Ontology-based methodology for error detection in software design

Allyson M. Hoss
Louisiana State University and Agricultural and Mechanical College, ahoss1@lsu.edu
ONTOGOY-BASED
METHODOLOGY
FOR
ERROR DETECTION
IN
SOFTWARE DESIGN

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Allyson M. Hoss
B.S., State University of New York at Albany, 1983
M.S.C.S., University of Texas at Arlington, 1988
August 2006
Acknowledgements

There are several people whom I would like to acknowledge for their support during the development of this dissertation. To Dr. Doris L. Carver, my supervising professor, I express my sincere gratitude for her guidance throughout my graduate studies. She helped me to stay focused, to continually move forward toward my goals, as well as improve my literary writing and presentation skills. She provided invaluable advice with regard to conference and journal paper submissions. She also encouraged me at those critical times when I questioned my own abilities and became the most frustrated. To my committee members, Dr. Donald H. Kraft, Dr. Jianhua Chen, Dr. Young H. Chun, Dr. Earnest Mendrela, and Dr. Xiaoyue Jiang, I am thankful for the time and effort spent reviewing my research and the helpful comments that contributed to the success of this dissertation. To the members of my software engineering group, I appreciate the helpful suggestions given during my practice presentations. To my parents, Rai K. Schmalz and Guy C. Schmalz, I am forever grateful for their guidance and wisdom through the years that enabled me to develop the emotional and educational background that has helped me to progress this far in life. To my husband, James W. Hoss, and my children, Erik and Emma, I am forever grateful for the steadfast love, support, and inspiration they have given me each and every day. Their continual belief that I would one day finish this dissertation became the rock that saved me many a time from drifting off into the endless void of never ending research. Lastly, I would like to acknowledge the financial support of this research via a Louisiana State University Internal Grant.
# Table of Contents

Acknowledgements ........................................................................................................... ii

Abstract ......................................................................................................................... vi

Chapter 1 Introduction ..................................................................................................... 1
  1.1 Software Design ......................................................................................................... 1
  1.2 Software Design Consistency .................................................................................... 2
  1.3 Object-Oriented versus Agent-Oriented Software Development ............................. 4
  1.4 Problem Statement and Approach ........................................................................... 8
  1.5 Dissertation Overview ............................................................................................ 10

Chapter 2 Review of Literature ...................................................................................... 11
  2.1 Related Research Areas ............................................................................................ 11
    2.1.1 Ontologies .......................................................................................................... 11
    2.1.2 Requirements Specification .................................................................................. 12
    2.1.3 Software Design .................................................................................................. 12
    2.1.4 Consistency Management ..................................................................................... 12
    2.1.5 Knowledge Integration ......................................................................................... 12
    2.1.6 Tool Integration ................................................................................................... 12
    2.1.7 Agents ................................................................................................................ 12
    2.2 Related Research Comparison ................................................................................ 13
      2.2.1 Ontology Research ............................................................................................. 13
      2.2.2 UML and Model Checking Research ................................................................. 15
      2.2.3 Semantic Web Languages Research ................................................................. 15
      2.2.4 Tool Integration Research .................................................................................. 16
      2.2.5 Integrating Objects and Agents Research ......................................................... 17
      2.2.6 Consistency Management Research ................................................................ 17
      2.2.7 Summary ........................................................................................................... 20

Chapter 3 Components for Integration .......................................................................... 21
  3.1 Overview .................................................................................................................. 21
  3.2 Ontologies ............................................................................................................... 21
    3.2.1 Introduction ......................................................................................................... 21
    3.2.2 Ontology Development ...................................................................................... 21
    3.2.3 Ontology Language ............................................................................................ 22
      3.2.3.1 Introduction .................................................................................................... 22
      3.2.3.2 OWL ............................................................................................................. 23
  3.3 UML and OCL .......................................................................................................... 24
  3.4 KAOS ......................................................................................................................... 25

Chapter 4 Methodology for Objects to Agents (MOA) ................................................. 30
  4.1 Introduction .............................................................................................................. 30
  4.2 OSSD Model ............................................................................................................ 30
  4.3 MOA ......................................................................................................................... 33
    4.3.1 Overview ............................................................................................................. 33
    4.3.2 Lexical and Semantic Analysis ......................................................................... 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.3 Goal Thesaurus and Behavior Thesaurus</td>
<td>36</td>
</tr>
<tr>
<td>4.3.4 UML to OSSD Transformation</td>
<td>37</td>
</tr>
<tr>
<td>4.3.4.1 Overview</td>
<td>37</td>
</tr>
<tr>
<td>4.3.4.2 High-level Algorithms</td>
<td>39</td>
</tr>
<tr>
<td>4.3.4.3 Detailed Algorithms</td>
<td>44</td>
</tr>
<tr>
<td>4.3.4.4 Summary of UML to OSSD Model Transformation</td>
<td>58</td>
</tr>
<tr>
<td>4.3.5 MOA Consistency Checking</td>
<td>60</td>
</tr>
<tr>
<td>4.3.5.1 Overview</td>
<td>60</td>
</tr>
<tr>
<td>4.3.5.2 Consistency Checking of OSSD Model Constructs</td>
<td>60</td>
</tr>
<tr>
<td>4.3.5.3 Consistency Checking of Source Language Views</td>
<td>61</td>
</tr>
<tr>
<td>4.3.5.4 Consistency Checking Algorithms</td>
<td>63</td>
</tr>
<tr>
<td>4.3.6 OSSD Model to KAOS Transformation</td>
<td>63</td>
</tr>
<tr>
<td>4.3.6.1 Overview</td>
<td>63</td>
</tr>
<tr>
<td>4.3.6.2 OSSD Model to KAOS Transformation Algorithm</td>
<td>63</td>
</tr>
<tr>
<td>4.4 Summary</td>
<td>65</td>
</tr>
<tr>
<td>Chapter 5 Elevator Case Study</td>
<td>66</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>66</td>
</tr>
<tr>
<td>5.2 UML Representation of the Elevator System</td>
<td>66</td>
</tr>
<tr>
<td>5.3 Applying MOA to the Elevator System Case Study</td>
<td>73</td>
</tr>
<tr>
<td>5.3.1 UML Design to OSSD Model</td>
<td>73</td>
</tr>
<tr>
<td>5.3.2 OSSD Model</td>
<td>77</td>
</tr>
<tr>
<td>5.3.2.1 OSSD Model of the Elevator System</td>
<td>77</td>
</tr>
<tr>
<td>5.3.2.2 OSSD Model in OWL Notation</td>
<td>78</td>
</tr>
<tr>
<td>5.3.3 Consistency Checking</td>
<td>81</td>
</tr>
<tr>
<td>5.4 KAOS</td>
<td>81</td>
</tr>
<tr>
<td>Chapter 6 Evaluation of Methodology</td>
<td>85</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>85</td>
</tr>
<tr>
<td>6.2 Ontological Evaluation of the OSSD Model</td>
<td>85</td>
</tr>
<tr>
<td>6.2.1 Introduction</td>
<td>85</td>
</tr>
<tr>
<td>6.2.2 BWW Model</td>
<td>86</td>
</tr>
<tr>
<td>6.2.3 BWW Model Evaluation of the OSSD Model</td>
<td>89</td>
</tr>
<tr>
<td>6.2.3.1 Representation Mapping Evaluation</td>
<td>89</td>
</tr>
<tr>
<td>6.2.3.2 Interpretation Mapping Evaluation</td>
<td>93</td>
</tr>
<tr>
<td>6.3 Error Detection</td>
<td>96</td>
</tr>
<tr>
<td>6.3.1 Introduction</td>
<td>96</td>
</tr>
<tr>
<td>6.3.2 Representative Consistency Rules</td>
<td>97</td>
</tr>
<tr>
<td>6.3.3 Representation of Consistency Rules</td>
<td>97</td>
</tr>
<tr>
<td>6.3.4 Application of Consistency Rules</td>
<td>99</td>
</tr>
<tr>
<td>6.4 Transformation</td>
<td>100</td>
</tr>
<tr>
<td>6.4.1 Introduction</td>
<td>100</td>
</tr>
<tr>
<td>6.4.2 Set of Semantically Significant Features</td>
<td>101</td>
</tr>
<tr>
<td>6.4.3 Evaluation of UML to OSSD to KAOS Transformation</td>
<td>104</td>
</tr>
<tr>
<td>6.5 Summary</td>
<td>104</td>
</tr>
<tr>
<td>Chapter 7 Summary</td>
<td>106</td>
</tr>
<tr>
<td>7.1 Dissertation Summary</td>
<td>106</td>
</tr>
</tbody>
</table>
7.2 Contributions..............................................................................................................107
7.3 Future Work................................................................................................................108

References........................................................................................................................110

Appendix
A. London Ambulance Service Computer Aided Dispatch Case Study.............................127

B. Additional Data for the Elevator Case Study................................................................135

Vita....................................................................................................................................141
Abstract

Improving the quality of a software design with the goal of producing a high quality software product continues to grow in importance due to the costs that result from poorly designed software. It is commonly accepted that multiple design views are required in order to clearly specify the required functionality of software. There is universal agreement as to the importance of identifying inconsistencies early in the software design process, but the challenge is how to reconcile the representations of the diverse views to ensure consistency. To address the problem of inconsistencies that occur across multiple design views, this research introduces the Methodology for Objects to Agents (MOA). MOA utilizes a new ontology, the Ontology for Software Specification and Design (OSSD), as a common information model to integrate specification knowledge and design knowledge in order to facilitate the interoperability of formal requirements modeling tools and design tools, with the end goal of detecting inconsistency errors in a design. The methodology, which transforms designs represented using the Unified Modeling Language (UML) into representations written in formal agent-oriented modeling languages, integrates object-oriented concepts and agent-oriented concepts in order to take advantage of the benefits that both approaches can provide. The OSSD model is a hierarchical decomposition of software development concepts, including ontological constructs of objects, attributes, behavior, relations, states, transitions, goals, constraints, and plans. The methodology includes a consistency checking process that defines a consistency framework and an Inter-View Inconsistency Detection technique. MOA enhances software design quality by integrating multiple software design views, integrating object-oriented and agent-oriented concepts, and defining an error detection method that associates rules with ontological properties.

Keywords: agent-oriented, consistency, error detection, knowledge integration, object-oriented, ontology, software design, KAOS, UML
1 Introduction

1.1 Software Design

Most software development projects include the basic software engineering activities of analysis, specification, design, coding, testing and maintenance. The techniques used to actually implement these activities vary greatly as is evident in the variety of approaches including the traditional Waterfall Model, Spiral Model, Controlled-Iteration Model, and Prototyping Model. Software product development usually starts with analysis of the problem to be solved and creation of a requirements specification. Requirements specify the needs and desires of the customer while specifications detail how the software product will fulfill those needs and desires. Creating a requirements specification requires frequent interaction with the anticipated end-users of the software product and results in a document detailing the objectives, requirements, alternatives, and constraints of the product being developed as well as the environment in which the product will exist. Software design follows specification and focuses on decomposing and detailing the architecture of the software product, including the interfaces among its internal and external interfaces, and the behavior of the software product. The goal of software design is to produce a complete, consistent, unambiguous software design in a high-level design language. Common design methodologies include object-oriented, function-oriented, and agent-oriented. Whatever the methodology followed, critical issues addressed in software design include concurrency, data control, flow control, error handling, exception handling, performance, and quality.

Improving software quality continues to be a critical issue in software development. The most recent report by the Standish Group shows that 74% of software development projects do not deliver what the customer wants, on time and within budget and 94% of software development projects undergo project restarts [Frantzen]. Some studies have shown that 80% of software development effort is expended to debug and redevelop, and that more than 50% of the reasons for the rework is due to inadequate, inconsistent and imprecise requirements specifications [Davis]. Incorporating formalization techniques into software development can increase the success rate of software development projects; however, many software development practitioners are reluctant to adopt formal software specification techniques due to difficulties such as poor tool feedback; cost; poor guidance; isolation from other software products and processes; the low level of abstraction; and limited scope [vanLamsweerde3]. Most software today is developed using informal specification methodologies that lack formalized verification techniques.

Software quality attributes include characteristics such as correctness, completeness, robustness, maintainability, portability, testability, traceability, security, and quality [Abran et al.]. Most software engineers consider quality to be the most important part of software design. Ensuring both correctness and completeness is critical to ensuring quality. Analysis of software development projects shows that the cost and difficulty of fixing errors increases significantly as the project progresses. The earlier in the development life cycle that errors are discovered, the less time, effort, and cost are required to fix them. Errors detected later in the development life cycle usually result in not completing a project on time or within budget. Undetected errors in a product delivered to the customer can cause problems ranging from simple annoyances such as restarting a computer to serious accidents affecting human lives, as well as loss of customers, decreased sales, and increased repair costs [Torres-Pomales]. Common design errors include incompleteness, inconsistency, and redundancy. This research focuses on error detection in software design with an emphasis on designs represented in the Unified Modeling Language (UML) [OMG1]. We define the Methodology for Object to Agents (MOA) that integrates multiple design views including both object-oriented and agent-oriented concepts to facilitate the detection of software design errors.
1.2 Software Design Consistency

One aspect of error detection is ensuring consistency among the multiple views of a design that are required to understand system functionality from various perspectives. Inconsistencies are one of the most common, and most elusive, errors in software design. Although most researchers have a general understanding of the meaning of consistency, few agree on a specific definition of this term with regards to software design. The following definitions are just a few of the definitions given for consistency in software requirements and design:

- “any situation in which two parts of a specification do not obey some relationship that should hold between them” [Easterbrook & Nuseibeh];
- “no conflicting requirements and no (unintentional) non-determinism” [Pap et al.];
- “Different submodels of a model are called consistent if they can be integrated into a single model with a proper semantics….consistency of submodels ensures the existence of an implementation: if consistency is ensured, an implementation of submodels is obtained by implementing the integrated model” [Engels3 et al.]; and
- “…the use of constraints, algorithms, and tools to check that information described in one deliverable … is not contradicted by information described in another deliverable” [Paige1].

Some research on consistency in software design attempts to define consistency by defining inconsistency. Such definitions range from simply “contradictory design decisions” [Lange et al.] to more complex definitions such as:

- “any situation in which two parts of a specification do not obey some relationship that should hold between them” [Easterbrook & Nuseibeh];
- “a design is inconsistent if the design conveys conflicting information about the system, and/or violates predefined constraints” [Liu];
- “an inconsistency occurs whenever some relationship that should hold (of a model) has been violated” [Easterbrook]; and
- “the simultaneous assertion of a fact \( \alpha \) and its negation \( \neg \alpha \)” [Hunter & Nuseibeh].

It is critical to specify the term “consistency” in a precise and formal method and that there exists an automated mechanism for verifying consistency [Engels2 et al.]. However, “The consistency conditions depend on the diagrams involved, the development process employed, and the current stage of the development” [Engels2 et al.]. Some approaches to addressing inconsistency problems in UML define consistency in highly detailed terminologies that are specific to the associated specification languages [Astesiano & Reggio]. A different classification scheme for design inconsistencies, given in [Liu], presents three classes of design description inconsistencies: redundancy, conformance to constraints and standards, and change. A framework for UML consistency, given in [Derrick et al.], analyzes the problem of consistency in UML from a viewpoints (partial specification) perspective.

In this research, inconsistency means that either there exists a conflict, disagreement or variation within a single fact, behavior or constraint, or there exists a conflict, disagreement or variation among a set of facts, behaviors or constraints. General examples of inconsistency include: references to one fact, behavior or constraint by more than one name (a.k.a. aliasing); contradictions between descriptions of a behavior, fact or constraint; or inaccurate descriptions of behaviors, facts or constraints. A consistent design does not violate predefined rules and constraints of syntax and semantics of its associated model. However, because UML is by its very nature a collection of
various modeling notations that is specifically designed to encourage considerable freedom of specification, it does not have a precisely unified semantics with which to clearly specify and verify consistency and completeness issues.

Consistency can be viewed from two perspectives:

- intra-consistency: (a.k.a. horizontal consistency [Engels2 et al.]) consistency between two or more diagrams within a specific model; typically, these diagrams are at different levels of abstraction; for example, consistency between two different UML sequence diagrams of the same system that arise between the initial version of an UML class diagram and an enhanced version of that same UML class diagram that has includes such modification as additional features, deleted features or error corrections; and

- inter-consistency: (a.k.a. vertical consistency [Engels2 et al.]) consistency between two or more models within a specific system; typically these diagrams are at the same level of abstraction; for example, consistency between a UML class diagram and a UML sequence diagram of the same system.

This research addresses both intra-consistency and inter-consistency problems among the UML subset consisting of class, object, sequence, collaboration and statechart diagrams.

A two-dimensional classification of inconsistencies divides inconsistencies into structural or behavioral based upon analysis of UML class, statechart and sequence diagrams [Wagemann]. This classification does not include inconsistencies that arise due to violations of UML well-formedness rules because this type of syntactic (or static semantic) inconsistencies is typically enforced via the use of Object Constraint Language (OCL) [OCL] well-formed rules and detected by existing UML CASE tools such as xlinkit [Nentwich1 et al] and [Nentwich2 et al], Argi.YNK [Robbins et al.] [Robbins & Redmiles], and Rational Rose [Rational].

A behavioral inconsistency describes system behavior that is “incomplete, incompatible or inconsistent with respect to existing behavior or definitions” [Wagemann]. Behavioral inconsistencies are sub-divided into model-instance conflicts (such as incompatible definitions that affect multiplicity, navigation and abstract objects) and instance-instance conflicts (such as invocable, observable and incompatible behavior conflicts). A structural inconsistency describes situations where the system structure is “incomplete, incompatible or inconsistent with respect to existing behavior or definitions [Wagemann]. Structural conflicts are sub-divided into model-model conflicts (such as inherited association conflicts and dangling (type) references), model-instance conflicts (such as missing instance definitions), and instance-instance conflicts (such as disconnected models). The definition of consistency utilized in this research is based on the classification, detection and resolution techniques for inconsistencies in requirements presented in Knowledge Acquisition in autOmated Specification (KAOS), a goal-oriented approach to requirements engineering [Van Lamsweerde8], [KAOS] (see Chapter 3 for more background on KAOS). The general definition of inconsistent given in this approach is “a set of descriptions is inconsistent if there is no way to satisfy those descriptions all together” [Van Lamsweerde8]. A more detailed definition defines inconsistency as the “presence of unresolved conflict among goals” and “agents not able to perform their responsibilities” [Ponsard]. KAOS also defines incompleteness as the “presence of hidden assumptions”, “goals not operationalized,” and “lack of responsibility assignment for some constraints” [Ponsard].
Inconsistencies arise in software design for various reasons. Significant contributing factors include the incremental and distributed nature of software development, the definition of multiple views of a software system, and interactions among numerous stakeholders including customers, users, designers, and developers. Each type of stakeholder can view the system models from different perspectives due to varying levels of experience and responsibilities as well as different goals. Often, the initial software specifications are not complete and/or evolve as the software development lifecycle progresses. Lack of information, mistakes, and uncertainty also contribute to both inconsistencies and incompleteness in software design.

Inconsistencies can result in misinterpretations and/or multiple interpretations of critical design issues. Inconsistencies in software development models can also lead to various other problems including: difficulties in proving properties of the system such as reliability and safety; schedule delays; cost increases; and maintenance difficulties.

Handling inconsistencies in software design has been a widely discussed and debated issue for many years. While tolerating inconsistencies is sometimes beneficial and/or necessary, it is critical to identify and manage such inconsistencies early in the software design process. Most researchers and practitioners agree that detecting inconsistencies early in the software design process can improve the quality of software design with the ultimate goal of improving the resulting software product. Although it is necessary to allow some inconsistencies to exist, it is important to be able to clearly identify them. “It is undetected inconsistency that causes the most problems...known inconsistencies can be tolerated, provided they are managed carefully” [Nuseibeh]. It may even be detrimental to force consistency at all times during the development lifetime in order to “maximize design freedom, to prevent premature commitment to design decisions, and to ensure all stakeholder views are taken into account.” [Nuseibeh & Easterbrook]. Additionally, “rather than seeking to build a single consistent model, software designers need to reason about the inconsistencies and dependencies between a set of inter-related partial models” [Easterbrook].

1.3 Object-Oriented versus Agent-Oriented Software Development

Two common paradigms for software development are the object-oriented (OO) and agent-oriented (AO) methodologies. These two paradigms share many similarities, primarily due to the fact that the AO methodology evolved from the OO methodology. OO software development itself evolved from structured programming in the early 1960’s but did not become commonly used until the mid 1980’s. The basic concepts of OO software development include organizing a software representation of the world into a sets of discrete hierarchically arranged objects that contain structure and behavior, and associating with each object four characteristics: identity, classification, polymorphism, and inheritance [Rumbaugh et al]. OO software development also introduced the concepts of encapsulation (information hiding) and data abstraction. Within the OO paradigm, objects interact with each other, via messages exchanged with other objects, based on the objects’ internal state(s) and behavior. The AO methodology evolved in the late 1990’s via a merging of concepts derived from the OO methodology and artificial intelligence. Both objects and agents have identity, state, and behavior; in addition, they both communicate via interfaces. However, there exist several significant differences between objects and agents especially with regard to behavior. First, there is general agreement that a software object is a representation of a real-world object or concept that has one or more states, maintained via its variables, and behavior, implemented via its methods or operations. However, numerous definitions exist for the use of the word “agent” in software design with no generally agreed upon single definition. Most definitions do agree that three characteristics are common to a software agent: autonomy, situatedness and flexibility [Jennings et al.]. For an agent to be considered autonomous it must be able to control it own actions and internal state without any direct intervention from people or other agents. A simple object is considered passive, while an
agent is considered active. An agent is situated if that agent can receive from its environment sensory input and act upon that input which then causes an environmental change. An agent is flexible if it recognizes and reacts to changes in its environment within a reasonable period of time, exhibits goal-directed behavior, and can interact with other agents and people to complete its operations. Agents contain additional structures to represent and act upon the more complex concepts of goals, beliefs, and plans. Agents communicate with each other either directly via a high-level meta-language, referred to as an agent communication language (ACL), or indirectly via “blackboards” or “whiteboards”, shared communication areas, rather than using the simple message passing of OO. Other significant differences between objects and agents are the languages used to describe them [Huhns]. Object-oriented languages utilize the class structure as the basic abstraction, the object as the basic building block, methods/messages as the basic computation model, interaction patterns as the design paradigm and encapsulation, inheritance, polymorphism as the basic architecture. Agent-oriented languages use agent type as the basic abstraction, the agent as the basic building block, the processes of perception, reasoning and action as the basic computation model, the goal/belief/intention triumvirate as the basic design paradigm, and the manager/peer architecture. To summarize, “agent-based computing promotes designing and developing applications in terms of autonomous software entities (agent), situated in an environment, and that can flexibly achieve their goals by interacting with one another in terms of high-level protocols and languages [Zambonelli].

To acquire a perspective of the interrelationships between object-oriented and agent-oriented languages, as well as their relationship with formal requirements modeling languages, Figure 1 shows the relationships among a sub-set of these languages (derived from similar methodology genealogies [Sudeikat et al.], [Henderson-Sellers & Gorton]). Over 50 different object-oriented languages and techniques contributed to the development of UP/RUP and eventually UML. From this object-oriented (OO) pool also emerged agent-oriented languages such as AUML [Bauer et al.], Australian AI Institute (AAII) [Kinny et al.], MESSAGE, MaSE and MAS Common KADS [Iglesias et al.]. Formal requirements were developed on the foundation of Requirements Engineering (RE) from which emerged numerous Requirements Specification Languages (RSLs). These RSLs can be loosely grouped into Goal-Oriented (such as KAOS and TROPOS), Algebraic (such as Larch [Guttag & Horning] and OBJ [Goguen & Winkler]), State-based (such as VDM [Woodman & Heal], Z [Spivey], SLABS [Zhu]), and Operational (such as LISP [McCarthy], Prolog [Clocksin & Mellish] and Smalltalk [Tomek]. Of particular interest to the research in this research is the KAOS RSL.

![Figure 1: OO, AO and Formal Requirements Language Development](image-url)
because it draws upon not only Requirements Engineering but also Knowledge Engineering (KE) and Machine Learning (ML).

The OO software paradigm has several advantages: it is a mature software development paradigm; a wide variety of applications have been developed using it; there exists numerous object-oriented based tools, operating systems, programming languages, and databases; the OO paradigm’s concepts of encapsulation, polymorphism, and inheritance facilitates modular software development, reuse, and independent modular development respectively [Huhns]; a recent empirical study shows that 94% of the companies surveyed indicate that they use OO in the development of large-scale, complex information systems; 92% believe its reuse capabilities beneficial, and 70% considered its quality better than that of traditional system development [Paetau]. Weaknesses of the OO software development include: insufficient abstraction and support for object interaction [Huhns]; and the focus on objects tends to result in a bottom-up approach to design that result in the creation of large, generic libraries that are “hardly more useful that the massive procedure libraries they made obsolete” [Coggins]. Additionally, a recent empirical study shows that 65% of the companies surveyed found it difficult to acquire experienced OO software developers and 49% encounter efficiency problems [Paetau].

The AO software paradigm has significant advantages over the OO software paradigm. Agents are well-suited to handling complex systems because they are able to autonomously “engage in flexible, high-level interactions”…”self-awareness reduces control complexity since the system’s control know-how is taken from a centralised repository and localised inside each individual problem solving component” [Jennings]. Agents can participate in multiple interactions via multiple threads. In order to exchange message in OO, an object must know the address and receiving method of the receiving object whereas in AO, agents communicate using an agent communication language, with common semantics, that does not requiring knowledge of the receiving agent’s address or methods which facilitates interoperability at a level higher than OO message passing. Disadvantages of AO include: unpredictable interaction behavior, patterns, outcomes [Jennings] and insufficient off-the-shelf, mature agent-oriented methodologies [Shehory].

While AO software development has steadily gained converts in recent years, numerous software developers are using an OO approach to software development. Many are yet to be convinced that agents are not merely complex objects in disguise. Additionally, some software products, such as small systems and performance constrained systems, will continue to be developed using object-oriented concepts because they can not justify and/or tolerate the higher overhead required by agent-oriented processing nor its potentially unpredictable behavior. However, “agent-based computing has the potential to significantly improve our ability to model, design and build complex, distributed software systems” [Jennings et al.]. It appears that object-oriented and agent-oriented software development will continue to coexist for the foreseeable future. Referring to agents and objects, the Object Management Group acknowledges that “there is a very real need for these two related technologies to co-exist, and even more, to become better integrated, so agents can interact with objects and vice versa” [Odell]. Lastly, numerous legacy systems exist, based on object-oriented design or that have object-oriented interfaces that will eventually need to interact with newer agent-oriented software systems. It is, therefore, critical that future software development address the integration of these two worlds.

The integration of objects and agents is an active research area. Some research suggests that an object can be transformed into an agent by [Wagner2]: “treating its information items as its beliefs or knowledge”; “adding further mental components such as perceptions (in the form of incoming messages) and commitments”; and “providing support for agent-to agent communication on the basis of a standard agent communication language”. Additionally, objects are slowly adopting agent-
oriented concepts such as partial autonomy and situatedness via active objects versus passive objects, independent threads of execution, cooperating autonomous processes, and reactive components [Zambonelli & Omicini].

One benefit of integrating objects and agents is the enhancement of software interoperability. A commonly used definition of interoperability is “the ability of two or more systems or components to exchange information and to use the information that has been exchanged” [IEEE1]. Given that this definition is only a general definition, researchers and practitioners developed several frameworks to further define levels of software interoperability. One such framework models software interoperability at different abstraction levels [Howie, Kunz & Law]: physical interoperability (byte stream), data-type interoperability (simple data types), specification-level interoperability (abstract data types), and semantic interoperability (logic and rules). One distinction between integration and interoperability in software development focuses on data source versus software system; specifically, six levels of software construct interoperability (object, component, application, system, enterprise, and community) versus three levels of data integration (syntactic, structural, and semantic) [Obrst]. Software developers typically achieve interoperability either by standardizing the interfaces between applications and/or implementing software wrappers. Two methods commonly used to standardize interfaces between applications include the Object Management’s Group Common Object Request Broker Architecture (CORBA) [OMG2] and extensible Markup Language (XML) based [XML].

Using the CORBA Interface Definition Language, software engineers define object interfaces to access procedures within any object, via a request sent to that object, regardless of that object’s location within a distributed environment, the programming language, or the implementation platform utilized to create that object. CORBA is an application middleware for distributed object-oriented applications. It does not address agent-oriented concepts or the integration of agents and objects.

XML is an application independent and human-readable markup language that facilitates syntactic interoperability via the standardization of document and data structure as well as metadata syntax. Tags surround data elements to provide some semantic meaning. A schema language, such as Document Type Definition (DTD) or XML Schema, defines the document specific vocabulary and hierarchical structures for specific XML documents (a.k.a a common grammar). XML simulates semantic interoperability only if the data is exchanged within the same domain, so that both sender and receiver agree on the semantics of that data. However, XML cannot provide true semantic interoperability because it focuses on structural relationships in a document and cannot interpret the data within that document with regards to different domains. The meaning of the data is implicitly understood or specified in documentation accompanying the DTD. It is possible to map between two different DTDs via extensible Style Language (XSL) Transformation stylesheets. But, this requires potentially high overhead if several different DTDs exist. XML alone cannot handle the integration of agent-oriented and object-oriented concepts.

Wrapping software consists of code extensions that facilitate access and modification to internal data structures through abstract interfaces. It is possible to create agent wrappers around object-oriented software to facilitate interactions between agents and objects. Unfortunately, the creation, maintenance, and performance of such wrappers is costly; a unique wrapper must be developed for each non-agent-oriented system; any changes to the such system interfaces require updates to these wrappers; and, system performance often degrades due to wrapper execution.

There is a need for a conceptually higher level, less costly, and more comprehensive method to integrate objects and agents.
1.4 Problem Statement and Approach

This research addresses the problem of poorly designed software by developing a methodology to detect errors resulting from multiple views of a software design. MOA integrates software specification knowledge with software design knowledge, as well as object-oriented concepts with agent-oriented concepts, into a common information model called the Ontology for Software Specification and Design (OSSD), in order to identify errors among multiple design perspectives. MOA also utilizes the OSSD Model to facilitate the interoperability of formal requirements modeling tools and software design tools to detect complex errors in software designs. MOA contributes to the software design verification process by facilitating the identification and addition of error detection rules above and beyond that provided by the tools it interconnects. As an application of this methodology, MOA transforms a software design into an instance of the OSSD Model and then into a requirements specifications in order to deduce consistency properties of the specifications. These properties are then used to improve the original design.

Numerous modeling languages can represent a design from diverse views, including UML, the OPEN Modeling Language (OML) [Firesmith et al.], Specification and Description Language (SDL) [IEC], Z, and Petri-nets. In this work, we represent the source design using UML. UML, one of the most commonly used informal software modeling techniques, has become a de facto standard for modeling software systems. One of the major benefits of using UML is the extensive collection of various modeling notations specifically designed to encourage considerable freedom of specification. These notations enable software designers to specify partially overlapping views of the system to be modeled as shown in Figure 2; however, this flexibility often introduces inconsistencies into a software design. Unfortunately, UML does not have a precisely unified semantics to clearly specify and verify consistency. It is virtually impossible to adequately verify and validate software designs without precise semantics. Considerable research has detailed the problems and inadequacies caused by the lack of precise semantics in UML. Numerous theories, research projects, and a few practical tools have been developed to address this lack of precise semantics in UML. This research addresses undetected errors resulting from multiple views of software designs represented in UML.

We represent the common integrated model using an ontology. An ontological model provides a model and application independent method of integrating heterogeneous design models. Other models considered as a basis for the OSSD Model included UML Profiles, the Common Warehouse Model [OMG5], the ADORA model [Glinz] and work being performed by the Precise UML Group [pUML]. None of these models provided enough independence from their respective underlying
Ontologies provide the conceptual independence needed for a truly integrated model. We chose from among the numerous ontology representation languages to represent the OSSD Model using the Web Ontology Language (OWL), the World Wide Web Consortium (W3C) Recommendation for ontology representation [OWL]. Representing the OSSD Model with OWL will enable future interoperability with a wide variety of software engineering tools. Additionally, there exist numerous ontology building tools based on OWL, such as Protégé [Gennari], a tool for ontology modeling and knowledge base acquisition. Protégé, which is widely used with over 26,000 registered users, has several advantages over comparable ontology development tools [Alani]. Lastly, there exists an OWL Plugin to Protégé that facilitates the development of ontologies in OWL [Knublauch et al.].

Many specification languages exist to assist software developers with detailing the requirements of a software product. We chose to narrow analysis of formal requirements modeling languages to those that are agent-oriented due to the growing importance and success of agent-oriented approaches to software development. Examples of agent-based formal specification languages include KAOS, TROPOS [Bresciani], MaSE [DeLoach], MESSAGE [Evans1 et al.], and SLABS [Zhu]. In this work, we chose to represent the target requirements specification in KAOS, a goal-oriented approach to requirements engineering that has been used successfully to detect and resolve conflicts in requirements engineering. KAOS includes a wide range of requirements engineering activities including meta-modeling, obstacle recognition, and conflict management. KAOS performs formal reasoning utilizing real-time temporal logic notation to prove the completeness and correctness of its refinement process, obstacle analysis and conflict analysis. Classification of inconsistencies within the KAOS framework includes product-level inconsistencies (such as terminology, designation, or structure clashes), and assertion inconsistencies (such as conflict, divergence, competition, obstruction, realizability and concern meta-relationships). Additionally, KAOS has associated with it commercially available tools that can perform consistency verification, including Objectiver Requirements Management platform [Delor et al.] and an extension to Objectiver called the FAUST Toolbox for Formal Requirements Specification Analysis [Ponsard et al.].

Figure 3 portrays a high-level view of MOA. MOA extracts structure, data and relationships from the UML design; abstracts them into an ontology-based integrated model; and creates a specification level representation of the original UML design in a formal, agent-oriented requirements modeling language, namely KAOS. MOA transforms a software design specified using UML into an OSSD
Model instance, upon which consistency checking is performed, and then into a KAOS representation of its associated requirements specifications level in order to utilize formal verification tools to deduce consistency properties of the specifications. The verification tool associated with the agent-oriented model then processes the generated specification and produces a report that lists the inconsistencies in the original UML design. For each inconsistency identified, the UML developer can then determine whether it should be resolved or permitted to exist. Any changes to the sources of these inconsistencies in the original UML design are manually updated. This research assumes that the UML design includes all available requirements level information. The existence of a formal and/or testable requirements specification is not relevant to this research. Some implementation details resident in the UML design are not represented in MOA if they are not relevant to generation of the specification level representation.

The primary motivation for this research is to improve the quality of software designs through enhanced error detection in order to improve the quality of the resulting software product. A second motivation is the need for improved methods to promote interoperability among different design methodologies. A final motivation addresses the need to improve software development tool interoperability that can help improve the design process. Interoperability in these last two motivations implies the capability of software components to interact cooperatively with each other. The National Institute of Standards and Technology (NIST) has defined within the recent Automated Methods for Integrating Systems (AMIS) project that “the object of the integration process is to get separately designed resources to work together to accomplish some end goal” [Barkmeyer].

1.5 Dissertation Overview

Chapter 2 reviews related research. Chapter 3 presents background information on the integration components that are integral to the methodology presented in this dissertation: ontologies, UML and OCL, and KAOS. Chapter 4 introduces MOA including it analysis, transformation and consistency checking algorithms. Chapter 5 presents an example application of the MOA via a case study analysis of an elevator system. Chapter 6 presents evaluations of the ontology model, the error detection, and transformation technique employed in this methodology. Chapter 7 includes a summary of this research and ideas for future work. Appendix A describes the application of MOA to a safety-critical, real-time, and distributed system case study, the London Ambulance Service (LAS) Computer Aided Dispatch (CAD) System [Finkelstein & Dowell].
2 Review of Literature

2.1 Related Research Areas

The research presented in this dissertation touches upon seven areas of related research as shown in the two Venn Diagrams given in Figure 4: Ontologies, Software Design, Requirements Specification,

![Figure 4: Related Research Areas](image)

Consistency Management, Knowledge Integration, Agents, and Tool Integration. This research, represented by MOA, is shown in the center of both Venn Diagrams. Although it is possible that additional overlaps exist between the two Venn Diagrams, the arrangement is Figure 4 portrays MOA’s relationship to related research in an easily understood format. This Section presents a brief overview of each of the seven related research areas with a narrowing focus on its relationship with MOA. Section 2.2 provides examples of related research sources that overlap two or more research areas and a discussion of those areas that overlap three or more areas.

