

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

VINICIUS MARCHANDT SOBRAL

**A MODEL-DRIVEN APPROACH TO THE CONCEPTUAL MODELING
OF SITUATIONS: FROM SPECIFICATION TO VALIDATION**

Vitória – ES, Brazil

2015

VINICIUS MARCHANDT SOBRAL

**A MODEL-DRIVEN APPROACH TO THE CONCEPTUAL MODELING
OF SITUATIONS: FROM SPECIFICATION TO VALIDATION**

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do título de Mestre em Informática.

Orientador: Prof. Dr. João Paulo Andrade Almeida

Coorientador: Prof^a. Dr^a. Patrícia Dockhorn Costa

Vitória – ES, Brazil

2015

VINICIUS MARCHANDT SOBRAL

**A MODEL-DRIVEN APPROACH TO THE CONCEPTUAL MODELING
OF SITUATIONS: FROM SPECIFICATION TO VALIDATION**

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do título de Mestre em Informática.

Aprovada em 28 de Outubro de 2015.

COMISSÃO EXAMINADORA

Prof. Dr. João Paulo Andrade Almeida - Orientador
Universidade Federal do Espírito Santo

Prof^a. Dr^a. Patrícia Dockhorn Costa – Coorientador
Universidade Federal do Espírito Santo

Prof. Dr. José Gonçalves Pereira Filho
Universidade Federal do Espírito Santo

Prof. Dr. Luís Ferreira Pires
Universidade de Twente, Holanda

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Setorial Tecnológica,
Universidade Federal do Espírito Santo, ES, Brasil)

S677m Sobral, Vinicius Marchandt, 1989-
A model-driven approach to the conceptual modeling of
situations: from specification to validation / Vinicius Marchandt
Sobral. – 2015.
119 f. : il.

Orientador: João Paulo Andrade Almeida.

Coorientador: Patrícia Dockhorn Costa.

Dissertação (Mestrado em Informática) – Universidade
Federal do Espírito Santo, Centro Tecnológico.

1. Ontologia. 2. Modelagem de informações. 3. Modelagem
de Situações. 4. Modelagem conceitual. 5. Validação de
modelos conceituais. 5. Verificação de modelos conceituais.
I. Almeida, João Paulo Andrade. II. Costa, Patrícia Dockhorn. III.
Universidade Federal do Espírito Santo. Centro Tecnológico. IV.
Título.

CDU: 004

ACKNOWLEDGEMENTS

I would like to thank my parents (Mariângela and Raul), for all the love, support, trust and eventual harsh words that turned into motivation that kept me going.

I thank my brother, Bruno, my best friend, for the times of joy, the talks (some very intellectual, some really stupid) and friendship.

I would like to thank my advisor, prof. João Paulo, and co-advisor, prof^a. Patrícia, for the knowledge shared, the discussions and the patience to have kept guiding me for those three years. I have learned a lot from you and I am honored to have worked with you both.

A special thanks to Izon Mielke, whose work served as solid basis for the development of mine.

Thanks John Guerson, João Moreira, Carlos Lins and Pedro Paulo for the advices, discussions and ideas that somehow contributed to this work.

Finally, I would like to thank each colleague that I had the pleasure of meeting in the university and living in a daily basis, especially in Nemo. Thanks Freddy, Tiago, Ernani, Victor, Cássio, Basseti, Layla, Filipe, Cadu (I did it!), just to name a few.

RESUMO

A modelagem de situações para aplicações sensíveis ao contexto, também chamadas de aplicações sensíveis a situações, é, por um lado, uma tarefa chave para o funcionamento adequado dessas aplicações. Por outro lado, essa também é uma tarefa árdua graças à complexidade e à vasta gama de tipos de situações possíveis. Com o intuito de facilitar a representação desses tipos de situações em tempo de projeto, foi criada a Linguagem de Modelagem de Situações (Situation Modeling Language - SML), a qual se baseia parcialmente em ricas teorias ontológicas de modelagem conceitual, além de fornecer uma plataforma de detecção de situação em tempo de execução. Apesar do benefício da existência dessa infraestrutura, a tarefa de definir tipos de situação é ainda não-trivial, podendo carregar problemas que dificilmente são detectados por modeladores via inspeções manuais. Esta dissertação tem o propósito de melhorar e facilitar ainda mais a definição de tipos de situação em SML propondo: (i) uma maior integração da linguagem com as teorias ontológicas de modelagem conceitual pelo uso da linguagem OntoUML, visando aumentar a expressividade dos modelos de situação; e (ii) uma abordagem para validação de tipos de situação usando um método formal, visando garantir que os modelos criados correspondam à intenção do modelador. Tanto a integração quanto a validação são implementadas em uma ferramenta para especificação, verificação e validação de tipos de situação ontologicamente enriquecidos.

ABSTRACT

The modeling of situation types for context-aware applications, also called situation-aware applications, is, on the one hand, a key task to the proper functioning of those applications. On the other hand, it is also a hard task given the complexity and the wide range of possible situation types. Aiming at facilitating the representation of those types of situations at design-time, the Situation Modeling Language (SML) was created. This language is based partially on rich ontological theories of conceptual modeling and is accompanied by a platform for situation-detection at runtime. Despite the benefits of the availability of this suitable infrastructure, the definition of situation types, being a non-trivial task, can still pose problems that are hardly detected by modelers by manual model inspection. This thesis aims at improving and facilitating the definition of situation types in SML by proposing: (i) the integration between the language and the ontological theories of conceptual modeling by using the OntoUML language, with the purpose of increasing the expressivity of situation type models; and (ii) an approach for the validation of situation type models using a lightweight formal method, aiming at increasing the correspondence between the created models' instances and the modeler's intentions. Both the integration and the validation are implemented in a tool for specification, verification and validation of ontologically-enriched situation types.

LIST OF FIGURES

Figure 1. Approach overview of the SML modeling improvement.	21
Figure 2. Context related to a user.	24
Figure 3. Intuitive vision of a context-aware application interacting with the user and its context (COSTA; ALMEIDA, 2007).	25
Figure 4. Example of situation instances in time (MIELKE, 2013).	27
Figure 5. Example of context model in the healthcare domain.	29
Figure 6. Fever situation.	30
Figure 7. Is Being Treated situation.	31
Figure 8. Hospital within Range situation.	31
Figure 9. Incompatible Treatments situation.	32
Figure 10. Allen relations and their converses (MIELKE, 2013).	33
Figure 11. Example timeline for situation Incompatible Treatments.	33
Figure 12. Intermittent Fever situation.	34
Figure 13. Example timeline for situation type Intermittent Fever.	34
Figure 14. Healthcare context model.	37
Figure 15. Inadequate Graduated situation example.	43
Figure 16. Example of Graduated situation using new SML elements.	43
Figure 17. Switch situation from (MIELKE, 2013).	44
Figure 18. Reviewed Switch situation type.	44
Figure 19. Risky NeedQuarantine situation illustrating the use of modes.	46
Figure 20. Fever situation reified in an OntoUML model.	47
Figure 21. Intermittent Fever situation reified in an OntoUML model.	48
Figure 22. Different possibilities of representing multiple participants and the respective reified representation.	49
Figure 23. Overload situation and respective interpretation.	50
Figure 24. Intermittent Fever situation and respective interpretation.	50
Figure 25. Has Any Ongoing Treatment situation and example timeline.	52
Figure 26. Suspicious Faraway Login from (COSTA et al., 2012).	53
Figure 27. Proposed Suspicious Faraway Login representation.	54
Figure 28. Approach for Qualitative Formal Relations.	56
Figure 29. Function example.	57

Figure 30. AccountUnderObservation from (MIELKE, 2013).	58
Figure 31. AccountUnderObservation with the self-reference node.....	58
Figure 32. Fragment of the SML metamodel depicting the main classes.....	60
Figure 33. Fragment of the SML metamodel showing referable elements and participants.....	61
Figure 34. Fragment of the SML metamodel showing situation type associations. ...	62
Figure 35. Fragment of the SML metamodel detailing nodes.	64
Figure 36. World Structure for simulation.	72
Figure 37. Intended and possible model instantiations adapted from (GANGEMI et al., 2005).....	82
Figure 38. Alloy Analyzer.	84
Figure 39. Simulation of fever situation.	86
Figure 40. Example of an inconsistent situation.	87
Figure 41. Alloy result example showing the situation is unsatisfiable/inconsistent. .	87
Figure 42. Run commands for detecting inconsistency.	88
Figure 43. Normal Fever situation.	89
Figure 44. Common Fever situation.	89
Figure 45. Alloy simulation for Common Fever and Normal Fever situations.....	90
Figure 46. No counterexample found. Valid assertion.	91
Figure 47. Run commands for detecting redundancy.	91
Figure 48. Healthy, Infected, Is Being Treated and Becomes Infected situations.	93
Figure 49. Possible Contagion situation.	93
Figure 50. Possible Contagion situation simulation (World 1).....	94
Figure 51. Possible Contagion situation simulation (World 2).....	95
Figure 52. Being Treated/Infected and correct Possible Contagion situation.....	96

