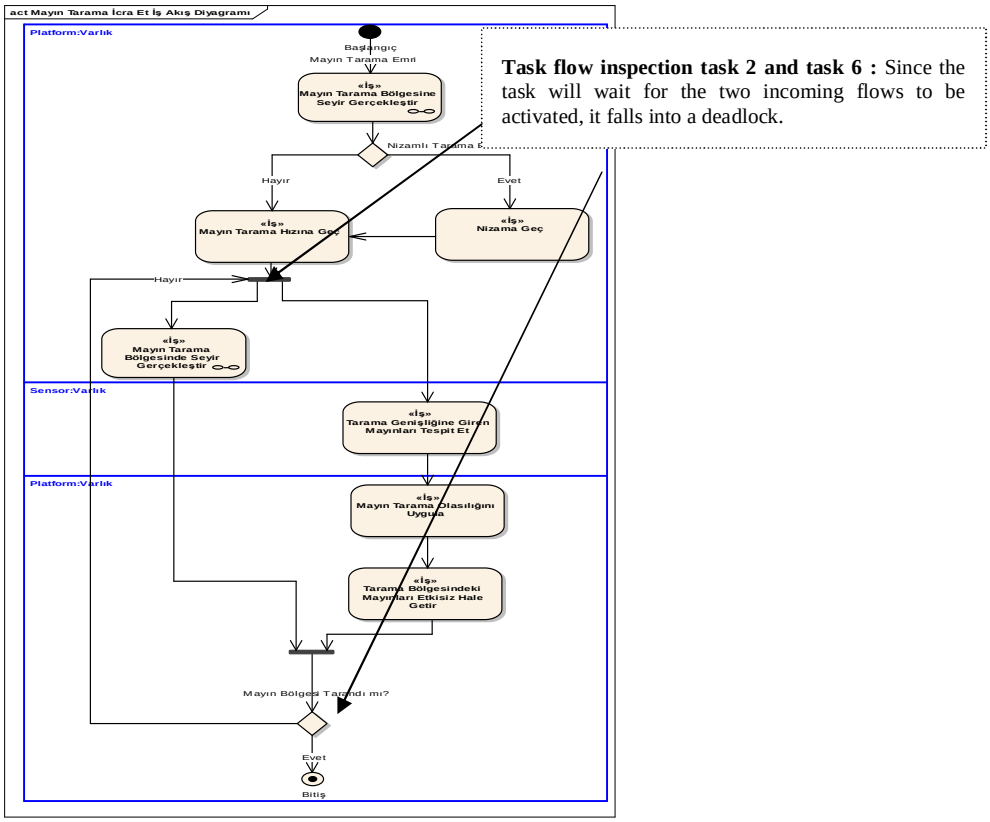
Note: There are 21 mission space diagrams in the model. It takes about 1-3 minutes to check interdiagram task 4 for a diagram by the search function of the project explorer tree of the tool. The same kind of finding for 7 mission space diagrams.