2.1.1 Ontologies

Ontologies have been utilized for many years in the fields of philosophy, linguistics and artificial intelligence. They are becoming a popular technique to solve problems in a variety of applications as described in a recent survey on ontology-based applications [Gargantilla]. Ontologies have become the underlying information model in a variety of software development areas including multi-agent systems, natural language processing, knowledge engineering, information retrieval, digital libraries, and electronic commerce. They offer the potential of supporting and integrating the difficult tasks of representing extensive and diverse knowledge, searching that knowledge, and presenting that knowledge in a user-friendly format. The O SSD Model is based on ontological concepts to represent software design and requirements specification knowledge. Chapter 3 contains additional background on ontologies.
2.1.2 Requirements Specification

Most software development projects include the basic software development concepts of analysis, specification, design, coding, testing and maintenance. The techniques used to actually implement these concepts vary greatly as is evident in the variety of approaches including the traditional Waterfall Model, Spiral Model, Controlled-Iteration Model, and Prototyping Model. There are numerous methods and languages for specifying software requirements, each with its own associated verification techniques and tools. Some tools and techniques are useful in improving the quality of software development by identifying errors early in the development process. Chapter 3 reviews one such requirements specification language, KAOS, which is the target specification language selected for integration with UML via MOA.

2.1.3 Software Design

The software design related research area focuses upon methods and tools to produce a complete, consistent, unambiguous software design in a high-level design language. The MOA assumes the existence of a software design that is then transformed into a requirements specification to facilitate the application of a formal requirements modeling tool to identify errors in the original software design. Chapter 1 provides an overview software design in general and Chapter 3 provides an overview of UML, the source design language selected for integration with KAOS via MOA.

2.1.4 Consistency Management

MOA is related to the numerous methodologies that have been developed to address software design inconsistencies. An overview of consistency in software design is provided in Chapter 1. Many of these methodologies are manual methods developed to detect inconsistencies while some are partially automated. Only a few of these automated approaches have tools available for industrial use. A limited number of approaches offer guidance on diagnosis, tracking, or resolution of software design problems.

2.1.5 Knowledge Integration

The goal of knowledge integration is to combine specialized knowledge from a variety of sources into one synthesized form that is better than the sum of its parts. MOA utilizes ontological concepts to integration software requirements specification knowledge with software design knowledge. This integrated model can then be used to detect errors in the software design as well as use to integrate other software engineering tools with the end goal of improving the quality of the software.

2.1.6 Tool Integration

Given the wide diversity of software engineering tools available to developers today, it is becoming increasingly important for these tools to be able to access common information sources and have a shared, common understanding of these sources. MOA provides one way to integrate software design tools with tools for software requirements specification.

2.1.7 Agents

Agent-oriented approaches to software development have been steadily gaining popularity in recent years as an alternative to the object-oriented methods. It is appealing to consider developing software that can react autonomously and/or cooperatively with other software agents to events its
environment. Acknowledging the importance of this trend, MOA transforms a software design into an agent-oriented requirements specification. Chapter 1 provides an overview of agent-oriented versus object-oriented software development.

2.2 Related Research Comparison

Figure 5 shows specific research sources in the seven related research areas presented in Section 2.1. Table 1 provides a key to identify the reference sources, represented as numbers in the two Venn diagrams, most closely related to this research. It lists sources with their corresponding reference identification. A source can be represented in one or both Venn Diagrams. A detailed review is given for sources that exist in three out of four research area in a Venn diagram.

2.2.1 Ontology Research

Related research methodologies employing ontologies to improve the software development process include the following.

1) An ontological engine is integrated into a CASE-tool that assists with the creation, verification, and validation of software artifacts (not designs) used throughout the software development life cycle, such as classes, patterns, and diagrams [Deridder].
2) Ontologies organize design knowledge on the functional decomposition of engineering devices based on functional ontologies into a framework of systematization in order to make that knowledge consistent and relatively domain independent [Kitamura & Mizoguchi].
3) A common ontological model integrates network management information models [Vergara]; a Merge and Map (M&M) method merges the network management information into the common model and then maps instances of each input model to the common model via a mapping ontology.
4) An agent-based requirements refinement model represents requirements as state transition diagrams uses a domain ontology for the detection, diagnosis, and resolution of semantic inconsistencies in software requirements specifications [Zhu & Zhi].
5) Ontologies integrate software engineering tools in a knowledge based system development environment to facilitate knowledge integration among software engineering tools in order to avoid redundancies and inconsistencies [Falbo et al.]; specifically, an ontology of software development process is created on top of domain ontologies of software development activities, procedures to be performed to carry out those activities, and resources required to complete those procedures.

6) Ontologies in an agent-based system, InfoSleuth, integrate heterogeneous, distributed information, and tools [Fowler et al.]; six types of agents (user agents, broker agents, ontology agents, resource agents, value mapping agents, and multi-resource query agents) interact with each other and reason via a common ontological model of information management.

<table>
<thead>
<tr>
<th>Ref. ID</th>
<th>Description</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aredo</td>
<td>Tool integrates UML and PVS for verification</td>
<td>1</td>
</tr>
<tr>
<td>Beato et al.</td>
<td>Tool to transform UML to SMV for formal verification</td>
<td>2</td>
</tr>
<tr>
<td>Botelho et al.</td>
<td>Integrating ontologies and databases with agent communication language</td>
<td>3</td>
</tr>
<tr>
<td>Brandao</td>
<td>Ontology as specification for verification of consistency of Multi-agent system design models</td>
<td>4</td>
</tr>
<tr>
<td>Briand et al.</td>
<td>Rules to detect inconsistencies in UML designs</td>
<td>5</td>
</tr>
<tr>
<td>Chen</td>
<td>Ontology for inconsistency handling in requirements specifications</td>
<td>6</td>
</tr>
<tr>
<td>Chinorean et al.</td>
<td>Integrating ontologies and databases with agent communication language</td>
<td>3</td>
</tr>
<tr>
<td>Corradini et al.</td>
<td>Agent-oriented approach to tool integration using wrappers and workflows</td>
<td>8</td>
</tr>
<tr>
<td>Deridder</td>
<td>Integrating ontologies into CASE tool for software artifact creating, verification, validation</td>
<td>9</td>
</tr>
<tr>
<td>Dong</td>
<td>Semantic Web environment to integrate formal specification languages</td>
<td>10</td>
</tr>
<tr>
<td>Egyed2</td>
<td>Pattern-based approach to integrating design views in UML</td>
<td>11</td>
</tr>
<tr>
<td>Falbo et al.</td>
<td>Software development process ontology for knowledge integration among SE tools</td>
<td>12</td>
</tr>
<tr>
<td>Fowler et al.</td>
<td>Agent-based system that utilizes ontologies to integrate heterogeneous, distributed information, and tools</td>
<td>13</td>
</tr>
<tr>
<td>Guizzardi et al.</td>
<td>Integrated agent-oriented methodology; knowledge management system</td>
<td>14</td>
</tr>
<tr>
<td>Jin</td>
<td>Ontology and tool adapters provide interoperability of software reengineering tools</td>
<td>15</td>
</tr>
<tr>
<td>Kalfoglou</td>
<td>Ontology to identify conceptual errors in software specifications</td>
<td>16</td>
</tr>
<tr>
<td>Kitjongthawonjul &amp; Khosla</td>
<td>Integration of objects and agents via task-based problem solving adapters</td>
<td>17</td>
</tr>
<tr>
<td>Kitamura &amp; Mizoguchi</td>
<td>Ontological organization of functional design knowledge</td>
<td>18</td>
</tr>
<tr>
<td>Kozlenkov &amp; Zisman3</td>
<td>Goal-Based; identify and resolve inconsistencies</td>
<td>19</td>
</tr>
<tr>
<td>Liu</td>
<td>Rule-based inconsistency classification</td>
<td>20</td>
</tr>
<tr>
<td>Mota</td>
<td>Mapping UML to NuSMV</td>
<td>21</td>
</tr>
<tr>
<td>Nentwich2 et al.</td>
<td>XML-based tool to check consistency of distributed and heterogenous documents</td>
<td>22</td>
</tr>
<tr>
<td>Perini</td>
<td>Integrates agent-oriented modeling tool with software verification tool</td>
<td>23</td>
</tr>
<tr>
<td>Ramalho &amp; Robin</td>
<td>Maps UML to a formal knowledge representation language for verification</td>
<td>24</td>
</tr>
<tr>
<td>Silva &amp; Lucena</td>
<td>Combines concepts of agents, objects, and UML into a multi-agent modeling language</td>
<td>25</td>
</tr>
<tr>
<td>Silva et al.</td>
<td>Integrating OO and AO concepts into an ontology for multi-agent systems</td>
<td>26</td>
</tr>
<tr>
<td>VanLamsweerde8</td>
<td>Goal-Oriented approach to detect, handle, resolve inconsistencies in requirements</td>
<td>27</td>
</tr>
<tr>
<td>Vergara et al.</td>
<td>Ontology for integrating network management tools</td>
<td>28</td>
</tr>
<tr>
<td>Zhu &amp; Zhi</td>
<td>Agent-based requirements refinement model including a domain ontology; detect, diagnose, resolve inconsistencies in software requirements</td>
<td>29</td>
</tr>
</tbody>
</table>
MOA differs from the related research in ontologies by combining object-oriented and agent-oriented concepts into its common model, and by utilizing its ontological common model, with associated ontological reasoning, to detect errors in the domain of software design.

2.2.2 UML and Model Checking Research

Related research integrating UML with model checking and/or theorem proving tools to verify UML designs includes the following.

1) The automatic mapping of UML diagrams (Class, Object, Statechart, Activity, and Collaboration) into a formal knowledge representation language, Concurrent Transaction Frame Logic (CTFL) programs is performed as a part of the Model-Oriented Development with Executable Logical Object Generation (MODELOG) project [Ramalho & Robin]; CTFL programs can then be processed by an inference engine to perform consistency and completeness verification as well as other model analysis, refinement, and refactoring.

2) A tool integrates UML and PVS that maps UML modeling constructs (obtained from UML Class, Sequence and Statechart Diagrams) into the specification language Prototype Verification System (PVS) for verification via PVS type-checkers, theorem-provers, and model-checkers [Aredo].

3) A Tool for the Active Behavior of UML (TABU) inputs a UML specification formatted in XMI and automatically generates a Symbolic Model Verifier (SMV) specification, which is then processed by a SMV tool model checker [Beato].

Model checking tools, such as SMV, input a description of a software system as a finite-state machine including properties of the system specified in temporal logic. The model checker then determines if the system satisfies those properties by performing a search of the state space defined by the state machine. If the search produces a state in which the temporal logic is not satisfied, it outputs the sequence of states leading up to the point at which the inconsistency was identified. For example, the types of properties that can be verified using TABU concern proof that a state machine and/or object activity is in a particular state, a signal or event is produced, and a comparison of attribute values.

Approaches utilizing model checkers and theorem provers are similar to the MOA because they attempt to integrate formal methods with semi-formal methods to verify UML behavior. However, these approaches verify only a limited number of consistency and completeness problems. MOA facilitates the definition of numerous syntactic and semantic rules to assist with error detection. Additionally, the model checkers and theorem provers do not provide the higher-level detailed error detection results or the obstacle recognition and conflict management techniques available in requirements engineering tools with which MOA is designed to interface, such as KAOS.

2.2.3 Semantic Web Languages Research

Related research utilizing Semantic Web [Berners-Lee] languages in software development includes the following.

1) Markup languages specify software requirements to facilitate detection and resolution of inconsistencies in those specifications via a CASE tool (SC-CHECK) [Chen]. The “semantic markup involves placing tags that point to pre-defined web-based ontologies for explicating the meaning of elements of a specification being marked up” [Chen]. The original software
requirements specifications, defined in one of three formats ranging from textual to informal (UML), to formal specifications (KAOS), are manually marked up in DAML+OIL (DARPA Agent Markup Language [DAML] + Ontology Inference Layer [W3C2]) format. The SC-CHECK tool combines editors for ontology and rule management, annotators for markup management, an ontology repository, a formal set of rules, a set of consistent specifications, and an inconsistency monitor to detect and resolve inconsistencies. The inconsistency monitor consists of an inference engine, theorem prover or reasoner that identifies if a specification violates specified consistency rules. The SC-CHECK tool is in its preliminary stages. The examples and case study given address only a very small subset of either the UML or KAOS languages.

2) The Semantic Web languages RDF (Resource Description Framework) [W3C3] and DAML create a Semantic Web environment that integrates different formal specification languages such as Z and CSP [Dong et al.].

3) An XML-based tool, xlinkit, facilitates the consistency checking of distributed and heterogeneous documents [Nentwich2 et al]. A document is any source of structured or semi-structured data represented in XML including software engineering documents such as requirements specifications, design models, and source code. Xlinkit utilizes a rule language, based on first-order logic, to specify assertions regarding consistency relationships between elements in the distributed documents; it associates constraints with the hyperlinks that interconnect elements of the distributed documents.

We utilize the semantic web language OWL to define the common model at the heart of MOA, the OSSID Model. The research closest in concept to MOA is xlinkit [Nentwich2 et al]. However, the syntactic checks performed by the xlinkit tool cannot contain the semantic information nor perform the semantic reasoning that is possible in the ontologically based MOA because XML focuses on structural relationships in a document and does not interpret the data within that document with regards to different domains. XML does provide the syntactic and structural interoperability upon which ontology languages can provide true semantic interoperability. “Ontologies in the form of logical domain theories and their knowledge bases offer the richest representations of machine-interpretable semantics for systems and databases in the loosely coupled world” [Obrst].

2.2.4 Tool Integration Research

Research integrating tools and/or software development methodologies includes the following.

1) Integration of AIXO (Any Input XML Output) wrappers to facilitate XML-based wrapping of tools, agents to manage and coordinate heterogeneous activities, and workflows to specify and coordinate the series of activities [Corradini et al.].

2) Integration of an agent-oriented modeling tool, TAOM, with software verification tools such as the T-TOOL, a type of model-checker [Perini]; both TAOM and the T-TOOL are based on the TROPOS Methodology for requirements engineering.

3) An approach to develop knowledge management systems [Guizzardi et al.], the Agent-oriented Recipe for Knowledge Management Systems Development, integrates two agent-oriented methodologies: the TROPOS Methodology for requirements engineering and the ontology-based Agent-Object-Relationship.

MOA is similar in concept to the integration of TAOM with the T-TOOL [Perini]. However, both TAOM and the T-TOOL focus on agent-oriented concepts while MOA integrates object-oriented and agent-oriented concepts.
2.2.5 Integrating Objects and Agents Research

Research addressing the integration of objects with agents focuses on either the development of agent-oriented systems utilizing new conceptual frameworks or the implementation of agents using OO concepts. Such research includes the following.

1) The Taming Agents and Objects (TAO) conceptual framework [Silva et al.] defines an ontology consisting of both OO and AO concepts essential for developing a multi-agent system (MAS). These concepts are grouped into three categories of abstraction:
   a) fundamental (objects and agents);
   b) grouping (the organizations and roles required to represent complex collaborations);
   and
   c) environmental (constraints, events, and characteristics of the environment in which the objects and agents exist).

The TAO conceptual framework combined with concepts from the UML metamodel is the basis for a MAS Modeling Language (MAS-ML) [Silva & Lucena] which, in turn, is the basis for a MAS ontology [Brandao et al.] used to verify the consistency of MAS design models.

2) Integrating OO domain ontologies and OO databases with an agent communication language (ACL) is the goal of an alternative approach to OO and AO integration [Botelho et al.]. In this research, they augment the ACL with OO domain ontological concepts and translate the ACL via a one-to-one mapping to the OO database entries.

3) Task-based problem solving adapters integrate object and agents into an integrated architecture for information system and database system development [Kitjongthawonkul & Khosla].

4) Considerable research exists regarding implementing agents using OO techniques by augmenting the OO methodologies and/or programming languages to accommodate AO concepts. A recent empirical study compares a pattern-oriented approach and an aspect-oriented approach to MAS design and implementation [Garcia et al.]. The Agent Unified Modeling Language (AUML) [Bauer et al.] is an extension of UML that provides modeling mechanisms for describing multi-agent interactions; it extends the OO concept of an active object and provides agent interaction protocols, agent roles, and agent lifelines including multiple threads of interaction.

MOA most closely resembles the TAO conceptual framework research [Silva et al.] and specifically its use in the MAS ontology to verify consistency of MAS design models [Brandao et al.]. However, MOA assists with error detection in object-oriented designs, specifically for designs specified using UML.

2.2.6 Consistency Management Research

Since the software design language used in this research is UML, this Section concludes with a more in-depth analysis of the related research regarding approaches to consistency management in software design and requirements engineering with a specific focus on software engineering with UML. In [Spandoudakis & Zisman], techniques and methods for handling inconsistencies are organized into the following six activities: detection of overlaps, detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, tracking of inconsistencies, and specification and application of a management policy for inconsistencies. A different overview of UML consistency
management organizes approaches into Meta-Modeling approaches, Constraint Language approaches, and Formal Notation approaches [Elaasar].

MOA can be categorized among the approaches that detect inconsistencies in UML designs based on the concept of mapping UML to the input specification required by model checking or theorem proving tools. These approaches include, but are not limited to, the Prototype Verification System (PVS) [Aredo], Concurrent Transaction Frame Logic (CTFL) [Ramalho], and NuSMV [Mota et al.]. These tools are useful in detecting inconsistencies; however, they do not provide the higher-level detailed verification results or the obstacle recognition and conflict management techniques of a formal requirements engineering tool such as those that support KAOS.

An empirical study quantifying inconsistency and incompleteness of UML designs divides approaches to solving UML inconsistency problems into two categories: complete approaches and partial approaches [Lange et al.]. A complete approach provides a formal semantic definition for all UML. A partial approach focuses upon defining the semantics for a subset of UML in order to assist with identifying inconsistencies. The category of partial approaches can be further subdivided into two groups: formal approaches in which subsets of UML designs are mapped to formal methods; and design-oriented approaches in which meta-model analysis of designs specified in UML and OCL format is performed to analyze design properties and then define meta-model consistency rules [Lange et al.]. The following examples of each approach would be placed in the overlap between software design and consistency management in Figure 5. Examples of formal partial approaches are algebraic abstract data types [Andre et al.], classical algebraic specifications [Astesiano & Reggio], description logic [Mens et al.] and [Wagemann], category-theoretic framework for analyzing fuzzy viewpoints [Sabetzadeh & Easterbrook], abduction [Nuseibeh & Russo], conceptual graphs [Sunetnanta & Finkelstein], attributed graph grammar [Tsiolakis & Ehrig], and graph transformation to a variety of formats that serve as input to a theorem prover that verifies system properties [Kyas & Fecher] and [Paige2]. Examples of design-oriented partial approaches are rule-based or expert systems [Briand et al.], [Liu] and [Suourrouille & Caplat], OCL constraints [Chiorean et al.], [Gomaa & Wijesekera] and [Bodeveix et al.], graph-grammar [Wagner et al.], pattern-based analysis [Egyed2], goal driven knowledge-based system [Kozlenkov & Zisman3], and based on XML [Nentwich2 et al]. MOA can be categorized as a design-oriented partial approach.

Few examples of complete approaches exist that attempt to provide a formal semantic definition for all UML. Considerable research has been performed during the past few years detailing the problems and inadequacies caused by the lack of precise semantics in UML [Andreopoulos]. Imprecise semantics make adequate verification and validation virtually impossible. Numerous theories, research projects, and a few practical tools have been developed to address this lack of precise semantics in UML. The underlying concept of most of these approaches is to formalize the semantics of UML. With formal verification, a property of the software specification is usually mathematically proven. Attempts to formalize UML have encountered numerous problems due to the very nature of UML including its “heterogeneous semi-formal notations”, it multiple viewpoint perspective, its extendable features (such as stereotypes and tagged values) and the fact that UML does not “prescribe any particular development process” [Andreopoulos]. There are basically three approaches to formalizing UML [Evans2 et al.]:

1) Supplemental Approach: transforming the semantics of the informal UML model to a formal specification language (such as Z, Object Z [Roe et al.], B [Marcano et al.]) or to an intermediate mathematical notation (such as Petri Nets and Kripke automata [VIATRA], and Abstract State Machines [Ober]) so that the UML semantics can be proven via the formal
semantics of the specification language itself or via a verification tool (such as model checkers [Engels1 et al.], and theorem provers [Paige2]); other intermediate formats include Algebraic Specifications [Peng] or Object Algebras [Hussmann] with which properties can also be proven mathematically;

2) O-O-extended Formal Language Approach: extending an existing formal notation (such as Z) with the object oriented features of UML thereby creating a new formal notation (such as Object-Z) so that the semantics of UML Meta-Model can be formalized and proven; and

3) Methods Integration Approach: incorporating formal specification notation into the informal UML meta-model in order to prove properties by manipulation of the graphical object-oriented model without reference to the underlying formalism.

A recent classification of consistency checking approaches defines three unique categories: consistency by analysis; consistency by monitoring; and consistency by construction [Snoek et al.]. Most approaches fall under the first category, consistency by analysis, in which inconsistency detection algorithms are developed and executed several times against developing models. Such algorithms are manual or automated, and result in a generated report that is used to update the original model. MOA can be classified as a consistency by analysis approach. Consistency by monitoring enables the incremental development of a model that is always consistent. Inconsistent updates to the model are simply not allowed. With consistency by construction, a tool automatically generates consistent specifications via automatic inference. We classify MOA as a consistency by analysis approach.

Several papers published recently define rules to detect inconsistencies in UML designs but most define only a handful of rules; one notable exception defines 100 rules [Briand]. Only a few papers define classifications or frameworks for organizing the types of inconsistencies. Examples of consistency frameworks are: a classification based on five design issues: syntax versus semantics, static versus dynamic, intra-model versus inter-model, multi-level, and error type [Elaasar & Briand]; a classification that presents three classes of design description inconsistencies: redundancy, conformance to constraints and standards, and change [Liu]; a constraint classification that addresses the various domains that are included in the development process: paradigmatic (typically those detectable by UML modeling tools), profiles and stereotypes, modeling process, target/implementation specific, and target domain specific [Suourrouille & Caplat]; a three-tier classification of inconsistencies based on a view integration framework that organizes over 50 different types of inconsistencies [Egyed1]; and, lastly, classification of seven intra-specification and inter-specification inconsistencies within UML structural, interaction and statechart diagrams: vocabulary, integrity, abstraction, definition, coherence, configuration, and contract [Kozlenkov & Zisman1]. This last classification is a component of a goal-based approach to discovering, recording, analyzing and resolving inconsistencies in software specifications written in UML in which axioms define goals that collectively represent the UML. MOA provides a consistency framework based on the constructs of the OSSD Model.

Although several goal-oriented approaches exist addressing inconsistencies in software specifications, we found only one other approach addressing inconsistencies in software design that utilizes goals. This approach, referred to as a goal-driven knowledge-based approach [Kozlenkov & Zisman3], is not based upon the KAOS approach to goal-oriented requirements engineering. In this approach, goals are defined via axioms to represent the UML meta-model as a knowledge base. Abduction is used to process information in this knowledge base. MOA includes the concept of goals its error detection process through inclusion of a goal construct in its OSSD Model.
2.2.7 Summary

We categorize MOA as a combination of overlap detection and inconsistency detection based on violations of consistency rules, a design-oriented partial approach with a unique ontological perspective that includes an integrated model that provides a model and application independent method of integrating heterogeneous design models, and a consistency by analysis approach. MOA differs from the related research in several ways.

1) MOA provides a common ontological model to integrate multiple views of software design. It is this ontological model that represents semantic design information, thereby enabling the application of ontological reasoning to assist with the detection of complex semantic errors in software designs.

2) MOA enables definition of semantic rules above and beyond the typical syntactic checks. Most software design consistency checks are syntactic, based on the well-formed rules (WFR) specified in the UML 2.0 Specification that address primarily the syntactic inconsistencies within a given UML diagram. MOA facilitates the definition of numerous syntactic and semantic rules to assist with error detection.

3) MOA integrates OO and AO concepts of software design in its error detection ontology. Few error detection techniques for software design take into consideration the integration of AO and OO concepts. As mentioned in Section 1.3, it is critical that future software development address the integration of these two worlds. Additionally, existing techniques that encompass solely object-oriented concepts, specifically UML related techniques such as profiles and stereotypes, can make it difficult to address the complexity and abstractions of the more frequently reoccurring agent-oriented concepts.

4) MOA introduces a new classification framework for consistency rules. This framework enables a broad definition of consistency rules that includes a wide variety of potential interactions.
3 Components for Integration

3.1 Overview

This chapter reviews the basic integration components used in MOA: ontologies; the source design language UML with OCL; and the target requirements specification language KAOS. Included with the overview of ontologies is an introduction to the ontology language in which the OSSD Model was developed. Additionally, a comparison of object-oriented versus agent-oriented software development is provided to show the relationships between UML and KAOS.

3.2 Ontologies

3.2.1 Introduction

Viewed simply, an ontology structures knowledge that consists of hierarchically arranged concepts, properties associated with these concepts, relationships between the concepts, and rules that govern these relationships. However, no standard definition of ontology exists. One of the more commonly quoted definitions, originating with [Gruber] and enhanced by [Borst], defines ontology as a formal, explicit specification of a shared conceptualization. An ontology is, therefore, an abstract model of some area of knowledge, also known as a domain, which is used to share information in that domain. It should consist of explicitly defined and generally understood concepts and constraints that are machine understandable.

An ontology should be formalized if it is to be understood and managed by a computer. Although there exists several different formal definitions of an ontology, the OSSD Model is based on one of the more commonly referenced definitions [Maedche & Staab] and is graphically portrayed by a simple example given in Figure 6:

- a set of Concepts C
- a set of Relations R
- two sets of strings describing lexical entries L: \( L^C \) and \( L^R \)
- a concept taxonomy: \( H^C \)
- a relation taxonomy: \( H^R \)
- two sets of relations associating concepts and relations with corresponding lexical entries: F and G
- a set of axioms describing constraints on the ontology: A

3.2.2 Ontology Development

There exist a variety of ontology development techniques. As with many aspects of ontologies, no standard ontology development methodology has yet emerged. A recent survey of the current approaches is given in [Cristani & Cuel]. In general, the process of building an ontology usually takes four steps including specification, conceptualization, formalization and implementation [Kayed]. Popular methodologies that have been used to build ontologies include: Toronto Virtual Enterprise [TOVE], ENTERPRISE [Uschold et al.], METHONTOLOGY [Fernandez et al.], and Ontology Development 101 [Noy & McGuinness].

The “Ontology Development 101” was selected from among the variety of ontology development methodologies because it is promoted as the beginner’s guide to ontology development using Protégé [Gennari], a tool for ontology modeling and knowledge base acquisition. It also includes guidelines
to: ensure that class hierarchies are correct; analyze class sibling relationships; permit multiple
inheritance; identify disjoint sub-classes; limit scope; and, assist with distinctions between class,
property, and instance definitions. The basic steps proposed by “Ontology Development 101” are: 1)
identify the domain and scope of the ontology, 2) evaluate reusing an existing ontology, 3) identify
important terminology to be used in the ontology, 4) identify classes and their hierarchical
relationship, 5) identify class properties, 6) define the characteristics (or facets) of the class
properties, and 7) create the class instances. Development of the OSSD Model followed steps 1
through 6 of the “Ontology Development 101”. Step 7 is repeated each time MOA is applied to a
unique UML design.

3.2.3 Ontology Language

3.2.3.1 Introduction

Numerous languages have been developed to represent ontologies. We analyzed two of these with
regards to representing the OSSD Model: the Knowledge Interchange Format (KIF) and the Web
Ontology Language (OWL). The Knowledge Interchange Format (KIF) [Genesereth] is a low-level
language based on first-order predicate logic. It is not intended as a user-level language. It has
extensions that can be used to represent definitions and meta-knowledge. Ontolingua [Farquhar] is
an example of an ontology-editing tool that is based on the KIF and developed by Stanford
University for the construction and maintenance of ontologies. OWL is the World Wide Web
Consortium (W3C) Recommendation for ontology representation. OWL has a significant advantage
over modeling in software design languages such as UML because UML does not support
specification of domain knowledge and domain constraints other than in textual format; however,
OWL does provide the capability to represent domain knowledge [Neuhold et al.].

Representing the OSSD Model with OWL will enable future interoperability with a wide variety of
software engineering tools. Protégé is widely used to build OWL ontologies, with currently over
26,000 registered users. It has several advantages over comparable ontology development tools [Alani]. Additionally, there exists an OWL Plugin to Protégé that facilitates the development of ontologies in OWL [Knublauch]. The next chapter provides an overview of OWL.

### 3.2.3.2 OWL

OWL evolved from earlier ontology modeling languages (Resource Description Framework (RDF), RDF Schema, and DAM+OIL) to provide a more expressive and powerful language to define, and reason with, ontologies on the World Wide Web. RDF is basically a data model used to make simple statements concerning resources (objects such as books, people, places, etc.) on the Web together with the relationships (properties such as “title”, “name”, “location”, etc.) between those resources. These simple statements are specified in the format object-attribute-value triplet corresponding to the resource, property and value. RDF Schema expands upon the capabilities of RDF by adding the concept of generalization enabling the definition of classes and subclasses of objects as well as subproperty relationships. RDF Schema also adds the ability to specify to which side of a relationship a resource can belong, either the domain or range. OWL builds upon RDF Schema by adding the abilities to specify logical expressions, equalities and inequalities, cardinality restrictions, required and optional properties, enumerated classes, and the concepts of symmetry in inverse. This additional expressiveness enables enhanced semantical specification of and reasoning with domain information. OWL uses XML syntax. OWL is used to describe a domain model by defining classes, properties, and individuals. Figure 7 gives an example of a partial OWL representation of the consumer-producer-decomposer relationships. Individuals are the instances of the ontology, the specific examples. The classes describe sets of individuals. The properties describe relationships between objects (object property) such as subclass, inverse, transitive, symmetric, and functional properties, and between objects and their data type values (data type properties) such as integer,

![Figure 7: OWL Ontology Example](image-url)

"grass"\rightarrow plant
\text{chemosynthetic}
\text{photosynthetic}
\text{subClassof}
\text{subClassof}
\text{subClassof}
\text{subClassof}
\text{subClassof}

producer
\text{subClassof}
\text{subClassof}
\text{eats}
\text{absorbs}
\text{isEatenBy}
\text{isDecomposedBy}
\text{absorbs}
\text{contains}

energy
sun

bacteria

\text{decomposer}

\text{subClassof}

\text{subClassof}

\text{subClassof}

\text{fungi}

soil

\text{scavenger}

\text{bacteria}

\text{decomposer}

\text{fruits}

\text{consumer}

\text{subClassof}

\text{subClassof}

\text{subClassof}

\text{subClassof}

"owl"

"squirrel"

"vulture"

"snake"

"consumer"

"producer"

"decomposer"

"plant"

"chemosynthetic"

"photosynthetic"

"herbivore"

"carnivore"

"omnivore"

"squirrel"

"vulture"

"snake"
string, boolean, date and time. It is possible to specify restrictions on properties such as one of, unionOf, allValuesFrom, someValuesFrom, minCardinality, and maxCardinality. There are three sublanguages of OWL that range from the simplest and easiest to implement to the most expressive: OWL Lite, OWL DL (Description Logics), and OWL Full. OWL Full is an extension of OWL DL which is an extension of OWL Lite.

3.3 UML and OCL

This research focuses on UML designs specified using the officially adopted standard UML 2.0 [OMG1] addressing two layers of the UML Architecture, shown in Table 2, specifically the UML Meta-model layer and the Model layer.

<table>
<thead>
<tr>
<th>UML Layer</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3: Meta-metamodel</td>
<td>Defines the language to specify meta-models</td>
<td>MetaClass, MetaAttribute, MetaOperation</td>
</tr>
<tr>
<td>M2: Meta-model</td>
<td>Defines the language to specify models</td>
<td>Class, interface, operation</td>
</tr>
<tr>
<td>M1: Model</td>
<td>Defines the language to specify user objects in a specific domain; Instance of meta-model</td>
<td>Car, customer</td>
</tr>
<tr>
<td>M0: User Objects</td>
<td>Defines a specific domain; Instance of the model</td>
<td>Honda Prelude ABC-123, John Smith</td>
</tr>
</tbody>
</table>

UML 2.0 includes the following 13 diagrams: Activity, Class, Communication, Component, Composite Structure, Deployment, Interaction Overview, Object, Package, Sequence, State Machine, Timing and Use Case Diagrams. This research analyzes a subset of UML diagrams that includes the Use Case, Class, Sequence and State Machine diagrams:

1. use case diagram: specifies the system’s functionality from the perspective of interactions with the user, also known as an actor; includes relationships between the system and its environment; typically is supplemented with considerable natural language descriptions;

2. class diagram: specifies the static structure of objects, including attributes and operations, and their relationships such as aggregation and generalization;

3. sequence diagram: specifies the dynamic behavior between objects represented as a chronological sequence of messages exchanged between objects;

4. state machine diagram: specifies the dynamic behavior within objects in terms of states and events.

This subset of UML includes a representative selection of UML diagrams specific to analysis and design and covers the representative diagrams of the user view, the structural view and the behavioral view. This research does not cover the implementation view.

UML designs analyzed in this research can include constraints specified via OCL. Typically, OCL is used in conjunction with UML to specify constraints utilizing constructs unavailable in UML but typically required for formal verification. It is possible to specify well-formed rules for the UML model using OCL that can in turn be used to assist with the verification of consistency of the UML design. OCL enables the software engineer to specify more precisely the behavior and constraints
associated with the system via the specification of invariants on classes, and pre-conditions and post-
conditions on operations. A class invariant is a statement about a property of a class that holds for all
objects of that class throughout the lifetime of each object. A post-condition is a statement about
conditions that exist after execution of an operation, basically what the operation should accomplish.
A pre-condition is a statement about conditions that exist before the execution of an operation,
basically the assumptions before the operation. Class invariants, pre-conditions and post-conditions
are usually specified as assertions. An assertion is a logical statement regarding one or more
variables. OCL expressions are declarative, specifying what constraints should be enforced but not
how they should be enforced, and side effect-free since they do not change the state of the system.
Currently, the semantics of OCL is based on UML semantics. OCL does not have a separate meta-
model [Warmer]. However, the current UML 2.0 OCL specification includes meta-modeling
diagrams that link it with UML 2.0 [OMG3].

3.4 KAOS

KAOS is a widely cited goal-oriented approach to requirements engineering that is currently being
incorporated into several emerging research projects including obstacle recognition [Brohez &
Gregoire], process control systems design [El-Maddah & Maibaum], derivation of event-based
specifications (SCR) from KAOS models [DeLandtsheer et al.], security requirements [Fontaine],
software architecture design [van Lamsweerde1], the reconciling of requirements with runtime
behavior [Feather et al.], and UML profiles [Heaven & Finkelstein]. KAOS covers a broad range of
requirements engineering activities including meta-modeling, specification methodology, obstacle
recognition, and conflict management. Goals are used to refer to the state(s) of the system that
should be achieved, maintained, ceased and/or avoided. KAOS uses real-time temporal logic
notation to perform formal reasoning to prove correctness and completeness of its refinement
process, its analysis of obstacles and conflicts. The different types of inconsistencies detectable in
the KAOS framework include intra-level inconsistencies among a process/product/instance tri-level
scope, product-level inconsistencies (such as terminology, designation, or structure clashes), and
assertion inconsistencies (such as conflict, divergence, competition, obstruction, realizability and
concern meta-relationships).

Research in requirements engineering during the past fifteen years has increasingly recognized the
importance of incorporating a goal-oriented view into its modeling, specification and reasoning [van
Lamsweerde5], [Kavakli & Loucopoulos]. The correspondence between goals and requirements are
that “requirements implement goals much the same way as programs implement design
specifications” [van Lamsweerde8]. Rather than focusing upon the system behavior and its
interactions with users, goal-oriented approaches make it “easier to investigate different ways of
achieving the stated goals and to detect and solve conflicts between them” [Regev & Wegmann].
Several goal-oriented approaches exist that are applicable to one or more of the four basic
requirements engineering activities: elicitation, negotiation, specification and validation. Such
approaches include KAOS, i* approach [iSTAR], Non-functional Requirements (NFR) Framework
[Mylopoulos et al.], Goal-Based Requirements Analysis Method (GBRAM) [Anton2 et al.], and
Goal-Questions-Metrics approach (GQM) [Basili et al.].

The KAOS approach covers a broad range of requirements engineering activities including meta-
modeling, specification methodology, obstacle recognition, and conflict management. KAOS
enables software engineers to identify high-level goals of the system to be built, both functional and
non-functional, and subsequently refine those goals into sub-goals and/or identify super-goals by
continually asking, in addition to “what” types of questions typical of requirements engineering, the
“why”, “how” and “when” type of questions. The sub-goals are in turn assignable to individual software components, hardware devices or humans, collectively referred to as agents. Goals are used to refer to the state(s) of the system that should be achieved, maintained, ceased and/or avoided. KAOS facilitates alternative goal refinement and alternative agent responsibility assignment enabling the development of alternative system proposals [van Lamsweerde4].