LIST OF TABLES

Table 1. Relations between the SML and OntoUML metamodels.....	64
Table 2. Structural patterns in Alloy created from situation elements.	74
Table 3. Examples of participant's quantification.....	76
Table 4. Example of FormalRelation patterns.....	78
Table 5. Patterns for nodes representation in Alloy.	78
Table 6. Types Representation in Alloy simulation model.	85
Table 7. Linear Branch - scenario description.	112
Table 8. Alternative Futures - scenario description.....	112
Table 9. Counterfactual Worlds - scenario description.	113
Table 10. Branch Depth - scenario description.....	113
Table 11. Population Size - scenario description.....	114
Table 12. Population Variability - scenario description.	115
Table 13. Population Growth - scenario description.	116
Table 14. Extension Size - scenario description.....	117
Table 15. Temporal Extension Size - scenario description.	118
Table 16. Extension Variability - scenario description.....	119

LIST OF LISTINGS

Listing 1. AllenLink's type must be compatible with the participants' temporality.	65
Listing 2. A CharacterizationLink must connect the same entities as its Characterization does.	65
Listing 3. A ContextFormalLink must connect the same entities as its FormalAssociation does.	66
Listing 4. A MediationLink must connect the same entities as its Mediation does.	66
Listing 5. The maximum number of a Participant's instances must be greater than or equal the minimum.	66
Listing 6. Only a multiple participant (max > 1) may have an image.	67
Listing 7. The source of an AllenLink must be a SituationParticipant.	67
Listing 8. The target of an AllenLink must be a SituationParticipant.	67
Listing 9. The source of an AttributeLink must be a Participant, a ModeReference or a ReferenceNode, which in this case must be a reference to a Participant or a ModeReference.	67
Listing 10. The target of an AttributeLink must be an AttributeReference.	68
Listing 11. The source of a CharacterizationLink must be a ModeReference.	68
Listing 12. The target of a CharacterizationLink must be an EntityParticipant or a ReferenceNode, which in this case must be a reference to a EntityParticipant.	68
Listing 13. The source of a FunctionParameter must be a Function.	68
Listing 14. The source of an InstantiationLink must be an EntityParticipant, a RelatorParticipant or a ReferenceNode, which in this case must be a reference to an EntityParticipant or RelatorParticipant.	68
Listing 15. The target of an InstantiationLink must be a TypeLiteral.	68
Listing 16. The source of a MediationLink must be a RelatorParticipant.	69
Listing 17. The target of a MediationLink must be an EntityParticipant.	69
Listing 18. Skeleton structure of a generated Alloy model including situations.	73
Listing 19. Situation uniqueness predicate in Alloy.	73
Listing 20. Situation continuity predicate in Alloy.	74
Listing 21. Skeleton of a situation axiom represented as a fact in Alloy.	76
Listing 22. Fever situation complete rule in Alloy.	80
Listing 23. Run command example in Alloy.	87

Listing 24. Assertion for checking redundancy of situations.....	90
Listing 25. Generic assertion to verify redundancy.	91

LIST OF ABBREVIATIONS/ACRONYMS

DAML - DARPA Agent Markup Language

EMF – Eclipse Modeling Framework

GLOSS - Global Smart Space

OCL – Object Constraint Language

OIL - Ontology Interchange Language

OMG – Object Management Group

OLED – OntoUML Lightweight Editor

OWL – Ontology Web Language

RefOntoUML – Reference OntoUML

SML – Situation Modeling Language

SPARQL - SPARQL Protocol and RDF Query Language

UFO – Unified Foundational Ontology

UML – Unified Modeling Language

XML - Extensible Markup Language

CONTENTS

1	INTRODUCTION	15
1.1	MOTIVATION	16
1.2	OBJECTIVES	19
1.3	APPROACH.....	19
1.4	STRUCTURE	21
2	THEORETICAL BACKGROUND.....	23
2.1	CONTEXT, CONTEXT AWARENESS AND CONTEXT-AWARE APPLICATIONS	23
2.2	SITUATIONS.....	25
2.3	SITUATION MODELING LANGUAGE (SML)	28
2.4	CONTEXT MODELING, THE UNIFIED FOUNDATIONAL ONTOLOGY (UFO) AND ONTOUML	35
2.4.1	The Ontologically Well-Founded UML Profile	36
2.4.2	The OntoUML Infrastructure and Validation Framework	39
2.5	CONCLUDING REMARKS.....	40
3	REVISITING THE SML METAMODEL.....	41
3.1	EXTENSIONS REGARDING A MORE EXPRESSIVE CONTEXT MODEL.....	41
3.1.1	Addressing Modal Properties.....	42
3.1.2	Addressing Qualities and Modes	45
3.2	OTHER EXTENSIONS/MODIFICATIONS.....	46
3.2.1	Cardinality of Participants	48
3.2.2	Mutability of Participants.....	51
3.2.3	Attribute Links.....	52
3.2.4	Primitive, Formal and Allen Comparative Relations	54
3.2.5	Functions.....	56
3.2.6	Self-Reference Node	57
3.3	THE NEW SML METAMODEL	59
3.4	CONSTRAINTS	65
4	AUTOMATIC TRANSFORMATION.....	70

4.1	WORLD STRUCTURE	71
4.2	STRUCTURAL CHARACTERISTICS OF SITUATIONS	72
4.3	THE SITUATION MODULE	75
5	ASSESSMENT APPROACH	81
5.1	QUALITY CRITERIA AND PROBLEMATIC SITUATION TYPES.....	81
5.2	VALIDATION/SIMULATION SCENARIOS.....	84
5.2.1	The Alloy Analyzer.....	84
5.2.2	Inconsistency.....	86
5.2.3	Redundancy	88
5.2.4	Unintended states.....	92
6	CONCLUDING REMARKS	97
6.1	CONTRIBUTIONS AND CONCLUSIONS.....	97
6.2	RELATED WORKS.....	98
6.2.1	Ontology-based Situation Specification	98
6.2.2	Situation Validation.....	100
6.3	LIMITATIONS AND FUTURE WORK.....	100
	BIBLIOGRAPHY.....	103

1 INTRODUCTION

Costa (2007) defines *context-aware applications* as applications that are “capable of autonomously adapting their behavior in response to context changes”. By context we mean any real world circumstance that can be used to characterize the situation of an entity. In (DEY, 2001) the author uses the term *information* (context information), which we here take as the representation of this circumstance in a computer system, such as *in memory data*. The goal of a context-aware application is to supply the user with services that are more adequate to his/her current (or a projected) situation, without any human interaction. This kind of technology gained importance in the field called *ubiquitous* or *pervasive computing* (HANSMANN, 2003), which holds that computers must be transparently integrated to the everyday environment.

In order to provide this effective interaction, the designer of such a system must be able to answer questions such as: what are the relevant types of entities that exist in the user’s environment (or context)? What are the particular combinations of entities that are relevant to us? These questions lead, respectively, to two distinct definitions in the design of a context-aware application: one that describes the entity types and relations of the application’s domain, which we call a *context model*; and one that establishes the situation of interest to the application and their rules, given by the combination of the context entities, which we call a *situation model* or *situation type model* (MIELKE, 2013). For this reason, context-awareness is also entitled situation-awareness, where a *situation* is considered a higher abstraction concept that deals with conceptual patterns regardless of how context information is materially obtained. Throughout this thesis we use the expression *situation-awareness* as a synonym to *context-awareness* and *situation-aware application* as the same as *context-aware application*.

As discussed by Kokar et al. in (KOKAR; MATHEUS; BACLAWSKI, 2009), “to make use of situation awareness [...] one must be able to recognize situations, [...] associate various properties with particular situations, and communicate descriptions of situations to others”. As stated in (COSTA et al., 2012), enterprise systems can

profit from the notion of situation and its adequate support both at design-time and at runtime. At *design-time*, behavior and policies can be defined in terms of the types of situations in which they apply, instead of various low-level conditions. This not only fosters separation of concerns through abstraction but also enables the definition of complex situation types by reusing previously defined situation types. At *runtime*, situation detection machinery can be employed, enabling timely reaction to current situations. Examples of approaches for simplifying design-time situation modeling can be seen in (COSTA, 2007), (COSTA et al., 2012), (MIELKE, 2013), (KOKAR; MATHEUS; BACLAWSKI, 2009) and (BAUMGARTNER et al., 2010) while approaches for runtime situation detection are present in (PEREIRA; COSTA; ALMEIDA, 2013), (RAYMUNDO et al., 2014) and (KOKAR; MATHEUS; BACLAWSKI, 2009).

In order to leverage the benefits of the notion of situation at design time, Costa et. al. (2012) have proposed a model-driven approach to the specification of situation types (and ultimately model-driven realization of situation detection). That approach consists in part of a Situation Modeling Language (SML), which is a graphical language for situation modeling, allowing the expression of primitive situation types and complex situation types (with temporal constraints when required). This means that the designer is able to specify the types of situations in which he/she is interested at a high-level of abstraction. The definitions can then be used to generate situation detection code automatically in a platform called SCENE (PEREIRA; COSTA; ALMEIDA, 2013), a rule-based situation detection platform that leverages on JBoss Drools engine (and its integrated Complex Event Processing platform). The SML language serves as basis for the work developed throughout this thesis.