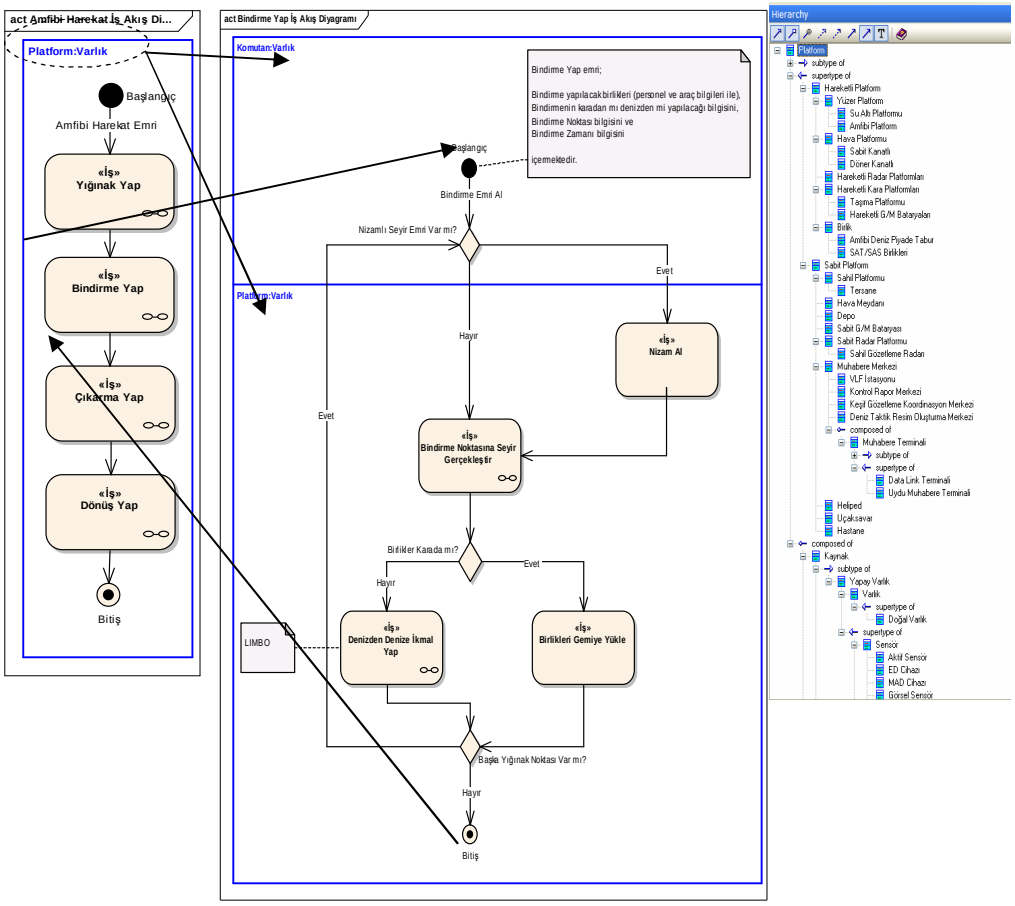
Inspection time of task-flow inspection tasks: 10 minutes



Inspection time: 5 minutes



Inspection time: 5 minutes



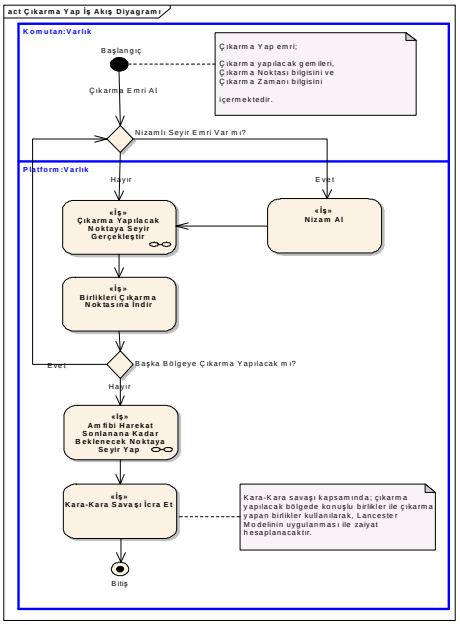
Inter diagram inspection task 4 (Refinement): During task 4, we have checked the refinement relation such that only entities composing or specializing the upper entity in the ontology can be associated to sub tasks of a structured task node. That is the assigned entity is decomposed to sub entities and assigned to tasks in the refining sub task-flow diagram, in respecting with the generalization or composition hierarchy defined in the entity ontology view. In the case study, the facility of the EA 6.5 tool to view the class hierarchy tree is used to obtain all the lower level entities transitively based on both aggregation and generalization relations. Note that only first sub level has been checked for each diagram, the deeper levels of activities are not checked to avoid duplicate checks. Because the lower level activities will are verified with the same inspection steps. In this way only one sub level of refinement check for each activity diagram will cover the whole model.

In this case although the “bindirme yap” task-flow is properly refined into a sub task-flow, its associated entity “platform” is not properly refined because when we check the entity hierarchy the “Komutan” entity is neither a subtype nor a part of platform entity. Inspecting 20 activity diagrams, we encountered 7 issues of this type