The basic structures in KAOS are goals, requirements, agents, objects and operations. A goal is basically “an objective the system should achieve through cooperation of agents in the software-to-be and in the environment” [vLamsweerde4]. An agent is either a person or a software/hardware component that is responsible for achieving one or more requirements [Objectiver1]. A requirement is “a low-level type of goal to be achieved by a software agent” [Objectiver1]. An object is a “thing of interest in the system whose instances share similar features, can be distinctly identified, and have specific behavior from state to state” [van Lamsweerde3]. Viewed from the meta-level, object specializations include entities, associations, events and agents. Entities are autonomous. Associations are subordinate. Events are instantaneous. Agents are active. Operations are input-output relations over objects that are used to define state transitions and are characterized by pre-conditions, post-conditions, and trigger conditions [van Lamsweerde8]. In the KAOS approach, constraints are obtained by formally refining high-level goals. Constraints can be specified on objects, processes and requirements. Constraints on objects are specified in a manner similar to class invariants. Constraints on requirements and processes are specified in a manner similar to necessary and sufficient pre-conditions and post-conditions.

KAOS goals are subdivided into functional and non-functional high-level goals and are expressed at the conceptual model level thereby ignoring specific system implementation issues. KAOS includes numerous requirements patterns that are useful when building the goal model. Goal refinement utilizes these patterns to refine high-level goals into combinations of low-level goals. Goals are refined into sub-goals and/or used to identify super-goals by continually asking, in addition to the “what” types of questions typical of requirements engineering, the “why”, “how” and “when” type of questions. Refinement stops once “a goal has been placed under the responsibility of a single agent” [Objectiver1]. These goals are organized into goal graphs with the business or strategic goals at the root and the system requirements at the leaves. Conflicts among goals arise if two goals in the same goal graph cannot be satisfied simultaneously or when two or more goals produce opposite actions under the same conditions. Obstacles are conflicts that prevent the achievement of goals.

All KAOS language constructs can be specified via a two-level structure: an outer declaration layer, which includes semi-formal goal diagrams with natural language descriptions, and an inner formal assertion layer that is used to formally define the construct and for formal reasoning. KAOS assists software engineers with identifying and resolving goal conflicts and obstacles to those goals. KAOS uses real-time temporal logic notation, originally developed by [Manna & Pnueli], to perform formal reasoning to prove correctness and completeness of its refinement process, its analysis of obstacles and conflicts. It performs this formal reasoning at the goal level to detect and resolve conflicts, generate obstacles, refine goals and operationalize goals. Each goal is represented as a rule in temporal logic. When the goals are specified formally the temporal logic, it is possible to derive goal refinement patterns via goal regression that are provable. Refinement patterns are used to decompose goals into sub-goals. Assuming that a sub-goal holds, the truth of the super-goal is inferred from the conjunction (or disjunction) of the sub-goals. Goals are negated to produce obstacles that are used to create and resolve new goals. A goal is assigned to an object or operation. Once derived, a goal refinement pattern can be reused with the necessity to re-prove. KAOS uses state-based specifications to specify operations.
KAOS utilizes the following temporal logic operators originally developed in [Manna & Pnueli]:

- ○ (in the next state)
- ● (in the previous state)
- ◊ (some time in the future)
- ♦ (some time in the past)
- □ (always in the future)
- ■ (always in the past)
- W (always in the future unless)
- U (always in the future until)

The following patterns of temporal behavior are used to classify goals [van Lamsweerde7]:

- **Achieve:** \( P \implies ◊ Q \) or **Cease:** \( P \implies ◊ \neg Q \)
- **Maintain:** \( P \implies Q \wedge W \wedge R \) or **Avoid:** \( P \implies ◊ \neg Q \wedge W \wedge R \)

where \( P, Q, \) and \( R \) are propositions

Therefore, goals are used to refer to the state(s) of the system that should be achieved, maintained, ceased and/or avoided. Goals do not refer to state transitions in the system. The *Achieve* and *Cease* goals are used to generate behaviors. The *Maintain* and *Avoid* goals are used to restrict behaviors. There also exist soft goals that are used to indicate behavioral preferences where there exist alternative behaviors. However, it is the *Achieve, Cease, Maintain* and *Avoid* goals that can be verified via goal satisfaction and formal reasoning.

Goals are additionally organized in taxonomic categories such as satisfaction, information, accuracy, security, safety, usability, etc. Goals have attributes, such as name, priority, and definition. Goals also have links. Intra-model links are utilized for goal refinement as well as obstruction and conflict analysis. Inter-model links are used for reference, operationalization and responsibility.

The KAOS Metamodell [van Lamsweerde2] has evolved considerably during the past ten years. The KAOS meta-level is composed of four sub-models: goal model, object model, agent responsibility model and operation model. The goal model focuses upon behavioral aspects including refinement, obstacle, and conflict analysis. The object model concerns conceptual issues such as agents (independent, active objects) and entities (independent, passive objects), and associations (dependent, passive objects) as well as the relationships among them such as specialization and aggregation. The agent responsibility model addresses the assignment of responsibility to agents and the corresponding interfaces. Lastly, the operation model concerns the behavior required of agents (the scenarios) to meet the requirements and includes and operationalization. Operationalization is the process of identifying and deriving operations and their domain pre-conditions and post-conditions for associated goals; what an agent needs to do in order to fulfill a goal. Operations are performed upon objects.

Before discussing the different types of inconsistencies detectable in the KAOS framework, it is important to cover the scope of managing inconsistency within this framework. This scope is viewed from the following three levels [van Lamsweerde8]:

- **process level:** describes requirements in terms of objectives, actors, and elaboration operators to produce artifacts; actors at this level include clients, users, domain specialists, requirements engineers, and software developers;
- **product level:** describes instances of the artifacts created in the process model to further describe the requirements model in terms of goals, agents, objects and operations;
- **instance level:** describes instances of the objects and operations created in the product level to describe operations executed on objects in the running system.
The KAOS framework includes the following inconsistencies:

A. Intra-level inconsistencies: inconsistencies involving two levels of scope that arise due to problems with the objectives and rules at the process level, or the requirements at the product level, or the states at the instance level:
   1. process-level deviation: violation of a process-level rule that occurs in the requirements engineering process; for example, assigning responsibility for a goal to two different agent types;
   2. instance-level deviation: violation of a product-level rule that occurs in the running system; for example, a specific instance of an agent failing to provide requirement constraints;

B. product level inconsistencies: problems with goals and requirements at the product level:
   1. terminology clash: multiple syntactic names given to a single real-world concept;
   2. designation clash: one syntactic name is given to multiple real-world concepts;
   3. structure clash: multiple structures are given to a single real-world concept;

C. assertion inconsistencies: problems among assertions that formalize a goal, or a requirement or an assumption; this type of inconsistency involves domain descriptions:
   1. conflict: two or more assertions are logically inconsistent in the domain descriptions; the negation of these assertions can be inferred from other assertions; also, if any one of these assertions no longer exist then the inconsistency no longer exists; for example (modified from [van Lamsweerde8]), the following three assertions are conflicting: (1) when a device is in operation it should be running; (2) when a device is in operation it should be running but when it is in start up it is not running; (3) a device should always be running;
   2. divergence: a conflict (as defined above) between assertions that occurs only if there is a boundary condition such that 1) a set of assertions become inconsistent within the domain that includes the boundary condition, 2) the removal of one or more of the assertions removes the inconsistency and 3) it is possible for the boundary condition to exist; a boundary condition is a specific combination of circumstances that results in conflicts between goals or requirements; for example (modified from [Letier]): Given the two assertions (1) a pump should be on when there exists high water and (2) a pump should be off when critical methane levels are detected, the boundary condition, high water level and critical methane level, results in a divergence; boundary conditions can be “formally derived by regressing the negation of one of the goal assertions through the domain theory extended with the other goal assertions” [van Lamsweerde7];
   3. competition: a type of divergence within a single goal or requirement; for example, a person is invited to attend two different meetings in which that person is able to attend each of the meetings separately but if he attends one meeting he can not attend the other meeting [van Lamsweerde8]; imagine the case where the meetings are held in two distant states on the same day and there is not enough time to travel from one state to the other between meetings;
   4. obstruction: a type of divergence that involves only one assertion; a boundary condition becomes an obstacle to the assertion of a goal; for example, a person is invited to attend a meeting to which that person can attend but then the meeting time changes and that person can no longer attend the meeting [van Lamsweerde8];
   5. realizability: “a goal can be assigned as the responsibility of an agent only if the goal is stated in terms of objects that are monitorable and controllable by the agent” [Letier];
   6. concern meta-relationship: “every vocabulary element used in the formal definition of goals must be declared in the object model” [Letier].
The KAOS approach has been used successfully to detect and resolve conflicts in requirements engineering. The research presented in this paper adopts the KAOS framework for the classification of requirements inconsistencies presented in [van Lamsweerde8] combined with its enhancements in [Letier]. The KAOS approach manages conflicts at the goal level in order to provide more flexibility in handling conflicts. Associated with the KAOS classification of inconsistency types, there exist within KAOS techniques for detecting and resolving inconsistency types based on this classification. In general terms, the problems of inconsistency are addressed in KAOS by “checking the meta-constraints” and “by using systematically formal refinement techniques and the pattern library” [Ponsard]. More specifically, inconsistencies are detected in KAOS by assertion regressing, divergence patterns and detection heuristics [van Lamsweerde8]. Inconsistencies are resolved in KAOS by avoiding boundary conditions, goal restoration, conflict anticipation, goal weakening, resolution patterns, alternative goal refinement, resolution heuristics and object refinement [van Lamsweerde8].

There does exist some incompleteness in the KAOS approach. Specifically, “inconsistencies are not explicitly represented as a KAOS object” therefore “focusing on a subset of inconsistencies is outside the scope of the framework” [Robinson]. Additionally, there exists a “lack of heuristic criteria that could direct the search for boundary conditions towards goals and domain formulas whose sub-formulas would be more likely to appear in prominent scenarios” [Spanoudakis & Zisman]. And lastly, “the current set of (divergence and obstacle) patterns has to be expanded to capture a larger part of the range of divergences that can be found in goal specifications for complex system” [Spanoudakis & Zisman].

Significant benefits of utilizing the KAOS model beyond detection and diagnosis of inconsistencies include [Objectiver1]: bi-directional traceability between the problem description and solution spaces; completeness criteria via refinement of all goals specified, assignment of all requirements to agents, justification of all operations, assignment of responsibility and order of operations; and reduction of ambiguity via glossary construction and validation.

KAOS has associated with it commercially available tools that can perform consistency verification, including Objectiver Requirements Management platform [Delor] and an extension to Objectiver called the FAUST Toolbox for Formal Requirements Specification Analysis [Ponsard et al.]. The KAOS model is incorporated into other tools including diagram editors. Additionally, Rational Rose modeling tools provide extension mechanisms for the KAOS meta-model. There is also a KAOS CASE-tool known as GRAIL [Ballant et al.]. Lastly, there exists a toolbox for Formal Requirements Specification Analysis [FAUST] that is based on the KAOS goal-oriented methodology.

One other example of similar research addressing both the UML and KAOS models exists but it has a different focus that applies goal-oriented requirements engineering techniques between system objectives and UML models to facilitate the development of precise software specifications [van Lamsweerde3]. KAOS starts with gathering information on requirements, assists the requirements engineers with modeling and creating the requirements specification documentation. KAOS addresses inconsistencies with regard to requirements. MOA starts with a UML design, transforms it to a KAOS specification, and performs analysis on that specification in order to address inconsistencies in UML models.
4 Methodology for Objects to Agents (MOA)

4.1 Introduction

Methodology for Object to Agents (MOA) is a methodology that integrates multiple views of a software design and combines object-oriented concepts with agent-oriented concepts to facilitate detection of errors arising from these multiple perspectives. Section 4.2 presents the Ontology for Software Specification and Design (OSSD) that was developed for use in MOA as the common model during the transformation of an informal software design into a formal agent-oriented requirements specification. Sections 4.3.1 to 4.3.4 define the MOA process including a high-level view of MOA processing including a novel utilization of thesauruses to analyze behavior and goals identified in the source language design followed by the algorithms to transform a UML design to an OSSD representation of that design. Section 4.3.5 describes the two forms of MOA consistency checking. Finally, Section 4.3.6 gives the algorithm to transform an OSSD representation into a target formal requirements specification, KAOS.

4.2 OSSD Model

The structure of the OSSD Model, shown in Figure 8, is a hierarchical decomposition of software development concepts that is intended for automated manipulation. The top level of the ontology consists of a Construct, which is subdivided into nine sub-constructs: Object, Attribute, Behavior, Relation, State, Transition, Goal, Constraint, and Plan. Object is subdivided into Event and Statebased; the latter is subdivided into Agent and Entity. An Event is an Object that has only one State with no duration of time. An Agent is an Object that Controls and/or Monitors the Behavior of other Objects. An Entity is an Object that has multiple States but does not Control or Monitor the behavior of other Objects. Both Agents and Entities can have Perform Behavior. An Event is the result of a Behavior. Attribute is subdivided into ObjectAttribute, RelationalAttribute, Visibility, and Multiplicity. Behavior is subdivided into Control, Monitor and Perform. Relation is subdivided into Association and Non-Association, the latter of which is further subdivided into Aggregation, Composition, and Generalization/Specialization. State is subdivided into Initial, Intermediate and Final. Transition is subdivided into Incoming and Outgoing. Goal is subdivided into Achieve, Maintain, Cease and Avoid. Constraint is subdivided into Precondition, Postcondition, Trigger, Guard and Action.

Properties in the OSSD Model depict both structural and behavior relationships between OSSD constructs. This wide latitude of interpretation is derived from the definition of an ontological “property”. The term “property” itself has numerous definitions. In the general sense, a property of something is often referred to as an attribute and describes a quality of that something and is used to describe that something; for example, the color, weight, and size of something. Within the Resource Description Framework (RDF), a property represents an attribute or relationship associated with a resource. “A property is a binary relation between Thing and Thing” [DSTC et al.]. Properties in the OSSD Model are assumed to imply the “has” relationship unless otherwise labeled. For example, each Object has ObjectAttribute(s), the Relation(s) in which it is involved, and for StateBased Objects, the State(s) in which the Object can exist.

Associated with each Behavior are the Attributes that it inputs and outputs, the Constraints it has, and the Goal that it operationalizes. Associated with each Goal are other Goals that the Goal depends on, the Agent for which the Goal is under the responsibility of, and the Object that the Goal concerns. Lastly, each Agent has a Plan that contains a sequence of Behavior(s).
The agent-oriented concepts of goal, belief, and intention are represented in the OSSD Model. Beliefs portray knowledge that an agent has of its environment and are represented via the Object.


Relation, Attribute, State, Transition and Constraint Constructs. Goals are the ultimate outcomes desired by an agent and are represented via the Goal Construct. Intentions are the goals that an agent is focusing on at a specific moment in time and are depicted via how the agent plans to work towards its selected goals based on its current knowledge. Intentions are represented via the Behavior Construct.

Because the terms Agents, Entities and Events are often used in software development with varying definitions, there is a need for additional clarification and refinement of their definitions. These refinements affect the transformation between software modeling and/or requirements languages and the OSSD Model.

An Agent is an Object that controls and/or monitors the behavior of other Objects. These “controlled” Objects are “outside” of the Agent, that is, they are not sub-components of the Agent. Agents interact with other Agents, control Entities, and react to Events based on sensory input from their environment. Agents execute their own thread of control and therefore cannot be a subcomponent of another Object. Agents send messages to other Objects and sometimes expect a response from those Objects. Therefore, they are, as part of their normal processing, partaking in a communication similar to that of an agent communication language. Lastly, since an Agent has control of its own actions and internal state without any direct intervention from people or other Agents, the receipt of a message cannot change the state of that Agent.

An Entity is an Object that has multiple States but does not control or monitor the behavior of other Objects unless those Objects are sub-components of the Entity. Entities typically perform operations at the request of Agents and typically send messages to Agents indicating an operation has been performed. The internal state of an Entity can be changed as a result of receiving a message from another Object.

An Event is an Object that has only one State with no significant duration of time. An Event occurs when some action has been performed by another Object. An Event can be as simple as a discrete change in an environment variable, including temporal variables, or the completion of a complex operation. The receipt of an Event by an Agent causes that Agent to perform some action. In UML 2.0, an event is defined as “the specification of a significant occurrence that has a location in time and space and can cause the execution of an associated behavior”….“in the context of state diagrams, an event is an occurrence that can trigger a transition” [OMG1]. In UML 2.0 each message in a Sequence Diagram is represented as an event in an associated State Machine Diagram. The definition of an OSSD Event is more restricted than a UML event because it does not include the request for an operation or the command from one Object to another Object, therefore UML call events are not considered OSSD Events. A message in a UML Sequence Diagram corresponds to an Event only if it indicates that some action has been performed.

The graphical notations of the OSSD Model are commonly used in ontological representations. Classes are depicted as rounded rectangles with solid lines showing sub-class relationships. This subclass relationship is typically referred to as an “Is-a” relation. A class can have associated with it one or more properties, indicated by dashed lines, which further define the class and link it conceptually with related classes. The two classes interconnected by a property can be identified as the “from” class and the “to” class, if required for clarification, via the direction of the arrow at the end of the dashed line. Classes are given in italics and capitalized while properties are given in italics and not capitalized. Instances of a class are indicated at the end of a double-headed arrow.
Similar models have influenced the development of the OSSD Model such as the ABC Metadata Model [Lagoze & Hunter] and the Methodology for Engineering Systems of Software Agents [Evans1 et al.]; however, the OSSD Model is not directly derived from any one of these ontologies but rather is developed based upon different concepts inherent in the ontologies.

4.3 MOA

4.3.1 Overview

The MOA includes both transformations and consistency checking. The transformation from the source language UML to the OSSD Model can be summarized as a combined lexical and semantic analysis of the UML Model diagrams, followed by the utilization of multiple mapping tables that enable the creation of an instance of the OSSD Model. The MOA consistency checking is a two-stage process that introduces a consistency framework and an Inter-View Inconsistency Detection Table, both of which are based on the OSSD Model. The final transformation from the OSSD Model to the target agent-based requirements specification language, KAOS, is accomplished by the use of two mapping tables.

Processing of the UML Class Diagrams is the first step in identifying the Object, Attribute, Relation and Behavior Constructs of the OSSD Model. The processing of the UML Sequence Diagrams refines the OSSD concept of Behavior and identifies the Constraints associated with Behavior. The processing of the UML StateMachine Diagram refines the OSSD concept of Constraints and identifies the States and Transitions in the OSSD Model. Lastly, the processing of the UML Use Case Diagram identifies the Goals associated with Objects and Behavior in the OSSD Model.

Section 4.3.4 provides details concerning the transformation of UML diagrams to the OSSD Model. Figures 9 and 10 show high-level views of the conceptual mappings between the UML Diagrams and the Model Constructs, and the MOA processing, respectively. The MOA algorithms are shown as the shaded areas in Figure 10. The first algorithm transforms a UML design into an instance of the OSSD Model. The second algorithm performs basic consistency processing on the OSSD Model. The third algorithm transforms the consistent OSSD Model instance into a KAOS specification.

Figure 9: High-Level View of UML to OSSD Mapping
4.3.2 Lexical and Semantic Analysis

The initial step consists of a lexical analysis that performs a part-of-speech tagging for each English word in the source language. This research utilizes the Suggested Upper Merged Ontology (SUMO) [Niles] WordNet [Miller] Browser [Sigma] to assist with the categorization of terminology used in the UML diagrams. SUMO is a large formal ontology that is available to the public and is currently mapped to the complete WordNet lexicon. WordNet is a lexical reference system for the English language that categorizes English words into parts of speech (noun, verb, adjective, adverb). It organizes words into sets of synonyms, referred to as synsets, gives definitions and provides semantic relations between the synsets. These relations include synonyms/antonyms, hypernyms/hyponyms (is-a relations with a broader and narrower definition), and meronyms/holonyms (similar to part/whole of the part-of or has-part relations). A partial view of the SUMO hierarchy is shown in Figure 11.

![Diagram of MOA Processing](image)

Figure 10. High-Level View of MOA Processing

The initial steps of the part-of-speech tagging include identifying each word in the source design as one of the typical English parts of speech: noun, verb, adjective, adverb, and preposition. All verbs are identified as either in past and present tense. If an English word has more than one possible part of speech interpretation, the context of the UML element determines the appropriate part of speech, defaulting to nouns for classes and attributes and verbs for operations and messages. For example, the English word “press” is sometimes interpreted as a verb describing the act of pressing something or as a noun describing a machine used for printing, a newspaper organization, a newspaper or magazine. Next, the SUMO/WordNet Browser determines the English word’s ontological classification within the SUMO Ontology. If a word has multiple meanings within the same part of speech, the user is asked to select the closest meaning from a list of possible definitions.
There is no standard format for specifying Use Case Diagrams; as a result, the processing of the UML Use Case Diagrams uses the guidelines given in [Gottesdiener]. Use cases identify the actors and actor interactions along with the goals associated with the roles that the actors play. Use Case Diagrams describe “what” a system does as opposed to the “how”. The frame of references is that of an observer external to the system. Use case diagrams relate to scenarios, which describe what happens during interactions with the system to be developed. A use case is a set of scenarios that accomplish a single task or goal. Actors represent the roles that people or system components play that initiate events in the scenarios. A stick figure represents an actor. A use case represents the primary goal of the actor. An oval containing a named description represents a use case. The line connecting an actor to a use case is a communication association. The rectangle around the set of use cases is the system boundary. The “includes” relationship shows sub-cases and the “extends” relationship shows Use Case alternatives, exceptions and error conditions. Additionally, for purposes of processing, the naming of the Use Cases conforms to the recommendations given in [Gottesdiener] which state:

- use the format “verb” + [qualified] “noun”
- use active verbs and not passive verbs
- avoid verbs that are vague such as “do” or “process”
- avoid low-level verbs that are database oriented such as “create”, “read”, “update”, “delete”, “get”, “insert”
- use “informative” verbs such as “analyze”, “discover”, “find”, “identify”, “inform”, “monitor”, “notify”, “query”, “request”, “search”, “select”, “state”, “view”
- only one actor goal per Use Case
- the format for events should be either “subject”+”verb”+”object” OR “time to <verb + object>”

Figure 11: Partial View of SUMO Hierarchy
4.3.3 Goal Thesaurus and Behavior Thesaurus

In addition to the SUMO/WordNet Browser and WordNet Database, two lists of keywords were specifically developed for the MOA, a Goal Thesaurus and a Behavior Thesaurus, assist with the classification of the OSSD Model Constructs Goal and Object. There are four types of Goals in the OSSD Model: Achieve, Maintain, Cease, or Avoid. There are three types of Behavior in the OSSD Model: Perform, Monitor, or Control. Goals and behaviors are divided into these categories based on their categorization in the KAOS methodology. While the meanings of perform, monitor, and control are obvious, the meaning of the goal classifications need further explanation. Terminology for Goals and Behavior is based on similar terms defined previously [van Lamsweerde5]. An object monitors/controls a second object if it observes/modifies the state of one or more variables of that second object. An object behavior is considered to perform if it actually executes a sequence of steps to complete a task or operation. Specifically, achieve and cease imply a desired goal will eventually be obtained or rejected while maintain and avoid imply that a desired goal is to be continuously held or rejected. Figure 12 gives a partial view of the Goal Thesaurus, and Figure 13 provides a partial view of the Behavior Thesaurus. The similar use of keywords in the repository created for the Privacy Goal Management Tool (PGMT), under development at the North Carolina State University [Anton1 et al.], inspired the development of the Goal Thesaurus and the Behavior Thesaurus.

Creating the Goal Thesaurus includes extracting synonyms for the key words “achieve”, “maintain”, “cease”, and “avoid” from a standard thesaurus. Creating the Behavior Thesaurus includes extracting synonyms for the key words “monitor” and “control” from a standard thesaurus. The perform type of behavior is too broad a category to capture its meaning in a listing of synonyms. Categorization of perform is a combination of SUMO and heuristics. The Goal Thesaurus assists with analyzing verbs from each UML Use Case name to create instances of goal classes in the OSSD Model. The Behavior Thesaurus assists with analyzing verbs from each UML Association name to determine if an OSSD Model instance of an OSSD Agent or Entity should be created.

The MOA transformation process includes heuristics. Heuristics have been applied recently to the transformation of natural language text into the Entity-Relationship (ER) Model [Omar et al.] and a UML Class Model [Harmain & Gaizauskas], and in the transformation between UML Diagrams [Selonen et al.]. While formal rules will always consistently produced correct results, heuristics will

![Figure 12: Goal Thesaurus](image-url)
usually produce correct results. The key is to clearly define the context in which the heuristic is to be applied in order to ensure produce the desired results and thereby enhance the confidence in it. Additionally, application of formalization techniques can be used to enhance confidence in the heuristic. Eventually, the heuristic must be tested and verified.

4.3.4 UML to OSSD Transformation

4.3.4.1 Overview

Figure 14 represents the transformation of a UML design to the OSSD Model as a UML Activity Diagram. A detailed overview of each step is given below including tables 3 through 5 that contain examples of the mappings from UML to OSSD for the UML Class, Sequence and StateMachine Diagrams. The first step identifies the Objects, Attributes, Relations and Behavior Constructs of the OSSD Model. The processing of the Sequence Diagrams refines the concept of Behavior and identifies the Constraints associated with Behavior. Each message in a Sequence Diagram produces a Behavior whether it corresponds to a UML signal or an operation call. The processing of the StateMachine Diagram refines the concept of Constraints and identifies the States and Transitions in the OSSD Model. The processing of the Use Case Diagram identifies the Goals associated with Objects and Behavior in the OSSD Model. Lastly, the information gathered and analyzed is combined into an instance of the OSSD Model for the UML design.

Four steps detail the classification of a UML Class as an OSSD Object. First, the English text used to describe the Class name is identified within the SUMO hierarchy as a possible Agent or an Entity. For example, if a Class is identified as a sub-level of the SUMO Entity:Physical:Object then it is potentially an Entity. If a Class is identified as a sub-level of the SUMO Entity:Abstract:Attribute:RelationalAttribute:SocialRole then it is potentially an Agent. Second, the association relationships between UML classes are analyzed based on a search through the Behavior Thesaurus for the English text used to describe the relationships. Relationships with a Monitor or Control type of behavior identify potential Agents and Entities. Third, the English text used to describe the UML operations within each Class are analyzed for their type of behavior. Operations in messages sent from an Agent to an Entity are assumed to utilize the present tense of the verb, thereby indicating a command. Similarly, operations in messages sent from an Entity to an Agent are assumed to utilize the past tense of the verb, thereby reporting to the Agent that some action has been performed or observed. UML operations that correspond to levels in the SUMO hierarchy under IntentionalProcess indicate
that such operations are associated with an *Agent* because these operations are deliberate actions initiated by an *Agent* and performed by either an *Agent* or *Entity*. Any Class that contains only *Perform* type of operations is classified as an *Entity*. Any *Entity* that has only one state is classified as an *Event*. Any Class that has either *Control* or *Monitor* type of behavior but that *Controls* or *Monitors* only one or more classes contained within that Class is classified as an *Entity* because that Class is actually controlling or monitoring itself. Any Class that has either *Control* or *Monitor* type of behavior that *Controls* or *Monitors* one or more classes not contained within that Class is classified as an *Agent*. Lastly, since the definition of an agent states that it must be able to control its own actions and internal state without any direct intervention from people or other agents, if the behavior of an Object caused by the receipt of a message from an Agent object results in a change in the state of that Object, then that Object is an *Entity*.

![Activity Diagram - UML to OSSD Transformation](image)

Figure 14: Activity Diagram - UML to OSSD Transformation
This research makes the following assumptions concerning the UML specifications. When a UML definition includes multiple English words, each new English word starts with a capital letter (e.g. TurnLightOn). UML Association Names are specified using directional indicators to enhance interpretation of the Association Name. If directional indicators are not specified with the UML Association names in the Class Diagrams then the Association is read from left to right for horizontally specified associations and from top to bottom for vertically specified association.

### 4.3.4.2 High-level Algorithms

Five high-level algorithms, shown in Figures 15 through 19 as A1-1 through A1-5, correspond to the five major activities shown in the Activity Diagram in Figure 14 that describe the transformation from UML diagrams to an instance of the OSSD Model that represents the UML design. Section 4.3.4.3 provides expanded and more formalized versions of these five algorithms.

#### A1-1: Process Class Diagram Algorithm

For each element in a UML Class Diagram:

a. identify UML definitions (e.g., class, operation);

b. perform an English language part of speech (POS) tagging using the SUMO/WordNet browser (e.g. noun, verb, adjective); for each verb, identify its English sub-POS (present/past) and determine its English language significance based on the SUMO ontology accessed via the WordNet mappings; if an English word has multiple SUMO/WordNet definitions then prompt the user to select the closest meaning from a list of SUMO/WordNet definitions

c. classify each UML relationship as an OSSD Relation based on the different relationships involving the UML classes (e.g. association, generalization, aggregation, composition);

d. update the Inter-View Inconsistency Detection Table

#### A1-2: Process Sequence Diagram Algorithm

For each message in a UML Sequence Diagram:

a. classify the message type (note: [ ] indicates optional);
   - Message Type A: {present tense verb}+[noun/adj]
   - Message Type B: [noun]+[past tense verb]+[adj]

b. rename UML operations in a message if necessary:
   - if a UML message with the same operation is sent to multiple Objects then rename the UML operation with the operation name suffixed by the UML Class name to which the message is sent
   - if an unnamed UML return message (dashed line with filled arrowhead) is sent corresponding to a synchronous message (solid line with a filled arrowhead) then name the Behavior using a Message Type B format corresponding to the last message sent from the UML Class receiving the return message

These algorithms provide a structured approach to converting UML specifications into an OSSD Model representation, ensuring accurate and meaningful translation of design intentions. The optional directional indicators in UML provide additional context for associations, enhancing interpretability further.
c. classify each UML operation as an Behavior;

if the sending UML Class and receiving UML Class are the same then
   classify the Behavior of the sending UML Class as Construct:Behavior:Perform AND
classify the Behavior of the receiving UML Class as Construct:Behavior:Perform
else
   if the UML operation corresponds to Message Type A then
      classify the Behavior of the sending UML Class as Construct:Behavior:Control AND
classify the Behavior of the receiving UML Class as Construct:Behavior:Perform
   else
      if the UML operation corresponds to Message Type B then
         classify the Behavior of the receiving UML Class as Construct:Behavior:Monitor
classify the Behavior of the sending UML Class as Construct:Behavior:Perform


d. classify each UML Class as an OSSD Object;

if UML Class name is identified as sublevel of SUMO Entity:Physical:Object then
   classify the UML Class as an OSSD Construct:Object:Statebased:Entity
else
   if UML Class name is identified as sublevel of SUMOEntity:Physical:Object:Agent OR
      Entity:Abstract:Attribute:RelationalAttribute:SocialRole then
      classify the UML Class as an OSSD Construct:Object:Statebased:Agent

   search Behavior Thesaurus for the verb specified in the UML Association Name
   if verb is not found then
      search WordNet Database for the verb AND
      repeat for each synonym identified for the verb
      search the Behavior Thesaurus for that synonym
      until verb is found in Behavior Thesaurus OR there are no more synonyms

   if the verb is found in the Behavior Thesaurus then
      if directional indicators have been specified next to the UML association name then
         if the verb is the type Control or Monitor then
            classify the UML Class on the “from” side of the association name
            as OSSD Construct:Object:Statebased:Agent
            classify the UML Class on the “to” side of the association name
            as OSSD Construct:Object:Statebased:Entity
         else
            if the verb is the type Control or Monitor then
               classify the UML Class to the left of or above the association name
               as Construct:Object:Statebased:Agent
               classify the UML Class to the right of or below the association name
               as Construct:Object:Statebased:Entity

            if all operations associated with the UML Class are of the OSSD type Perform then
               if UML Class has only one state then
                  classify the UML Class as an OSSD Construct:Object:Statebased:Event
               else
                  classify the UML Class as an OSSD Construct:Object:Statebased:Entity
               else
                  if the Control and/or Monitor type operations of the UML Class refer only to Class(es)
                     contained within that UML Class then
                     classify the UML Class as an OSSD Construct:Object:Statebased:Entity
                  else classify the UML Class as an OSSD Construct:Object:Statebased:Agent

Figure 16b: Figure continued
e. classify each UML Class Attribute as an OSSD Attribute either as ObjectAttributes (including properties visibility, and multiplicity) or RelationAttributes ToObject and FromObject (including properties role and multiplicity) and associate them with the OSSD Objects

f. associate Behavior with OSSD Objects and Attributes according to sends message to and the inputs and outputs for each message; the ordering of the messages exchanged between UML objects is captured in the OSSD Model by simply ordering the creation of the properties (e.g. has Behavior0, has Behavior1)

g. associate each OSSD Relation with its corresponding OSSD Objects and Attribute(s)

h. update the Inter-View Inconsistency Detection Table

A1-3: Process StateMachine Diagram Algorithm

For each state and transition in a UML StateMachine Diagram;

a. classify each UML State as an OSSD State and Initial, Intermediate, or Final;

b. classify each UML Transition as an OSSD Transition and Incoming or Outgoing;
   link all Transitions in a given State using the followed by property

c. classify each UML Constraint as an OSSD Constraint and Precondition, Postcondition, Guard, or Trigger based on the following:

   Precondition: state related attributes and values associated with Incoming Transition; these are attached to the UML transition via a UML Note;

   Postcondition: state related attributes and values associated with Outgoing Transition; these are attached to the UML transition via a UML Note;

   Guard: conditional statement of non-state attributes and values associated Incoming Transition

   Trigger: behavior associated with Incoming Transition; associate Trigger with Behavior

   Action: behavior associated with a Transition that is performed as a result of the Transition

d. associate each OSSD State with its State-Based Object, Transition, Constraints and State Contains;

e. associate each OSSD Transition with its Constraint and Behavior;

f. recheck each OSSD Object classified previously as an Agent to determine if its state is changed by a different Object:
   for each State in the StateMachine Diagram
   if the state of the UML class can be changed by a UML message that UML Class receives then
   UML Class is an OSSD Construct: Object: Statebased: Entity

g. update the Inter-View Inconsistency Detection Table

Figure 17: Process StateMachine Diagram Algorithm
A1-4: Process Use Case Diagram Algorithm

For each Use Case defined in the UML Use Case Diagram:

a. identify the OSSD Objects in the Use Case scenarios via a simple matching of the Actor(s) and any nouns described in the scenarios of the Use Case with the OSSD Objects already identified; nouns referenced in scenario lines containing other Use Case names are processed in the subordinate Use Case;

b. identify the Behavior that is described in the Use Case via a simple matching of the verbs described in the scenarios of the Use Case with the Behavior already identified; verbs referenced in scenario lines containing other Use Case names are processed in the subordinate Use Case;

c. name the Goal by reversing the main verb and noun in the Use Case name; change the verb to noun or past tense;

d. identify the dependency relationships between Goals based on the nesting of UML Use Cases

e. classify the Goals (Achieve, Maintain, Avoid, Cease) based on the verb specified in the Use Case Name:

   search Goal Thesaurus for the verb specified in the Use Case Name
   if verb is not found then
      search WordNet Database for the verb
      repeat for each synonym identified for the Use Case verb
      search the Goal Thesaurus for that synonym
      until an Goal Category has been found OR
      there are no more synonyms
   if verb is found then
      classify the Goal according to the goal category identified

f. associate each Goal with the Behavior identified for the associated verbs from the Use Case

g. associate each Goal with the OSSD Objects they concern and the Agents the Goal is under the responsibility of based on the Objects identified with the Use Case

h. update the Inter-View Inconsistency Detection Table

Figure 18: Process Use Case Diagram Algorithm

A1-6: Build the OSSD Model Algorithm

Build an instance of the OSSD Model for the UML design:

a. create an OSSD Construct Object State-based Agent / Entity or Construct Object Event for each OSSD Object; if an Agent is created then create an OSSD Plan

b. create an OSSD Construct Relation Association or Non-Association for UML Association for each NonAssociation create the appropriate General or Composition sub-trees;
   link each Relation with its associated OSSD Construct Object via the has property

Figure 19a: Build OSSD Model Algorithm
c. create an OSSD Construct Attribute ObjectAttribute / RelationAttribute (ToObject, FromObject) for each UML Attribute
   link each OSSD Construct Attribute ObjectAttribute / RelationAttribute with the OSSD Construct Object or OSSD Construct Relation Association corresponding to the UML Class or Association to which the OSSD Construct Attribute belongs based on the UML elements via the has property;

d. create an OSSD Construct Attribute Visibility for each UML Visibility
   link each OSSD Construct Attribute Visibility with the corresponding OSSD Construct Attribute ObjectAttribute via the has property;

e. create an OSSD Construct Attribute Multiplicity for each UML Multiplicity
   link each OSSD Construct Attribute Multiplicity with the corresponding OSSD Construct Attribute ObjectAttribute / RelationalAttribute via the has property;

f. create an OSSD Construct Attribute Role for each UML Role
   link each OSSD Construct Attribute Role with the corresponding OSSD Construct Attribute RelationalAttribute via the has property;
   link each OSSD Construct Attribute Role with the corresponding OSSD Construct State-based Agent via the performed by property;

g. create an OSSD Construct Behavior (Perform, Control, Monitor) for each Perform, Control, or Monitor Behavior associated with Message Type A
   link each Behavior with its sending OSSD Construct Object State-based Agent or Entity via the has property
   link each Behavior with its receiving OSSD Construct Object State-based Agent or Entity via the sends message property if that Behavior is either Control or Monitor
   link each Behavior with its receiving OSSD Construct Event via the causes property if that Behavior is Perform and then that Event with its receiving OSSD Construct Object State-based Agent or Entity via the sends message property
   link each Behavior with its associated input and output OSSD Construct Attribute via the inputs and outputs properties respectively

h. create OSSD Construct Object Event for each Perform Behavior associated with Message Type B
   link each Behavior with newly created OSSD Construct Object Event via the causes property
   link each Event with its receiving OSSD Construct Object State-based Agent or Entity via the sends message property

Figure 19b: Figure continued
i. create an OSSD Construct State (Initial, Intermediate, Final) for each UML State;
   link each OSSD Construct State with its contained OSSD Construct State(s) via contains property
   link each OSSD Construct State with its associated OSSD Construct Behavior via entry property
   link each OSSD Construct State with its associated OSSD Construct Behavior via do property
   link each OSSD Construct State with its associated OSSD Construct Behavior via exit property

j. create an OSSD Construct Transition (Incoming, Outgoing) for each UML State;
   link each OSSD Construct Transition with its subsequent OSSD Construct Transition via the followed by property
   link each OSSD Construct State with its corresponding OSSD Construct Transition (Incoming, Outgoing) via the has property
   link each OSSD Construct Transition (Incoming, Outgoing) to its corresponding OSSD Construct State via the from and to properties respectively

k. create an OSSD Construct Constraint (Precondition, Postcondition, Trigger, Guard, Action) for each UML Constraint;
   link each OSSD Construct Constraint with its corresponding OSSD Construct Transition via the has property;
   link each OSSD Construct Constraint with its corresponding Behavior via contains property
   link each OSSD Construct Constraint with its corresponding OSSD Attribute RelationalAttribute via the has property

l. create an OSSD Construct Goal (Achieve, Maintain, Cease, Avoid) for each UML Goal identified
   link each OSSD Construct Goal with its associated OSSD Construct Object State-based Agent via the under responsibility of property
   link each OSSD Construct Goal with its associated OSSD Construct Object via concerns property
   link each OSSD Construct Goal with its associated OSSD Construct Behavior via operationalizes property
   link each OSSD Construct Goal with its associated OSSD Construct Goal via depends on property

Figure 19c: Figure continued

4.3.4.3 Detailed Algorithms

Five detailed transformation algorithms, shown in Figures 20 through 24 as A1-1 through A1-5, correspond to the five high-level algorithms given in Figures 15 through 19 that describe the transformation from UML diagrams to an instance of the OSSD Model that represents the UML design. These algorithms utilize supplemental algorithms, shown in Figure 25. The transformation algorithms utilize tables whose names and formats are shown in Figure 26.