1.1 MOTIVATION

In this thesis we treat situation modeling as a conceptual modeling activity such as it is described in (MYLOPOULOS, 1992). In the referred work, the author states that model descriptions (or just models) must be created for the purpose of understanding and communication between humans, not machines. To provide this effective communication, those models must accurately reflect the modeler's intentions and be

consistent with the entities that exist in the domain, qualities that have been discussed and recognized by the traditional conceptual modeling community (MOODY et al., 2003). In addition to their role in understanding and communication, when used directly as computational artifacts, e.g. in a model-driven process, models are used to automatically generate software, affecting directly the behavior, data and functionalities of a system. Thus, semantic errors in the conceptual model imply possible problems in the generated system. For example, if a situation model is inadequate and used to generate a situation-aware implementation, the implemented system may fail to inform the user of a relevant situation or may fail to respond to a relevant situation. Further, the system may incorrectly detect situations even in circumstances in which they do not actually exist.

Computer tools, languages and frameworks are usually designed to offer flexibility and usability to the user. Modeling tools, in particular, facilitate the task of creating models and offer features to help creating *higher-quality* ones (i.e. one that is syntactically and semantically correct, adequate to its uses (functional), etc.), such as specific purposes menus and metamodels, validation mechanisms, simulation, among others. The SML language has been created with this purpose, aiming at simplifying the definition of situation types at the modeling level by means of a domain-specific visual language and providing integration with a situation detection platform at runtime. However, despite the benefits of the availability of a suitable modeling language and a code generation infrastructure, the definition of situation types, being a non-trivial task, can still bear problems that are hardly detected by modelers by manual model inspection. For example, since situations consist of particular combinations of context elements, their combinations into complex situations may lead to the creation of *inconsistent*, *redundant* or *unintended* situation type definitions. Those problematic definitions may result, respectively, in failing to detect an important situation, multiple responses for one single situation or in the detection of inexistent/unintended ones. Given the challenging nature of the situation modeling activity, designers could profit from additional support in order to assess the quality of the situation type models they produce.

Most of the aforementioned problems are related to semantic, domain-specific aspects of the situation models, which can only be avoided if a proper model assessment mechanism is employed at design-time. Model assessment is crucial for

the production of high-quality conceptual models in general, and is especially relevant if the models are employed in a model-driven approach, with the generation of deployable code from models. In our case, we generate situation detection code directly from situation type models in SML, hence the key role of model assessment. It allows model rectification at an early phase to make the created models less prone to error and to reflect accurately the modeler's intention.

Furthermore, the same complexity of creating accurate models encouraged deeper studies in conceptual representations that would most faithfully express the real-world entities and relationships. Thus, ontologies (GUARINO; OBERLE; STAAB, 2009) emerged as artifacts that materialized those studies and aimed at increasing the semantic expressiveness of models in the field of software engineering. Ontologies capture the subtleties of the different entities in a domain and are especially important (and widely adopted) for system interoperability, since they are formalized, allow reuse and substantially help the common understanding between people or software agents (e.g. allowing a more precise mapping of concepts between heterogeneous systems and reducing errors of false agreement). In (GUIZZARDI, 2005) the author presented ontological foundations that gave rise to modeling primitives for ontology-driven conceptual models, encompassing established works on areas that ranged from philosophy to linguistics. SML relies on some of those primitives for the creation of its context model, e.g. by distinguishing between entities and their contexts and specializing the latter according to other ontological distinctions.

Despite of this solid grounding, the SML language only uses a small portion of the ontological categorizations from the aforementioned works. Some important aspects were left aside making it less expressive when compared to other works in the area of ontology-driven conceptual modeling. As example, we can mention the lack of support for dynamic (or contingent) classification and for modal meta-properties of classes and associations such as rigidity and immutability. While this is not originally supported in SML, it is key to modeling certain situations in reality. For example, dynamic classification mechanisms can be used to represent situations concerning an entity's phases (such as a person's life phases: child, teenager and adult, a disease's phases: contagious, non-contagious) or the dynamics of role playing (such as a person's role through life: student, employee, husband/wife, patient).

1.2 OBJECTIVES

This thesis addresses situation type modeling enhancement with two primary objectives, namely extending the SML language and increasing its expressivity, and providing an approach to assess these enhanced situation type models.

With respect to the first objective, we aim at improving the expressivity of SML, integrating its context model with a consolidated ontology modeling language called OntoUML, whose meta-model has been designed to comply with the ontological distinctions and axioms of a theoretically well-grounded foundational ontology, named the Unified Foundational Ontology (UFO) (GUIZZARDI, 2005). This requires that we also extend/adapt SML's concepts to conform to the more expressive context model. With this, we seek to increase the quality and expressivity of the created situation types by using a context model that has clear real-world semantics while also expanding the possibilities of creation of situation models by including novel elements that can be combined to provide new constraint for the situation types.

With respect to the second objective, we propose an assessment method by using a lightweight formal method in an approach that is analogous to the one employed in (BENEVIDES et al., 2009) for ontology-based conceptual models in OntoUML. We use a simple but expressive logic-based language called Alloy (JACKSON, 2006), which is shipped with a sophisticated analyzer. In order to retain the ease-of-use of SML, the approach should minimize the need for the modeler to learn a new formal language. With this we seek to be able to identify problematic scenarios resulting from inconsistent, redundant situation types in a systematic and automated way, and scenarios resulting from unintended situation types for which we provide common examples that can be used in other domains, requiring a minor adaptation.

1.3 APPROACH

In order to achieve the integration between SML and OntoUML (our first objective) it is necessary to understand the commonalities between the two languages, i.e. identify the concepts with similar semantics. Since SML is divided in a context and a

situation model, this integration regards only the context model at first, allowing us to directly reference the respective OntoUML concepts from the situation model, thus using an OntoUML model as the context model. Besides identifying these commonalities, OntoUML brings novel semantic distinctions that have consequences to the situation models. Therefore, the SML language needs to be revised to accommodate the new definitions, which could result in expanding, modifying or excluding some of its constructs. Since we revise the SML metamodel, some opportunities for improvement beyond the integration with OntoUML are also identified and acted upon.

In order to provide an assessment method of situation type models (our second objective), we develop an automatic transformation of SML models into Alloy, using the existing OntoUML validation infrastructure and adding a *situation module* to it. The result of the transformation is an Alloy specification model which is then used to perform a series of simulation tests. In addition to simulation, it is possible to perform exhaustive verification of logical propositions in the Alloy model to detect problematic scenarios. Regarding the simulation, it is possible to check the correspondence between the instances of the written/drawn model and the original intentions of the modeler, to gain confidence that the allowed instances are only the intended ones. Regarding the verification, it is possible to locate inconsistent and redundant situation types, guaranteeing the validity of the situation models for the scope we define. The simulations we provide may be used as guidelines and reproduced in other domains with minor adaptations, while the verification is automated to completely validate the models regarding inconsistency and redundancy. This is important because not everyone is familiar with formal languages, especially with Alloy, and our intention is simplifying situation type modeling while also providing an effective method to validate situation models.

Figure 1 depicts the overview of the approach described. Transformation T1 between OntoUML and Alloy already exists and serves as bases for the transformation T2 which is developed in this thesis. The dependencies represented by the numbers (1) and (2) are novel contributions that we provide. Besides, we also contribute with the approach for simulation and validation of situation types, represented by (3), considering that the visual simulation already exists as part of the Alloy Analyzer tool.

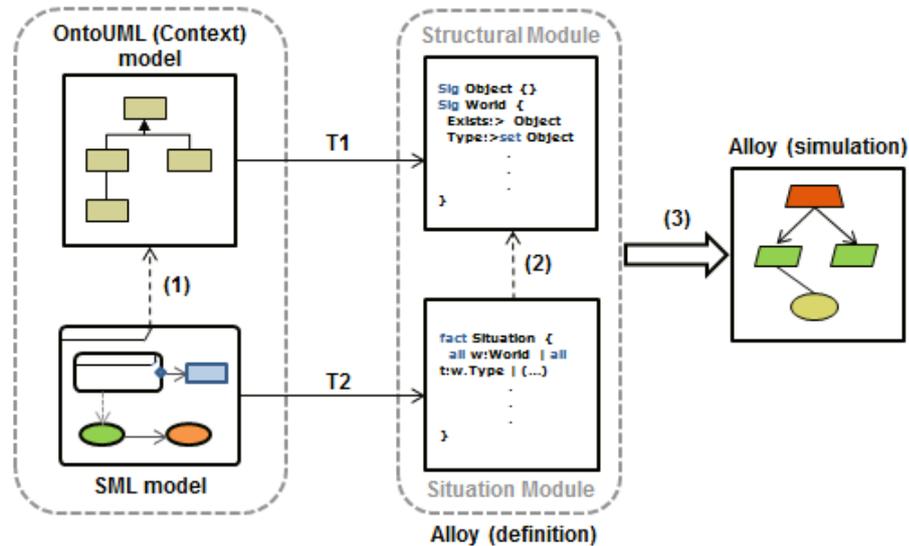


Figure 1. Approach overview of the SML modeling improvement.

Regarding the integration of metamodels we adopt Ecore as a metamodeling language, which is part of the Eclipse Modeling Framework (EMF)¹. Ecore was chosen since both metamodels were originally developed in this language, facilitating our integration. The choice for Alloy for the simulation and validation of situation types is a result of its successful application in the assessment of OntoUML models (BENEVIDES et al., 2009) (SALES, 2014), which means that, since we are also integrating the SML model with OntoUML, it also allow us to reuse the Alloy infrastructure created for the latter and extend it to support situations.