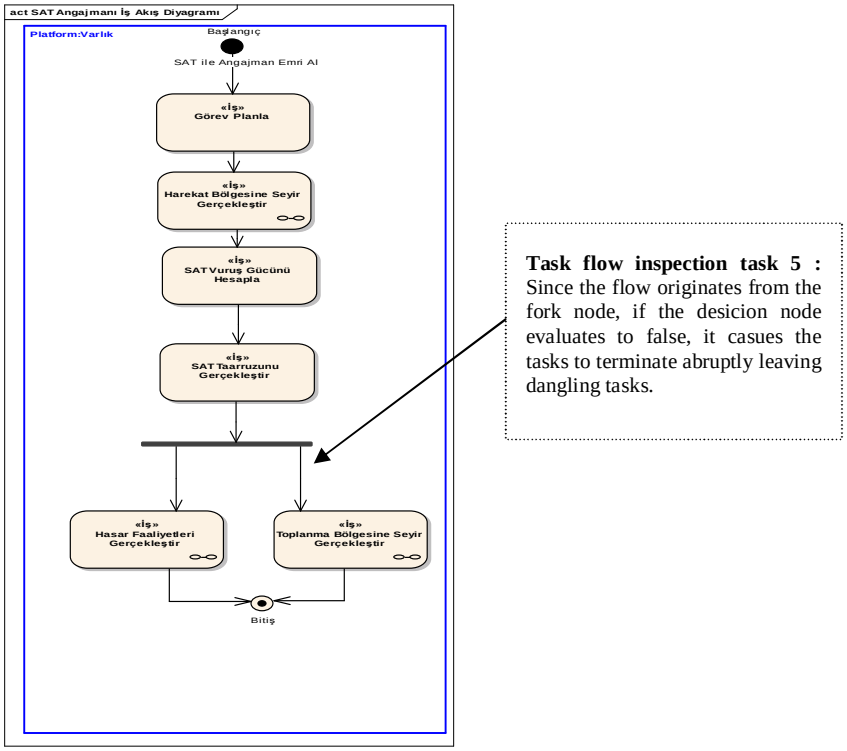
Inspection time: 10 – 20 minutes

Inter diagram inspection task 4 (Refinement):

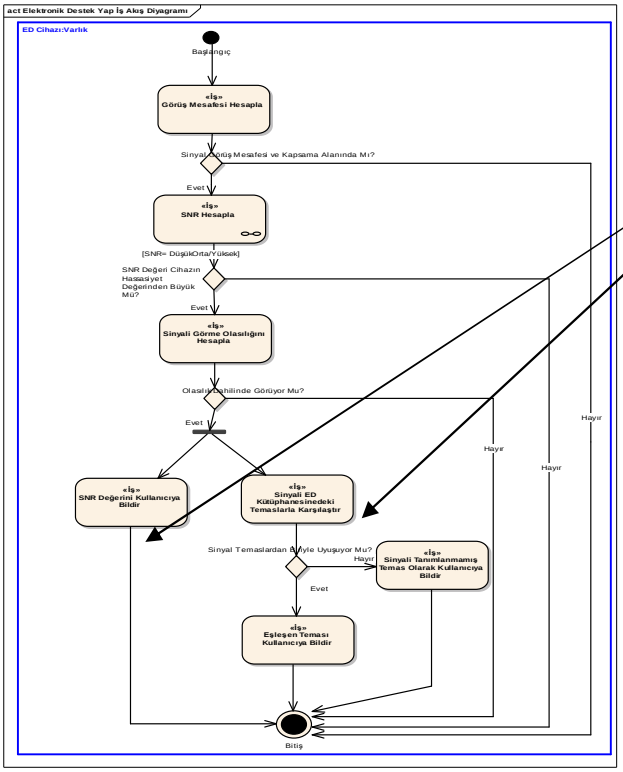
In this case although the “çıkarma yap” task-flow is properly refined into a sub task-flow, but its associated entity “platform” is not properly refined because when we check the entity hierarchy the “Komutan” entity is neither a subtype or a part of “platform” entity.