For use in the MOA transformation algorithms, we formally define followings sets:

1) WCLD is the set of words in the UML Class diagrams
2) WSQD is the set of words in the UML Sequence diagrams

3) WSMD is the set of words in the UML StateMachine diagrams

4) WUCD is the set of words in the UML Use Case diagrams

5) WUC is the set of words in the UML Use Cases

6) E is the set of UML elements \{class, operation, attribute, association, generalization, …\} ⊂ E

7) R is the set of relationships in the UML diagrams; \( R \subseteq E \)
\{association, generalization, aggregation, composition\} \( \subseteq R \)

8) SW is set of SUMO/WordNet words

9) SWC is the set of SUMO/WordNet classifications
\{entity:physical:object, entity:physical:process:motion, …\} \( \subseteq SWC \)

10) V is set of verbs; PastV is set of past tense verbs; Present V is set of present tense verbs
\( V \subseteq SW \); \{PastV, PresentV\} \( \subseteq V \)

11) A is the set of adjectives
\( A \subseteq SW \)

12) N is the set of nouns
\( N \subseteq SW \)

13) OSSD_Behavior_Thesaurus is the set of verbs divided into Control and Monitor verbs
\{Control_Verbs, Monitor_Verbs\} \( \subseteq OSSD_Behavior_Thesaurus \)
\{administer, advise, call, command, instruct, …\} \( \subseteq Control_Verbs \)
\{oversee, regulate, rule, supervise, check, …\} \( \subseteq Monitor_Verbs \)

14) OSSD_Goal_Thesaurus is the set of verbs divided into Achieve, Maintain, Avoid, and Cease verbs
\{Achieve_Verbs, Maintain_Verbs, Avoid_Verbs, Cease_Verbs\} \( \subseteq OSSD_Goal_Thesaurus \)
\{accomplish, determine, confirm, find, execute, close, …\} \( \subseteq Achieve_Verbs \)
\{keep, hold, observe, manage, assist, support, provide, …\} \( \subseteq Maintain_Verb \)
\{nullify, avert, deny, void, prevent, reject, forbid, reject, …\} \( \subseteq Avoid_Verbs \)
\{desist, halt, drop, finish, quit, delete, destroy, interrupt, …\} \( \subseteq Cease_Verb \)

15) OSSD_Model is the set of elements in the OSSD Model
\{OSSD_Constructs, OSSD_Properties\} \( \subseteq OSSD_Model \)

16) OSSD_Constructs is the set of constructs in the OSSD Model
\{Object, Attribute, Behavior, Relation, State, Transition, Goal, Constraint, Plan\} \( \subseteq OSSD_Constructs \)
\{Statebased, Event\} \( \subseteq Object \)
\{ObjectAttribute, RelationAttribute, Visibility, Role, Multiplicity\} \( \subseteq Attribute \)
{Control, Perform, Monitor} ∈ Behavior
{Association, NonAssociation} ∈ Relation
{Initial, Intermediate, Final} ∈ State
{Incoming, Outgoing} ∈ Transition
{Achieve, Avoid, Cease, Maintain} ∈ Goal
{Action, Guard, Trigger, Precondition, Postcondition} ∈ Constraint
{Agent, Object} ∈ Statebased
{Generalization, Aggregation, Composition} ∈ Non-Association
{Subclass, Superclass} ∈ Generalization
{Whole, Part} ∈ Aggregation
{Whole, Part} ∈ Composition
{ToObject, FromObject} ∈ RelationAttribute

17) OSSD_Properties is the set of properties in the OSSD Model {causes, concerns, dependsOn, do, entry, exit, followedBy, from, has, inputs, operationalizes, outputs, performedBy, sendMsgTo, to, underResponsibilityOf} ∈ OSSD_Properties

18) T is the set of MOA transformation tables; \( \{ t1, t2, t3, t4, t5, t6, t7, t8 \} \) ∈ T

19) \( t1 \) is the UML Class Element and POS Tagging Table where \( \{ t11, t12, \ldots, t1i \} \) ∈ t1 and \( \{ \text{uml, e, p, sp, swc, dw, parms} \} \) ∈ t1i where

- \( \text{uml} = \text{UML name,} \{ w1, w2, \ldots, wi \} \) ∈ \( \text{uml, w} \) ∈ WCLD
- \( e = \text{UML element,} \ e \) ∈ E
- \( p = \text{POS,} \ p \in \{ \text{verb, noun, adjective, preposition} \} \)
- \( sp = \text{SubPOS,} \ p \in \{ \text{past, present, future} \} \)
- \( swc = \text{SUMO/WordNet classification,} \ swc \in \text{SWC} \)
- \( dw = \text{Defined within UML} \)
- \( dwc = \text{Defined within UML classification} \)
- \( \text{parms} = \text{Parameters} \)

20) \( t2 \) is the MOA Relation Classification Table where \( \{ t21, t22, \ldots, t2i \} \) ∈ t2 and \( \{ \text{an, rel, at, r, m} \} \) ∈ t2i where

- \( \text{an} = \text{UML association name} \)
- \( \text{rel1} = \text{OSSD Relation, rel} \in \{ \text{from, superclass, whole} \} \)
- \( \text{rel2} = \text{OSSD Relation, rel} \in \{ \text{to, subclass, part} \} \)
- \( \text{at1} = \text{OSSD Relation Attribute, at} \in \text{WCLD} \)
- \( \text{at2} = \text{OSSD Relation Attribute, at} \in \text{WCLD} \)
- \( r = \text{OSSD Role} \)
- \( m = \text{OSSD Multiplicity} \)

21) \( t3 \) is the MOA Behavior Classification Table where \( \{ t31, t32, \ldots, t3i \} \) ∈ t3 and \( \{ \text{op, mt, parms, so, soc, ro, roc} \} \) ∈ t3i where

- \( \text{op} = \text{UML operation} \)
- \( \text{mt} = \text{message type,} \ mt \in \{ A, B \} \)
- \( \text{parms} = \text{message parameters} \)
- \( \text{so} = \text{OSSD sending Object} \)
- \( \text{soc} = \text{OSSD sending Object classification} \)
- \( \text{ro} = \text{OSSD receiving Object} \)
roc = OSSD receiving Object classification

22) t4 = MOA Object Classification where \{t41, t42, \ldots, t4i\} \subset t4
and \{cn, c, swc, b, oc\} \subset t4i where where
   cn = UML class name
   c = UML composition type, c \in \{TOP, SUB\}
   swc = SUMO/WordNet classification
   b = list of OSSD Behavior associated with cn
   oc = OSSD Object classification

23) t5 = MOA State, Transition, Constraints Classification Table Part 1
   where \{t51, t52, \ldots, t5i\} \subset t5 and \{cn, sn, sc, enb, db, exb, itf, ott\} \subset t5i where
   cn = UML class name
   sn = UML state name
   sc = OSSD State classification where \{initial, intermediate, final\} \subset sc
   enb = OSSD Entry Behavior
   db = OSSD Do Behavior
   exb = OSSD Exit Behavior
   itf = OSSD Incoming Transition From
   ott = OSSD Outgoing Transition To

24) t6 = MOA State, Transition, Constraints Classification Table Part 2
   where \{t61, t62, \ldots, t6i\} \subset t6 and \{cn, tn, itf, ott, c, cc, tl\} \subset t6i where
   cn = UML class name
   tn = OSSD transition number
   itf = OSSD Incoming Transition From
   ott = OSSD Outgoing Transition To
   clist = OSSD Constraint list
   cc = OSSD Constraint classif. where \{precondition, postcondition, guard, trigger, action\} \subset c
   tlist = OSSD Transition list

25) t7 = MOA Goal Classification Table where \{t71, t72, \ldots, t7i\} \subset t7
   and \{ucn, a, el, b, g, gc\} \subset t7i where
   ucn = UML Use Case name
   a = OSSD Agent
   el = OSSD Entity list where e1 \in \{el1, el2, \ldots, eli\}
   b = OSSD Behavior
   g = OSSD Goal
   gc = OSSD Goal classification

26) t8 = Inter-view Inconsistency Detection Table where \{t81, t82, \ldots, t8i\} \subset t8
   and \{uml, ossd, cld, sqd, smd, ucd\} \subset t8i where
   uml = UML element name
   ossd = OSSD element
   cld \in \{Y, N\}
   sqd \in \{Y, N\}
   smd \in \{Y, N\}
   ucd \in \{Y, N\}
Figure 20a: Process Class Diagram Algorithm

for each \( w, w \in \text{WCLD} \)

/* create entries in UML Class Element and POS Tagging Table */
identify \( \text{uml} \), \( \text{uml} = \{w_1, w_2, \ldots, w_i\} \) and \( w \in \text{WCLD} \)
identify \( e \) corresponding to \( \text{uml} \)
create a new entry \( t1a \) in \( t1 \)
set \( t1a.\text{uml} = \text{uml} \), \( t1a.e = e \)
/* end create entries in UML Class Element and POS Tagging Table */
for each \( t1a, t1a \in t1 \)

/* update UML Class Element and POS Tagging Table */
for each \( w, w \in t1a.\text{uml} \)
find \( \text{sw} \) and \( w = \text{sw} \) via the SUMO/WordNet browser
if not found prompt user to select \( \text{sw} \) in SW
identify \( \text{swc} \) corresponding to \( \text{sw} \)
identify \( p \) corresponding to \( \text{sw} \)
if \( p = \text{verb} \) identify \( \text{sp} \)
set \( t1a.p = p \), \( t1a.sp = sp \), \( t1a.\text{swc} = \text{swc} \)
if \( t1a.e = \{\text{operation}\} \)
identify parameters \( \text{parms} \), \( \text{parms} \in \text{WCLD} \) associated with \( t1a.w \)
set \( t1a.\text{parms} = \text{parms} \)
if \( t1a.e = \{\text{attribute}\} \) or \( t1a.e = \{\text{operation}\} \)
identify \( t1b, t1b \in t1 \) such that
\( t1b.e = \{\text{class}\} \) and \( t1a.w \) is defined within \( t1b.w \)
set \( t1.a.\text{dw} = t1b.w \)
if \( t1a.e = \{\text{class}\} \)
identify \( t1b, t1b \in t1 \) such that
\( t1b.e = \{\text{class}\} \) and \( t1a.w \) is defined within \( t1b.w \)
set \( t1.a.\text{dwc} = "C" \) or "A" (to be refined)
identify \( t1b, t1b \in t1 \) such that
\( t1b.e = \{\text{class}\} \) and \( t1a.w \) is a sub-class of \( t1b.w \)
set \( t1.a.\text{dwc} = "G" \)
if \( t1a.e = \{\text{association}\} \)
/* create entry in OSSD Relations Classification Table */
create a new entry \( t2a \) in \( t2 \)
set \( t2a.an = \text{uml} \)
identify \( t1b, t1b \in t1 \) such that
\( t1b.e = \{\text{class}\} \) and \( t1b \) is the from end of association \( t1a \)
set \( t2a.\text{rel1} = \text{from} \) /* note: directional indicators may affect “to” and “from” */
set \( t2a.at1 = t1b.\text{uml} \)
identify role of \( t1b \)
set \( t2a.r = \text{role} \)
identify multiplicity of \( t1b \)
set \( t2a.m = \text{multiplicity} \)
identify \( t1b, t1b \in t1 \) such that
\( t1b.e = \{\text{class}\} \) and \( t1b \) is the to end of association \( t1a \)
set \( t2a.\text{rel2} = \text{to} \) /* note: directional indicators may affect “to” and “from” */
set \( t2a.at2 = t1b.\text{uml} \)
identify role of \( t1b \)
set \( t2a.r = \text{role} \)
identify multiplicity of \( t1b \)
set \( t2a.m = \text{multiplicity} \)
for each w, w ∈ WSQD
  /* create and update entries in UML Class Element and POS Tagging Table */
  execute A1-S2(w, WSQD, e) /* get UML element */
  execute A1-S1(e, t8a) /* update Inter-view Inconsistency Detection Table */
  set t8a.sqd = "Y"
/* end update UML Class Element and POS Tagging Table */

if t1a.dwc = “G”
  identify t1b, t1b ∈ t1 such that
  t1b.e = {class} and t1b is the superclass of the association t1a
  set t2a.rel1 = ‘superclass
  set t2a.at1 = t1b.uml
  identify t1b, t1b ∈ t1 such that
  t1b.e = {class} and t1b is the to subclass of the association t1a
  set t2a.rel2 = ‘to’
  set t2a.at2 = t1b.uml
if t1a.dwc = “C” or “A”
  identify t1b, t1b ∈ t1 such that
  t1b.e = {class} and t1b is the whole side of the association t1a
  set t2a.rel1 = ‘whole
  set t2a.at1 = t1b.uml
  identify t1b, t1b ∈ t1 such that
  t1b.e = {class} and t1b is the to part side of the association t1a
  set t2a.rel2 = ‘part’
  set t2a.at2 = t1b.uml
/* end create entry in OSSD Relations Classification Table */
execute A1-S1(t1a.uml, t8a) /* update Inter-view Inconsistency Table */
set t8a.cld = “Y”
/* end update UML Class Element and POS Tagging Table */

A1-2: Process Sequence Diagram Algorithm
execute A1-S2(w, s, e) /* get UML element */
if e = {class}
    if e = sending object
        set t3a.so = so
    else set t3a.ro = ro
/* end create entries in UML Class Element and POS Tagging Table */
for each t3a, t3a ∈ t3
/* update entries in UML Class Element and POS Tagging Table */
for each t3b, t3b ∈ t3
    if t3a.op = t3b.op /* operations have same name */
        set t3a.op = concat(t3a.op, t3a.ro)
        set t3b.op = concat(t3b.op, t3b.ro)
    if t3a.mt = {null}
        set t3a.mt = B
        /* set t3a.op to behavior of last message sent from UML class receiving return msg*/
        /* use B format */
/* classify OSSD Behavior */
if t3a.so = t3a.ro
    set soc = “Perform” and roc = “Perform”
else
    if t3a.mt = A
        set soc = “Control”
        set roc = “Perform”
    else
        if t3a.mt = B
            set roc = “Monitor”
            set soc = “Perform”
soc = soc
    set t3a.ro = roc
/* end classify OSSD Behavior */
/* update Inter-view Inconsistency Table entry*/
execute A1-S1(t3.op, t8a)
set t8a.sqd = “Y”
execute A1-S1(t3so, t8a)
set t8a.sqd = “Y”
execute A1-S1(t3ro, t8a)
set t8a.sqd = “Y”
/* end update entries in UML Class Element and POS Tagging Table */
for each t1a, t1a ∈ t1
    if t1a.e = {class}
        /* process a class */
        for each t3a, t3a ∈ t3
            if t1a.uml = t3a.so or t1a.uml = t3a.ro
                /* create entry in OSS Object Classification Table */
                create an entry t4a in t4
                set t4a.cn = t1a.uml
                if t1a.uml = t3a.so
                    add t3a.so to t4.b
                else
                    add t3a.ro to t4.b
Figure 21b: Figure continued
identify t1a.uml as whole top level or part sub level and set to t4a.c
set t4a.swc = t1a.swc

/* end create entries in OSSD Object Classification Table */

/* process a class */
for each t4a, t4a ∈ t4
/* classify OSSD Object */
if t4a.swc is a sublevel of SUMO Entity:Physical:Object then
    set t4a.oc = Entity
else
if t4a.swc is a sublevel of SUMO Entity:Physical:Object then
    set t4a.oc = Entity
else if t4a.swc is a sublevel of (SUMO Entity:Physical:Object:Agent or
Entity:Abstract:Attribute:RelationalAttribute:SocialRole)
    set t4a.oc = Agent
for each t2a, t2a ∈ t2
/* check each association of current class */
if t2a.rel1 = “from”
if t2a.at1 = t4a.cn or t2a.at2 = t4a.cn
    set av = null
    repeat for each v, v ∈ OSSD_Behavior_Thesaurus
        /* find association verb in Behavior Thesaurus */
        if v = t2a.an
            set av = v
            until av <> null or end of OSS_Behavior_Thesaurus
        /* if av= null then repeat search WordNet Database for t2a.an AND */
        /* repeat for each synonym identified for the verb */
        /* search the Behavior Thesaurus for that synonym */
        /* until verb is found in Behavior Thesaurus OR there are no more synonyms */
        if av <> null /* verb is Control or Monitor */
            if t2a.at1 = t4a.cn
                set t4a.oc = “Agent” /* the “from” side of the association */
            else
                set t4a.oc = “Entity” /* the “to” side of the association */
        /* check each association of current class */
        if all entries in list t4a.b is “Perform”
            if t4a.cn has only one state
                set t4a.oc = “Event”
            else
                set t4a.oc = “Entity”
        else /* some Behavior is “Control” and/or “Monitor” */
            set t4a.oc = “Entity”
    for each t3a, t3a ∈ t3
        if t3a.so = t4a.cn and t3a.ro <> t4a.cn
        set t4a.oc = “Agent”
/* end classify OSSD Object */
execute A1-S2(w, s, e) /* get UML element */
set c = e /* first word to get is the Class name of the StateMachine diagram */
for each w, w ∈ WSMD
execute A1-S2(w, s, e) /* get UML element */
execute A1-S1(e, t8a) /* update Inter-view Inconsistency Detection Table */
set t8a.smd = "Y"
if e = {state}
/* create new table entries in OSSD State, Transition, Constraints Classification Tables */
create a new entry t5a in t5
set t5a.sn = w
classify t5a.sn as one of {initial, intermediate, final} and set to t5a.sc
set transition count, tn=0
/* end create new table entries */
else
/* update table entry */
if e = {transition}
/* process a transition */
increment tn
if w is an incoming transition of t5a.sn
add the state from which the transition is incoming to t5a.itf
if w is an incoming transition of t5a.sn
add the state to which the transition is outgoing to t5a.ott
create a new entry t6a in t6 t5a.sn
set t6a.cn = c
set t6a.tn = tn
set t6a.itf = state from which the transition is incoming
set t6a.ott = state to which the transition is outgoing
add constraints on transition w to t6a.clist
identify the constraint types and add to t6a.cc
/* Precondition: state related attributes and values associated w/ Incoming Transition*/
/* Postcondition: state related attributes and values associated w/ Outgoing Transition */
/* Guard: conditional stmt of non-state attributes and values assoc. w/ Incoming Transition */
/* Trigger: behavior assoc. with Incoming Transition; associate Trigger w/ Behavior */
/* Action: behavior assoc. with Transition performed as a result of Transition */
add tn to t6a.tlist
/* process a transition */
else if e = {entry operation}
add e to t5a.enb
else if e = {do operation}
add e to t5a.db
else if e = {exit operation}
add e to t5a.exb
/* update table entry */
/* end create new table entries */
/* end process each word in StateMachine diagram */

Figure 22: Process StateMachine Diagram Algorithm
for each w, w ∈ WUCD and each w, w ∈ WUC

/* process each word in Use Case diagrams and Use Cases */
execute A1-S2(w, s, e) /* get UML element */
execute A1-S1(e, t8a) /* update Inter-view Inconsistency Detection Table */
set t8a.ucd = “Y”
if e = {class}
execute A1-S3(e, c) /* get OSSD Object Classification */
if e = {Use Case name}
set ucn = e
if c = {Entity}
add e to el
else if e = {Behavior}
set b = Behavior
else if c = {Agent}
set a = e
find a in t7
if a is not found in t7
/* create new table entries in */
/* OSSD State, Transition, Constraints Classification Table Part 2 */
create a new entry t7a in t7
set t7a.ucn = ucn
set t7a.a = a
add el to t7a.el
set t7b = b
set ucv = verb in ucn
set vp = past tense of ucv
set n = noun in ucn
set g = n + vp
set t7a.g = g
/* classify goals */
set gv = null
repeat for each v, v ∈ OSSD_Goal_Thesaurus
/* find Use Case verb in Behavior Thesaurus */
if ucv = v
set av = ucv
until av <> null or end of OSS_Goal_Thesaurus
/* if av= null then repeat search WordNet Database for ucv AND */
/* repeat for each synonym identified for ucv */
/* search the Goal Thesaurus for that synonym */
/* until ucv is found in Goal Thesaurus OR there are no more synonyms */
if av <> null
if t7a.gc = v.classification /* Maintain, Cease, Avoid, Achieve */
/* end process each word in Use Case Diagrams */
A1-5: Build the OSSD Model Algorithm

for each t1a, t1a ∈ t1
  if t1a.e = {class} /* create Objects */
    execute A1-S3(t1a.e, c) /* get OSSD Object Classification */
    if c = {agent}
      create c, c ∈ OSSD_Constructs and c ∈ Agent; assign instance = t1a.uml
      create c, c ∈ OSSD_Constructs and c ∈ Plan; assign instance = t1a.uml
    else if c = {entity}
      create c, c ∈ OSSD_Constructs and c ∈ Entity; assign instance = t1a.uml
    else create c, c ∈ OSSD_Constructs and c ∈ Event; assign instance = t1a.uml
  else if t1a.e = {association} /* create Relations and RelationAttributes */
    find t2a, t2a ∈ t2 such that t2a.an = t1a.e
    if t2a.rel1 = {whole} or {part}
      create c, c ∈ OSSD_Constructs and c ∈ {Composition, Aggregation}
    else if t2a.rel1 = {Superclass} or {Subclass}
      create c, c ∈ OSSD_Constructs and c ∈ Generalization
    else
      create c, c ∈ OSSD_Constructs and c ∈ Association; assign instance = t2a.an
      create c, c ∈ OSSD_Constructs and c ∈ FromObject; assign instance = t2a.at1
      create c, c ∈ OSSD_Constructs and c ∈ ToObject; assign instance = t2a.at2
      create c, c ∈ OSSD_Constructs and c ∈ Role; assign instance = t2a.r
      create c, c ∈ OSSD_Constructs and c ∈Multiplicity; assign instance = t2a.m
  else if t1a.e = {attribute} /* create Object Attributes */
    find t3a, t3a ∈ t2 such that t2a.an = t1a.e
    create c, c ∈ OSSD_Constructs and c ∈ ObjectAttribute
  else if t1a.e = {operation} /* create Behaviors */
    find t3a, t3a ∈ t3 such that t3a.op = t1a.e
    if t3a.soc = {Perform}
      create c, c ∈ OSSD_Constructs and c ∈ Perform; assign instance = t3a.op
      if t3a.mt = {B} /* Message type is B */
        create c, c ∈ OSSD_Constructs and c ∈ Event; assign instance = t3a.op
    else if t3a.soc = {Monitor}
      create c, c ∈ OSSD_Constructs and c ∈ Monitor; assign instance = t3a.op
    else
      create c, c ∈ OSSD_Constructs and c ∈ Control; assign instance = t3a.op
      if t3a.roc = {Perform}
        create c, c ∈ OSSD_Constructs and c ∈ Perform; assign instance = t3a.op
        if t3a.mt = {B} /* Message type is B */
          create c, c ∈ OSSD_Constructs and c ∈ Event; assign instance = t3a.op
      else if t3a.roc = {Monitor}
        create c, c ∈ OSSD_Constructs and c ∈ Monitor; assign instance = t3a.op
      else
        create c, c ∈ OSSD_Constructs and c ∈ Control; assign instance = t3a.op
    for each t5a, t5a ∈ t5 /* create States */
    if t5a.sc = {Initial}
      create c, c ∈ OSSD_Constructs and c ∈ Initial; assign instance = t5a.sn
    else if t5a.sc = {Intermediate}
      create c, c ∈ OSSD_Constructs and c ∈ Intermediate; assign instance = t5a.sn
    else
      create c, c ∈ OSSD_Constructs and c ∈ Final; assign instance = t5a.sn

Figure 24a: Build the OSSD Model Algorithm
for each t6a, t6a \in t6 /* create Transitions */
create c, c \in OSSD_Constructs and c \in Incoming; assign instance = t6a.itf
create c, c \in OSSD_Constructs and c \in Outgoing; assign instance = t6a.ott
for each cl, cl \in t6.clist /* create Constraints */
  if cl = \{Trigger\}
    create c, c \in OSSD_Constructs and c \in Trigger; assign instance = cl
  else if cl = \{Guard\}
    create c, c \in OSSD_Constructs and c \in Guard; assign instance = cl
  else if cl = \{Precondition\}
    create c, c \in OSSD_Constructs and c \in Precondition; assign instance = cl
  else if cl = \{Postcondition\}
    create c, c \in OSSD_Constructs and c \in Postcondition; assign instance = cl
else
  create c, c \in OSSD_Constructs and c \in Action; assign instance = cl
for each t7a, t7a \in t7 /* create Goals */
  if t7a.g = \{Achieve\}
    create c, c \in OSSD_Constructs and c \in Achieve; assign instance = t7a.g
  else if t7a.g = \{Maintain\}
    create c, c \in OSSD_Constructs and c \in Maintain; assign instance = t7a.g
  else if t7a.g = \{Avoid\}
    create c, c \in OSSD_Constructs and c \in Avoid; assign instance = t7a.g
else
  create c, c \in OSSD_Constructs and c \in Cease; assign instance = t7a.g

/* link all OSSD Constructs via OSSD Properties */
for each c1, c1 \in Object
/* link Objects, Relations, RelationAttributes */
  for each c1 in t2 where t2.rel1 = c1
    create a property p1, p1 \in OSSD_Properties and p1 = \{has\}
    link c1 with r1, r1 \in Relation where t2.an = r1 via p1
    create a property p2, p2 \in OSSD_Properties and p2 = \{has\}
    link r1 with a1, a1 \in ToObject where t2a.at1 = a1 and t2.at1 = c1 via p2
    create a property p3, p3 \in OSSD_Properties and p3 = \{has\}
    link r1 with a2, a2 \in FromObject where t2.at2 = a2 via p3
    create a property p4, p4 \in OSSD_Properties and p4 = \{has\}
    link a1 with a3, a3 \in Role and t2.r = a3 via p4
    create a property p5, p5 \in OSSD_Properties and p5 = \{has\}
    link a1 with a4, a4 \in Multiplicity and t2.m = a4 via p5
    create a property p5, p5 \in OSSD_Properties and p5 = \{has\}
    link a2 with a5, a5 \in Role and t2.r = a5 via p5
    create a property p6, p6 \in OSSD_Properties and p6 = \{has\}
    link a2 with a6, a6 \in Multiplicity and t2.m = a6 via p6
    create a property p7, p7 \in OSSD_Properties and p7 = \{performedBy\}
    link a3 with agent1, agent1 \in Agent and t2.at1 = agent1 via p7
    create a property p8, p8 \in OSSD_Properties and p8 = \{performedBy\}
    link a5 with agent2, agent2 \in Agent and t2.at2 = agent2 via p8
/* link Behavior with Objects and Attributes */
for each b1 in t3 where t3.op = b1
    create a property p9, p9 \in OSSD\_Properties and p9 = \{has\}
    link b1 with so1, so1 \in Object where t3.so = so1 via p9
    if b1_mt = \{A\}
        create a property p10, p10 \in OSSD\_Properties and p10 = \{sendMessageTo\}
        link b1 with ro1, ro1 \in Object where t3.ro = ro1 via p10
    else
        create a property p11, p11 \in OSSD\_Properties and p11 = \{causes\}
        link b1 with e1, e1 \in Event where b1 = e1 via p11
        create a property p12, p12 \in OSSD\_Properties and p12 = \{sendMessageTo\}
        link e1 with ro1, ro1 \in Object where t3.ro = ro1 via p12
        create a property p13, p13 \in OSSD\_Properties and p13 = \{inputs\}
        link b1 with a7, a7 \in Attributes where t3.parms = at7 via p13
        create a property p14, p14 \in OSSD\_Properties and p14 = \{outputs\}
        link b1 with a8, a8 \in Attributes where t3.parms = at8 via p14
/* link State with Objects, Behavior, Transitions, Constraints */
for each s1 in t5 where t5.sn = s1
    create a property p15, p15 \in OSSD\_Properties and p15 = \{has\}
    link s1 with o1, o1 \in Object where t5.cn = o1 via p15
    create a property p16, p16 \in OSSD\_Properties and p16 = \{entry\}
    link s1 with b2, b2 \in Behavior where t5.enb = b2 via p16
    create a property p17, p17 \in OSSD\_Properties and p17 = \{do\}
    link s1 with b3, b3 \in Behavior where t5.db = b3 via p17
    create a property p18, p18 \in OSSD\_Properties and p18 = \{exit\}
    link s1 with b4, b4 \in Behavior where t5.exb = b4 via p18
    for each itf1 in t5.itf
        create a property p19, p19 \in OSSD\_Properties and p19 = \{has\}
        link s1 with t1, t1 \in IncomingTransition where itf1 = t1 via p19
        create a property p20, p20 \in OSSD\_Properties and p20 = \{from\}
        link t1 with s1 where via p20
    for each otf1 in t5.otf
        create a property p21, p21 \in OSSD\_Properties and p21 = \{has\}
        link s1 with t2, t2 \in OutgoingTransition where otf1 = t2 via p21
        create a property p22, p22 \in OSSD\_Properties and p22 = \{to\}
        link t2 with s1 where via p22
    find otf1 in t6
    for each t3, t3 \in t6.tlist
        create a property p23, p23 \in OSSD\_Properties and p23 = \{followed\_by\}
        link t3 with otf1 via p23
        create a property p24, p24 \in OSSD\_Properties and p24 = \{has\}
        link otf1 with ct1, ct1 \in Constraint where t6.ott = otf1 via p24
    for each b5, b5 \in t6.cl
        create a property p25, p25 \in OSSD\_Properties and p25 = \{contains\}
        link b5 with b6, b6 \in Behavior and b6 = b5 via p25
        if b5 = \{Action\} or b5 = \{Trigger\}
            create a property p26, p26 \in OSSD\_Properties and p26 = \{has\}
            link b5 with b7, b7 \in Behavior and b7 = b7 via p25
Figure 24c: Figure continued.
/* link Goals with Objects */
for each g1 in t6 where t6.g = g1
    for each o2, o2 ∈ t7.o
        create a property p27, p27 ∈ OSSD_Properties and p27 = {concerns}
        link g1 with o2 via p27
    for each b6, b6 ∈ t7.b
        create a property p28, p28 ∈ OSSD_Properties and p28 = {contains}
        link g1 with plan1, plan1 ∈ Plan where plan1 = t6.a via p28
        create a property p29, p29 ∈ OSSD_Properties and p29 = {operationalizes}
        link g1 with b7 where b7 ∈ Behavior and b6=b7 via p29
        create a property p30, p30 ∈ OSSD_Properties and p30 = {has}
        link plan1 with agent2, agent2 ∈ Agent and t6.a = agent2 via p30
    create a property p31, p31 ∈ OSSD_Properties and p31 = {underResponsibilityOf}
    link g1 with agent2 where agent2 ∈ Agent and t6.a = agent2 via p31

A1-S1: Get Inter-view Inconsistency Table Entry
input: uml /* UML element */
output: t8a /* entry in Inter-view Inconsistency Table */
find t8a, t8a ∈ t8 and t8a.uml = uml
    if not found
        create a new entry t8a in t8
        set t8a.cld = t8a.sqd = t8a.smd = t8a.ucd = “N”
        set t8a.uml = uml
    return t8a

A1-S2: Get UML Element
input: w /* UML word */
    s /* set of words from a UML diagram */
output: e /* UML element */
set e = null
for each t1a, t1a ∈ t1
    if w = t1a.uml
        e = w
if e = null
    for each w1, w1 ∈ s
        concatenate w1 to w
    for each t1a, t1a ∈ t1
        if w = t1a.uml
            e = w
    until e <> null or end of t1
return e

A1-S3: Get OSSD Object Classification
input: o /* OSSD Object */
output: c /* OSSD Classification */
set c = null
for each t4a, t4a ∈ t4
    if o = t4a.cn
        c =t4a.oc
until c <> null or end of t4
return c
4.3.4.4 Summary of UML to OSSD Model Transformations

Tables 3 through 5 summarize the transformations from the UML design to an instance of the OSSD Model. Table 3 shows the one-to-one mapping of major elements from a UML Class Diagram to the OSSD Model. Table 4 shows the one-to-one mapping subset of major elements from a Sequence Diagram to the OSSD Model. Table 5 shows the one-to-one mapping of major elements from a UML StateMachine Diagram to the OSSD Model. Some UML elements do not have a mapping to the OSSD Model because they concern implementation details that are not utilized in the target requirements specification language.