1.4 STRUCTURE

This thesis is further structured as follows:

- Chapter 2 compiles the most important background knowledge for the understanding of this thesis, including an overview of context, context-awareness and context-aware applications, the concepts of situation and situation types, the SML language, and UFO/OntoUML in the context of situation-aware applications;

¹ <https://www.eclipse.org/modeling/emf/>

- Chapter 3 revisits the SML metamodel to provide the aimed integration with OntoUML. We describe the novel features to be included in the language, depicting examples of situation types with these features, and present the resulting revised metamodel;
- Chapter 4 describes our transformation of SML models to Alloy, using the improved SML metamodel. We present input and output patterns that are the backbone of the transformation, so that the Alloy models preserve the semantics of the original SML models in the transformation;
- Chapter 5 presents an assessment example for detecting and correcting problematic situation types still at the modeling phase. The examples shown can be used as guidelines and replicated for a number of other situation types and domains;
- Chapter 6 presents our concluding remarks, discussing related work and proposing topics for further investigation;
- In Appendix A we show the complete Alloy situation module used in the transformation;
- Finally, Appendix B lists some simulation scenarios in Alloy taken from (SALES, 2014), which were inspired in modeling anti-patterns of ontology models in OntoUML.

2 THEORETICAL BACKGROUND

In this chapter we present the theoretical basis that is required for the development and understanding of this work. Section 2.1 introduces basic concepts in context, context-awareness and context-aware applications; section 2.2 explains the situation concept, which is the main concept used throughout the entire thesis; section 2.3 presents the Situation Modeling Language (SML), a domain-specific language for modeling situation types which is the core of this work and for which we present enhancements and an assessment approach; finally, section 2.4 introduces UFO, the foundational ontology that comprises the set of ontological theories for describing real-world phenomena, and OntoUML, the modeling language that reflects the conceptual distinctions underlying UFO and which we use as the new context model.

2.1 CONTEXT, CONTEXT AWARENESS AND CONTEXT-AWARE APPLICATIONS

Dey (2001) affirms that “Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.” However, we distinguish context from information and consider context to be a real-world circumstance (which still characterizes the situation of an entity), such as the real-world temperature of a Person, and then classify context *information* as the data extracted from this circumstance, e.g. a computer data representing this temperature. Thus, if a circumstance is used to characterize a participant of the user-application interaction it is context, and it is only relevant when applied to something that exists, which is called an Entity. Examples of context are the location of a user, the lighting of an environment, the direction of a mobile device, among others. Figure 2 illustrates an intuitive context model, showing a person and many contexts that can be associated to this person (proximity, location, network access through a device, etc.).



Figure 2. Context related to a user.

Context awareness deals with the ability of applications to use information about the user's environment (context) to activate services according to his/her current need/situation. The use of context makes an application more user-centered and provides that it offers more adequate services, improving the user experience. Since the beginning of the 90s contributions to the context awareness area have been made, particularly in the Artificial Intelligence community. Currently, with the development of mobile technologies and proliferation of portable multifunctional devices (e.g. smartphones, tablets, etc.), context has become a highlighted topic in Computer Science. It receives special interest in the *Ubiquitous/Pervasive Computing* field (WEISER, 1999), which supports the vision in which computation is transparently integrated to the common life environment.

"A *context-aware application* is a distributed application that adapts its behavior according to its users' context" (COSTA; ALMEIDA, 2007). In (DEY, 2001) the author presents a similar definition, however Dey also describes what is offered by a context-aware application: "A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task". Thus, a context-aware application may be understood as an application that, in order to offer relevant information and services to the user captures and makes use of context. An application can, for example, use the location of the user to inform him about the availability of nearby services such as restaurants, stores, etc.

Figure 3 shows an intuitive vision of a user, its context and the context-aware application.

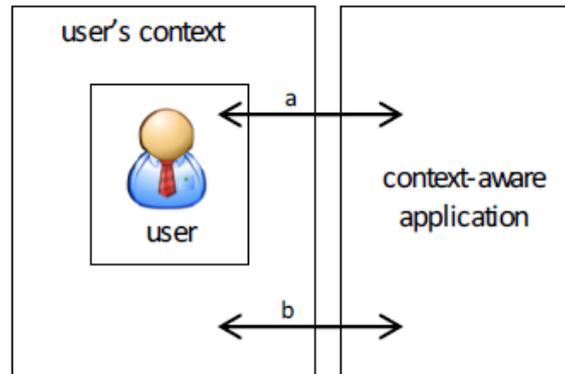


Figure 3. Intuitive vision of a context-aware application interacting with the user and its context (COSTA; ALMEIDA, 2007).

In Figure 3 the “a” arrow shows that the user and the context-aware application interact. Similarly, the “b” arrow indicates that the user’s context and the application also interact. An interaction represented by “a” may be inputs provided by the user, while “b” interactions may be the capturing of contextual information by the application autonomously.

In general, this kind of application stores contextual information (e.g. information captured by sensors) to infer higher-level abstraction contexts, which we call situations. Those situations are then used in specific decision making, be it presentation, services call or storage. The next sub-section describes in more details the situation concept.

2.2 SITUATIONS

A situation (COSTA et al., 2006) represents *state-of-affairs* that are of interest to a context-aware application. They are composite entities whose constituents are other context entities, their properties and the relations in which they are involved (COSTA; ALMEIDA, 2007). Situations support us in conceptualizing certain “parts of reality that can be comprehended as a whole” (ROSEMANN; RECKER, 2006). This notion enables designers, maintainers and users to abstract from the lower-level entities

and properties that stand in a particular situation and to focus on the higher-level patterns that emerge from lower-level entities in time. A situation-aware application, usually, is not only interested in the values associated to certain contextual information, but also in the meaning that this value represents. For example, an application may want to know if a person has a fever, i.e. if the temperature is above some value, not necessarily interested in the exact temperature value.

In our work, situations are composed by entities originated from an ontology model that describes a conceptualization, i.e. a view of a certain domain. A situation is itself a genuine ontological element composed by other elements (COSTA et al., 2006). As examples of situations we have that “John has a fever”, “John is running and has access to his cellphone”, “John is driving and is in danger of suffering an imminent epileptic attack”, “John and Paul are within 10 meters from each other” and “a suspicious withdrawal is occurring in account number 87346-0”.

Situations are frequently reified (BARWISE, 1989) (COSTA et al., 2006) or taken as a state of an object (KOKAR; MATHEUS; BACLAWSKI, 2009), which allows not only to identify the situation as a fact, but also to make references to properties of the situation itself. For example, it is possible to reference the duration of a specific situation, if it is currently occurring or if it is a past one. This allows inferring, for example, that a situation “John has fever” occurred yesterday and lasted two hours. The temporal aspect of a situation also allows making references to time changing, such as “John’s temperature is rising” or “bank account number 87346-0 has had suspicious withdrawals in the past 30 days”.

A situation type (KOKAR; MATHEUS; BACLAWSKI, 2009) is about considering general characteristics of situations of a particular sort. An example of a situation type is “Patient has fever”, which can be instantiated multiple times by patients like “John” and “Paul”, which in turn are said to “be febrile”. Therefore “John has fever” and “Paul has fever” are examples of instances of the situation “Patient has fever”. This example alerts for the need of referencing types and entities, like “Patient”, as part of the specification of a situation type. The many types that exist in a domain and may be referred to by a situation type must be specified in advance, for which reason we employ conceptual context models in addition to situation models. The same can be said to “be febrile” which, in this case, is defined in terms of entity’s properties that

instantiate the type *Patient* (attribute *corporal temperature*, also defined in a context model). Situation detection, i.e. situation type instantiation, requires detection of entity types' instances, whose properties satisfy restrictions set in the situation type.

A situation is said to be *active* while the properties of the entities that compose the situation satisfy the restrictions captured in the situation type specification. A situation ceases to exist when those properties no longer satisfy the defined restrictions. In this case, the situation becomes a *past* situation. Figure 4 provides a graphical representation of three situation's life cycle (S1, S2, S3), instantiating the same situation type. The vertical axis represents the possible states-of-affair of the domain entities. The horizontal axis represents the time passing.

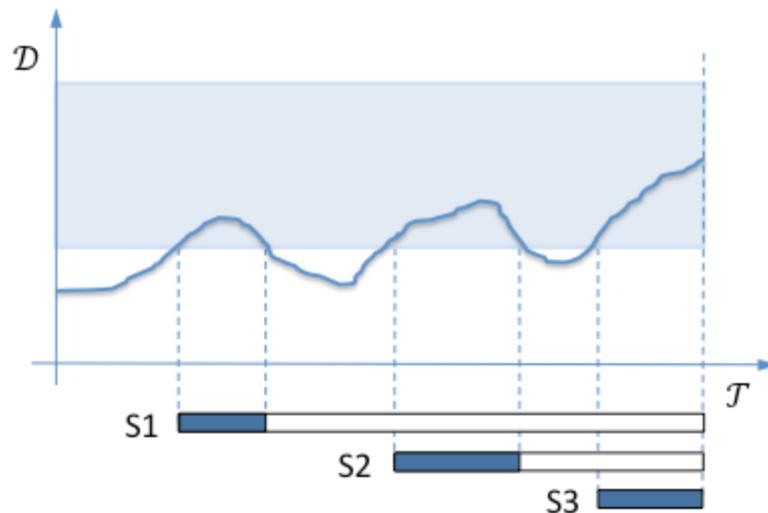


Figure 4. Example of situation instances in time (MIELKE, 2013).