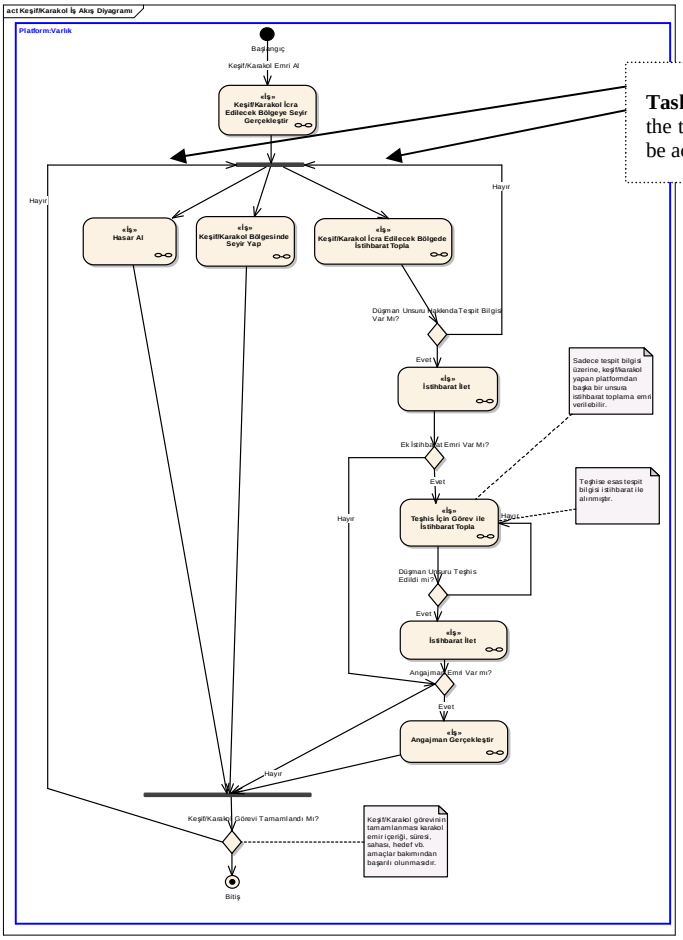
Note : We have inspected the 15 of the 45 task-flow diagrams and encountered 4 issues of this type.



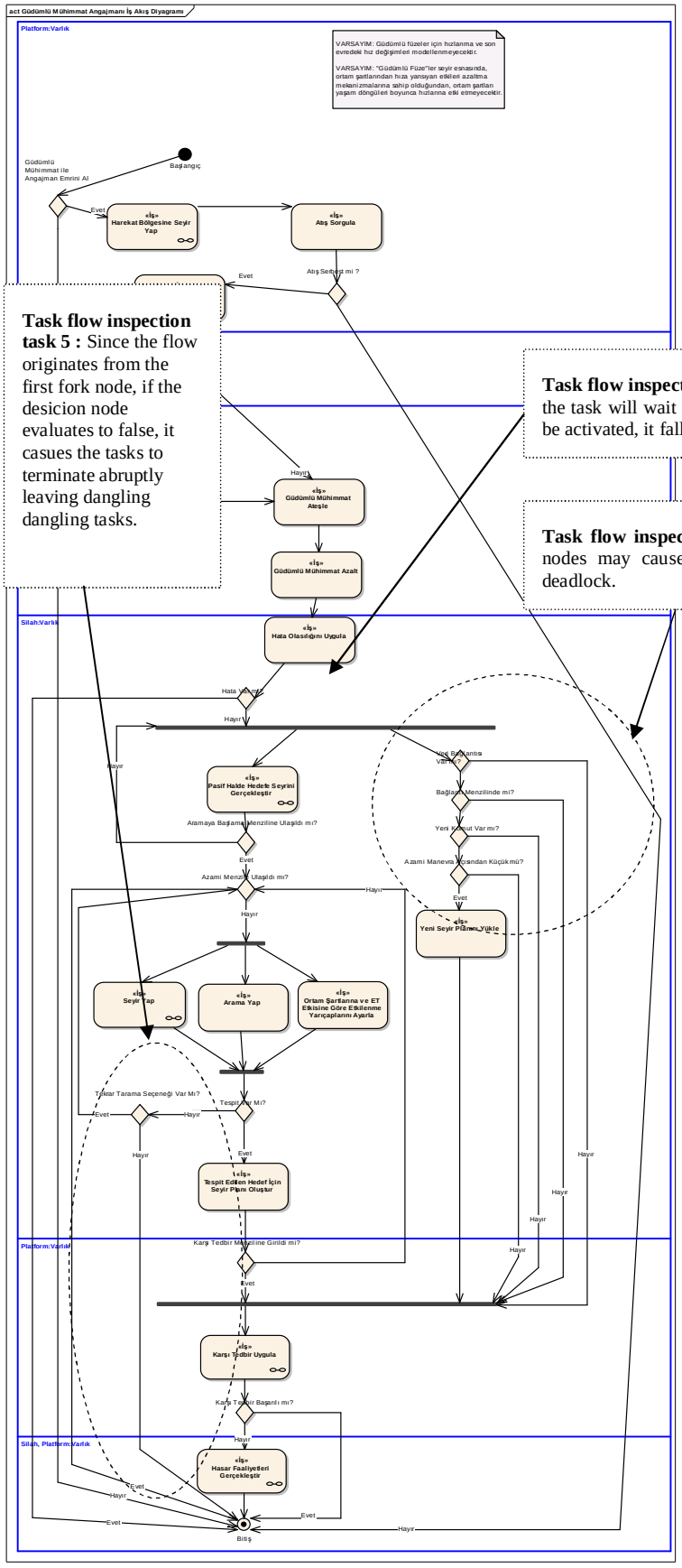
Task flow inspection task 5 : Since the flow originates from the fork node, if the desicion node evaluates to false, it casues the tasks to terminate abruptly leaving dangling tasks.