Table 3: UML Class Diagram Classification

<table>
<thead>
<tr>
<th>UML 2.0</th>
<th>OSSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Construct:Object: {StateBased: {Agent or Entity}} or Event</td>
</tr>
<tr>
<td>AttributeName</td>
<td>Construct:Attribute:ObjectName</td>
</tr>
<tr>
<td>AttributeType</td>
<td>An implementation detail not represented in OSSD</td>
</tr>
<tr>
<td>AttributeVisibility</td>
<td>Construct:Attribute:ObjectName property has Visibility</td>
</tr>
<tr>
<td>AttributeMultiplicity</td>
<td>Construct:Attribute:ObjectName property has Multiplicity</td>
</tr>
<tr>
<td>Operation</td>
<td>Construct:Behavior: {monitor, control, perform}</td>
</tr>
<tr>
<td>OperationParameter (input)</td>
<td>Construct:Behavior property inputs Attribute</td>
</tr>
<tr>
<td>OperationParameter (output)</td>
<td>Construct:Behavior property outputs Attribute</td>
</tr>
<tr>
<td>OperationVisibility</td>
<td>Construct:Behavior property {inputs or outputs} Attribute:ObjectName property Visibility</td>
</tr>
<tr>
<td>OperationType</td>
<td>An implementation detail not represented in OSSD</td>
</tr>
</tbody>
</table>
Association
AssociationType¹
Association AggregationKind²
AssociationOwningName
AssociationOwningRole
AssociationOwningMultiplicity
AssociationOwningConstraint
AssociationOwningNavigability
AssociationOwnedName
AssociationOwnedRole
AssociationOwnedMultiplicity
AssociationOwnedConstraint
AssociationOwnedNavigability

¹ UML 2.0 Association Types include: binary, n-ary
² UML 2.0 Association AggregationKind: none (simple association), aggregation, composition

Transformed as multiple leaves of the Relation:Association construct
See Association above

Table 4: UML Sequence Diagram Classification

<table>
<thead>
<tr>
<th>UML 2.0</th>
<th>OSSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectLifeline</td>
<td>Match with OSSD Object</td>
</tr>
<tr>
<td>SynchronousMessage Operation Name</td>
<td>Match with Behavior</td>
</tr>
<tr>
<td>SynchronousMessage Arguments</td>
<td>Match with Behavior{Inputs or Outputs}</td>
</tr>
<tr>
<td>ReturnFromSynchronousMessage</td>
<td>Match with OSSD Event</td>
</tr>
<tr>
<td>Asynchronous Message</td>
<td>Match with Behavior</td>
</tr>
<tr>
<td>StateInvariantIcon</td>
<td>Match with OSSD State</td>
</tr>
<tr>
<td>SelfReferenceMessageOperation Name</td>
<td>Match with Behavior</td>
</tr>
<tr>
<td>SelfReferenceMessageArguments</td>
<td>Match with Behavior{Inputs or Outputs}</td>
</tr>
<tr>
<td>StateInvariantConstraint</td>
<td>Match with OSSD Constraint</td>
</tr>
<tr>
<td>DurationConstraint</td>
<td>Match with OSSD Constraint</td>
</tr>
<tr>
<td>TimeConstraint</td>
<td>Match with OSSD Constraint</td>
</tr>
</tbody>
</table>

Table 5: UML StateMachine Classification

<table>
<thead>
<tr>
<th>UML 2.0</th>
<th>OSSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateType¹</td>
<td>Construct:State property contains State</td>
</tr>
<tr>
<td>InitialState</td>
<td>Construct:State:Initial</td>
</tr>
<tr>
<td>FinalState</td>
<td>Construct:State:Final</td>
</tr>
<tr>
<td>IntermediateStateName</td>
<td>Construct:State</td>
</tr>
<tr>
<td>IntermediateStateEntryTransition</td>
<td>Construct:State property has Transition:incoming</td>
</tr>
<tr>
<td>IntermediateStateExitTransition</td>
<td>Construct:State property has Transition:outgoing</td>
</tr>
<tr>
<td>StateEntryAction</td>
<td>Construct:State property Entry Behavior</td>
</tr>
<tr>
<td>StateDoActivity</td>
<td>Construct:State property has Behavior</td>
</tr>
<tr>
<td>StateExitAction</td>
<td>Construct:State property Exit Behavior</td>
</tr>
<tr>
<td>Transition¹</td>
<td>Construct:Transition</td>
</tr>
<tr>
<td>TransitionTrigger</td>
<td>Construct:Transition property has Constraint:Trigger</td>
</tr>
<tr>
<td>TransitionGuard</td>
<td>Construct:Transition property has Constraint:Guard</td>
</tr>
<tr>
<td>TriggerEvent on ExternalTransition</td>
<td>Construct:Transition:{incoming or outgoing} has Constraint:Trigger</td>
</tr>
<tr>
<td>Condition on External Transition</td>
<td>Construct:Transition:{Incoming or Outgoing} has Constraint:{Precondition or Postcondition}</td>
</tr>
<tr>
<td>Guard on External Transition</td>
<td>Construct:Transition:{Incoming or Outgoing} has Constraint:Guard</td>
</tr>
<tr>
<td>Action on ExternalTransition</td>
<td>Construct:State property {Entry or Exit} Behavior (corresponding to Incoming or Outgoing transition)</td>
</tr>
<tr>
<td>DecisionNode</td>
<td>Construct:Transition property followed by Transition</td>
</tr>
</tbody>
</table>

¹ State Type includes: simple, composite, submachine, submachine state
² TransitionType includes: basic, fork, join
4.3.5 MOA Consistency Checking

4.3.5.1 Overview

MOA identifies basic consistency problems during the transformation of the UML Model into the OSSD Model. Although UML CASE tools used to produce the UML Diagrams do have some inconsistency detection capabilities, such as those performed by the Rose Model Checker [Moors], a universally accepted set of consistency checks does not exist. Furthermore, these consistency checks are usually based on the well-formed rules (WFR) specified in the UML 2.0 Specification. These WFRs address primarily the syntactic inconsistencies within a given UML diagram such as naming, visibility, and scope. UML provides few explicitly defined inter-diagram consistency rules. “There exists no general techniques for specifying semantic (and, in particular, behavioral) consistency constraints” [Engels4 et al.].

Consistency checking is a two-stage process. The first stage, which begins once the OSSD Model has been created for a specific set of source language diagrams, concerns consistency checking of the OSSD constructs. Rules attached to the properties in the OSSD Model facilitate this stage of the consistency checking process. The second stage introduces an Inter-View Inconsistency Detection technique, which is based on the Consistency framework and inter-diagram consistency rules of the source language. Section 4.3.5.2 introduces a consistency framework that organizes these rules and Section 4.3.5.3 introduces the Inter-View Inconsistency Detection technique.

4.3.5.2 Consistency Checking of OSSD Model Constructs

This research defines a consistency framework based on the OSSD Model. This framework organizes rules for inconsistency detection based on interactions among the set of ontological constructs, O, where O = \{Agent, Entity, Event, Goal, Relation, State, Behavior, Constraint\} and Plan ∉ O since Plan ∉ \{Behavior1, Behavior2, … Behaviorn\}. This framework does not include the Plan construct because it represents the combination of Behavior constructs and so would cause unnecessary redundancy in the framework. This framework includes both syntactic and semantic inconsistencies. Rules from the consistency framework are attached to properties of the OSSD Model. Figure 27 shows the organization of the consistency framework into 36 categories. In the

<table>
<thead>
<tr>
<th></th>
<th>OBJECT</th>
<th>ATTRIBUTE</th>
<th>BEHAVIOR</th>
<th>GOAL</th>
<th>RELATION</th>
<th>STATE</th>
<th>TRANSITION</th>
<th>CONSTRAINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>OO</td>
<td>OA</td>
<td>OB</td>
<td>OG</td>
<td>OR</td>
<td>OS</td>
<td>OT</td>
<td>OC</td>
</tr>
<tr>
<td>Attribute</td>
<td>AA</td>
<td>AB</td>
<td>AG</td>
<td>AR</td>
<td>AS</td>
<td>AT</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td>Behavior</td>
<td>BB</td>
<td>BG</td>
<td>BR</td>
<td>BS</td>
<td>BT</td>
<td>BC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goal</td>
<td>GG</td>
<td>GR</td>
<td>GS</td>
<td>GT</td>
<td>GC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relation</td>
<td>RR</td>
<td>RS</td>
<td>RT</td>
<td>RC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>SS</td>
<td>ST</td>
<td>SC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T T</td>
<td>TC</td>
<td></td>
</tr>
<tr>
<td>Constraint</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CC</td>
</tr>
</tbody>
</table>

Figure 27: Consistency Framework

contents of the consistency framework table, consistency rules are labeled based on acronyms created by reading the consistency framework table first by row followed by column. For example, an
OB_Rule1 concerns the relationship between Object and Behavior, represented in the consistency framework as O for the object row and B for the behavior column.

In this research, we formally define a set of OSSD_Consistency_Rules where OSSD_Consistency_Rules ⊂ All_Rules. Figure 28 gives two examples of such consistency rules, OB_Rule1 and OR_Rule1, where {OB_Rule1, OB_Rule2} ⊂ OSSD_Consistency_Rules.

OB_Rule1: This category includes rules affecting an Object’s Behavior as given in its definition (UML Class Diagram) and the Object’s use (or lack of use) of that Behavior (UML Sequence Diagram). An example of this OB_Rule1 is: a message sent from an Object must be associated with a Behavior of that Object. Attaching the axiom given in Figure 28, specified in first order predicate logic, to the OSSD property has that links a Construct:Object:State-based and Construct:Behavior enforces this OB_Rule1.

OR_Rule1: This category includes rules affecting Relations defined for an Object (obtained from a UML Class Diagram) and Behavior of that Object as represented by messages that an Object sends (obtained from a UML Sequence Diagram). An example of this OR_Rule1 is: for a message to be exchanged between one Object and a second Object there must be a Relation defined between them. This OR_Rule1 is enforced by the combination of the OB_Rule1 above and executing the following axiom, specified in first order predicate logic, attached to the OSSD property has that links Construct:Object-State-based with Construct:Behavior.

4.3.5.3 Consistency Checking of Source Language Views

The Inter-View Inconsistency Detection technique for processing of UML diagrams is based on the consistency framework. The primary purpose of the technique is to identify inconsistencies in the definitions of model elements across the partial, overlapping views of the design. In this research, we formally define a set of OSSD_Inter-View_Consistency_Rules where OSSD_Inter-View_Consistency_Rules ⊂ All_Rules. Figure 29 contains three examples of such rules: IC_Rule1, IC_Rule2, and IC_Rule3 where {IC_Rule1, IC_Rule2, IC_Rule3} ⊂ IC_Rules.
The meaning of these rules is as follows:

IC_Rule1: an OSSD *Object* must be defined in a UML Class Diagram and referenced in at least one UML Sequence Diagram and one UML StateMachine Diagram

IC_Rule2: an OSSD *Relation* must be defined in a UML Class Diagram and referenced in at least one UML Sequence Diagram

IC_Rule3: an OSSD *Behavior* must be defined in a UML Class Diagram and referenced in at least one UML Sequence Diagram and one UML StateMachine Diagram

<table>
<thead>
<tr>
<th>UML</th>
<th>OSSD</th>
<th>Class Diagram</th>
<th>Sequence Diagram</th>
<th>StateMachine Diagram</th>
<th>Use Cases or Use Case Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassName</td>
<td>Agent3</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>AssociationName</td>
<td>Association6</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>OperationName</td>
<td>Behavior5</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
Additional rules can be added to the Inter-View Inconsistency Detection Table based on types of UML Diagrams. For example, a consistency rule that requires each actor in a UML Use Case diagram to be associated with a Class in a UML Class Diagram can be added. Given the knowledge that each Class in a UML Class Diagram is represented in OSSD as a *Construct:Object*, verifying this rule requires a simple check in the Inter-View Inconsistency Detection Table to show a one to one correspondence between each Actor in UML Use Case Diagram and some *Construct:Object*.

The algorithms that transform a UML design to OSSD Model instance (see Chapter 4 Section 4.3.4.2) add new entries to this Inter-View Inconsistency Detection Table each time a unique UML element is identified.

### 4.3.5.4 Consistency Checking Algorithm

The Consistency Checking Algorithm is given in Figure 30.

**Figure 30: Consistency Checking Algorithm**

```
A2-1: Consistency Checking Algorithm

for each p, p ∈ OSSD_Properties
    if p contains rule r, r ∈ OSSD_Conistency_Rules
        execute r
for each r, r ∈ OSSD_Inter-View_Conistency_Rules
    execute r
```

### 4.3.6 OSSD Model to KAOS Transformation

#### 4.3.6.1 Overview

Transformation from the OSSD Model to an agent-based model produces an agent-oriented requirements specification that is used as input to an appropriate verification tool in order to detect inconsistencies. Many agent-oriented specification techniques already have verification tools for detecting inconsistencies associated with them. The target language we utilize is the KAOS specification language. KAOS defines an entity as an autonomous object that is not dependent upon other objects and an agent as an object that has both behavior and choice; however, KAOS does not describe in detail how an entity and an agent differ [Silva et al.]. From the numerous examples given in literature on KAOS, only agents can perform operations. Entities do not perform operations implying that if an object performs an operation then it must be some type of agent. Therefore, since OSSD *Entities* perform behavior but are not agents, a *Monitor/Control Behavior* of an OSSD Agent that has the OSSD property *sends a message* to an OSSD *Entity* (and therefore corresponds to a *Perform Behavior* of that *Entity*) transforms to an operation of the corresponding KAOS agent. Lastly, the transformation of the OSSD *Constraint* to KAOS is based on a related KAOS transformation application [Van Hung].

#### 4.3.6.2 OSSD Model to KAOS Transformation Algorithm

Figure 31 gives the OSSD to KAOS Transformation Algorithm. The algorithm produces a textual KAOS specification. Figure 32 gives an example of the template for the definition of a KAOS agent. KAOS details are in bold print and OSSD Model references are in italics. The textual specification is not an executable KAOS specification but is used to enter information into a KAOS tool.
A2-1: OSSD to KAOS Transformation Algorithm

A3-1. Using Tables 7 and 8, transform each:
- OSSD Agent, Entity, Event into a KAOS Agent, Entity, Event;
- OSSD Attribute into KAOS attribute;
- Behavior into KAOS operation;
  if the Monitor or Control Behavior of an Agent sends a message to an Entity then
  the Behavior of that message becomes a KAOS Operation performed by that Agent;
  the Attributes included in that message become associated with that Agent
  according to the type of Behavior (either Monitor or Control)

A3-2. transform each OSSD Relation into a KAOS Relation using Table 7;

A3-3. transform each OSSD State, Transition and Constraint into a KAOS state variables, transition
variables, and constraints using Tables 7 and 8;

A3-4. transform each Goal into a KAOS Goal using Table 8.

Figure 31: OSSD Model to KAOS Transformation Algorithm

Agent Construct:Object:Statebased:Agent
  Has Construct:Object property has
  Construct:Attribute:ObjAttrib
  Inherited from Construct:Object property has
  Construct:Relation:Non-Association:Generalization:Subclass property has
  Construct:Relation:Non-Association:Generalization:Superclass
  Monitors Construct:Object:Statebased property has
  Construct:Behavior:Monitor property sends message to
  Construct:Object:Statebased /
  Construct:Object:Statebased property has
  Construct:Behavior:Monitor property inputs
  Construct:Attribute:ObjAttrib
  Controls Construct:Object:Statebased property has
  Construct:Behavior:Control property sends message to
  Construct:Object:Statebased /
  Construct:Object:Statebased property has
  Construct:Behavior:Control property outputs
  Construct:Attribute:ObjAttrib
  ResponsibleFor Construct:Goal property under responsibility of
  Construct:Object:Statebased:Agent
  DependsOn Construct:Object:Statebased:Agent
  For Construct:Goal property under responsibility of
  Construct:Object:Statebased:Agent
  Performs Construct:Object:Statebased property has
  Construct:Behavior:Perform
End

Figure 32: Template for Specification of a KAOS Agent
Table 7: Mapping OSSD to KAOS Meta-Objects and Meta-Attributes

<table>
<thead>
<tr>
<th>OSSD</th>
<th>KAOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct:Object: Statebased: Entity or Agent</td>
<td>Object:Entity</td>
</tr>
<tr>
<td>Construct:Object: Statebased: Agent</td>
<td>Object:Agent</td>
</tr>
<tr>
<td>Construct:Object: Event</td>
<td>Object:Event</td>
</tr>
<tr>
<td>Construct:Relation: Association</td>
<td>Object:Association:ApplicationSpecific</td>
</tr>
<tr>
<td>Construct:Relation: NonAssociation: Generalization</td>
<td>Object:Association:Builtin:IsA</td>
</tr>
<tr>
<td>Construct:Relation: NonAssociation: Aggregation or Composition</td>
<td>Object:Association:Builtin:part of</td>
</tr>
<tr>
<td>Construct:Attribute: ObjAttrib</td>
<td>Attribute:Range</td>
</tr>
<tr>
<td>Construct:Behavior</td>
<td>Operation</td>
</tr>
<tr>
<td>Construct:Plan</td>
<td>Scenario</td>
</tr>
</tbody>
</table>

Table 8: Mapping Table for OSSD to KAOS Meta-relationship Mappings

<table>
<thead>
<tr>
<th>OSSD</th>
<th>KAOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property concerns</td>
<td>Concerns</td>
</tr>
<tr>
<td>Construct:Relation</td>
<td>Link</td>
</tr>
<tr>
<td>Construct:Relation:Association property has RelationAttribute</td>
<td>Link:Role</td>
</tr>
<tr>
<td>property has Role</td>
<td></td>
</tr>
<tr>
<td>Construct:Relation:NonAssociation: Generalization property has RelationAttribute property has Multiplicity</td>
<td>Link:Multiplicity</td>
</tr>
<tr>
<td>property has Multiplicity</td>
<td></td>
</tr>
<tr>
<td>Construct:Relation:Association property has RelationAttribute: [ToObject OR FromObject] property has Multiplicity</td>
<td>Link:Position</td>
</tr>
<tr>
<td>Construct:Attribute property has ObjectAttribute property has Multiplicity</td>
<td>ValuesIn: Multiplicity</td>
</tr>
<tr>
<td>Construct:Behavior:Monitor</td>
<td>Monitoring</td>
</tr>
<tr>
<td>Construct:Behavior:Control</td>
<td>Control</td>
</tr>
<tr>
<td>Construct:Behavior:Perform</td>
<td>Performance</td>
</tr>
<tr>
<td>Property under responsibility of</td>
<td>Responsibility</td>
</tr>
<tr>
<td>Construct:Goal property</td>
<td>DependsOn</td>
</tr>
<tr>
<td>property dependsOn Construct:Goal</td>
<td></td>
</tr>
<tr>
<td>Property inputs</td>
<td>Input</td>
</tr>
<tr>
<td>Property outputs</td>
<td>Output</td>
</tr>
<tr>
<td>Construct:Behavior property causes</td>
<td>Cause</td>
</tr>
<tr>
<td>Construct:State property has Transition:Incoming hasConstraint:Precondition</td>
<td>Operation: DomPre</td>
</tr>
<tr>
<td>Construct:State property has Transition:Outgoing hasConstraint:Postcondition</td>
<td>Operation: DomPost</td>
</tr>
<tr>
<td>Construct:State property has Transition:Incoming hasConstraint:Trigger</td>
<td>Operationlization: ReqTrigFor</td>
</tr>
<tr>
<td>Construct:State property has Transition:Incoming hasConstraint:Guard</td>
<td>Operationlization: ReqPre</td>
</tr>
<tr>
<td>Construct:State property has Transition:Outgoing hasConstraint:Guard</td>
<td>Operationlization: ReqPost</td>
</tr>
</tbody>
</table>

4.4 Summary

This chapter introduced the Methodology for Object to Agents (MOA), the Ontology for Software Specification and Design (OSSD); two methods of consistency checking executed during MOA utilizing axioms attached to properties of the OSSD Model and a three-dimensional Inter-View Inconsistency Detection table; and lastly the high-level algorithms describing the transformation from a UML design, to an OSSD Model instance, and then to a KAOS textual specification.
5 Elevator Case Study

5.1 Introduction

The Elevator System case study explains the methodology developed in this research. The basic requirements for this Elevator System are:

- elevator services 3 floors
- floor buttons exist on each floor to call the elevator:
  - one Up floor button on the 1st and 2nd floors and
  - one down floor button on the 2nd and 3rd floor
- 3 floor buttons exist within the elevator car for the user to select desired floor (1, 2, 3)
- elevator car contains buttons for stopping the elevator, opening doors, and closing doors
- all buttons have a light that turns on when pressed and turns off when the elevator arrives at the associated floor
- elevator doors are controlled by a timer after each stop
- elevator has two doors; one inner door is attached to the elevator car; one outer door is attached to each floor
- each elevator door has a door sensor which detects if the door is open or closed and detects if something is blocking the doorway which prevents the door from closing
- elevator car has a motor that moves the elevator up and down
- elevator car has two sensors; a floor sensor that identifies where it is located based on reading a floor identification tape on the inside of the elevator shaft; a weight sensor detects if the maximum weight has been exceeded which prevents the elevator doors from closing.

In a typical software development project, multiple teams exist to develop the elevator car, the elevator controller, the elevator motor, the elevator doors, the elevator button panels as well as teams that focus on the performance and safety aspects of the overall system. This team organization is similar to that suggested in the development work of the Viewpoint Framework for integrating multiple perspectives in software design [Finkelstein92]. When the time comes to integrate the work of these teams, typically meetings are held and manual reviews are conducted to discover inconsistencies. This progress can be extremely time-consuming and error-prone when there exist numerous interactions to consider.

Section 5.2 contains an example subset of UML diagrams developed for the Elevator System case study including Use Cases and Use Case, Sequence, Class, and StateMachine Diagrams. Section 5.3 provides example mappings from the UML design to the OSSD Model, a sample of OSSD Model instances created for the Elevator System, a sample of the OSSD Model represented in OWL notation, and examples of MOA consistency checking applied to the OSSD Model instance created for the Elevator System. Lastly, Section 5.4 contains Sections of a KAOS specification created for the Elevator System case study, examples of goal patterns that are produced during the KAOS processing, and a discussion of the error detection that is performed using the KAOS specification for the Elevator System.

5.2 UML Representation of the Elevator System

Several examples of Use Cases for the Elevator System are given in Figures 30 through 39. Although there does not exist a universally accepted Use Case format, the structure of the following Use Cases consists of the basic and commonly used subSections. Nested Use Cases are underlined for ease of understanding.
Use Case name: Request Elevator  
Primary Actor(s): Passenger, Elevator Controller  
Precondition: Passenger is at a floor and wants to ride an elevator car  
Postcondition: Elevator car is stopped at the passenger’s floor; Elevator doors are open  
Scenario:  
Passenger presses an elevator call up button or down button  
Elevator Controller turns on the call button light  
Elevator Controller requests Move Elevator  
Elevator Controller turns off the call button light  
Elevator Controller requests Open Doors  
Alternative Scenario:  
Passenger presses an elevator call up button or down button  
Elevator Controller turns on the call button light  
Elevator car is at the Passenger’s floor  
Elevator Controller turns off the call button light  
Elevator Controller requests Open Doors

Figure 33: UML Use Case: Request Elevator

Use Case name: Open Doors  
Primary Actor(s): ElevatorController, Door Controller  
Precondition: Elevator car is stopped; Elevator Doors are closed  
Postcondition: Elevator car is stopped; Elevator Doors are open  
Scenario:  
Elevator Controller requests doors open  
Door Controller requests Inner Door open and Outer Door open simultaneously  
Inner Door executes open and Outer Door executes open simultaneously

Figure 34: UML Use Case: Open Doors

Use Case name: Move Elevator  
Primary Actor(s): Elevator Controller, Elevator Car  
Precondition: Elevator car at a floor that is not the requested floor  
Postcondition: Elevator car is at the requested floor; Elevator doors are open  
Scenario:  
Elevator Controller requests elevator car move to passenger’s requested floor  
Elevator Car moves up or down based on the current and requested floor locations  
Floor Sensor informs Elevator Car of arrival at each floor  
When the current floor is the requested floor, the Elevator Car stops the elevator

Figure 35: UML Use Case: Move Elevator

Use Case name: Request Floor  
Primary Actor(s): Passenger  
Precondition: Passenger is in the elevator car  
Postcondition: Passenger is at the requested floor; Elevator doors are open  
Scenario:  
Passenger presses an elevator car floor button  
Elevator Controller turns on the elevator car floor button light  
Elevator Controller requests Close Doors  
Elevator Controller requests Move Elevator  
Elevator Controller turns off the elevator car floor button light  
Elevator Controller requests Open Doors

Figure 36: UML Use Case: Request Floor
Use Case name: Close Doors  
Primary Actor(s): Elevator Controller, Door Controller  
Precondition: Elevator car is stopped; Elevator Doors are open  
Postcondition: Elevator car is stopped; Elevator Doors are closed  
Scenario:  
- Elevator Controller requests Ensure Safe Door Operation and Prevent Exceeding Elevator Weight Limit simultaneously  
- Elevator Controller requests doors close  
- Door Controller requests Inner Door close and Outer Door close simultaneously  
- Inner Door executes close and Outer Door executes close simultaneously  

Figure 37: UML Use Case: Close Doors

Use Case name: Ensure Safe Door Operation  
Primary Actor(s): Elevator Controller, Door Sensor  
Precondition: There is no obstruction to the elevator doors  
Postcondition: There is no obstruction to the elevator doors  
Scenario:  
- Elevator Controller requests check for door obstruction  
- Door Controller requests check for inner door obstruction and outer door obstruction simultaneously  
- Inner Door Sensor indicates inner door not obstructed  
- Outer Door Sensor indicates outer door not obstructed  
Alternative Scenario:  
- Elevator Controller requests check for door obstruction  
- Door Controller requests check for inner door obstruction and outer door obstruction simultaneously  
- Inner Door Sensor indicates inner door is obstructed  
- Door Controller rings Inner Door Alarm

Figure 38: UML Use Case: Ensure Safe Door Operation

Use Case name: Prevent Exceeding Elevator Weight Limit  
Primary Actor(s): Elevator Controller, Elevator Car, Weight Sensor  
Precondition: Maximum weight limit has not been reached  
Postcondition: Maximum weight limit has not been reached  
Scenario:  
- Elevator Controller requests check for excess weight  
- Elevator Car requests check for excess weight  
- Weight Sensor indicates no excess weight  
Alternative Scenario:  
- Elevator Controller requests check for excess weight  
- Elevator Car requests check for excess weight  
- Weight Sensor indicates excess weight  
- Elevator Car rings Car Alarm

Figure 39: UML Use Case: Prevent Exceeding Elevator Weight Limit

Figures 40 to 46 contain a subset of UML diagrams developed for the Elevator System case study including a Use Case diagram, a Sequence diagram showing a passenger’s request for an elevator, a StateMachine diagram for the door controller, and a Class diagram of the elevator system.
Figure 40: Elevator System Use Case Diagram

Figure 41: Passenger Request Sequence Diagram
Figure 42: OpenDoors and CloseDoors Sequence Diagrams

Figure 43: Move Elevator Sequence Diagram
Figure 44: Elevator System Class Diagram
Figure 45: Elevator Controller State Machine Diagram

Figure 46: DoorController State Machine Diagram
5.3 Applying MOA to the Elevator System Case Study

5.3.1 UML Design to OSSD Model

Tables 9 through 14 contain examples of the mappings from the UML design to the OSSD Model for the Elevator System case study. Appendix B provides additional table entries. In Table 9, each UML Class element is listed by name, type of UML Class element, its part-of-speech tagging, the SUMO/WordNet association for each POS, the UML element within which it is defined (if the UML element is a UML Class then it is given with additional information detailing its classification as G=generalization, A=aggregation, C=composition), and any parameters associated with the UML Class element.

It is significant to note the classification of the UML Classes “ElevatorCar” and “DoorController” as OSSD Entities. While some agent-oriented approaches might consider one or both of these to be agents, OSSD considers them to be Entities. The “ElevatorCar” is an Entity even though it “controls” and “monitors” other devices, for example the “Motor” and “Weight Sensor” UML Classes respectively, these UML Classes are components of the “ElevatorCar”. Therefore, the “ElevatorCar” is basically controlling and monitoring itself, which does not make it an Agent. The “DoorController” is an Entity even though it “controls” and “monitors” other UML Classes that are not components of itself, specifically the “InnerDoor” and “OuterDoor”, the state of the “DoorController” is directly controlled by the “ElevatorController” via the “OpenDoors” and “CloseDoors” operations. Only an Agent can change the state of that Agent.

<table>
<thead>
<tr>
<th>UML Name</th>
<th>UML Element</th>
<th>POS: SubPOS</th>
<th>SUMO/WordNet</th>
<th>Defined within UML (G/A/C)</th>
<th>Params</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elevator Car</td>
<td>Class</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>Elevator System/C</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Entity:Physical:Object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>Attribute</td>
<td>Noun</td>
<td>Entity:Abstract:Attribute</td>
<td>ElevatorCar</td>
<td>None</td>
</tr>
<tr>
<td>Moving</td>
<td>Attribute</td>
<td>Verb</td>
<td>Entity:Physical:Process: Motion</td>
<td>ElevatorCar</td>
<td>None</td>
</tr>
<tr>
<td>Location</td>
<td>Attribute</td>
<td>Noun</td>
<td>Miscellaneous Relation:Located</td>
<td>ElevatorCar</td>
<td>None</td>
</tr>
<tr>
<td>Move To</td>
<td>Operation</td>
<td>Verb:present Adjective</td>
<td>Entity:Physical:Process: Motion Not available</td>
<td>ElevatorCar</td>
<td>F D</td>
</tr>
<tr>
<td>Elevator</td>
<td>Operation</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorCar</td>
<td>F</td>
</tr>
<tr>
<td>Arrived</td>
<td></td>
<td></td>
<td>Entity:Physical:Process: Motion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weight</td>
<td>Operation</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorCar</td>
<td>None</td>
</tr>
<tr>
<td>Exceeded</td>
<td></td>
<td>Verb</td>
<td>Entity:Abstract:Attribute: Relational Attribute</td>
<td>ElevatorCar</td>
<td>None</td>
</tr>
<tr>
<td>Weight</td>
<td>Class</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorCar/C</td>
<td>None</td>
</tr>
<tr>
<td>Sensor</td>
<td></td>
<td></td>
<td>Entity:Physical:Object</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floor</td>
<td>Class</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorCar/C</td>
<td>None</td>
</tr>
<tr>
<td>Alarm</td>
<td>Class</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorCar/C</td>
<td>None</td>
</tr>
<tr>
<td>Button</td>
<td>Class</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorCar/C</td>
<td>None</td>
</tr>
<tr>
<td>Panel</td>
<td></td>
<td></td>
<td>Entity:Physical/Object</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 10: MOA Relations Classification

<table>
<thead>
<tr>
<th>UML assoc.</th>
<th>OSSD Relation</th>
<th>OSSD Relation</th>
<th>OSSD Role</th>
<th>OSSD Mult.</th>
</tr>
</thead>
<tbody>
<tr>
<td>controls</td>
<td>Association:Rel.Attrib:From.Obj</td>
<td>DoorController</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>controls</td>
<td>Association:Rel.Attrib:To.Obj</td>
<td>InnerDoor</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>controls</td>
<td>Association:Rel.Attrib:From.Obj</td>
<td>DoorController</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>controls</td>
<td>Association:Rel.Attrib:To.Obj</td>
<td>OuterDoor</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>controls</td>
<td>Association:Rel.Attrib:From.Obj</td>
<td>ElevatorController</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>controls</td>
<td>Association:Rel.Attrib:To.Obj</td>
<td>ElevatorCar</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>requests</td>
<td>Association:Rel.Attrib:From.Obj</td>
<td>Passenger</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>requests</td>
<td>Association:Rel.Attrib:To.Obj</td>
<td>ElevatorCar</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>presses</td>
<td>Association:Rel.Attrib:From.Obj</td>
<td>Passenger</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>presses</td>
<td>Association:Rel.Attrib:To.Obj</td>
<td>Button</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>uses</td>
<td>Association:Rel.Attrib:From.Obj</td>
<td>ElevatorController</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>uses</td>
<td>Association:Rel.Attrib:To.Obj</td>
<td>Timer</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>communicates with</td>
<td>Association:Rel.Attrib:From.Obj</td>
<td>ElevatorController</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>communicates with</td>
<td>Association:Rel.Attrib:To.Obj</td>
<td>DoorController</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>unnamed</td>
<td>Nonassociation:General:Superclass</td>
<td>Door</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>unnamed</td>
<td>Nonassociation:General:Subclass</td>
<td>InnerDoor</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>unnamed</td>
<td>Nonassociation:General:Subclass</td>
<td>OuterDoor</td>
<td>none</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 11: MOA Behavior Classification

<table>
<thead>
<tr>
<th>UML Operation</th>
<th>Msg Type</th>
<th>Msg Parms</th>
<th>Sending Object / OSSD Classification</th>
<th>Receiving Object / OSSD Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>PressButton</td>
<td>A</td>
<td>BT, F</td>
<td>Passenger/Control</td>
<td>Floor/Perform</td>
</tr>
<tr>
<td>ButtonPressed</td>
<td>B</td>
<td>none</td>
<td>Floor/Perform</td>
<td>ElevatorController/Monitor</td>
</tr>
<tr>
<td>TurnLightOn</td>
<td>A</td>
<td>BT, F</td>
<td>ElevatorController/Control</td>
<td>Floor/Perform</td>
</tr>
<tr>
<td>UpdateDestination</td>
<td>A</td>
<td>none</td>
<td>ElevatorController/Perform</td>
<td>ElevatorController/Perform</td>
</tr>
<tr>
<td>GetNextDestination</td>
<td>A</td>
<td>F</td>
<td>ElevatorController/Perform</td>
<td>ElevatorController/Perform</td>
</tr>
<tr>
<td>TurnLightOff</td>
<td>A</td>
<td>BT, F</td>
<td>ElevatorController/Control</td>
<td>Floor/Perform</td>
</tr>
<tr>
<td>OpenDoors</td>
<td>A</td>
<td>none</td>
<td>ElevatorController/Control</td>
<td>DoorController/Perform</td>
</tr>
<tr>
<td>OpenOuterDoor</td>
<td>A</td>
<td>none</td>
<td>DoorController/Control</td>
<td>OuterDoor/Perform</td>
</tr>
<tr>
<td>OpenInnerDoor</td>
<td>A</td>
<td>none</td>
<td>DoorController/Control</td>
<td>InnerDoor/Perform</td>
</tr>
<tr>
<td>InnerDoorOpened</td>
<td>B</td>
<td>none</td>
<td>InnerDoor/Perform</td>
<td>DoorController/Monitor</td>
</tr>
<tr>
<td>OuterDoorOpened</td>
<td>B</td>
<td>none</td>
<td>OuterDoor/Perform</td>
<td>DoorController/Monitor</td>
</tr>
<tr>
<td>DoorsOpened</td>
<td>B</td>
<td>none</td>
<td>DoorController/Perform</td>
<td>ElevatorController/Monitor</td>
</tr>
<tr>
<td>Start</td>
<td>A</td>
<td>none</td>
<td>ElevatorController/Control</td>
<td>Timer/Perform</td>
</tr>
<tr>
<td>Stop</td>
<td>A</td>
<td>none</td>
<td>ElevatorController/Control</td>
<td>Timer/Perform</td>
</tr>
<tr>
<td>EnterElevator</td>
<td>A</td>
<td>none</td>
<td>ElevatorController/Control</td>
<td>Passenger/Perform</td>
</tr>
<tr>
<td>TimeOut</td>
<td>A</td>
<td>none</td>
<td>Timer/Perform</td>
<td>ElevatorController/Perform</td>
</tr>
<tr>
<td>CloseDoors</td>
<td>A</td>
<td>none</td>
<td>ElevatorController/Control</td>
<td>DoorController/Perform</td>
</tr>
<tr>
<td>CloseOuterDoor</td>
<td>A</td>
<td>none</td>
<td>DoorController/Control</td>
<td>OuterDoor/Perform</td>
</tr>
<tr>
<td>CloseInnerDoor</td>
<td>A</td>
<td>none</td>
<td>DoorController/Control</td>
<td>InnerDoor/Perform</td>
</tr>
<tr>
<td>InnerDoorClosed</td>
<td>B</td>
<td>none</td>
<td>InnerDoor/Perform</td>
<td>DoorController/Monitor</td>
</tr>
<tr>
<td>OuterDoorClosed</td>
<td>B</td>
<td>none</td>
<td>OuterDoor/Perform</td>
<td>DoorController/Monitor</td>
</tr>
<tr>
<td>DoorsClosed</td>
<td>B</td>
<td>none</td>
<td>DoorController/Perform</td>
<td>ElevatorController/Monitor</td>
</tr>
</tbody>
</table>
### Table 12: MOA Object Classification

<table>
<thead>
<tr>
<th>UML Class</th>
<th>UML Compos.</th>
<th>SUMO/WordNet Classification</th>
<th>OSSD Behavior</th>
<th>OSSD Classif</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElevatorController</td>
<td>Whole top level</td>
<td>Entity:Physical:Object</td>
<td>Control, Monitor, Perform</td>
<td>Agent</td>
</tr>
<tr>
<td>DoorController</td>
<td>Whole top level</td>
<td>Entity:Physical:Object</td>
<td>Control, Monitor, Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>ElevatorCar</td>
<td>Whole top level</td>
<td>Entity:Physical:Object</td>
<td>Control, Monitor, Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>ButtonPanel</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>WeightSensor</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>Motor</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>InnerDoor</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>Door</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>DoorSensor</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>OuterDoor</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>Passenger</td>
<td>Whole top level</td>
<td>Entity:Abstract:Attribute:RelationalAttribute:SocialRole</td>
<td>Perform</td>
<td>Agent</td>
</tr>
</tbody>
</table>

### Table 13a: MOA State, Transition, Constraints Classification Table Part 1

<table>
<thead>
<tr>
<th>UML Class</th>
<th>UML State</th>
<th>OSSD State Classif.</th>
<th>OSSD Entry Behavior</th>
<th>OSSD Exit Behavior</th>
<th>OSSD Incoming Transition/From</th>
<th>OSSD Outgoing Transition/To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Door Controller</td>
<td>Elevator Idle</td>
<td>Initial</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Door Controller</td>
<td>Destination Request</td>
<td>Intermd</td>
<td>TurnLight On(BT,F)</td>
<td>Update Destination()</td>
<td>GetNext Destination()</td>
<td>ElevatorIdle</td>
</tr>
<tr>
<td>Door Controller</td>
<td>Waiting</td>
<td>Intermd</td>
<td>Start Timer()</td>
<td>None</td>
<td>GetNext Destination()</td>
<td>ElevatorAt Floor</td>
</tr>
<tr>
<td>Door Controller</td>
<td>Preparing To Move</td>
<td>Intermd</td>
<td>none</td>
<td>Close Doors()</td>
<td>none</td>
<td>ElevatorIdle, Waiting, Destination Request</td>
</tr>
<tr>
<td>Door Controller</td>
<td>Elevator Moving</td>
<td>Intermd</td>
<td>none</td>
<td>MoveTo (F)</td>
<td>none</td>
<td>PreparingTo Move</td>
</tr>
<tr>
<td>Door Controller</td>
<td>Elevator AtFloor</td>
<td>Intermd</td>
<td>TurnLight Off(BT,F)</td>
<td>none</td>
<td>Open Doors()</td>
<td>Elevator Moving, PreparingTo Move</td>
</tr>
<tr>
<td>Door Controller</td>
<td>Doors Closed</td>
<td>Initial</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Door Controller</td>
<td>Opening Doors</td>
<td>Intermd</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>DoorsClosed, DoorsOpen, DoorsOpen</td>
</tr>
<tr>
<td>Door Controller</td>
<td>Doors Open</td>
<td>Intermd</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>DoorsOpen</td>
</tr>
<tr>
<td>Door Controller</td>
<td>Closing Doors</td>
<td>Intermd</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>DoorsOpen, OpeningDoors</td>
</tr>
</tbody>
</table>
### Table 13b: MOA State, Transition, Constraints Classification Table Part 2