For simplicity purposes Figure 4 only considers a one-dimensional property (e.g. temperature) of a unique entity (e.g. John of kind Person). When the involved temperature value hits a specific limit in the domain, established at the situation type specification (e.g. temperature greater than 37°C) an instance of this situation type is created (the situation becomes active). When the property no longer satisfies the restrictions of the situation type, the situation is deactivated, becoming a past situation (white area of S1). In case the property returns to meet the requirements of the situation type, a new instance is created.

2.3 SITUATION MODELING LANGUAGE (SML)

The Situation Modeling Language (SML) (COSTA et al., 2012) is a graphical language for modeling situation abstractions in a situation-aware application scenario. The language was created with the purpose of facilitating the definition of situations types at design-time. SML allows the expression of primitive situations and complex situations involving the composition of situations (with temporal constraints when required). A modeling infrastructure for the language was created, and is composed by a metamodel in the Eclipse EMF's Ecore language, a graphical editor, which is a model-driven Eclipse plug-in developed with the Obeo Designer tool, and an automatic transformation to a rule-based situation detection platform that leverages on JBoss Drools engine (and its integrated Complex Event Processing platform).

A *situation type* in SML is always a *derived type* which exists *iff* a specific derivation rule is satisfied. SML is a special language that allows us to establish those rules in a graphical manner, instead of using a logic proposition or other text-based definition. We call this graphical definition a *situation type definition* or *situation type model*. A situation type definition in SML is a composition of two kinds of models: a context model and a situation type model. The context model is a structural model that defines the classes of entities and relationships that exist in the modeled domain, which in turn are referred by the situation type model entities. In order to define context models, Mielke (2013) has proposed a novel metamodel which incorporates ontological distinctions resulting from researches in the conceptual modeling area. In turn, the situation type model defines situation as patterns of the context model classes' instances, and SML defines a concrete syntax for creating those models. Figure 5 represents a context model in the healthcare domain created with the mentioned distinctions and which is used in the next situation examples.

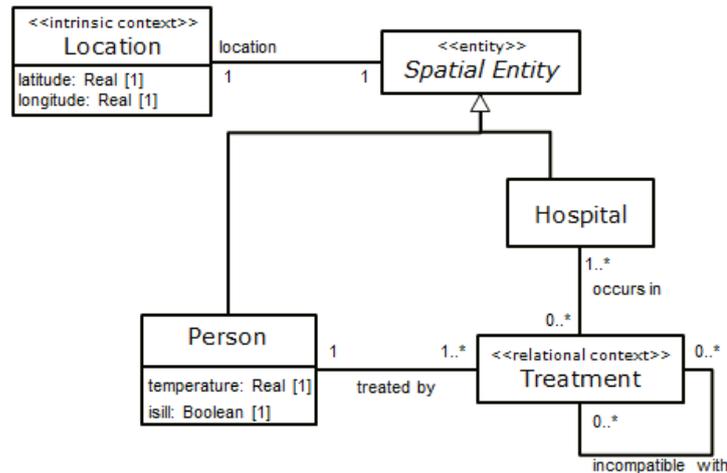


Figure 5. Example of context model in the healthcare domain.

In the example depicted there are two main entity classes, namely *Person* and *Hospital*, which specialize the abstract class *Spatial Entity*. The *entity* property (given by the respective stereotype) is passed down from the most general to the most specific elements in the hierarchy. Every *Spatial Entity* has a location, which is an *intrinsic context* or, using a more familiar term, an attribute of the respective entity, which means it inheres in its entity. Temperature is also an intrinsic context. Conversely, a *relational context* has its own identity, meaning they are genuine classes but dependent on two or more other elements, i.e. they do not inhere in none of them. For example, a *Treatment* is existentially-dependent of one or more *Hospitals* and one *Person*. Finally, a *Person* may or may not be on a treatment, which is stated by the optional cardinality (0..* where * stands for an unbound limit).

Figure 6 illustrates a *situation type* relevant in the healthcare domain, namely a *Fever* situation type, which is represented in SML as a rounded rectangle and happens when a person's temperature is above 37 degrees Celsius. The elements depicted in Figure 6 are references to the homonymous ones created in the context model. Each element composing a particular situation is shown inside the bordered rectangle that represents that situation. In this case we represent *Person* as an *Entity Participant*, which is modeled as a blue rectangle, along with its temperature's *Attribute Reference*, represented as a white ellipse. To constrain the temperature value to be greater than 37 degrees we represent a *Literal* with the respective value as an orange oval shape and connect a "greater than" *Comparative Relation* between

them. Finally, the small diamond at the border represents a *Situation Type Parameter*, indicating the entity Person may be referred by composite situations.

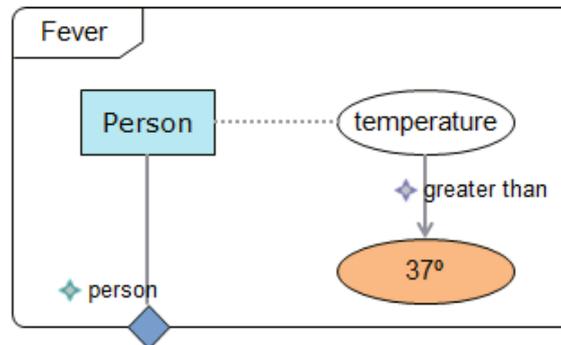


Figure 6. Fever situation.

Comparative Relations in the situation type model are the counterparts of the context model's comparative formal relations, which establish a relation between two entities according to a specific intrinsic context (such as temperature). Comparative relations simply restrict if an information holds or not, e.g. if a person is *older than* other or a value is *greater than* other. Some formal relations, such as "greater than", are *built-in* in the SML language and hold directly between any elements of the model (without any intervening element). Domain-specific formal relations may be introduced in the context model and referred in SML.

Figure 7 depicts the *Is Being Treated* situation. Two Entity Participants are represented there, namely a Person and a Hospital. The purple diamond represents a reference to the relational context Treatment, which is an instance of the so-called *Relator Participant* metaclass. It is connected to its respective Entity Participants, according to the relations established in the context model. This particular situation indicates simply that a Person has an ongoing Treatment occurring in some Hospital.

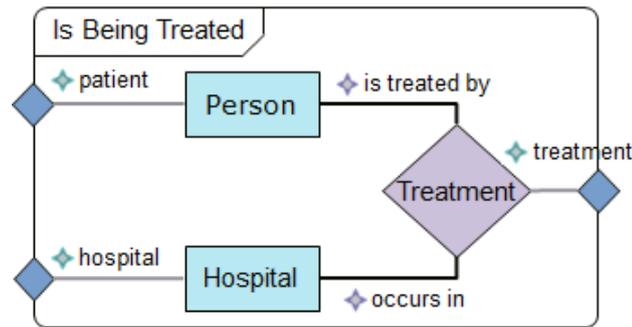


Figure 7. Is Being Treated situation.

Figure 8 shows the *Hospital within Range* situation. This situation is a relevant healthcare emergency situation, in which there is a possible risk state (here depicted by a person being ill) and the user is within some distance from a hospital. A situation-aware application could use it to promptly locate (e.g. using a GPS) nearby hospitals, enabling the taking of actions like contacting the hospital with an emergency message or calculating a route to it and sending to the user, allowing him to drive there immediately. This situation introduces the *Function* element, which is a reference to a formal association from the context model in which we can derive some value and use it for comparison. Distance is an example of this kind of association, as shown in Figure 8.

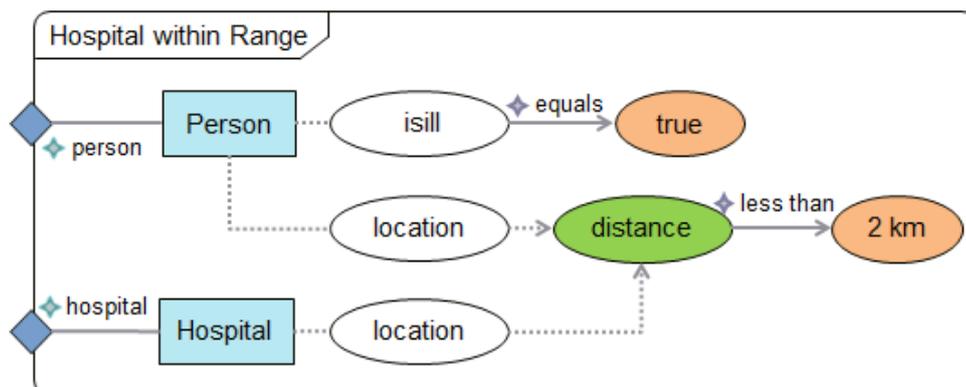


Figure 8. Hospital within Range situation.