Task flow inspection task 5: Since the flow originates from the fork node, if the decision node evaluates to false, it causes the tasks to terminate abruptly leaving dangling tasks.



Task flow inspection task 2 and task 6 : Since the task will wait for the two incoming flows to be activated, it falls into a deadlock.



Task flow inspection task 5 : Since the flow originates from the first fork node, if the desicion node evaluates to false, it casues the tasks to terminate abruptly leaving dangling dangling tasks.

Task flow inspection task 2 and task 6 : Since the task will wait for the two incoming flows to be activated, it falls into a deadlock.

Task flow inspection task 3.3: The decision nodes may cause the activity to go in to a deadlock.

4. Summary and Conclusion

We have performed an application of the approach to a real life conceptual model. The mission space view of this model included 70 missions represented in 21 mission space diagrams. Task-flow view was represented with 397 tasks in 45 task-flow diagrams and 95 entities represented in 16 entity ontology diagrams.

One important observation in the case study was the model tree browser of the EA 6.5 tool proved to be very helpful for inter-diagram verification tasks. For the specific case study, during the pre-inspection phase, we were able to tailor the order of execution of inspection tasks to be more effective. We have used a perspective oriented inspection strategy for effectiveness reasons. For this, we have conducted the inter-diagram inspection tasks not as a standalone activity but rather decided to perform the inter-diagram task related to each of diagram type just after finishing the intra-diagram inspection for that diagram.

Although, 150 issues had been identified and corrected in previous verification and validation activities, by applying our approach we were able to identify 30 additional semantic issues which were non-trivial and important.

VITA

Ö. Özgür Tanrıöver holds Computer Engineering B.Sc., Information Systems M.Sc.(2002) and Science & Technology Policy M.Sc. degrees (2001) from Middle East Technical University (METU), Ankara, Turkey. He has worked as a research assistant at METU - CSTPS (Center for Science and Technology Policy Research) between 1998 and 2005. Since then, he has been working as an expert (Certified Information Systems Auditor) in information management department at the Banking Regulation Agency of Turkey. His current research interests are software quality assurance, information systems audit and conceptual modeling. Below is the list of publications by the author:

- Tanrıöver, Ö., Bilgen, S., "An Inspection Approach for Conceptual Models of the Mission Space in a UML based Domain Specific Notation" , submitted for publication in Requirements Engineering Journal, Springer-Verlag, Berlin, 2008.
- Tanrıöver, Ö., Bilgen, S., "An Inspection Approach for Conceptual Models for the Mission Space Developed in Domain Specific Notations of Uml", Simulation Interoperability Workshop, SISO,Orlando- USA, Fall 2007.
- Tanrıöver, Ö., Bilgen, S., "An Inspection Approach for Conceptual Models in Notations derived from UML: A Case Study", IEEE, ISCIS Proceedings, 2007.
- Ö. Tanriöver , O. Demirörs, "A Software Workforce Organization Assessment Model", European Software Process Improvement Conference, EuroSPI' 2003, University of Graz , Austria, 10^{th}-12^{th} , Dec, 2003.
- O. Demirörs, Ö. Tanriöver, C. Hoşver, "Software Engineering Coverage of Computer Engineering Programs", TMMOB, Chamber of Electrical Engineers Journal, 2001.
- O. Demirörs, Ö. Tanrıöver, C. Hoşver, "Software Engineering Coverage of Graduate Programs: A Developing Country Perspective", Proceeding of Informatics Summit, 2001 September, Istanbul.