<table>
<thead>
<tr>
<th>UML Class</th>
<th>OSSD Transition Number</th>
<th>OSSD Transition IncomingFrom/OutgoingTo</th>
<th>OSSD Constraint</th>
<th>OSSD Constraint Classification</th>
<th>OSSD Transition followed by OSSD Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Door Controller 1</td>
<td>1</td>
<td>ElevatorIdle/DesinationRequest</td>
<td>ButtonPressed(BT,F), [BT=FloorButton V BT=DownButton V BT=UpButton]</td>
<td>Trigger, Guard</td>
<td>4</td>
</tr>
<tr>
<td>Door Controller 2</td>
<td>2</td>
<td>ElevatorIdle/PreparingToMove</td>
<td>ButtonPressed(BT,F), [BT=CloseButton]</td>
<td>Trigger, Guard</td>
<td>5 or 6</td>
</tr>
<tr>
<td>Door Controller 3</td>
<td>3</td>
<td>ElevatorIdle/ElevatorAtFloor</td>
<td>ButtonPressed(BT,F), [BT=OpenButton]</td>
<td>Trigger, Guard</td>
<td>8</td>
</tr>
<tr>
<td>Door Controller 4</td>
<td>4</td>
<td>DestinationRequest/PreparingToMove</td>
<td>none</td>
<td>none</td>
<td>5 or 6</td>
</tr>
<tr>
<td>Door Controller 5</td>
<td>5</td>
<td>PreparingToMove/ElevatorMoving</td>
<td>DoorsClosed() [F &lt;&gt; ElevatorLocation]</td>
<td>Trigger, Guard</td>
<td>7</td>
</tr>
<tr>
<td>Door Controller 6</td>
<td>6</td>
<td>PreparingToMove/ElevatorAtFloor</td>
<td>DoorsClosed() [F = ElevatorLocation]</td>
<td>Trigger, Guard</td>
<td>8</td>
</tr>
<tr>
<td>Door Controller 7</td>
<td>7</td>
<td>ElevatorMoving/ElevatorAtFloor</td>
<td>ElevatorArrived(F)</td>
<td>Trigger</td>
<td>8</td>
</tr>
<tr>
<td>Door Controller 8</td>
<td>8</td>
<td>ElevatorAtFloor/Waiting</td>
<td>DoorsOpened()</td>
<td>Trigger</td>
<td>9 or 10</td>
</tr>
<tr>
<td>Door Controller 9</td>
<td>9</td>
<td>Waiting/PreparingToMove</td>
<td>TimedOut(), [F &lt;&gt; null]</td>
<td>Trigger, Guard</td>
<td>5 or 6</td>
</tr>
<tr>
<td>Door Controller 10</td>
<td>10</td>
<td>Waiting/ElevatorIdle</td>
<td>TimedOut(), [F = null]</td>
<td>Trigger, Guard</td>
<td>1 or 2 or 3</td>
</tr>
</tbody>
</table>

### Table 14: MOA Goal Classification

<table>
<thead>
<tr>
<th>UML Use CaseName</th>
<th>OSSD Agent(s)</th>
<th>OSSD Entity</th>
<th>OSSD Behavior</th>
<th>OSSD Goal</th>
<th>OSSD Goal Classif.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Elevator</td>
<td>Passenger</td>
<td>Floor, CallButton UpButton</td>
<td>PressButton</td>
<td>Elevator Request</td>
<td>Achieve</td>
</tr>
<tr>
<td>Request Elevator</td>
<td>Elevator</td>
<td>Floor, CallButton UpButton, Light</td>
<td>TurnLightOn</td>
<td>Elevator Request</td>
<td>Achieve</td>
</tr>
<tr>
<td>Request Elevator</td>
<td>Elevator</td>
<td>ElevatorCar</td>
<td>See sub goal</td>
<td>Elevator Movement</td>
<td>Achieve</td>
</tr>
<tr>
<td>Request Elevator</td>
<td>Elevator</td>
<td>Floor, CallButton UpButton, Light</td>
<td>TurnLightOff</td>
<td>Elevator Request</td>
<td>Achieve</td>
</tr>
<tr>
<td>OpenDoors</td>
<td>Elevator</td>
<td>DoorController</td>
<td>OpenDoors</td>
<td>DoorsOpened</td>
<td>Achieve</td>
</tr>
<tr>
<td>OpenDoors</td>
<td>none</td>
<td>DoorController, InnerDoor</td>
<td>OpenInnerDoor</td>
<td>DoorsOpened</td>
<td>Achieve</td>
</tr>
<tr>
<td>OpenDoors</td>
<td>none</td>
<td>DoorController, OuterDoor</td>
<td>OpenOuterDoor</td>
<td>DoorsOpened</td>
<td>Achieve</td>
</tr>
<tr>
<td>Move Elevator</td>
<td>Elevator</td>
<td>ElevatorCar</td>
<td>MoveTo</td>
<td>Elevator Movement</td>
<td>Achieve</td>
</tr>
<tr>
<td>Move Elevator</td>
<td>none</td>
<td>ElevatorCar, Motor</td>
<td>MoveUp</td>
<td>Elevator Movement</td>
<td>Achieve</td>
</tr>
<tr>
<td>Move Elevator</td>
<td>none</td>
<td>ElevatorCar, Motor</td>
<td>MoveDown</td>
<td>Elevator Movement</td>
<td>Achieve</td>
</tr>
<tr>
<td>Move Elevator</td>
<td>none</td>
<td>FloorSensor</td>
<td>ElevatorArrived</td>
<td>Elevator Movement</td>
<td>Achieve</td>
</tr>
<tr>
<td>Move Elevator</td>
<td>none</td>
<td>ElevatorCar, Motor</td>
<td>Stop</td>
<td>Elevator Movement</td>
<td>Achieve</td>
</tr>
</tbody>
</table>
Table 15: Inter-View Inconsistency Detection

<table>
<thead>
<tr>
<th>UML Class</th>
<th>OSSD</th>
<th>Class Diagram</th>
<th>Sequence Diagram</th>
<th>State Machine Diagram</th>
<th>Use Cases or Use Case Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElevatorCar</td>
<td>Entity</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Passenger</td>
<td>Agent</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>OpenDoors</td>
<td>Behavior</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ButtonPressed</td>
<td>Behavior</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Controls</td>
<td>Association</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>WeightSensor</td>
<td>Entity</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

5.3.2 OSSD Model

5.3.2.1 OSSD Model of the Elevator System

Figure 47 shows a partial view of the OSSD Model created for the elevator system described in Figures 40 to 46. Instances are attached to the leaves of the OSSD Model via a double-headed arrow and are enclosed in double quotation marks. Each element from the UML diagram is represented as an instance in the OSSD Model. Each OSSD Model element is suffixed by an integer that is incremented for each UML element processed. For example, in the OSSD Model in Figure 47, the “ElevatorController” from the UML Class diagram is represented as an Agent [Construct0, Object0, Statebased0, Agent0], which has the Association “controls” [Construct1, Relation1, Association1], which connects “ElevatorController” with “ElevatorCar” [Construct4, Object4, Statebased4, Entity4]. The numeric suffixes associated with the leaf names are assigned as the leaves are created and do not correspond directly with semantically related Constructs (e.g. Behavior0 may or may not be performed by Agent0).
Figure 48 shows a partial view of structural relationships between the elevator car and elevator doors in the Elevator System using the Elevator System example. Figure 49 shows a detailed expansion of the OSSD Model for the Elevator System described in the UML diagrams in Figures 40 to 46. To simplify the pictorial view of the OSSD Model for the Elevator System, these figures show only the significant classes and properties. Some super-classes and paths connecting upward to the Construct level are omitted to simplify the diagrams and ease their understanding.

5.3.2.2 OSSD Model in OWL Notation

Figure 50 provides examples of OWL notation for portions of the OSSD Model. It specifies the high level constructs using OWL as well as provides an example a rule specified in SWRL notation [SWRL]. This rule implements the OB_Rule1 given in Figure 28 in Chapter 4 Section 4.3.5.2. The disjunctive clause in the head of this rule is not standard SWRL; it is based on FOL RuleML [Boley et al.]
Figure 49: OSSD Model - Elevator System Detailed View
Figure 50: Partial OWL Notation for Elevator Case Study
5.3.3 Consistency Checking

The UML designs in Figures 40 through 46 contain the following seeded inconsistencies.

1) The class “weight sensor” exists in the Class Diagram but no reference to that class exists in either the StateMachine or Sequence Diagram.

2) The Sequence Diagram shows the Elevator Controller class exchanging messages with the Floor class but the Class Diagram does not show an association link between these classes.

3) The Door Controller StateMachine Diagram shows detection of an obstruction between the doors that prevents the doors from closing, but the Passenger Request Sequence Diagram that indicates the doors should close after a timeout has occurred without any indication of detection of an obstruction between the doors.

Consistency checking performed during the transformation to the OSSD Model detects the first two inconsistencies. The first inconsistency involving the “weight sensor” is detected via the IC_Rule1 (see Figure 29) and the Inter-View Inconsistency Table shown in Table 15 in Chapter 5 Section 5.3.1 which shows the UML Class “Weight Sensor” is identified in the Class and Use Case diagrams but not in the Sequence or StateMachine diagrams. The second inconsistency concerning the missing association link is detected via two axioms OB-1 and OR-1 (based on the OB_Rule1 and OR_Rule1 axioms given in Figure 28 in Chapter 4 Section 4.3.5.2). These axioms are shown in Figure 51 located on the property has linking State-Based (representing “ElevatorController”) to Behavior (representing “TurnLightOn”). Although this diagram shows that axiom OB-1 is true it shows that axiom OR-1 is not true thereby identifying the inconsistency. Figure 52 shows that “ElevatorController” participates in only three Associations involving the “ElevatorCar”, "DoorController", and “Timer”. No Association exists between “ElevatorController” and “Floor”. Figure 52 also shows that both axiom OB-1 and OR-1 are true for the message sent between the “ElevatorController” and the “ElevatorCar” to perform the “MoveTo” behavior.

The third inconsistency in the Elevator system is detected during the KAOS processing of the Elevator System specification. This inconsistency is an example of a divergence that is easily detected in the KAOS processing. The KAOS processing will identify from the KAOS specification of the Elevator System the two assertions (1) elevator doors should close after a given timeout period and (2) elevator doors should not close if the door sensor detects an obstruction. Then the KAOS system will identify the boundary condition, “timeout” and “obstruction detected” which results in a divergence. In KAOS processing, boundary conditions can be “formally derived by regressing the negation of one of the goal assertions through the domain theory extended with the other goal assertions” [van Lamsweerde7].

5.4 KAOS

Figure 52 contains Sections of a KAOS specification for the Elevator System that is generated by the transformation.

Figure 53, adapted from [Objectiver1], shows an example of a generic KAOS goal pattern for a “system satisfying stakeholder’s needs” which, when applied to the Elevator System, will produce an initial goal pattern shown in Figure 54, also adapted from [Objectiver1]. Additional reiterations and expansions of the KAOS goal patterns will facilitate the KAOS identification of conflicts and goals as well as inconsistencies.
The KAOS specification is transformed into XML format in order for it to be accessible to the FAUST Toolbox. Transformation into XML is beyond the scope of this research. Results of the verification processing would then be used to manually update the original UML design.

The third inconsistency residing within the UML design is an example of a divergence that is easily detected in the KAOS processing. The KAOS processing will identify from the KAOS specification of the Elevator System the two assertions (1) elevator doors should close after a given timeout period and (2) elevator doors should not close if the door sensor detects an obstruction. Then the KAOS system will identify the boundary condition, “timeout” and “obstruction detected” which results in a divergence. In KAOS processing, boundary conditions can be “formally derived by regressing the negation of one of the goal assertions through the domain theory extended with the other goal assertions” [van Lamsweerde7].

Figure 51: Relation Inconsistency
<table>
<thead>
<tr>
<th>Agent ElevatorController</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hs</strong></td>
</tr>
<tr>
<td>ElevatorDestinationList,</td>
</tr>
<tr>
<td>ElevatorDirection,</td>
</tr>
<tr>
<td>CurrentLocation</td>
</tr>
<tr>
<td><strong>Inherited from</strong></td>
</tr>
<tr>
<td>none</td>
</tr>
<tr>
<td><strong>Monitors</strong></td>
</tr>
<tr>
<td>DoorController/DoorsState</td>
</tr>
<tr>
<td><strong>Controls</strong></td>
</tr>
<tr>
<td>ElevatorCar/Location, State</td>
</tr>
<tr>
<td><strong>ResponsibleFor</strong></td>
</tr>
<tr>
<td>ElevatorMovement, WeightLimitSafety</td>
</tr>
<tr>
<td><strong>DependsOn</strong></td>
</tr>
<tr>
<td>DoorController For DoorsOpened, DoorsClosed, DoorSafety</td>
</tr>
<tr>
<td><strong>Performs</strong></td>
</tr>
<tr>
<td>UpdateDestinationList, GetNextDestination, StartTimer, StopTimer</td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entity ElevatorCar</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Has</strong></td>
</tr>
<tr>
<td>State, Location, Moving</td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event ElevatorArrived(Location)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Has</strong></td>
</tr>
<tr>
<td>Location</td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Association Controls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Links</strong> ElevatorController {mult 1..1}, ElevatorCar {mult 1..1}</td>
</tr>
<tr>
<td><strong>Has</strong> none</td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal Maintain [DoorsClosedWhileMoving]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Concerns</strong> ElevatorCar/InnerDoor, Floor/OuterDoor</td>
</tr>
<tr>
<td><strong>AndRefines</strong> DoorSafety</td>
</tr>
<tr>
<td><strong>UnderResponsibilityOf</strong> ElevatorController</td>
</tr>
<tr>
<td><strong>Operationalizes</strong> ElevatorCarOperation</td>
</tr>
<tr>
<td><strong>OperationalizedBy</strong> OpenDoors, CloseDoors, MoveTo</td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation OpenDoors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong> e:ElevatorCar, id:InnerDoor, od:OuterDoor</td>
</tr>
<tr>
<td><strong>Output</strong> e: ElevatorCar/State, id: InnerDoor/State, od: OuterDoor/State</td>
</tr>
<tr>
<td><strong>DomPre</strong> e.State=&quot;stopped&quot;, id.State = &quot;closed&quot;, od.State = &quot;closed&quot;</td>
</tr>
<tr>
<td><strong>DomPost</strong> e.state=&quot;stopped&quot;, id.State = &quot;open&quot;, od.State = &quot;open&quot;</td>
</tr>
<tr>
<td><strong>ReqPre</strong> DoorsClosedWhileMoving: ¬e.moving</td>
</tr>
<tr>
<td><strong>ReqTrig</strong> DoorSafety: AtFloor=true ^ ¬e.moving</td>
</tr>
<tr>
<td><strong>CausedBy</strong> ElevatorArrived</td>
</tr>
<tr>
<td><strong>PerformedBy</strong> DoorController</td>
</tr>
<tr>
<td><strong>Operationalizes</strong> DoorOperation</td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation MoveTo</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong> e:ElevatorCar, f:FloorButton</td>
</tr>
<tr>
<td><strong>Output</strong> e: ElevatorCar/Location</td>
</tr>
<tr>
<td><strong>DomPre</strong> e.Location &lt;&gt; f.Floor, e.State = &quot;doors closed&quot;</td>
</tr>
<tr>
<td><strong>DomPost</strong> e.Location = f.Floor, e.State = &quot;doors closed&quot;</td>
</tr>
<tr>
<td><strong>ReqPre</strong> DoorsClosedWhileMoving</td>
</tr>
<tr>
<td><strong>CausedBy</strong> ee.ButtonPressed(Floor,f)</td>
</tr>
<tr>
<td><strong>PerformedBy</strong> ElevatorCar</td>
</tr>
<tr>
<td><strong>Operationalizes</strong> ElevatorCarOperation</td>
</tr>
<tr>
<td><strong>End</strong></td>
</tr>
</tbody>
</table>

Figure 52: Partial KAOS Specification for Elevator System
transporation requests satisfied in a safe, efficient, usable and cheap way

Figure 54: Partial KAOS Goal Pattern for the Elevator System
6 Evaluation of Methodology

6.1 Introduction

We evaluated this research from three perspectives: evaluating the ontology represented as the OSSD Model; evaluating the error detection; and lastly, evaluating the transformation from the source language to target language, specifically UML to KAOS. No single evaluating technique or method addresses all three aspects; therefore, we address each aspect separately.

6.2 Ontological Evaluation of the OSSD Model

6.2.1 Introduction

Ontology development is slowly moving from an art to a science. The development of ontology evaluation methodologies is a significant factor in this progression. A variety of approaches are available to evaluate the quality of an ontology ranging from simply identifying typical problems encountered in taxonomic knowledge [Gomez-Perez], to ontological comparison with a generally agreed upon sound ontological model such as Bunge-Wand-Weber (BWW) Model [Wand & Weber2], to formal ontology evaluation methods such as OntoClean [Guarino & Welty], and to commercially available ontology evaluation support tools such as ODEval [Falbo2 et al.]. Recently, a meta-ontology approach, referred to simply as O² [Gangemi et al.], integrates several ontology evaluation methods and introduces a variety of ontology evaluation metrics. In this latter approach, ontologies are evaluated based on structure, functionality, and usability.

A recent survey of ontology evaluation techniques [Brank et al.] organizes ontological evaluation approaches into broad categories including methods that are based on comparison with a “golden standard” ontology, comparison with domain knowledge specific to the ontology, manual comparison against predefined standards, and empirical evaluation of the ontology. From a slightly different viewpoint, an approach to evaluate reference models (conceptual frameworks) organizes research methods into empirical and analytical perspectives [Fettke & Loos]. The analytical perspective is further sub-divided based on the quality criteria utilized, either ad-hoc (including metric-based, feature-based, and text-based evaluations) or theory-driven (including evaluations based on ontologies and meta-models). Evaluation of the OSSD Model utilizes the theory-driven analytical perspective (specifically ontology-based evaluation) in conjunction with a “golden standard” ontology.

We considered several approaches for evaluating the OSSD Model. The Gomez-Perez approach is useful but not complex enough to perform a full evaluation of an ontology. The O² evaluation method is a promising technique that provides numerous metrics but does not provide adequate information as to the interpretation of the results of applying such metrics; it lacks the range specification for each metric that is required to provide an understanding of the empirical data gathered by applying the metrics. Research implies that future versions will provide “patterns of good/bad quality based on correlation between success stories, user satisfaction feedback, and measures” [Gangemi et al.]. The OntoClean approach requires significant training [Hartmann et al.]. Insufficient information is publicly available to implement the ODEval method. The BWW Model, on the other hand, has both a wealth of information available regarding its application and has been used successfully to evaluate numerous modeling methods and modeling grammars (such as structured, data-centered, object-oriented, and process grammars) including the Entity-Relationship Model [Wand et al.], Reference Models [Fettke & Loos], Process Modeling Techniques [Rosemann1
6.2.2 BWW Model

An ontology developed by Bunge [Bunge] became the basis for the development of three ontological models (a representation model, a state-tracking model, and a good decomposition model) to evaluate information systems modeling techniques and grammars. The Bunge-Wand-Weber (BWW) Representation Model, hereafter referred to simply as the BWW Model, is the most commonly used of the three models to represent the structure and behavior of the real world. The state-tracking model analyzes the representation of dynamics from the real world while the good decomposition model evaluates the subsystem organization of a model. Ontological analysis utilizing the BWW Model is based on two types of mappings, representation mapping and interpretation mapping, as shown in Figure 55 (adapted from [Wand & Weber1]). With representation mapping, the BWW Model constructs are mapped onto the constructs of the grammar or modeling technique under evaluation (hereafter, referred to as the evaluated model). With interpretation mapping, the evaluated model constructs are mapped onto BWW Model constructs. As a result of these two mappings, it is possible to identify four potential weaknesses of the evaluated model [Fettke & Loos]. The representation mapping can reveal construct incompleteness if there exists one or more BWW Model construct that cannot be mapped to any construct in the evaluated model; construct redundancy (ambiguous mapping) occurs if there exists at least one BWW Model construct that can be mapped to multiple constructs in the evaluated model. The interpretation mapping can identify construct excess if there exists one or more evaluated model construct that cannot be mapped to any construct in the BWW Model; construct overload is revealed if there exists at least one evaluated model construct that can be mapped to multiple constructs in the BWW Model.

The BWW Model represents domain structure and behavior by defining approximately 50 ontological concepts. From a high-level, things represent the world (structural relationships between things portrayed via composite/component, class/kind) and own characteristics referred to as properties. Things are able to interact with each other within the systems in the environment.
according to *transformations* on *properties* that are affected into change by *events* based on *transformation laws*. Detailed explanations of the more commonly used BWW constructs are given in Table 16. The following descriptions of the BWW Model are based on a recent evaluation of

<table>
<thead>
<tr>
<th>BWW Construct</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thing(concrete/conceptual)</td>
<td>Elementary units in the real world (perceived modeled)</td>
</tr>
<tr>
<td>Primitive/Component Composite</td>
<td>Not divisible into other things / a thing that is part of a composite thing Divisible into two or more related primitive things</td>
</tr>
<tr>
<td>Property/Attribute</td>
<td>Characteristic belonging to a thing; can not be directly observed; modeled by a function; sub-types of properties include:</td>
</tr>
<tr>
<td>Intrinsic / Mutual (Relational)</td>
<td>belongs to a single, individual thing; inherent / belongs to two or more related things</td>
</tr>
<tr>
<td>Hereditary / Emergent</td>
<td>belongs to both composite and component thing / belongs only to a composite thing</td>
</tr>
<tr>
<td>InGeneral / InParticular</td>
<td>belongs to a group of things / belongs to an individual in a group of things</td>
</tr>
<tr>
<td>Property Function</td>
<td>“maps the thing into some value”; “represents how a property changes over time” “being incomposition of another thing or, complementary, of having another thing as a component”</td>
</tr>
<tr>
<td>Whole-part Relation</td>
<td>Two or more things that have a common property / Two or more things that have a common set of two or more properties</td>
</tr>
<tr>
<td>Class / Kind and Sub-Kind</td>
<td>“the vector of values for all property functions of a thing”</td>
</tr>
<tr>
<td>Conceivable State Space</td>
<td>“the set of all states that the thing might ever assume”</td>
</tr>
<tr>
<td>State Law</td>
<td>Property function value restriction “lawful because of natural laws or human laws”; is a property of a thing</td>
</tr>
<tr>
<td>Lawful State Space</td>
<td>“set of states of a thing that comply with the state laws of the thing”</td>
</tr>
<tr>
<td>Process</td>
<td>“ordered sequence of events on, or states of, a thing”</td>
</tr>
<tr>
<td>Event</td>
<td>State change “effected via a transformation”</td>
</tr>
<tr>
<td>Conceivable Event Space</td>
<td>“set of all possible events that can occur in the thing”</td>
</tr>
<tr>
<td>Transformation</td>
<td>“mapping from a domain comprising states to a codomain comprising states”; a mapping from one state to another</td>
</tr>
<tr>
<td>Lawful Transformation</td>
<td>“defines which events in a thing are lawful”; is a property of a thing; indicates transformations from lawful state to lawful state</td>
</tr>
<tr>
<td>Lawful Event Space</td>
<td>“set of all events in a thing that are lawful”</td>
</tr>
<tr>
<td>History</td>
<td>“chronologically ordered states that a thing traverses in time”</td>
</tr>
<tr>
<td>Acts on / Coupling</td>
<td>“a thing acts on another thing if its existence affects the history of the other thing”</td>
</tr>
<tr>
<td>System</td>
<td>A set of things in which “couplings exist among things in the two subsets”</td>
</tr>
<tr>
<td>System Composition</td>
<td>The component things in a system</td>
</tr>
<tr>
<td>System Environment</td>
<td>Things outside of the system that interact with things in the system</td>
</tr>
<tr>
<td>System Structure</td>
<td>“set of couplings that exist among things in the system and things in the environment of the system”</td>
</tr>
<tr>
<td>Subsystem</td>
<td>Subsets of a system</td>
</tr>
<tr>
<td>System Decomposition</td>
<td>Subsystem set totally inclusive within a system</td>
</tr>
<tr>
<td>Level Structure</td>
<td>“a partial order over the subsystems in a decomposition to show which subsystems are components of other subsystems or the system itself”</td>
</tr>
<tr>
<td>External Event / Internal Event</td>
<td>“an event that arises in a thing, subsystem or system by virtue of the action of something in the environment of the thing, subsystem or system. The before-state of an external event is always stable. The after-state may be stable or unstable” / “an event that arises in a thing, subsystem or system by virtue of the lawful transformations in the thing in the environment of the thing, subsystem or system. The before-state of an internal event is always unstable. The after-state may be stable or unstable”</td>
</tr>
<tr>
<td>Stable / Unstable State</td>
<td>“a state in which a thing, subsystem or system will remain unless forced to change by virtue of the action of a thing in the environment (an external event)” / “a state that will be changed into another state by virtue of the action of transformation in the system”</td>
</tr>
</tbody>
</table>
UML using the Bunge-Wand-Weber Model, which includes a synthesis of several sources describing the BWW Model [Opdahl & Henderson-Sellers2].

A review of the inter-relationships among the BWW Model concepts is beneficial before performing the BWW evaluation of the OSSD Model. Within the BWW Model, an object is either a concrete thing (something that is, or can be perceived by someone as, a specific object) or a conceptual thing (a model of a thing). A composite thing may contain one or component things. A thing possesses one or more properties. A property cannot exist without a thing. A property cannot possess other properties. A property of a concrete thing is also referred to as a substantial property while a property of a conceptual thing is also referred to as a formal property or attribute. As an example of this fine distinction, the color of a thing is an attribute that corresponds to the property reflection of a wavelength [Leppanen]. The complexity of the concept of a property function is clarified in the following manner: “In the BWW Model, an attribute (that stands for a BWW-property) is represented as a property function of time, which maps the property onto different property values” [Opdahl & Henderson-Sellers1]. In simpler terms, a property is modeled as an attribute. An attribute / property is characterized by three classifications Hereditary/Emergent, InGeneral/InParticular, and Intrinsic/Mutual. These classifications are not mutually exclusive of each other, for example, a property can be Hereditary and Intrinsic. Properties/attributes of a composite thing can be either hereditary (belonging to both the composite thing and the component things) or emergent (associated with the composite thing as a whole). An example of a simple emergent property would a sum of component parts. InGeneral/InParticular indicates belonging to a group as a whole or to only a specific member of a group. Intrinsic/Mutual imply belonging to only one thing or belonging to two or more things based on a relationship between those things. A kind is a collection of things that share two or more properties/attributes that are not shared by any thing outside of that collection. A class is a collection of things that all possess the same one property. Law and law statement are properties/attributes that restrict the property/attribute of a thing and specify property relationships. Properties/attributes that do not restrict other properties/attributes are referred to as value properties/attributes. An event causes the state of one or more properties of a thing; events can be internal if caused by a change in state of a thing as a result of a transformation law that applies to that same thing; external events are due to state changes of one thing caused by actions of a different thing; a transformation describes the change from one state to another state.

Few of the modeling methods and grammars evaluated via BWW Model produce a comprehensive one to one mapping with the BWW Model. Usually, mismatches identify weaknesses of the evaluated model, such as the BWW evaluation of UML to represent concrete problem domains [Opdahl & Henderson-Sellers2]. However, sometimes the mismatches reveal perceived problems with the BWW Model. Although critics of the BWW Model state that it lacks understandability, objectivity, guidance, and completeness [Rosemann2 et al.], and that analytical results of applying the BWW Model sometimes contradict conceptual modeling practice [Shanks], the overall process of evaluating a model using the BWW model is useful in refining, correcting, and justifying components of a model. It is this latter justification that enables developers of a model to prove why their model should be considered ontologically sound even if it does not map completely to the BWW Model. Additionally, the BWW Model is useful when combined with other ontologies, such as performed with the Workflow Management System to evaluate UML with regard to business to business workflows [Dussart et al.].
6.2.3 BWW Model Evaluation of the OSSD Model

A high-level view of the OSSD Model depicts the world (of software engineering requirements and design) as represented by instances of the OSSD Construct Object {Agent, Entity, and Event}. These OSSD Model instances own characteristics that are represented by Construct:Attribute. OSSD Model Agents and Entities interact with each other according to the Construct:Behavior that affects the states of the Construct:Attribute(s) that in turn cause Construct:Object:Events based on Construct:State, Construct:Transition, Construct:Constraint, Construct:Plan, Construct:Goal, and on the axioms associated with those properties between these Constructs. As described earlier when introducing the OSSD Model, properties within the OSSD Model depict both structural and behavior relationships between OSSD constructs. Sections 6.2.3.1 and 6.2.3.2 provide the results of performing a BWW Model Representation evaluation of the OSSD Model and the results of performing a BWW Model Interpretation evaluation of the OSSD Model.

6.2.3.1 Representation Mapping Evaluation

Table 17 contains the detailed results of the BWW Model Representation Mapping of the OSSD Model. The correspondence between the BWW Model and the OSSD Model is based on similar analysis performed to identify the correspondence between UML and the BWW Model [Opdahl & Henderson-Sellers2], [Dussart et al.] [Evermann & Wand1].

Table 18 contains a high-level summary of the representation mapping with only the construct incompleteness and construct redundancy errors listed. If no incompleteness or redundancy exists, the table entry is filled with dashes rather than textual comments to improve readability of the table. An analytical discussion comparing the evaluation results of the OSSD Model follows the tables. The representation mapping of the BWW Model constructs onto the constructs of the OSSD Model reveals potential construct incompleteness if there exists one or more BWW Model construct that cannot be mapped to any construct in the OSSD Model, and construct redundancy (ambiguous mapping) if there exists at least one BWW Model construct that can be mapped to multiple constructs in the OSSD Model.

With regard to construct incompleteness, the BWW Model analysis shows that the OSSD Model is ontologically complete given the scope restrictions of the OSSD Model. Eight BWW Model constructs that relate to the BWW concept of system composition/decomposition and environment cannot be mapped to OSSD Model constructs because the scope of the BWW ontology (the world) is considerably larger than the scope of the OSSD Model (software requirements specification and design). Additionally, this research narrows the scope of software requirements specification and design to include object definition and behavioral interaction but excludes system composition/decomposition and environment. Research related to OSSD, the adaptation of the BWW Model to the Off-the-Shelf Information Systems (OISR) Framework [Soffer et al], similarly narrows the scope of the BWW Model evaluation. With regard to its importance, a recent ontological analysis of process modeling techniques utilizing the BWW Model shows that while 58% of the techniques support the system concept only 17% support the subsystem and environment BWW constructs [Rosemann1 et al.]; additionally, the most commonly supported ontological constructs in these process modeling techniques include transformation, property, event, lawful transformation, coupling, state, system, external event, well-defined event, class, and thing. With regard to these commonly supported ontological constructs, the OSSD Model supports all but the external event.
With regard to **construct redundancy**, three BWW Model constructs can be mapped to multiple constructs in the OSSD Model. Construct redundancy is not a significant problem when “the

<table>
<thead>
<tr>
<th>BWW Construct</th>
<th>UML</th>
<th>OSSD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thing</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primitive</td>
<td>Object, Actor</td>
<td><strong>Construct:Object:</strong> <strong>State-based:</strong> {Agent or Entity}</td>
</tr>
<tr>
<td>(Component)</td>
<td>/ Aggregate object</td>
<td>An Instance of a Construct:Object that does have associated with it Construct:Relation:Non-Association:Composition:Part or Construct:Relation:Non-Association:Aggregation:Part</td>
</tr>
<tr>
<td>/ Composite</td>
<td></td>
<td>/ An Instance of an Construct:Object that does have associated with it Construct:Relation:Non-Association:Composition:Whole, or Construct:Relation:Non-Association:Aggregation:Whole</td>
</tr>
<tr>
<td><strong>Property:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intrinsic</td>
<td>Attribute, Property</td>
<td><strong>Construct:Attribute</strong></td>
</tr>
<tr>
<td>/ Mutual</td>
<td>/ Association</td>
<td><strong>Construct:Attribute:</strong> <strong>RelationAttribute</strong></td>
</tr>
<tr>
<td>Hereditary</td>
<td>No match</td>
<td>Attribute assoc. with Composite Thing that is also in its Component Things</td>
</tr>
<tr>
<td>/ Emergent</td>
<td>/ No match</td>
<td>/Attribute assoc. with Composite Thing that is not in its Component Things</td>
</tr>
<tr>
<td>InGeneral</td>
<td>No match</td>
<td>Attribute assoc. with all instances of a Construct</td>
</tr>
<tr>
<td>/ InParticular</td>
<td>/ No match</td>
<td>/ Attribute assoc. with one instance of a Construct</td>
</tr>
<tr>
<td>Whole-Part Relation</td>
<td>aggregation</td>
<td><strong>Construct:Relation:NonAssociation:</strong> Generalization: {Superclass,Sub-class}</td>
</tr>
<tr>
<td><strong>Class / Kind and Sub-Kind</strong></td>
<td>Class (stereotype), Generalization</td>
<td>Two or more instances of a Construct:Object that have only one common Construct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Two or more instances of a Construct:Object that have the relationship Construct:Relation:NonAssociation: Generalization: {Superclass,Sub-class}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All Construct:States associated with Construct:Object:Statebased</td>
</tr>
<tr>
<td><strong>State</strong></td>
<td>State</td>
<td><strong>Construct:State:</strong> {Initial, Intermediate, or Final}</td>
</tr>
<tr>
<td><strong>ConceivableStateSpace</strong></td>
<td>StateMachine</td>
<td>All Construct:States associated with Construct:Object:Statebased</td>
</tr>
<tr>
<td><strong>State Law</strong></td>
<td>Precondition, Guard, Multiplicity</td>
<td>Construct:Constraint:Precondition, Construct:Constraint:Guard</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Construct:attribute:Multiplicity, and Construct:Goal: {Achieve, Maintain, Cease, Avoid}</td>
</tr>
<tr>
<td><strong>Lawful State Space</strong></td>
<td>No match</td>
<td>All Construct:States with related Construct:Transitions and Construct:Constraints associated with Construct:Object:Statebased</td>
</tr>
<tr>
<td><strong>Process</strong></td>
<td>Use Case</td>
<td><strong>Construct:Plan</strong></td>
</tr>
<tr>
<td><strong>Event</strong></td>
<td>Event</td>
<td><strong>Construct:Object:</strong> <strong>Event</strong></td>
</tr>
<tr>
<td><strong>ConceivableEventSpace</strong></td>
<td>No match</td>
<td>All Construct:Object:Events associated with Construct:Object:Statebased that are a caused by associated Construct:Behavior</td>
</tr>
<tr>
<td><strong>Transformation</strong></td>
<td>Operation, Activity</td>
<td><strong>Construct:Behavior:</strong> {Perform, Monitor, Control}</td>
</tr>
<tr>
<td><strong>Lawful Transformation</strong></td>
<td>Transition, Action, Postcondition</td>
<td>Construct:Transition, Construct:Constraint:Action</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Construct:Constraint:Postcondition, Construct:Constraint:Trigger</td>
</tr>
<tr>
<td><strong>Lawful Event Space</strong></td>
<td>No match</td>
<td>All Construct:Object:Events associated with Construct:Object:Statebased that are a caused by associated Construct:Behavior and constrained by Construct:Constraint</td>
</tr>
<tr>
<td><strong>History</strong></td>
<td>Object Lifeline</td>
<td>All Construct:States of a Thing associated with all Construct:Plan</td>
</tr>
<tr>
<td><strong>System</strong></td>
<td>Composite</td>
<td>All instances of Construct:Object that are related via Construct:Relation:Association</td>
</tr>
</tbody>
</table>

*table continued*
overlapping modeling constructs represent disjunctive subtypes of the ontological concepts [Opdahl & Henderson-Sellers2]. Therefore, construct redundancy due to disjunctive subtypes is not listed in Table16 for the mapping of a BWW Thing, Property, State, Behavior, or ActsOn/Coupling. A BWW Thing can be mapped to Construct:Object:Statebased:Agent or Construct:Object:Statebased:Entity. However, this construct redundancy is not significant because it is a result of disjunctive subtypes of the Construct:Object:Statebased. The distinction between a Primitive Thing and a Composite Thing depends on Construct:Relation:Non-Association:{Composition, Aggregation}:{Part, Whole} associated with the Object that in turn indicates the BWW Whole-Part relation property. A BWW Property can be mapped to Construct:Attribute:ObjectAttribute or Construct:Attribute:RelationAttribute. However, this construct redundancy is not significant because it is a result of disjunctive subtypes of the Construct:Attribute. The distinction between an Intrinsic Property and a Mutual Property is dependent on whether the Property belongs to a single, individual Thing or to two more related Things. The characteristics of a BWW Property (Hereditary/Emergent, InGeneral/InParticular, Whole-Part Relation) further describe the Property and are not actually a direct part of the mapping of the BWW Property. The mapping of the BWW State, Transformation, ActsOn/Coupling each map to sub-types of the OSSD Model Construct:State, Construct:Behavior, and Construct:Attribute:RelationAttribute respectively. To reiterate, this redundancy is not significant because it is a result of the disjunctive subtypes of the OSSD Model constructs.