Figure 9 illustrates an example of a situation type composed of another situation type. The occurrences of composing situation types are instances of the *Situation Participant* metaclass and are represented within the composite situation as nested rounded rectangles in gray. The *Incompatible Treatments* situation type is defined here with two occurrences of the situation *Is Being Treated* that *overlap* in time, for

the same person. *Incompatible with* is an example of domain-specific formal relation defined in the context model and used to constrain the situation type so that it only applies for risk situations where the treatments (actually it would be the treatment's types, but it is abstracted from the model since it is an intrinsic property of treatments related by a formal relation) are incompatible with each other.

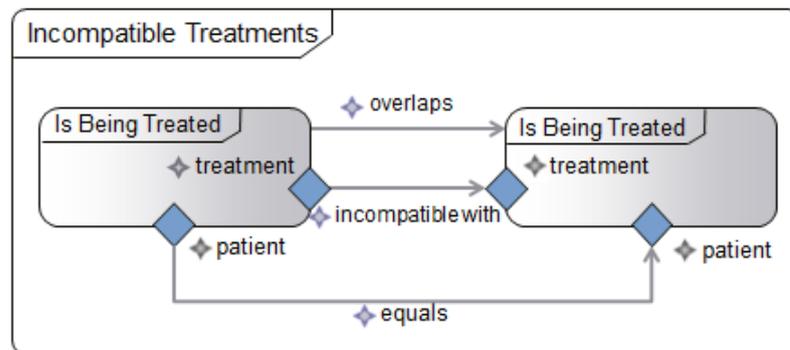


Figure 9. Incompatible Treatments situation.

The directed arrow *equals* connecting the bordering diamonds is another *built-in* formal relation that constrains the occurrences of *Is Being Treated* such that they must include the participation of the same person (regardless of the hospital used in the treatment). These bordering diamonds represent the *Situation Parameter Reference* metaclass and are references to the entities that participate in the situation which are of interest to the composite situation being defined, which in turn are defined by Situation Type Parameters.

The other directed arrow labeled with *overlaps* defines a constraint referring to a temporal formal relation between the situation type instances, in such way that both occurrences must overlap in time. SML allows composition of situations using the temporal formal relations defined by Allen (ALLEN, 1983), such as their converse relations, all which are shown by Figure 10.

	Ativa - Ativa	Ativa - Inativa	Inativa - Inativa
A Before B B After A		A  B 	A  B 
A Meets B B Met by A		A  B 	A  B 
A Overlaps B B Overlapped by A	A  B 	A  B 	A  B 
A Finishes B B Finished by A			A  B 
A Includes B B During A	A  B 	A  B 	A  B 
A Starts B B Started by A	A  B 	A  B 	A  B 
A Coincides B			A  B 

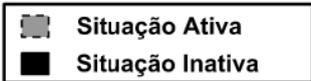


Figure 10. Allen relations and their converses (MIELKE, 2013).

Figure 11 depicts an example timeline for an instance of *Incompatible Treatments*, in terms of two occurrences of situation *Is Being Treated* (for the same person). In this example the situation only exists when both occurrences exist simultaneously.

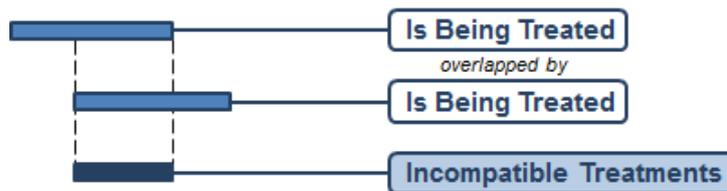


Figure 11. Example timeline for situation *Incompatible Treatments*.

Figure 12 depicts a more complex situation type called *Intermittent Fever*, in which we define additional constraints. The *Intermittent Fever* situation type is defined by two *Fever* occurrences (for the same person), in which the first occurrence must have ceased at most 2 hours earlier than the second. This temporal formal relation is specified by the directed arrow *before*, which is parameterized with lower and upper time limits (between 0 and 2h). *Past Situation Participants*, such as the first occurrence of Situation *Fever*, are graphically represented by nested rounded rectangles in white.

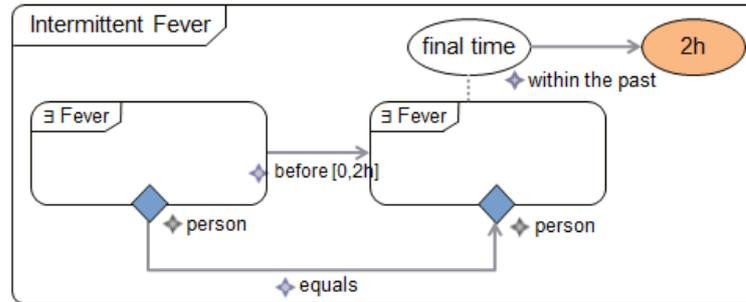


Figure 12. Intermittent Fever situation.

The icon for the existential quantifier (\exists) indicates that any instance of situation *Fever* for a given person can be matched here, as long as it respects the constraints in the situation type. In this case, both the past and the current composing situations have the existential quantifier, which is a requirement for the continuity of the situation.

Figure 13 shows an example timeline for an instance of *Intermittent Fever*, in terms of three occurrences of *Fever* situations, for the same person. The situation begins to exist when the second occurrence of situation *Fever* begins within 2 hours from the end of the first *Fever* occurrence. When the second instance ends, the *Intermittent Fever* instance continues to exist thanks to the constraint that indicates that one *Fever* instance should have occurred within the past 2 hours. As a third *Fever* begins, the *Intermittent Fever* situation is maintained with a new couple of *Fever* instances, thanks to the existential quantifier in the situation type rule. The existential quantifier indicates that any instance of a *Fever* situation that matches the constraints at a time will validate and participate in the situation. In short, the instances of *Fever* can change during the existence of the situation. The *Intermittent Fever* will only cease to exist when 2 hours have passed from the last *Fever* instance.

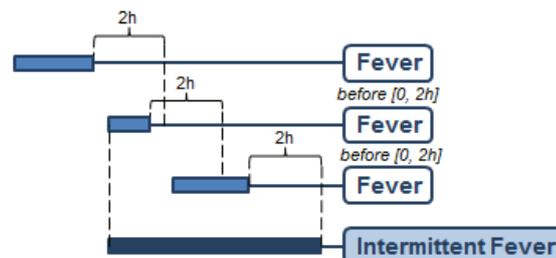


Figure 13. Example timeline for situation type Intermittent Fever.

Finally, SML also has a formal semantics which is defined in (COSTA et al., 2012). For every situation type model, it considers that the situation is instantiated when the

constraints defined in the situation type specification are satisfied, while it is finalized when these same constraints cease to be valid. Also, for every situation type, two axioms are assumed: the situation is unique for a particular set of entities in a particular point in time, which we address in this thesis as *situation uniqueness*; and if the set of entities remains stable in subsequent points of time, the situation is also the same, which we name *situation continuity*.

2.4 CONTEXT MODELING, THE UNIFIED FOUNDATIONAL ONTOLOGY (UFO) AND ONTOUML

According to (OLIVÉ, 2007), for an information system to perform its duties it must have some general knowledge about its domain and functions. Traditionally, we call this knowledge a conceptual model. Guizzardi (2005) reinforces the importance of conceptual modeling by saying that conceptual specifications are used to support understanding, problems solving and communication between the *stakeholders* about some specific domain. Once the understanding and accordance about a domain is achieved, the conceptual specification is used as a blueprint for the subsequent stages of the system development. In situation-aware applications we can say that the knowledge about the domain is represented by the context model, for it is used as a start point to the creation of situation representations.

Contextual modeling share similarities with conceptual modeling, in general, and ontology-based conceptual modeling, in particular. For instance, the concepts used in (COSTA et al., 2012) are derived from the modeling theories presented in (GUIZZARDI, 2005), which together forms the base of what we call the Unified Foundational Ontology (UFO). As we explained, we intend to extend the use of those theories in situation type modeling by using a more expressive context model, instead of the current one. We use OntoUML, a language which is also derived from UFO but allows the expression of important semantic distinctions that were not originally included in SML, such as dynamic classification of entities, e.g. the classification of a person regarding its life phases, such as Baby, Child, Teenager, Adult and Elder. Next we present the OntoUML language with a modeling example, describing the foundations from UFO underlying each concept.

2.4.1 The Ontologically Well-Founded UML Profile

UFO was presented in (GUIZZARDI, 2005), as ontological foundations for the most fundamental concepts in conceptual modeling. These foundations comprise a number of ontological theories, which are built on established work on philosophical ontology, cognitive psychology, philosophy of language and linguistics. UFO, whose concepts are used as a reference for the creation of ontologically well-founded models, is focused on providing foundations for the most fundamental and widespread constructs for conceptual modeling languages, namely, types and type taxonomies, roles, attributes, attribute values and attribute value spaces, relationships and part-whole relations. OntoUML is a language for constructing ontology-based conceptual models that was conceived using the concepts from UFO, which provides real-world semantics for the language constructs representing these concepts.

OntoUML can be seen as a *lightweight* extension of the Unified Modeling Language (UML), which is extended through its profile mechanism to support the new concepts, and uses a UML-like concrete syntax. It was first proposed in (GUIZZARDI, 2005) and updated in (ALBUQUERQUE; GUIZZARDI, 2013). The language has been successfully employed in a number of industrial projects in several different domains, such as Oil and Gas (GUIZZARDI et al., 2010), Complex Digital Media Management (CAROLO; BURLAMAQUI, 2011), Telecommunications (BARCELOS et al., 2011), and Government (BASTOS et al., 2011). Figure 14 represents an OntoUML model in the healthcare domain depicting some of its categorizations. The example model is used as reference in the examples throughout this thesis, although not preventing us from using other examples if judged necessary for better understanding.

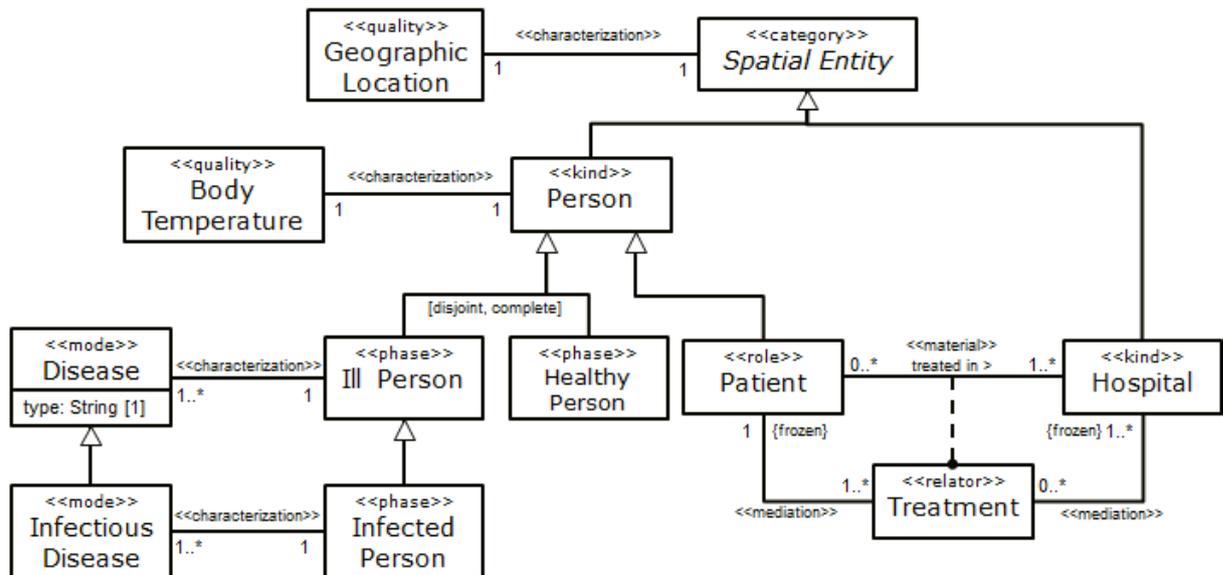


Figure 14. Healthcare context model

The model is a more expressive version of the one shown in Figure 5, since the categorizations included imbue the classes (or types) with extra characteristics. The *kind* stereotype in classes Person and Hospital states that they are *rigid* and provide *identity* for its instances. *Rigidity* regards the necessity (in the modal sense) of a class' instance to instantiate its class. A rigid classification is a static one, i.e. a rigid type's instance instantiates this type throughout its life. In the example given, a person (or hospital) can never cease to be a person (or hospital) without ceasing to exist. Other examples of stereotypes that carry the rigid property are *subkind* and *relator*, while a *category* (like the Spatial Entity class) aggregates rigid classes. The *identity principle* of an object is a sort of "function" that asserts whether two objects are the same or not. It is also what allows one to count objects. Sets (as in the mathematical notion) have one of the most simple identity principles: two sets are the same if, and only if, they contain the same elements. In OntoUML, objects must always follow a unique identity principle that cannot change throughout its existence. A subkind differs from a kind for it does not provide identity for its instances.

Following the classifications from the model, the *phase* stereotype in classes Ill Person and Healthy Person and the *role* stereotype in class Patient indicate that they are *anti-rigid*. Conversely from rigid classes, an *anti-rigid* instance may cease to instantiate its class at some point in time. In the modal sense, an anti-rigid instance

possibly instantiates its class or, more formally, a class *C* is anti-rigid iff for all its instances, there will be a possible world *w* in which they exist but do not instantiate *C*. Anti-rigidity represents dynamic (or contingent) classification, which was not possible previously in SML since it assumed a static (rigid) classification for all its entities. Thus, we could not specialize the type person as we did in Figure 14 since, for example, a person couldn't change from healthy to ill. The difference between roles and phases is that the former defines contingent properties of an instance exhibited in a relational context (e.g. a person is a patient contingently and only if it is having an ongoing treatment) while the latter defines changes in intrinsic properties of its instances (e.g. an person is ill thanks to an intrinsic characteristic, i.e. a disease). Finally, regarding the rigidity property and in addition to the mentioned stereotypes, OntoUML also defines *rolemixin*, which aggregates anti-rigid classes, and *mixin*, which aggregates both rigid and anti-rigid classes at the same time, establishing what we call a *semi-rigid* type.

Furthermore, the *mode* (Disease and Infectious Disease), *relator* (Treatment) and *quality* (Body Temperature and Geographic Location) stereotypes stand for what is called *moments* (or tropes) in OntoUML. Conversely to the classes presented so far, which are highly independent entities, moments are existentially dependent on the objects they exist in, i.e. their bearers, meaning that while they exist, their bearer's instances cannot change. In a common nomenclature, moments can be seen as the objectification of properties or attributes of objects. Relators are moments that represent objectifications of relational properties (e.g. the treatment is an entity that "connects" a patient and a hospital through the relation "is treated in"), whilst modes and qualities stand for moments that objectify intrinsic properties of the bearer and are also called *intrinsic moments*. Qualities are objectification of properties that evaluate (are projected) into a certain value space (e.g. mass, volume, color, body temperature, name, geographic location, etc.). Modes, conversely, cannot be directly evaluated in terms of a single value space, like someone's headache, intentions, and beliefs and so on.

The classification described implies the existence of different types of relations. For instance, *material* relations capture types of relations whose existence depends on relator types. The relation "is married to" between a husband and a wife is classified as material because it requires a marriage for it to be true (its truth-maker). Opposed

to the material relations, *formal* relations can hold between objects despite the existence of other entities. Those that are reducible to the comparison of values of qualities of the related objects are called *Domain Comparative Formal Relation (DCFR)*. For instance, the relation “being taller than” is a DCFR because it can be reduced to the comparison of the height property of two objects. Formal relations that cannot be reducible this way are further classified in: *characterization*, which stands for the inherence relation that holds between moments and the entities they characterize; *mediation*, which represents the relations between entities and the truth-makers of material relations; and *derivations*, which represent the relation between Material relations and their truth-makers, namely the relators.

2.4.2 The OntoUML Infrastructure and Validation Framework

In (CARRARETTO, 2010) the author presented an infrastructure to create OntoUML models, which was composed by a metamodel in the Ecore language and syntactical constraints to restrict the creation of models according to the rules of its foundational ontology. This infrastructure allowed the development of a model-based environment composed of many tools that improved the creation of the OntoUML model. Those tools functions range from model construction, verbalization and code generation to formal verification and validation (SALES; GUIZZARDI, 2014) (GUERSON; ALMEIDA, 2015) (ZAMBORLINI; GUIZZARDI, 2010). We are particularly interested in the validation framework for OntoUML (BENEVIDES et al., 2009), as we intend to extend it to include situation validation.

The framework uses the Alloy language to validate OntoUML models by using the Alloy language (JACKSON, 2006). Alloy is a logic language based on set theory, which is supported by an Analyzer that, given a context, exhaustively generates possible instances for a given specification and also allows automatic checking of assertions' consistency. In the OntoUML validation framework, the generated instances of a given conceptual model are organized in a branching-time temporal structure, thus, serving as a visual simulator for the possible dynamics of element creation, classification, association and destruction. It allows a modeler to visualize a representation of snapshots in this world structure, which are states admissible by

the models current axiomatization. This enables modelers to detect unintended, redundant or inconsistent model instances and take the proper measures to rectify them.

2.5 CONCLUDING REMARKS

In this chapter, we have presented basic background information that is necessary for properly understanding this work. In the following chapters, this knowledge will be important as we describe the work developed.

In chapter 3, we revisit the SML metamodel and investigate the concepts of UFO/OntoUML and SML to be able to achieve the aimed integration. We start by describing the features that we judge necessary given the inclusion of the OntoUML metamodel, showing examples of novel situation types that are only possible with the inclusion of the new features, and finish by presenting the new metamodel created in a metamodeling tool.

In chapter 4 and chapter 5, we present, respectively, the transformation of SML models to Alloy and the assessment approach for detecting and correcting problematic situation types in situation-aware applications. Both tasks rely on the OntoUML modeling infrastructure described in this chapter, which is extended to support situation type definitions, using the improved SML metamodel. In the transformation, specifically, we present input and output patterns that are the backbone of the transformation, so that the Alloy models are consistent with and reflect the same constraints as the original SML models.