The significant construct redundancy concerns three BWW Model constructs (StateLaw, LawfulTransformation, and UnstableState) that can be mapped to multiple constructs in the OSSD Model.

A BWW State Law can be mapped to four OSSD Constructs. Of these four, Construct:Constraint:Precondition and Construct:Constraint:Guard are a logically correct mapping because both logically restrict the values of Attributes before entering a given State. The two remaining OSSD Constructs pose an interesting dilemma. The OSSD Construct Construct:Attribute:Multiplicity logically restricts the occurrences of an Attribute (and therefore is a State Law by definition) but this restriction is independent of any State that the Object associated with the Attribute is currently in. Therefore, it should not be moved in the OSSD Model to Constraint and should remain with Attribute. We base the mapping of BWW State Law to OSSD Construct:Goal on considerable analysis performed with regard to goals that justifies its mapping based on the understanding that “goals are used to express constraints on the possible states a thing can be in” [Heymans et al.].

Lawful Transformation can be mapped to four OSSD Constructs. Construct:Constraint:Postcondition, Construct:Constraint:Trigger, and Construct:Constraint:Action are a logically correct mapping because all logically restrict the Behavior of Attributes. Postcondition restricts the Behavior expected within a given state by specifying the expected values of the Attributes after completion of a given State. Construct:Constraint:Trigger represents the Event that must occur, in conjunction with
the Guard and Precondition, in order for a Transition to fire. Similarly, Action is a Behavior that affects the value(s) of Attributes with the condition that no other Behavior can occur concerning the related Object until that Action completes. A Construct:Transition logically groups the restrictions on Behavior and Attribute values that must occur for an Object to transform from one State to another.

Unstable State can be mapped to three OSSD Constructs. The BWW Model makes the distinction between Stable States and Unstable States based on the occurrence of an External Event or Internal Event. Since the OSSD Model does not yet support the concepts of System Environment and Subsystem, it is not possible to make this distinction. The distinction of Construct:State:Initial, Construct:State:Intermediate, and Construct:State:Final is based on existence in the UML design. It could be removed if determined to be unnecessary after transformations to other models as deemed it so.

In summary, the above analysis of the BWW Model representation mapping of the OSSD Model shows that the OSSD Model is ontologically complete and non-redundant with regard to the most commonly used ontological constructs and within the narrowed scope of software specification and design. This analysis does reveal two ontological inadequacies in the OSSD Model that will be addressed its future development, specifically addressing system composition/decomposition and environment.

Table 18: Analysis Summary of the BWW Model Representation Mapping of OSSD

<table>
<thead>
<tr>
<th>BWW Construct</th>
<th>Incompleteness</th>
<th>Redundancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class / Kind</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conceivable State Space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State Law</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lawful State Space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conceivable Event Space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lawful Transformation</td>
<td></td>
<td>maps to 4 OSSD constructs</td>
</tr>
<tr>
<td>Lawful Event Space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>History</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acts On / Coupling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Composition</td>
<td>no match</td>
<td></td>
</tr>
<tr>
<td>System Environment</td>
<td>no match</td>
<td></td>
</tr>
<tr>
<td>System Structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subsystem</td>
<td>no match</td>
<td></td>
</tr>
<tr>
<td>SystemDecomposition</td>
<td>no match</td>
<td></td>
</tr>
<tr>
<td>Level Structure</td>
<td>no match</td>
<td></td>
</tr>
<tr>
<td>External Event / Internal Event</td>
<td>no match / no match</td>
<td>/</td>
</tr>
<tr>
<td>Stable / Unstable State</td>
<td>no match</td>
<td>/ maps to 3 OSSD constructs</td>
</tr>
</tbody>
</table>
### 6.2.3.2 Interpretation Mapping Evaluation

Table 19 contains the results of the BWW Model Interpretation Mapping of the OSSD Model. Again, the correspondence between the BWW Model and the OSSD Model is based on similar analysis performed to identify the correspondence between UML and the BWW Model [Opdahl & Henderson-Sellers2], [Dussart et al.] [Evermann & Wand1]. In most cases, the OSSD Model

<table>
<thead>
<tr>
<th>OSSD Construct</th>
<th>BWW Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Construct:Object</strong></td>
<td></td>
</tr>
<tr>
<td>Two or more instances of <code>Construct:Object:Statebased:[Agent, Entity]</code> that have only one common <code>Construct</code></td>
<td>Class</td>
</tr>
<tr>
<td>Two or more instances of <code>Construct:Object:Statebased:[Agent, Entity]</code> that have the relationship <code>Construct:Relation:Non-Association:Generalization:Superclass</code> and <code>Construct:Relation:Non-Association:Generalization:Subclass</code> respectively</td>
<td>Kind and sub-kind</td>
</tr>
<tr>
<td><strong>Construct:Object:Event</strong></td>
<td>Event</td>
</tr>
<tr>
<td>All <code>Construct:Object:Events</code> associated with <code>Construct:Object:Statebased</code> that are caused by associated <code>Construct:Behavior</code></td>
<td>ConceivableEventSpace</td>
</tr>
<tr>
<td>All <code>Construct:Object:Events</code> associated with <code>Construct:Object:Statebased</code> that are caused by associated <code>Construct:Behavior</code> and constrained by <code>Construct:Constraint</code></td>
<td>LawfulEventSpace</td>
</tr>
<tr>
<td><strong>Construct:Attribute</strong></td>
<td></td>
</tr>
<tr>
<td><code>Construct:Attribute:ObjectAttribute</code></td>
<td>Property:Intrinsic</td>
</tr>
<tr>
<td><code>Construct:Attribute:RelationAttribute</code></td>
<td>Property:Mutual</td>
</tr>
<tr>
<td><code>Construct:Attribute:RelationAttribute:ToObject</code></td>
<td>Acts On</td>
</tr>
<tr>
<td><code>Construct:Attribute:RelationAttribute:FromObject</code></td>
<td>Acts On</td>
</tr>
<tr>
<td><code>Construct:Attribute:Multiplicity</code></td>
<td>State Law</td>
</tr>
<tr>
<td><code>Construct:Attribute:Visibility</code></td>
<td>no match</td>
</tr>
<tr>
<td><code>Construct:Attribute:Role</code></td>
<td>no match</td>
</tr>
<tr>
<td><code>Attribute</code> assoc. with <code>Composite Thing</code> that is also in its <code>Component Things</code></td>
<td>Property:Hereditary</td>
</tr>
<tr>
<td><code>Attribute</code> assoc. with <code>Composite Thing</code> that is not in its <code>Component Things</code></td>
<td>Property:Emergent</td>
</tr>
<tr>
<td><code>Attribute</code> assoc. with all instances of a <code>Construct</code></td>
<td>Property:InGeneral</td>
</tr>
<tr>
<td><code>Attribute</code> assoc. with one instance of a <code>Construct</code></td>
<td>Property:InParticular</td>
</tr>
<tr>
<td><strong>Construct:Relation</strong></td>
<td></td>
</tr>
<tr>
<td><code>Construct:Relation:Association</code></td>
<td>SystemStructure</td>
</tr>
<tr>
<td>All instances of <code>Construct:Object</code> that are related via <code>Construct:Relation:Association</code></td>
<td>System</td>
</tr>
<tr>
<td><code>Construct:Relation:Non-Association:Generalization:Superclass</code></td>
<td>Kind</td>
</tr>
<tr>
<td><code>Construct:Relation:Non-Association:Generalization:Subclass</code></td>
<td>Sub-Kind</td>
</tr>
<tr>
<td><code>Construct:Relation:Non-Association:Aggregation:{Whole, Part}</code></td>
<td>Whole-part Relation</td>
</tr>
<tr>
<td><code>Construct:Relation:Non-Association:Composition:{Whole, Part}</code></td>
<td>Whole-part Relation</td>
</tr>
<tr>
<td><code>Construct:Goal:[:Achieve, Maintain, Cease, Avoid]</code></td>
<td>State Law</td>
</tr>
<tr>
<td><code>Construct:Behavior:[:Perform, Monitor, Control]</code></td>
<td>Transformation</td>
</tr>
<tr>
<td></td>
<td>table continued</td>
</tr>
</tbody>
</table>
elements listed in column one correspond to the significant upper-level OSSD Constructs (such as Object, Behavior, Goal, etc). Sub-levels of these OSSD Constructs (such Object:State-based:Agent) are not considered unique constructs that must be mapped to different BWW constructs. These sub-levels of these OSSD constructs are given in Table 19 only if an explicit mapping to a BWW Model construct must be identified.

Table 20 contains a high-level summary of the interpretation mapping with only the construct excess and construct overload errors listed. If no excess or overload exists, the table entry is filled with dashes rather than textual comments to improve readability of the table. An analysis comparing the evaluation results of the OSSD Model follows the tables. The interpretation mapping of the OSSD Model constructs onto the BWW Model constructs reveals construct excess if there exists one or more OSSD Model constructs that cannot be mapped to any construct in the BWW Model, and construct overload if there exists at least one OSSD Model construct that can be mapped to multiple constructs in the BWW Model.

It is possible to tolerate Construct excess in some circumstances. It is “only problematic if the construct is clearly intended (at least in part) to represent phenomena in or aspects of the problem domain, as opposed to, e.g., representing characteristics of the proposed software or information system [Opdahl & Henderson-Sellers2]”. The OSSD construct Visibility represents a characteristic of the source UML design and is not a significant feature of requirements specification. Therefore, this construct could be removed from the OSSD Model without significant loss to its purpose. However, the OSSD construct Role is a significant agent-oriented concept that is used to indicate capability (knowledge) and responsibility for specific tasks based on specific goals. Role is also used in UML design to name each end of an association. Therefore, the construct Role should be allowed to exist in the OSSD Model.

There are two other ontological concepts of the OSSD Model cannot be mapped directly into the BWW Model, the OSSD property and OSSD axiom. It is not possible to make the naïve mapping of OSSD property to BWW property because the OSSD property specifies a variety of relationships among OSSD Model Constructs whereas the BWW property specifies a fixed and very limited set of
characteristics and relationships between things. As already identified, only the OSSD Object:Statebased:{Agent,Entity} can be considered a BWW Thing, therefore the OSSD property can not be mapped directly to the BWW property. To force the OSSD property to be considered a BWW property would severely limit its usefulness in ontological freedom as well as force the ontological model to revolve solely around the OSSD Object, thereby pushing it toward an object-oriented representation rather than an ontological representation. Additionally, the OSSD property provides the basis for the inconsistency detection capabilities of the OSSD Model. Therefore, permitting construct excess with regard to the OSSD property is justified. A similar reasoning can be applied to the OSSD concept of axiom. It could naively be mapped to the BWW transformation law but should not for the same reasons as the OSSD property. Furthermore, the BWW Model actually does link its constructs implicitly in their textual descriptions producing a similar result as the OSSD properties and axioms. For example, in the BWW Model, a History is a set of states chronologically ordered that a thing traverses in time. “Chronologically ordered” is, in a sense, an axiom. Additionally, it is implied that a Thing “has” a History. However, the BWW Model does not specify

<table>
<thead>
<tr>
<th>OSSD Construct</th>
<th>Excess</th>
<th>Overload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct:Object</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Object:Statebased:{Agent,Entity}</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Object:Event</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Attribute</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Attribute:ObjectAttribute</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Attribute:RelationAttribute</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Attribute:RelationAttribute:ToObject</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Attribute:RelationAttribute:FromObject</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Attribute:Multiplicity</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Attribute:Visibility</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Attribute:Role</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Relation</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Relation:Association</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Relation:NonAssociation:Generalization:Superclass</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Relation:NonAssociation:Generalization:Subclass</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Relation:Non-Association:Aggregation:{Whole,Part}</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Relation:Non-Association:Composition:{Whole,Part}</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Goal:{Achieve, Maintain, Cease, Avoid}</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Behavior</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Constraint</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Constraint:Action</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Constraint:Guard</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Constraint:Precondition</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Constraint:Postcondition</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Constraint:Trigger</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:Transition</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:State</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:State:Initial</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:State:Intermediate</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Construct:State:Final</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Property</td>
<td>no match</td>
<td>----------</td>
</tr>
<tr>
<td>Axiom</td>
<td>no match</td>
<td>----------</td>
</tr>
</tbody>
</table>

Table 20: Analysis Summary of the BWW Model Interpretation Mapping of OSSD
a construct to model this axiomatic restriction or the “has” relationship. Therefore, the construct excess with regard to the OSSD Model property and axiom should be allowed.

With regard to **construct overload**, at first glance there appears to be significant construct overload because several OSSD Model constructs can be mapped to several BWW Model Constructs (e.g. *Construct:Constraint* can be mapped to either *Lawful Transformation* or *State Law* depending on the sub-type of *Constraint*). However, assuming the correlation of the statement that construct redundancy is not a significant problem when “the overlapping modeling constructs represent disjunctive subtypes of the ontological concepts [Opdahl & Henderson-Sellers2]”, then construct overload is not a significant problem if it occurs due to analysis of an intermediate level of the OSSD ontology and given that construct overload does not exist in the leaf-levels of that intermediate level.

In summary, the above analysis of the BWW Model interpretation mapping of the OSSD Model shows that the construct excess and overload existing in the OSSD Model is not problematic. The analysis shows that the OSSD Construct Visibility should be removed from the OSSD Model due to construct excess. However, the OSSD Construct Role should not be removed, even though it is deemed as construct excess, due to the importance of the concept of role to software design. Additionally, the OSSD Model concepts of *property* and *axiom* should be allowed to remain as defined due to their ontological importance in defining relationships among the OSSD Model constructs that provide the basis for its inconsistency detection capabilities.

### 6.3 Error Detection

#### 6.3.1 Introduction

Just as there are limitless design solutions for a given problem, there are limitless errors that can occur in any given design. Since the scope of error detection within MOA is limited to inconsistency errors, we focus on the specification of errors of consistency. Additionally, given that the software design and requirements specification addressed in this research are UML and KAOS respectively, the errors detected focus on UML design errors in general as well as errors detectable by a KAOS requirements engineering tool. Additionally, MOA adds value to the software design verification process by facilitating the identification and addition of error detection rules above and beyond that provided by the tools it interconnects.

It is possible to compile a list of commonly detected errors, as is performed by a UML CASE tool such as the Rose Model Checker [Moors]. However, these consistency checks are usually based on the well-formed rules (WFR) specified in the UML 2.0 Specification that address primarily the syntactic inconsistencies within a given UML diagram such as naming, visibility, and scope. The consistency checking within commercial tools based on UML remain limited [Kozlenkov & Zisman3]. Additionally, “there exist no general techniques for specifying semantic (and, in particular, behavioral) consistency constraints.” [Engels4 et al.]. Therefore, consistency rules gathered from a variety of other approaches to consistency management became axioms in the OSSD Model via axioms associated with the properties and the rules defined for the Inter-View Inconsistency Detection Table. Section 6.3.2 lists a subset of these consistency rules, Section 6.3.3 describes how they are incorporated into the OSSD Model, and Section 6.3.4 shows how intentionally seeded errors in the UML Case Study are detectable via these rules.
6.3.2 Representative Consistency Rules

The following consistency rules were selected randomly from the following sources: [Briand], [Kielland], and [Ohnishi]. Some of these rules are integrated into the OSSD Model in Chapter 4 Section 4.3.5.

The following rules were obtained from Rules from [Ohnishi]:
Each Actor in a Use Case Diagram should be associated with a Class in a Class Diagram;
Each UML Class should have a State Machine associated with it.
Each UML Class in a Class Diagram should be associated with at least one Object Lifeline in a Sequence diagram.

The following rules were obtained from [Briand]:
“Each object (in a sequence diagram) must be an instantiation of a class in a Class diagram”;
“For each message between two object (in a sequence diagram) there has to be a valid path (navigable) between them”;
“Each operation that is invoked in a state transition must be defined in a Class diagram”;
“A class cannot be a part in more than one composition”.

The following rules were obtained from [Kielland]:
Role names specified for an association must be unique within that association;
Attribute names specified in a given Class must be unique within that Class.

The following rules were obtained from [Quatrani]:
there exists a one to one correspondence between messages and behavior of a receiving class;
there exists either an association or aggregation between two interacting objects;
each class must participate in at least one scenario;
each operation specified in a class is used in at least one scenario;
each object specified in a sequence diagram is defined in a class in the class diagram;
each message in a sequence diagram is represented in a StateMachine diagram.

6.3.3 Representation of Consistency Rules

This Section describes the integration of the consistency rules given in Section 6.3.2 into the OSSD Model via axioms associated with the properties and the rules defined for the Inter-View Inconsistency Detection Table. Figure 56 describes these consistency rules, specified in first order predicate logic. The OSSD transformation tables described in Chapter 4 Section 4.3.4.2 contain the information to implement functions referenced in Figure 56. For example, the functions UML_Class and UML_Actor utilize information in the UML Class Element and POS Tagging table. The functions in-Class-Diagram and in-Sequence-Diagram utilize information in the Inter-view Inconsistency Detection table.

Each Actor in a Use Case Diagram should be associated with a Class in a Class Diagram
IC_Rule4 in Figure 56 represents this rule.

Each UML Class should have a State Machine associated with it
IC_Rule5 in Figure 56 represents this rule.
Each UML Class in a Class Diagram should be associated with at least one Object Lifeline in a Sequence diagram
IC_Rule6 in Figure 56 represents this rule.
Each object (in a sequence diagram) must be an instantiation of a class in a Class diagram
IC_Rule7 in Figure 56 represents this rule.

For each message between two objects (in a sequence diagram) there has to be a valid path (navigable) between them

IC_Rule7 in Figure 56 in conjunction with OB_Rule1 and OR_Rule1 in Figure 28 in Chapter 4 Section 4.3.5.3 represent this rule.

Each operation that is invoked in a state transition must be defined in a Class diagram
IC_Rule7 in Figure 56 represents this rule.

A class cannot be a part in more than one composition
This consistency rule would become an OO Consistency rule in the Consistency framework and added to OSSD Model by attaching the axiom shown as OO_Rule1 in Figure 56 to the property has that connects Construct:Object with Construct:Relation.

Role names specified for an association must be unique within that association
This consistency rule would become an AR Consistency rule in the Consistency framework and added to OSSD Model by attaching the axiom shown as AR_Rule1 in Figure 56 to the property has that connects Construct:Object with Construct:Relation:Association.

Attribute names specified in a given Class must be unique within that Class
This consistency rule would become an AA Consistency rule in the Consistency framework and added to OSSD Model by attaching the axiom shown as AA_Rule1 in Figure 56 to the property has that connects Construct:Object with Construct:ObjectAttribute.

6.3.4 Application of Consistency Rules

We used the methodology to perform the detection and diagnosis of consistency errors associated with the consistency rules given in Section 6.3 by intentionally violating a subset of these rules in the Elevator case study by first listing the rule, indicating what change to the UML diagrams in Chapter 5 must occur to violate the rule, and then showing how the rule violation would be detected in the Consistency checking. Details regarding Consistency checking are given in Chapter 5. Two other examples applications of Consistency detection are given in Chapter 6.

Error Detection #1 via Inter-View Inconsistency Detection Table

**Rules:** Each UML Class should have a State Machine associated with it; Each UML Class in a Class Diagram should be associated with at least one Object Lifeline in a Sequence diagram

**Violation:** The WeightSensor class is identified in UML Use Case and Class diagrams but not in either UML Sequence or StateMachine Diagrams

**Detection:** Section 5.3.3 shows how this violation is detected.

Error Detection #2 via Consistency framework Rules

**Rule:** Role names specified for an association must be unique within that association
Violation: Two role names are specified for the controls association between ElevatorController and ElevatorCar.

Detection: Figure 57 shows a partial representation of the OSSD Model that includes the Elevator Controller and Elevator Car with the axiom, AR-1, for this rule indicated. The rule would be executed during step A2-1 of the Consistency Checking Algorithm. Figure 57 shows that both Role names specified for RelationalAttributes associated with the Association “Controls” are not unique. Therefore, this inconsistency would be detected via the axiom, AR-1.

We demonstrated the error detection capabilities of MOA by integrating a variety of consistency rules into the OSSD Model via axioms associated with the properties and rules defined for the Inter-View Inconsistency Detection Table, and then intentionally seeding errors in a source design of the Elevator case study to show that the methodology presented in this research performs the appropriate error detection.

![Figure 57: Error Detection via Consistency framework Rules](image)

6.4 Transformation

6.4.1 Introduction

Ideally, software developers verify the transformation of one model to another in such a way that it proves the equivalence of the source and target models. Although one can verify the syntactic correctness of the target model via a simple parsing of the target specification produced as a result of the transformation, proving the semantic equivalence of the two models is not such a trivial task. “A common correctness criterion for translation systems is that they preserve semantics, i.e., the meaning of the source and the translation has to be the same. This is not necessarily desirable…since it should be perfectly admissible to perform abstractions or semantic shifts as part of the translation” [Chalupsky]. This is particularly true with regard to the model transformations between UML and the
OSSD Model, the OSSD Model and KAOS, and in effect, UML to KAOS. The target KAOS requirements level specification contains a subset of the functionality provided in the source UML design level specification. By definition, certain aspects of a software design are not required in a requirements level specification. Therefore, the task of verifying the transformation becomes considerably more difficult.

Since the transformation from UML to the OSSD Model to KAOS requires abstractions and semantic shifts, it is reasonable to expect the KAOS specification to be an abstraction of the UML design.

![Figure 58: Evaluation of Model Transformation](image)

However, there does not currently exist a model transformation technique that can handle verification of such abstractions. This research presents a unique methodology to model transformation evaluation, portrayed graphically in Figure 58, that can evaluate such abstractions and semantic shifts. This evaluation assumes that significant software design and requirements level concepts should be maintained throughout the model transformation from source model to target model. It evaluates the transformation from UML to OSSD Model to KAOS by showing that the set of semantically significant features of software requirements specification and design are represented in the UML Design, the OSSD Model, and the KAOS specification. Section 6.4.2 discusses the identification and determination of which features are selected for evaluation, and Section 6.4.3 presents an evaluation of the MOA transformations using the semantically significant feature set.

### 6.4.2 Set of Semantically Significant Features

We utilized IEEE Recommended Practices for Software Requirements Specifications (SRS) [IEEE2] and Software Design Descriptions (SDS) [IEEE3] to assist with the determination of which software requirements level and design level features to include in a set of semantically significant features used to evaluate model transformations. We organized these features in the familiar tri-view of system modeling popularized by the Object Modeling Technique [Rumbaugh et al.]: data (object structure and behavioral definitions), function (transformation of values and inter-dependencies of data), and control (event and state change of data with regard to sequence and time). We present the semantically significant feature set below preceded by the subSections of each of the IEEE recommendations from which it has been developed.

The second chapter, “Overall Description”, of the IEEE recommended SRS [IEEE2] describes the product perspective, product functions, user characteristics, constraints, assumptions and
dependencies. The product perspective “describes how the software operates inside various constraints” including system interfaces, user interfaces, hardware interfaces, software interfaces, communication interfaces, operations, memory and site adaptation requirements (the latter two issues not relevant for MOA analysis). The product functions chapter describes what the software will do and any “logical relationships among variables”. The user characteristics chapter describes the users of the proposed system to justify certain requirements (not relevant for MOA analysis). The constraints subSection includes hardware limitations, interfaces to other applications, parallel operations, control functions, signal handshake protocols, reliability requirements, criticality of the application, and safety and security considerations, and regulatory policies, audit functions, and higher-order language requirements (the latter three issues are not considered relevant for MOA analysis). Lastly, the assumptions and dependencies chapter gives potential changes that might affect the requirements that are not specifically design constraints (not considered relevant for MOA analysis).

Each requirement given in the IEEE SRS [IEEE2] includes: descriptions of external interfaces (including names, input/output, valid range/accuracy/tolerance, units of measure, timing, relationships to other inputs/outputs, and formatting for data and commands (formats for screens/windows and end messages are not considered relevant for MOA analysis)); functions (including input validity checks, operation sequence, abnormal situation responses, parameter effects, and input/output relationships); performance requirements (both static and dynamic); logical database requirements (not considered relevant for MOA analysis); and design constraints (including limitations of hardware and other standards requirements).

We present the semantically significant feature set (SSFS) in two stages of development to explicitly show its derivation from the IEEE standards. We formatted the first stage in an abbreviated textual format to show how material obtained from IEEE SRS recommendations (written in italics), material obtained from IEEE SDS recommendations (written in bold), and additional details added via this research that are above and beyond IEEE recommendations (written in bold italics) contribute to the development of the SSFS. The second stage gives the analytical format used throughout the remainder of the evaluation of transformation process. The organization of the SSFS builds on the familiar object/dynamic/functional models developed in the Object Modeling Technique (OMT) [Rumbaugh et al.] as a foundation but reorganizes and expands upon its concepts forming a new Representation/Behavior/Process (RBP) model. This RBP model is not intended for use as a design model but rather as a evaluation model. The RBP model organizes evaluation information based on Representation (which defines the objects, attributes, states, and relationships within the proposed system), Behavior (which defines the operations/functions/methods and their interrelationships/interfaces that are associated with the system objects as well as the corresponding state transitions), and Process (which defines the interrelationships of the system objects’ behavior represented via sequences of operations/functions/methods restricted by constraints/dependencies). A significant concept introduced in the RBP Model concerns Goals, which are associated with the Behavior of object(s). Additionally, we represented the concept of constraints/dependencies in each of the three sub-models of Representation, Behavior, and Process because we address this concept differently in each of the model. Lastly, we interconnected the concepts of interface and method under the sub-model Behavior via the concept Relation.

(1) Representation

*product functions chapter describes what the software will do and any “logical relationships among variables”; constraints subSection includes hardware limitations*;
design entity attributes (identification, type, purpose, subordinates, resources, processing, data); decomposition descriptions; design constraints (including limitations of hardware and other standards requirements); design entity dependencies;
actor(s), object(s), event(s), object state/substate(s), relationship (containment, generalization/specialization);

(2) Behavior
product perspective “describes how the software operates inside various constraints” including system interfaces, user interfaces, hardware interfaces, software interfaces, communication interfaces operations; constraints subSection includes interfaces to other applications; external interfaces (including names, input/output, valid range/accuracy/tolerance, units of measure, timing, relationships to other inputs/outputs, and formatting for data and commands; design entity interface; functions (including operations, abnormal situation responses, parameter effects); performance requirements (both static and dynamic); input validity checks; relationship (association); actions/behavior, function/methods/return values; messaging; state transition(s), preconditions, postconditions, exceptions, time; goals;

(3) Process
product perspective “describes how the software operates inside various constraints”; product functions chapter describes what the software will do and any “logical relationships among variables”; constraints subSection includes parallel operations, control functions, signal handshake protocols, reliability requirements, criticality of the application, and safety and security considerations;
performance requirements (both static and dynamic); functions (including operation sequence);
scenario(s);

The second stage of development for the Representation/Behavior/Process (RBP) model is presented below by rearranging the above IEEE concepts into a new and simplified format. The terms “Object”, “Action” and “Sequence” are used in the RBP Model in their most general terms.

(1) REPRESENTATION:
OBJECT: (actor(s), object(s), event(s));
ATTRIBUTES: (design entity attributes);
STATES: object state/substate(s);
RELATIONS: (“logical relationships among variables”; decomposition descriptions; design entity dependencies; relationship (containment, generalization/specialization));
CONSTRAINTS: hardware limitations; design entity dependencies; design constraints (including limitations of hardware and other standards requirements);

(2) BEHAVIOR:
ACTION: (how the software operates; functions (including operations, abnormal situation responses, parameter effects); actions/behavior, function/methods/return values; messaging);
ATTRIBUTES: (including names, input/output, valid range/accuracy/tolerance, units of measure, timing, relationships to other inputs/outputs, and formatting for data and commands);
RELATIONS: (system interfaces, user interfaces, hardware interfaces, software interfaces, communication interfaces operations; external interfaces; design entity interface); relationship (association);
TRANSITIONS: state transitions;
CONSTRAINTS: interfaces to other applications, performance requirements (both static and dynamic); input validity checks; preconditions, postconditions, exceptions, time;
GOALS: (what the software will do; reliability requirements, criticality of the application, and safety and security considerations; standards requirements);

(3) PROCESS:
SEQUENCE: (product perspective “describes how the software operates inside various constraints”; functions (including operation sequence); scenario(s));
CONSTRAINTS: (parallel operations, control functions, signal handshake protocols; reliability requirements, criticality of the application, and safety and security considerations; performance requirements (both static and dynamic)).

In this section we presented the semantically significant feature set (SSFS). We showed how it was derived from the IEEE standards. Lastly, we detailed the three sub-models of the SSFS. Section 6.4.3 utilizes the SSFS to evaluate MOA transformations.

6.4.3 Evaluation of UML to OSSD to KAOS Transformation

To evaluate the transformation of a UML design to OSSD to KAOS we show that the SSFS represents the significant software design and requirements level concepts in the source UML model, the intermediate OSSD Model, and the target KAOS model. Table 21 gives generic examples of the SSFS as represented in the UML Model, the OSSD Model and the KAOS model to show the transformation mappings between these three models.

The format given in Table 21 is the basis for the evaluation of transformation process from three viewpoints: structural, behavioral, or process. The representation viewpoint organizes the design and requirements specification information based on the structural objects within the presented system and associates the behavior of that system with those objects and the processes in which each behavior is a part. The behavioral viewpoint’s organization is based on the behavior of the presented system and associates the objects of that system with that behavior and the processes in which each behavior is a part. We based the process viewpoint’s organization on the processes of the presented system and associate the objects of that system with that behavior and the processes in which each behavior is a part.

This evaluation of the transformation process does not provide an exact one to one correspondence; however, it does successfully show that semantically significant features identified in a source design are represented in both the OSSD Model and target specification. Since the transformation from UML to OSSD Model to KAOS requires abstractions and semantic shifts, and currently no model transformation technique exists that can handle evaluation of such abstractions, we evaluated such abstractions and semantic shifts via evaluating transformation of the semantically significant feature set. We showed that significant software design and requirements level concepts are maintained throughout the model transformation from source model to target model.

6.5 Summary

We evaluated this research via a combined evaluation of its ontology, error detection capabilities, and transformations. We showed that the OSSD Model is ontologically sound by evaluating it using a generally agreed upon ontologically sound model, the BWW Model. Focusing on the most
commonly used ontological constructs and the narrowed scope of software specification and design, the analysis shows that OSSD Model is ontologically complete and non-redundant; additionally, the model does not have construct excess or overload. We demonstrated the error detection capabilities MOA by randomly selecting consistency rules from other consistency management techniques, incorporating them into the OSSD Model via axioms attached to properties and rules defined for the Inter-View Inconsistency Detection Table, and intentionally seeding errors in a source design of the Elevator case study to show that the methodology can successfully detect errors in the source design. We evaluated the transformation from source design to target specification by showing that a set of semantically significant features identified in a source design is represented in both the OSSD Model and target specification.

<table>
<thead>
<tr>
<th>SSFS</th>
<th>UML</th>
<th>OSSD</th>
<th>KAOS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Representation:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>Class, Diagram:</td>
<td>Object (Agent/Entity/Event), Instance</td>
<td>Agent/Entity/Event</td>
</tr>
<tr>
<td></td>
<td>Class, Object (instance)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attributes</td>
<td>Class Diagram:</td>
<td>Attribute:ObjectAttribute</td>
<td>Attribute</td>
</tr>
<tr>
<td></td>
<td>Object Attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relations</td>
<td>Class Diagram:</td>
<td>Relation:NonAssociation/ Generalization, Aggregation, Composition</td>
<td>Inherited From</td>
</tr>
<tr>
<td></td>
<td>Generalization, Aggregation,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Composition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constraints</td>
<td>Class Diagram:</td>
<td>Visibility,Multiplicity</td>
<td>None, Multiplicity</td>
</tr>
<tr>
<td></td>
<td>Visibility, Multiplicity</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Behavior:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>Class Diagram and Sequence Diagram:</td>
<td>Behavior</td>
<td>Operation</td>
</tr>
<tr>
<td></td>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attributes</td>
<td>Class Diagram and Sequence Diagram:</td>
<td>Attribute</td>
<td>Attribute</td>
</tr>
<tr>
<td></td>
<td>Attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relations</td>
<td>Class Diagram and Sequence Diagram:</td>
<td>Relation:Association</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>Association</td>
<td></td>
<td></td>
</tr>
<tr>
<td>States</td>
<td>StateMachine Diagram: State</td>
<td>State</td>
<td>Operation:DomPre (source state)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Operation:DomPost (destination state)</td>
</tr>
<tr>
<td>Transitions</td>
<td>StateMachine Diagram: Transition</td>
<td>Transition</td>
<td>None</td>
</tr>
<tr>
<td>Constraints</td>
<td>StateMachine Diagram: Transition Constraints</td>
<td>Constraint</td>
<td>Operationalization:ReqTrigFor, Operationalization:ReqPre, Operationalization:ReqPost</td>
</tr>
<tr>
<td>Goals</td>
<td>None</td>
<td>Goal</td>
<td>Goal</td>
</tr>
<tr>
<td>Process:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence</td>
<td>Sequence Diagram: Object Lifeline</td>
<td>Plan</td>
<td>Scenario</td>
</tr>
<tr>
<td>Constraints</td>
<td>Sequence Diagram: Constraints</td>
<td>Constraint</td>
<td>Operationalization:ReqTrigFor, Operationalization:ReqPre, Operationalization:ReqPost</td>
</tr>
</tbody>
</table>
7 Summary

7.1 Dissertation Summary

This research introduces an error detection methodology for software design, the Methodology for Object to Agents (MOA), which utilizes a common ontology-based model, Ontology for Software Specification and Design (OSSD) Model. MOA integrates multiple views of a software design to facilitate the interoperability of formal requirements modeling tools and software design tools with the ultimate goal of error detection in software designs. Inconsistency errors are the focus of the error detection in this work. The importance of identifying inconsistencies early in a software development project is recognized by software engineers as one of the keys to a successful project; however, few tools and techniques exist which apply formal inconsistency detection techniques at the software design level. MOA was defined to facilitate the detection of software design errors arising from multiple views of a design. It utilizes the concept of ontologies to define a common information model, the OSSD Model, which integrates object-oriented and agent-oriented approaches to software design. It is this ontological representation that enables the application of ontological reasoning to assist with semantic error detection in software designs. MOA defines a new form of error detection performed utilizing a combination of rules associated with the ontological properties of the OSSD Model, an Inter-View Inconsistency Detection technique, and a consistency framework. The focus of error detection was narrowed to inconsistency errors. MOA contributes to the software design process by integrating multiple views of software design, integrating object-oriented and agent-oriented concepts, and providing an error detection method for software designs. Additionally, MOA facilitates flexible error management by providing a technique to detect errors but not mandating immediate correction. Some software engineering tools enforce constraints by requiring correction before the software development process can continue. However, it is often necessary to live with inconsistency, assuming that it will be resolved at some time in the future. It is the identification of inconsistencies and the tracking of them that are most critical.

Three motivations for this research were: enhancing software design quality via error detection; integrating object-oriented with agent-oriented concepts and software specification with software design knowledge into one common model; and, creating an software methodology and tool integration component, in the form of an ontology. This research spans several related research areas including: ontologies, software design, requirements specification, consistency management, knowledge integration, agents, and tool integration.

This dissertation introduces MOA and the OSSD Model; it provides a unique definition and use of goal and behavior thesauruses to transform a software design to an OSSD representation of that design; it defines two forms of consistency checking; lastly, it provides the algorithms to transform source design into an instance of the OSSD Model and then transform an OSSD instance into a target formal requirements specification. The OSSD Model is a hierarchical decomposition of software development concepts including ontological constructs of objects, attributes, behavior, relations, states, transitions, goals, constraints and plans. Each of these constructs is further ontologically defined, such as decomposing objects into agent, entities or events. In addition to the hierarchical relationships, the OSSD Model contains properties that provide additional behavior relationships among OSSD constructs. Attached to these properties are rules that used to specify semantic relationship among the OSSD constructs and facilitate error detection. MOA includes both transformations and consistency checking. The initial transformation process includes both lexical and semantic analysis of a source software design that utilizes multiple mapping tables in its algorithm to create an instance of the OSSD Model. The consistency checking is a two-stage process assisted by a consistency framework and Inter-View Inconsistency Detection Table. The final
transformation process produces the target requirements specification from the OSSD Model instance via a set of simple mapping tables.

After providing details regarding MOA and the OSSD Model, we demonstrated MOA via two case studies: an elevator system (Chapter 5) and a computer-aided ambulance dispatch system (Appendix A). A subset of UML Use Case, Class, Sequence, and StateMachine diagrams of each case study was seeded with consistency errors. MOA transformed the multiple views of each case study into an instance of the OSSD Model and then into a KAOS requirements specification. Consistency checking successfully detected two of the seeded errors in each case study. The third error in each case study is easily detectable via the KAOS processing of the generated requirements specifications. Finally, an evaluation of MOA’s ontological representation, error detection capabilities, and transformations showed that: the OSSD Model is ontologically complete, non-redundant, and does not have construct excess or overload based on its comparison with a generally agreed upon ontologically sound BWW Model; the error detection capabilities of MOA did successfully detect design errors; and the transformation of a set of semantically significant features was successfully performed from source design to target specification.

7.2 Contributions

This research contributes to improving the quality of software design in the following ways.

1) It provides a unique methodology to detecting errors in software design arising from multiple views of that design. It has the capability to detect not only simple syntactic errors but also more complex semantic errors. This research performs error detection utilizing a combination of rules associated with the ontological properties of its common model, an Inter-View Inconsistency Detection technique, and a consistency framework. It is this ontological representation that enables the application of ontological reasoning to assist with semantic error detection in software designs. Most software design consistency checks are syntactic, based on the well-formed rules (WFR) specified in the UML 2.0 Specification that address primarily the syntactic inconsistencies within a given UML diagram.

2) It facilitates a systematic approach toward developing a comprehensive and high-level error detection rule set via its consistency framework. This framework, which includes the ontological elements of the OSSD Model, enables a broad definition of consistency rules that includes a wide variety of potential interactions among software design constructs. Additionally, while most software design consistency checks are syntactic, based on UML’s well-formed rules, MOA enables the creation of semantic rules above and beyond the typical syntactic checks. Ontological reasoning can be applied to these rules to assist with detecting complex design errors.

3) It enhances the semantic interoperability of software modeling tools. MOA facilitates the integration of informal software modeling tools with formal requirements specification tools to apply the error detection capabilities of the formal tools to an informal software design.

4) It includes a unique integrated ontology for object-oriented and agent-oriented concepts that minimizes the difficulties of mapping between these two paradigms, while reaping the benefits of each approach. Since it appears that both object-oriented and agent-oriented software development will continue to coexist for the foreseeable future, it is critical that future software development address the integration of these two worlds. Additionally, there exist few error detection techniques for software design that take into consideration the integration of AO and OO concepts.
5) It can reduce the development time and effort as compared with other error detection techniques because it integrates existing tools that have individually undergone development and testing. It also reduces development effort and time to transform among a variety of software engineering models by utilizing a common information model. The common information model reduces the number of transformations to only $2n$ (where $n$ is the number of software engineering models) rather than the $n^2 - n$ transformations required to transform between each pair of models. The common information model also minimizes the effect of changes to one software engineering model thereby requiring changes to only the transformation between the common model and the modified model.

6) It requires no additional training or expertise to reap the benefits of formal methods. No operational knowledge of the formal software modeling tool is required to detect inconsistency design errors.

7) Because the OSSD Model is ontology-based and defined using OWL, it has the potential to become a part of a knowledge-based system for software design within Semantic Web environments by enabling communication and knowledge sharing among agents such as Software Design Agents [Brazier et al.], or agents within distributed design environments such as the Intelligent Agent Based Collaborative Design Information Management and Support Tools (IDIMS) project [Tormey et al.], or as an integration component to facilitate the semantic interoperability of aerospace architectures [Kogut & Heflin]. Figure 59 shows a graphical view of potential interoperability of MOA.

![Figure 59: Future MOA Interoperability](image)

7.3 Future Work

The research presented in this dissertation develops a new type of error detection tool for software design. Future work includes the following.
1) The OSSD Model will be built using the Protégé ontology modeling and knowledge base acquisition tool [Gennari] that will create an OWL representation of the OSSD Model.

2) The OSSD Model consistency rules will be specified using the Semantic Web Rule Language (SWRL) [Horrocks et al.], a recent W3C proposal for semantic rule languages [SWRL]. SWRL extends OWL by introducing rule axioms that enable ontological reasoning beyond the basic axioms included in OWL (such as subclass and equivalentClass). The Protégé ontology modeling and knowledge base acquisition tool that will be used to build the OSSD Model has a SWRL plugin editor that facilitates the interactive creation and editing of SWRL rules.

3) Updates to the original source design will be automated based on the errors detected from the formal target specification consistency analysis. Currently, results of the error detection performed by the software specification tools are not automatically applied to the original software design; however, an evaluation and prioritization of the errors identified must be performed before the original design is updated.

4) MOA will be applied to integrate multiple software design languages with multiple agent-oriented specification languages.

5) Knowledge from requirements specifications created before the source design will be integrated into the OSSD Model of that design.
References


Appendix A: London Ambulance Service Computer Aided Dispatch Case Study

We apply MOA to a portion of a well-known and often utilized case study, the London Ambulance Service (LAS) Computer Aided Dispatch (CAD) System, which was used as a common case study at the 8th International Workshop on Software Specification and Design (IWSSD-8) [Finkelstein & Dowell]. This case study is considerably more complex than the elevator system case study. The LAS CAD System is a safety-critical, real-time, distributed system that receives emergency calls, dispatches ambulances based on medical need and availability of resources, and tracks the allocation of resources to emergency calls.

A sample of the basic system users and locations is:

- Dispatcher at the Central Ambulance Control
- Ambulance driver at each ambulance
- Hospital emergency room supervisor at each hospital emergency room
- Locations in the ambulance service jurisdiction are partitioned into sectors
- Incidents are geographically widely distributed

A portion of the basic system functionality is:

- Call taking: receiving emergency calls; recording incident details
- Dispatching ambulances: identifying nearest available ambulances; communicating with ambulance drivers; monitoring ambulance status; transporting patient(s) to nearest available hospital
- Time constraints: an ambulance should be dispatched within 3 minutes of receiving a call; an ambulance should arrive at the location of the incident within 14 minutes after the first call is received

Figures 60 and 61 give examples of Use Cases describing the LAS CAD system functionality.

<table>
<thead>
<tr>
<th>Use Case name:</th>
<th>Provide Ambulance Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Actor(s):</td>
<td>Dispatcher, Computer-AidedDispatch, ER Supervisor</td>
</tr>
<tr>
<td>Precondition:</td>
<td>Open incident does not exist for Caller</td>
</tr>
<tr>
<td>Postcondition:</td>
<td>Incident is completed for Caller</td>
</tr>
<tr>
<td>Trigger:</td>
<td>Caller Makes Emergency Call</td>
</tr>
</tbody>
</table>

Scenario:
Caller Makes Emergency Call
Dispatcher requests Computer-AidedDispatch to create an incident
Computer-AidedDispatch Dispatches Ambulance
ER Supervisor Updates Resource Status
Dispatcher requests Computer-AidedDispatch to close the incident

Figure 60: UML Use Case: Provide Ambulance Service
Use Case name: Dispatches Ambulance
Primary Actor(s): Computer-AidedDispatch, Ambulance Driver
Precondition: Open incident, location, section, resource status and ambulance status data are current
Postcondition: Ambulance is assigned in < 3 minutes; Ambulances arrives at incident location < 14 minutes
Trigger: Computer-AidedDispatch creates an incident

Scenario:
Computer-AidedDispatch identifies nearest available ambulance
Computer-AidedDispatch identifies nearest available hospital
Computer-AidedDispatch sends incident information to nearest available ambulance
Computer-AidedDispatch sends incident information to nearest available hospital
Ambulance driver Updates Ambulance Status
Ambulance driver Updates Resource Status
Ambulance arrives at location of incident

Figure 61: UML Use Case: Provide Ambulance Service

Nested Use Cases are underlined for ease of understanding. Figures 62 through 65 provide several UML diagrams that model the LAS CAD system. These diagrams are based on a conglomeration of previously specified software requirements specification of the London Ambulance Service [Allen],

Figure 62: CAD Use Case Diagram
We have extended these diagrams and seeded them with consistency errors to demonstrate our approach.

Figure 63: CAD Class Diagram

Figure 64. Ambulance StateMachine Diagram
Tables 22 through 29 contain partial examples of the mappings from UML to OSSD for the LAS CAD case study.

Table 22: UML Class Element and Part of Speech (POS) Tagging

<table>
<thead>
<tr>
<th>UML Name</th>
<th>UML Element</th>
<th>Part of Speech (POS) SubPOS</th>
<th>SUMO/WordNet</th>
<th>Defined within UML (G/A/C)</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambulance</td>
<td>class</td>
<td>noun</td>
<td>Entity:Physical:Object</td>
<td>computer-aided dispatch system/C</td>
<td>none</td>
</tr>
<tr>
<td>Ambulance</td>
<td>noun</td>
<td>noun</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>location</td>
<td>attribute</td>
<td>noun</td>
<td>Entity:Physical:Object</td>
<td>incident</td>
<td>none</td>
</tr>
<tr>
<td>caller</td>
<td>class</td>
<td>noun</td>
<td>Entity:Abstract:Attribute: RelationalAttribute:SocialRole</td>
<td>computer-aided dispatch system/C</td>
<td>none</td>
</tr>
</tbody>
</table>
### Table 23: OSSD Relations Classification

<table>
<thead>
<tr>
<th>UML association</th>
<th>OSSD Relation</th>
<th>OSSD Relation Attribute</th>
<th>OSSD Role</th>
<th>OSSD Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>informs</td>
<td>Association:Rel.Attrib:From Obj</td>
<td>Dispatcher</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>Association:Rel.Attrib:To.Obj</td>
<td>Computer-Aided Dispatch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unnamed</td>
<td>NonAssociation:Aggreg:Whole</td>
<td>Resource</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>NonAssociation:Aggreg:Part</td>
<td>Ambulance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 24: OSSD Behavior Classification

<table>
<thead>
<tr>
<th>UML Operation</th>
<th>Msg Type</th>
<th>Message Parameters</th>
<th>Sending Object / OSSD Classification</th>
<th>Receiving Object / OSSD Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignIncident</td>
<td>A</td>
<td>none</td>
<td>Computer-AidedDispatch/Control</td>
<td>AmbulanceDriver/Perform</td>
</tr>
<tr>
<td>ambulanceArrived</td>
<td>B</td>
<td>none</td>
<td>ER Supervisor/Perform</td>
<td>Computer-AidedDispatch/Monitor</td>
</tr>
</tbody>
</table>

### Table 25: OSSD State-based Object Classification

<table>
<thead>
<tr>
<th>UML Class</th>
<th>UML Composition</th>
<th>SUMO/WordNet Classification</th>
<th>OSSD Behavior</th>
<th>OSSD Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatcher</td>
<td>Whole top level</td>
<td>Entity:Abstract:Attribute:RelationalAttribute:SocialRole</td>
<td>Control, Perform</td>
<td>Agent</td>
</tr>
<tr>
<td>Incident</td>
<td>Whole top level</td>
<td>Entity:Physical:Process</td>
<td>Perform</td>
<td>Entity</td>
</tr>
</tbody>
</table>

### Table 26: MOA State, Transition, Constraints Classification Table Part 1

<table>
<thead>
<tr>
<th>UML Class</th>
<th>UML State</th>
<th>OSSD State Classif.</th>
<th>OSSD Entry Behavior</th>
<th>OSSD Do Behavior</th>
<th>OSSD Exit Behavior</th>
<th>OSSD Incoming Transition/From</th>
<th>OSSD Outgoing Transition/To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ambulance</td>
<td>IdleinLot</td>
<td>Initial</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>EnroutetoLot</td>
<td>Enrouteto Incident</td>
</tr>
<tr>
<td>Ambulance</td>
<td>EnrouteTo Incident</td>
<td>Intermed</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>IdleinLot, AtHospital</td>
<td>AtIncident, EnrouteToLot</td>
</tr>
<tr>
<td>Ambulance</td>
<td>AtIncident</td>
<td>Intermed</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>EnrouteTo Incident</td>
<td>EnrouteTo Hospital, EnrouteToLot</td>
</tr>
</tbody>
</table>

### Table 27: MOA State, Transition, Constraints Classification Table Part 2

<table>
<thead>
<tr>
<th>UML Class</th>
<th>OSSD Trans. Num.</th>
<th>OSSD Transition IncomingFrom/OutgoingTo</th>
<th>OSSD Constraint Classif.</th>
<th>OSSD Constraint followed by OSSD Transition Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ambulance</td>
<td>1</td>
<td>IdleinLot/EnrouteToIncident</td>
<td>assignIncident, {ambulance.status=available ^ ambulance.location&lt;&gt; incident.location}</td>
<td>Trigger, Precondition 2 or 3</td>
</tr>
<tr>
<td>Ambulance</td>
<td>2</td>
<td>EnrouteToIncident/AtIncident</td>
<td>arrived, [location=incident]</td>
<td>Trigger, Guard 4 or 5</td>
</tr>
<tr>
<td>Ambulance</td>
<td>3</td>
<td>EnrouteToIncident/EnrouteToLot</td>
<td>incidentCancelled</td>
<td>Trigger 9</td>
</tr>
<tr>
<td>Ambulance</td>
<td>4</td>
<td>AtIncident/EnrouteToHospital</td>
<td>PatientOnBoard</td>
<td>Trigger 6</td>
</tr>
</tbody>
</table>
Table 28: MOA Goal Classification Table

<table>
<thead>
<tr>
<th>UML UseCase Name</th>
<th>OSSD Agent</th>
<th>OSSD Entity</th>
<th>OSSD Behavior</th>
<th>OSSD Goal</th>
<th>OSSD Goal Classif.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides Ambulance Service</td>
<td>Caller</td>
<td>none</td>
<td>getName, getAddress, getDescription</td>
<td>IncidentReported</td>
<td>Achieve</td>
</tr>
<tr>
<td></td>
<td>Dispatcher</td>
<td>none</td>
<td>GetNextCall</td>
<td>GetCallerData, AvailableToReceiveCall</td>
<td>Achieve</td>
</tr>
<tr>
<td></td>
<td>ER Supervisor</td>
<td>Incident</td>
<td>resourceCheck, getResourceStatus, assignResource</td>
<td>ResourcesAvailable, ResourcesAssigned</td>
<td>Maintain</td>
</tr>
<tr>
<td></td>
<td>Computer-Aided Dispatch</td>
<td>Incident</td>
<td>createIncident, selectAmbulance, selectHospital, assignIncident</td>
<td>IncidentCreated, IdentifyNearestAmbulance, IdentifyNearestHospital, AmbulanceDispatched, TrackAmbulance, IncidentClosed</td>
<td>Achieve</td>
</tr>
</tbody>
</table>

Table 29: Inter-View Inconsistency Detection Table

<table>
<thead>
<tr>
<th>UML Element</th>
<th>OSSD Element</th>
<th>Class Diagram</th>
<th>Sequence Diagram</th>
<th>StateMachine Diagram</th>
<th>Use Cases or Use Case Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller</td>
<td>Agent</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>assignIncident</td>
<td>Behavior</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>incidentCancelled</td>
<td>Behavior</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

Figure 66 shows a partial view of the OSSD Model for the LAS CAD system described in the UML diagrams given in Figures 62 through 65. Figure 66 also shows an example of two rules the OB_Rule1 and OR_Rule1, shown in Figure 28 and discussed in more detail in Section 4.3.5.2. To simplify the pictorial view of the OSSD Model for the LAS CAD System, these figures show only the significant classes and properties. Some super-classes and paths connecting upward to the Construct level are omitted to simplify the diagrams and ease their understanding.

Figure 67 contains Sections of a KAOS specification for the LAS-CAD System that is generated at the end of the MOA transformation.

We seeded errors into the UML design of the LAS CAD case study given in Figures 62 through 65. The following two errors can be correctly detected during the creation and consistency processing of the OSSD Model:

1) an inconsistency between the Sequence Diagram showing the Computer-Aided Dispatch class exchanging messages with the Ambulance Driver class but the Class Diagram does not describe an association link between the Computer-Aided Dispatch and the Ambulance Driver classes;
2) an inconsistency between the Ambulance StateMachine Diagram showing “incidentCancelled” which is not specified in either the Class or Sequence Diagrams;
3) the ambulance assigned to the incident is unable to arrive at the incident location within the required 14 minutes due to unexpected events such as traffic gridlock.
The first inconsistency concerns a missing association link in the UML Class Diagram that is detected via two OSSD axioms OB-1 and OR-1 (based on the OB_Rule1 and OR_Rule1 axioms given in Figure 28 in Chapter 4 Section 4.3.5.2). Figure 66 shows these axioms on two properties: 1) the property \textit{has} linking \textit{State-based} (representing “Computer-Aided Dispatch”) to \textit{Behavior} (representing “assignIncident”); and 2) the property \textit{has} linking \textit{State-based} (representing “Computer-Aided Dispatch”) to \textit{Behavior} (representing “setIncident Type”). This diagram shows that first axiom pair reveals an inconsistency: axiom OB-1 is true but axiom OR-1 is not true. Figure 66
shows that “Computer-AidedDispatch” participates in only one Association, “Manages”, which involves the Entity “Incident” and includes the Behavior “setIncidentType. However, the axiom OR-1 is violated because no Association exists between “Computer-AidedDispatch” and “AmbulanceDriver” to support the exchange of the message associated with the Behavior “assignIncident”.

The second inconsistency involving “incidentCancelled” is detected via the IC_Rule3 associated the Inter-view Inconsistency Detection Table (refer to Figure 29 in Chapter 4 Section 4.3.5.3). Figure 64 shows that the Behavior “incidentCancelled” is defined in the StateMachine diagram but not in Class or Sequence diagrams and so violates IC_Rule3.

The remaining seeded error would be identified during the KAOS processing of the LAS CAD specification. The KAOS processing includes obstacle generation that would identify such an event, as well as numerous other potential conflicts and obstacles, and then recommend alternative solutions to those obstacles and/or conflicts.
## Appendix B: Additional Data for the Elevator System Case Study

### Table 30: UML Class Element and Part of Speech (POS) Tagging

<table>
<thead>
<tr>
<th>UML Name</th>
<th>UML Element</th>
<th>Part of Speech (POS) SubPOS</th>
<th>SUMO/WordNet</th>
<th>Defined within UML (G/A/C)</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Door Controller</td>
<td>Class</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorSystem/ C</td>
<td>None</td>
</tr>
<tr>
<td>Doors State</td>
<td>Attribute</td>
<td>Noun</td>
<td>Entity:Physical:Object or Miscellaneous:</td>
<td>DoorController</td>
<td>None</td>
</tr>
<tr>
<td>Motor</td>
<td>Class</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorCar/C</td>
<td>None</td>
</tr>
<tr>
<td>Move Up</td>
<td>Operation</td>
<td>Verb:present, Adverb</td>
<td>Entity:Physical:Process: Motion</td>
<td>Motor</td>
<td>None</td>
</tr>
<tr>
<td>Move Down</td>
<td>Operation</td>
<td>Verb:present, Adverb</td>
<td>Entity:Physical:Process: Motion</td>
<td>Motor</td>
<td>None</td>
</tr>
<tr>
<td>Elevator Controller</td>
<td>Class</td>
<td>Noun</td>
<td>Entity:Physical:Object or Miscellaneous:</td>
<td>Elevator System</td>
<td>None</td>
</tr>
<tr>
<td>Elevator Destination List</td>
<td>Attribute</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>Elevator Controller</td>
<td>None</td>
</tr>
<tr>
<td>Elevator Direction</td>
<td>Attribute</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>Elevator Controller</td>
<td>None</td>
</tr>
<tr>
<td>Timer</td>
<td>Attribute</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorSystem</td>
<td>None</td>
</tr>
<tr>
<td>At Floor</td>
<td>Attribute</td>
<td>Preposition, Noun</td>
<td>Not found</td>
<td>Elevator Controller</td>
<td>None</td>
</tr>
<tr>
<td>Button Pressed</td>
<td>Operation</td>
<td>Noun</td>
<td>Entity:Physical:Object</td>
<td>Elevator Controller</td>
<td>BT F</td>
</tr>
<tr>
<td>Elevator Arrived At</td>
<td>Operation</td>
<td>Verb:past, Preposition</td>
<td>Entity:Physical:Process: Motion</td>
<td>Elevator Controller</td>
<td>EL</td>
</tr>
<tr>
<td>Doors Opened</td>
<td>Operation</td>
<td>Noun</td>
<td>Entity:Physical:Process: Motion</td>
<td>Elevator Controller</td>
<td>None</td>
</tr>
<tr>
<td>Doors Closed</td>
<td>Operation</td>
<td>Noun</td>
<td>Entity:Physical:Process: Motion</td>
<td>Elevator Controller</td>
<td>None</td>
</tr>
</tbody>
</table>

*table continued*
<table>
<thead>
<tr>
<th>Operation</th>
<th>Operation Type</th>
<th>Adjective/Verb</th>
<th>Entity Type</th>
<th>Controller</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Door Sensor</td>
<td>Class Noun Noun</td>
<td>Entity:Physical:Object Entity:Physical:Object</td>
<td>DoorController/C</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Door</td>
<td>Class Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorCar/C, Floor/C</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>State Attribute</td>
<td>Noun</td>
<td>Entity:Abstract:Attribute</td>
<td>Door</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Open</td>
<td>Operation Verb:present</td>
<td>Entity:Physical:Process: Motion</td>
<td>Door</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Close</td>
<td>Operation Verb:present</td>
<td>Entity:Physical:Process: Motion</td>
<td>Door</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>DoorAlarm</td>
<td>Class Noun</td>
<td>Entity:Physical:Object</td>
<td>Door/C</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Alarm</td>
<td>Class Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorSystem/C</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>DoorAlarm</td>
<td>Class Noun</td>
<td>Entity:Physical:Object</td>
<td>Alarm/G</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>CarAlarm</td>
<td>Class Noun</td>
<td>Entity:Physical:Object</td>
<td>Alarm/G</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Ring Alarm</td>
<td>Operation Verb:past</td>
<td>RelationalAttribute: SoundAttribute Entity:Physical:Object</td>
<td>Alarm</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Inner Door</td>
<td>Class Adjective Noun</td>
<td>Relation:Subclass:Located Entity:Physical:Object</td>
<td>Door/G</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Outer Door</td>
<td>Class Adjective Noun</td>
<td>Relation:Subclass:Located Entity:Physical:Object</td>
<td>Door/G</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Floor</td>
<td>Class Noun</td>
<td>Entity:Physical:Object</td>
<td>ElevatorSystem/C</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Passenger</td>
<td>Class Noun</td>
<td>Entity:Abstract:Attribute: SocialRole</td>
<td>Elevator System/A</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

*table continued*
<table>
<thead>
<tr>
<th></th>
<th>Class</th>
<th>Noun</th>
<th>Entity: Physical: Object</th>
<th>Floor/C,</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>UpButton</td>
<td>Class</td>
<td>Noun</td>
<td>Entity: Physical: Object</td>
<td>Floor/C,</td>
<td>None</td>
</tr>
<tr>
<td>Dwn Button</td>
<td>Class</td>
<td>Noun</td>
<td>Entity: Physical: Object</td>
<td>Floor/C,</td>
<td>None</td>
</tr>
<tr>
<td>Floor Button</td>
<td>Class</td>
<td>Noun</td>
<td>Entity: Physical: Object</td>
<td>ButtonPanel/C</td>
<td>None</td>
</tr>
<tr>
<td>Open Button</td>
<td>Class</td>
<td>Noun</td>
<td>Entity: Physical: Object</td>
<td>ButtonPanel/C</td>
<td>None</td>
</tr>
<tr>
<td>Close Button</td>
<td>Class</td>
<td>Noun</td>
<td>Entity: Physical: Object</td>
<td>ButtonPanel/C</td>
<td>None</td>
</tr>
<tr>
<td>Alarm Button</td>
<td>Class</td>
<td>Noun</td>
<td>Entity: Physical: Object</td>
<td>ButtonPanel/C</td>
<td>None</td>
</tr>
<tr>
<td>Down Button</td>
<td>Class</td>
<td>Noun</td>
<td>Entity: Physical: Object</td>
<td>Button/G</td>
<td>None</td>
</tr>
<tr>
<td>Alarm Button</td>
<td>Class</td>
<td>Noun</td>
<td>Entity: Physical: Object</td>
<td>Button/G</td>
<td>None</td>
</tr>
<tr>
<td>UpButton</td>
<td>Class</td>
<td>Noun</td>
<td>Entity: Physical: Object</td>
<td>CallButton/G</td>
<td>None</td>
</tr>
<tr>
<td>Down Button</td>
<td>Class</td>
<td>Noun</td>
<td>Entity: Physical: Object</td>
<td>CallButton/G</td>
<td>None</td>
</tr>
<tr>
<td>Light</td>
<td>Class</td>
<td>Noun</td>
<td>Entity: Physical: Object</td>
<td>Button/C</td>
<td>None</td>
</tr>
<tr>
<td>Turn Light On</td>
<td>Operation</td>
<td>Verb: present</td>
<td>Noun</td>
<td>Entity: Physical: Process: Motion</td>
<td>Button</td>
</tr>
<tr>
<td>Turn Light Off</td>
<td>Operation</td>
<td>Verb: present</td>
<td>Noun</td>
<td>Entity: Physical: Process: Motion</td>
<td>Button</td>
</tr>
<tr>
<td>Presses</td>
<td>Association</td>
<td>Verb: present</td>
<td>Entity: Physical: Process: Motion</td>
<td>ElevatorSystem</td>
<td>None</td>
</tr>
<tr>
<td>Start</td>
<td>Operation</td>
<td>Verb: present</td>
<td>Entity: Physical: Process: Motion</td>
<td>Timer</td>
<td>None</td>
</tr>
<tr>
<td>Stop</td>
<td>Operation</td>
<td>Verb: present</td>
<td>Entity: Physical: Process: Motion</td>
<td>Timer</td>
<td>None</td>
</tr>
</tbody>
</table>
### Table 31: OSSD Relations Classification

<table>
<thead>
<tr>
<th>UML assoc.</th>
<th>OSSD Relation</th>
<th>OSSD Relation Attribute</th>
<th>OSSD Role</th>
<th>OSSD Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>unnamed</td>
<td>Nonassociation:General:Superclass</td>
<td>Button</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:General:Subclass</td>
<td>CallButton</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:General:Subclass</td>
<td>FloorButton</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:General:Subclass</td>
<td>OpenButton</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:General:Subclass</td>
<td>CloseButton</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:General:Subclass</td>
<td>AlarmButton</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>unnamed</td>
<td>Nonassociation:Aggreg:Whole</td>
<td>ElevatorCar</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:Aggreg:Part</td>
<td>InnerDoor</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:Aggreg:Part</td>
<td>WeightSensor</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:Aggreg:Part</td>
<td>ButtonPanel</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:Aggreg:Part</td>
<td>Motor</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>unnamed</td>
<td>Nonassociation:Aggreg:Whole</td>
<td>Floor</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:Aggreg:Part</td>
<td>OuterDoor</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:Aggreg:Part</td>
<td>CallButton</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>unnamed</td>
<td>Nonassociation:Aggreg:Whole</td>
<td>Door</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:Aggreg:Part</td>
<td>DoorSensor</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td>unnamed</td>
<td>Nonassociation:Aggreg:Whole</td>
<td>Button</td>
<td>none</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Nonassociation:Aggreg:Part</td>
<td>Light</td>
<td>none</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 32: OSSD Behavior Classification

<table>
<thead>
<tr>
<th>UML Operation</th>
<th>Msg Type</th>
<th>Message Params</th>
<th>Sending Object / OSSD Classification</th>
<th>Receiving Object / OSSD Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>MoveTo</td>
<td>A</td>
<td>F, D</td>
<td>ElevatorController/Control</td>
<td>ElevatorCar/Perform</td>
</tr>
<tr>
<td>MoveUp</td>
<td>A</td>
<td>none</td>
<td>ElevatorCar/Control</td>
<td>Motor/Perform</td>
</tr>
<tr>
<td>ElevatorArrived</td>
<td>B</td>
<td>EL</td>
<td>FloorSensor/Perform</td>
<td>ElevatorCar/Monitor</td>
</tr>
<tr>
<td>MoveDown</td>
<td>A</td>
<td>none</td>
<td>ElevatorCar/Control</td>
<td>Motor/Perform</td>
</tr>
<tr>
<td>Stop</td>
<td>A</td>
<td>none</td>
<td>ElevatorCar/Control</td>
<td>Motor/Perform</td>
</tr>
<tr>
<td>Stopped</td>
<td>B</td>
<td>none</td>
<td>Motor/Perform</td>
<td>ElevatorCar/Monitor</td>
</tr>
<tr>
<td>ElevatorArrived</td>
<td>B</td>
<td>none</td>
<td>ElevatorCar/Perform</td>
<td>ElevatorController/Monitor</td>
</tr>
<tr>
<td>EnterElevator</td>
<td>A</td>
<td>none</td>
<td>ElevatorController/Control</td>
<td>Passenger/Perform</td>
</tr>
</tbody>
</table>

### Table 33: OSSD State-based Object Classification

<table>
<thead>
<tr>
<th>UML Class</th>
<th>UML Composition</th>
<th>SUMO/WordNet Classification</th>
<th>OSSD Behavior</th>
<th>OSSD Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Door</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>Floor</td>
<td>Whole top level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>Button</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>Light</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>CallButton</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>UpButton</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>DownButton</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>AlarmButton</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>OpenButton</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>CloseButton</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
<tr>
<td>FloorButton</td>
<td>Part sub level</td>
<td>Entity:Physical:Object</td>
<td>Perform</td>
<td>Entity</td>
</tr>
</tbody>
</table>

138
### Table 34: MOA State, Transition, Constraints Classification Table Part 2

<table>
<thead>
<tr>
<th>UML Class</th>
<th>OSSD Trans. Num.</th>
<th>OSSD Transition From/To</th>
<th>OSSD Constraint</th>
<th>OSSD Constraint Classif.</th>
<th>OSSD Transition followed by OSSD Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Door Controller 11</td>
<td>DoorsClosed/OpeningDoors</td>
<td>OpenDoors(), (ElevatorCar.State=stopped) ^ (CurrentFloor = Requested Floor)</td>
<td>Trigger, Precondition</td>
<td>2 or 3</td>
<td></td>
</tr>
<tr>
<td>Door Controller 12</td>
<td>OpeningDoors/DoorsOpen</td>
<td>OuterDoorOpened() ^ InnerDoorOpened()</td>
<td>Trigger</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Door Controller 13</td>
<td>OpeningDoors/ClosingDoors</td>
<td>CloseDoors() [NotObstructed()]</td>
<td>Trigger, Guard</td>
<td>5 or 6 or 7</td>
<td></td>
</tr>
<tr>
<td>Door Controller 14</td>
<td>DoorsOpen/ClosingDoors</td>
<td>CloseDoors() [NotObstructed()]</td>
<td>Trigger, Guard</td>
<td>5 or 6 or 7</td>
<td></td>
</tr>
<tr>
<td>Door Controller 15</td>
<td>ClosingDoors/OpeningDoors</td>
<td>OpenDoors()</td>
<td>Trigger</td>
<td>2 or 3</td>
<td></td>
</tr>
<tr>
<td>Door Controller 16</td>
<td>ClosingDoors/OpeningDoors</td>
<td>[Obstructed()] OpenDoors()</td>
<td>Guard, Action</td>
<td>2 or 3</td>
<td></td>
</tr>
<tr>
<td>Door Controller 17</td>
<td>ClosingDoors/DoorsClosed</td>
<td>OuterDoorClosed() ^ InnerDoorClosed()</td>
<td>Trigger</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Table 35: MOA Goal Classification Table

<table>
<thead>
<tr>
<th>UML UseCase Name</th>
<th>OSSD Agent</th>
<th>OSSD Entity</th>
<th>OSSD Behavior</th>
<th>OSSD Goal</th>
<th>OSSD Goal Classif.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Floor</td>
<td>Passenger</td>
<td>ElevatorCar, ButtonPanel, FloorButton</td>
<td>PressButton</td>
<td>Floor Request</td>
<td>Achieve</td>
</tr>
<tr>
<td>Request Floor</td>
<td>Elevator Controller</td>
<td>ElevatorCar, ButtonPanel, FloorButton, Light</td>
<td>TurnLightOn</td>
<td>Floor Request</td>
<td>Achieve</td>
</tr>
<tr>
<td>Request Floor</td>
<td>Elevator Controller</td>
<td>ElevatorCar</td>
<td>See sub goal</td>
<td>DoorsClosed</td>
<td>Achieve</td>
</tr>
<tr>
<td>Request Floor</td>
<td>Elevator Controller</td>
<td>ElevatorCar</td>
<td>See sub goal</td>
<td>Elevator Movement</td>
<td>Achieve</td>
</tr>
<tr>
<td>Request Floor</td>
<td>Elevator Controller</td>
<td>ElevatorCar, ButtonPanel, FloorButton, Light</td>
<td>TurnLightOff</td>
<td>Floor Request</td>
<td>Achieve</td>
</tr>
<tr>
<td>Request Floor</td>
<td>none</td>
<td>DoorController</td>
<td>See sub goal</td>
<td>Doors Opened</td>
<td>Achieve</td>
</tr>
<tr>
<td>CloseDoors</td>
<td>Elevator Controller</td>
<td>DoorSensor</td>
<td>See sub goal</td>
<td>Door Operation Safety Ensurement</td>
<td>Achieve</td>
</tr>
<tr>
<td>CloseDoors</td>
<td>Elevator Controller</td>
<td>WeightSensor</td>
<td>See sub goal</td>
<td>Excessive Weight Prevention</td>
<td>Achieve</td>
</tr>
<tr>
<td>CloseDoors</td>
<td>Elevator Controller</td>
<td>DoorController</td>
<td>CloseDoors</td>
<td>Doors Closed</td>
<td>Achieve</td>
</tr>
</tbody>
</table>

*table continued*
<table>
<thead>
<tr>
<th>CloseDoors</th>
<th>none</th>
<th>DoorController, InnerDoor</th>
<th>CloseInnerDoor</th>
<th>Doors Closed</th>
<th>Achieve</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloseDoors</td>
<td>none</td>
<td>DoorController, OuterDoor</td>
<td>CloseOuterDoor</td>
<td>Doors Closed</td>
<td>Achieve</td>
</tr>
<tr>
<td>Ensure Safe Door Operation</td>
<td>Elevator Controller</td>
<td>DoorController</td>
<td>Check Obstruction</td>
<td>Door Operation Safety Guarantee</td>
<td>Maintain</td>
</tr>
<tr>
<td>Ensure Safe Door Operation</td>
<td>none</td>
<td>DoorController, DoorSensor, InnerDoor, OuterDoor</td>
<td>DoorNot Obstructed</td>
<td>Door Operation Safety Guarantee</td>
<td>Maintain</td>
</tr>
<tr>
<td>Ensure Safe Door Operation</td>
<td>none</td>
<td>DoorController, DoorSensor, InnerDoor, OuterDoor</td>
<td>DoorObstructed</td>
<td>Door Operation Safety Guarantee</td>
<td>Maintain</td>
</tr>
<tr>
<td>Ensure Safe Door Operation</td>
<td>none</td>
<td>DoorController, DoorAlarm</td>
<td>RingAlarm</td>
<td>Door Operation Safety Guarantee</td>
<td>Maintain</td>
</tr>
<tr>
<td>Prevent Exceeding Elevator Weight Limit</td>
<td>Elevator Controller</td>
<td>ElevatorCar</td>
<td>CheckWeight</td>
<td>Excessive Weight Prevention</td>
<td>Avoid</td>
</tr>
<tr>
<td>Prevent Exceeding Elevator Weight Limit</td>
<td>none</td>
<td>ElevatorCar, WeightSensor</td>
<td>NoExcess Weight</td>
<td>Excessive Weight Prevention</td>
<td>Avoid</td>
</tr>
<tr>
<td>Prevent Exceeding Elevator Weight Limit</td>
<td>none</td>
<td>ElevatorCar, WeightSensor</td>
<td>ExcessWeight</td>
<td>Excessive Weight Prevention</td>
<td>Avoid</td>
</tr>
<tr>
<td>Prevent Exceeding Elevator Weight Limit</td>
<td>none</td>
<td>ElevatorCar, Elevator CarAlarm</td>
<td>RingAlarm</td>
<td>Excessive Weight Prevention</td>
<td>Avoid</td>
</tr>
</tbody>
</table>
Allyson M. Hoss is a doctoral candidate at the Louisiana State University in Baton Rouge. She received a Bachelor of Science in business administration/management information systems from the State University of New York at Albany, and a Master of Science in computer science from the University of Texas in Arlington. She was employed for several years as a Member of Technical Staff at NEC America in Irving, Texas. Her current research interests include software engineering, knowledge representation, ontologies, and the Semantic Web.