3 REVISITING THE SML METAMODEL

The task of improving the expressivity of situation type models relies on the integration of SML with OntoUML. Although SML concepts, specifically its context model concepts, have been conceived based on ontological foundations, its context modeling capabilities are still not as expressive as languages such as OntoUML, whose elements are closest to the categorizations provided by UFO. Because of the difference in expressivity, we need to revisit the SML metamodel in order to achieve the aimed integration, adapting or including new concepts with the purpose of complying with an OntoUML context model.

Besides changing the SML metamodel to be in accordance with the OntoUML concepts, we also address other issues that we identified as being improvement opportunities for the language. Those other modifications are not directly related to the inclusion of the OntoUML model, but came up in modeling scenarios that were considered after the language was created. In any case, the list of improvements we describe next are a result of a series of exemplification and simulation of situation types, as well as the study of the intersections between the concepts of the old and new (OntoUML-based) context metamodels.

The examples we provide in this section use as reference the OntoUML model of Figure 14 (the healthcare context model presented in section 2.4.1). In section 3.3 we present the new SML metamodel that incorporates the solutions proposed.

3.1 EXTENSIONS REGARDING A MORE EXPRESSIVE CONTEXT MODEL

The integration with OntoUML brings new possibilities of situation type definitions. The novelties include the dynamic classification of entities, which results from the modality of the types in OntoUML, and the inclusion of the Mode stereotype, a category for intrinsic moments that was not addressed in the previous SML metamodel. Next, we describe the extensions that were made to SML that are the result of the inclusion of this more expressive context model.

3.1.1 Addressing Modal Properties

A particular and perhaps the most important benefit of the inclusion of OntoUML as context model is the reference to the classes' modal properties, and the possibilities that arise from dynamic classification of instances. While an instance of a rigid class instantiates that class for as long as it exists, an instance of a non-rigid (or anti-rigid) class may cease to instantiate the class without ceasing to exist. Such is true, for example, when a person who is a teenager becomes an adult, and later an elder, while still being the same person; or when a student from some educational institution graduates, thus ceasing to be a student while still existing as a person.

Dynamic classification can be partially dealt with currently in SML, e.g., we may use past situations to indicate whether a situation participant was an instance of some type in the past. Nevertheless, this is a limited solution, as we cannot address the cases in which a participant is *no longer* an instance of this same type in the present. The example of Figure 15, for instance, depicts an attempt to model a student's graduation situation using native SML. The *Enrolled* situation type in the example states that a person is enrolled in a university (he/she is a student), while the second situation type indicates there exists a person who was in an Enrolled situation in the past (he/she was a student). In this case, the past participation only states that the person involved was a student at some time in the past, not requiring that the person is no longer a student in the present.

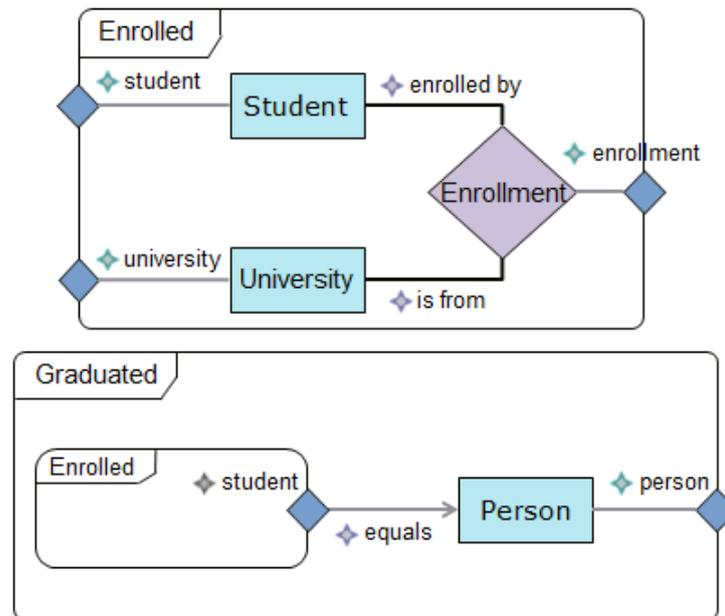


Figure 15. Inadequate Graduated situation example.

In the original version of SML, entities were not considered to be able to change their type during their existence. For that reason, it is not possible to model the exemplified situation using only the original language's constructs. We address this issue by proposing an *instance of* relation which allows to explicitly express whether an element instantiates or not a specific type at a particular point in time. It allows one to talk specifically about instantiations that no longer hold. Figure 16 depicts the Graduated situation, now including the information that the Person is no longer a student, indicated by the “negated” *instance of* relation (represented by the exclamation mark before the relation's name). The light yellow ellipse represents a new element, namely a *Type Literal*, used in instantiation relations to represent a type (not an instance, like a regular Entity Participant).

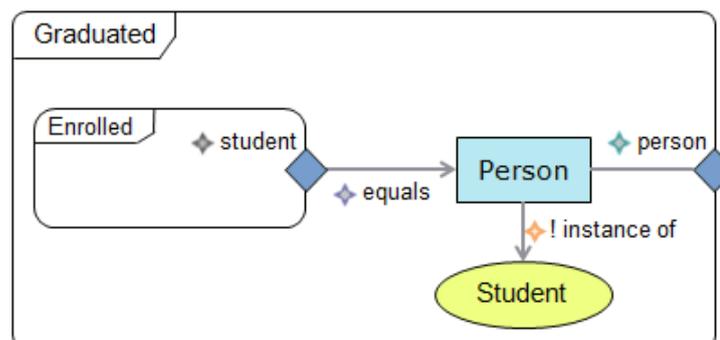


Figure 16. Example of Graduated situation using new SML elements.

Using instance of relations also allows solving a problem of representation of “past specialization” (OLIVÉ, 2001). In (MIELKE, 2013), the author presented the *Switch* situation, which was a remake of the homonymous situation presented in (COSTA, 2007). This situation is activated when a device switches its connection from a WLAN network to a Bluetooth network and is depicted in Figure 17.

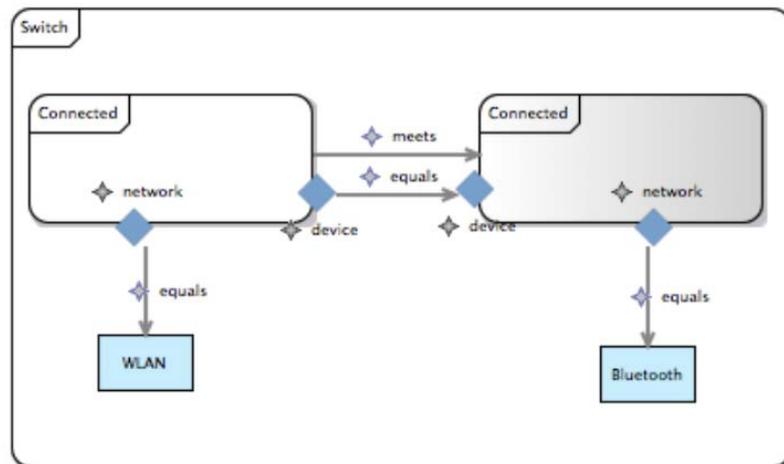


Figure 17. Switch situation from (MIELKE, 2013).

The example depicted, however, suffers from a side-effect, i.e. by representing WLAN and Bluetooth as entities in the situation type the modeler is requiring that both networks exist by the time of the occurrence of the situation Switch. This is not necessarily true since the WLAN network may have ceased to exist without compromising the situation. To avoid this kind of behavior, we propose using the instance of relation with a type literal, indicating that the entity inside the situation participant is of that respective type without requiring that the object instance continue to exist. The reviewed Switch situation model is depicted in Figure 18.

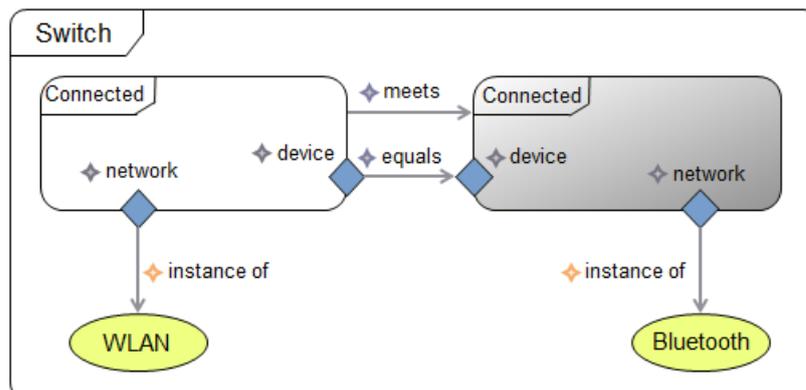


Figure 18. Reviewed Switch situation type.

3.1.2 Addressing Qualities and Modes

The inclusion of OntoUML as context model introduces specializations for the so-called *intrinsic context* in former SML, namely qualities and modes. Although the definition of intrinsic context in SML is the same as of intrinsic moment in UFO, it was mostly used to represent qualities, such as attributes like color or weight. Those former intrinsic contexts were then referred to in SML with the purpose of comparing it to a literal value, which represented a projection in the particular quality's value space (quality structure). The fever situation of Figure 6 illustrates this condition, where body temperature is an intrinsic context (Quality) and 37° is a value in the body temperature's quality structure. Another example of quality is the Geographic Location in our context model of Figure 14. We have chosen to support the representation of qualities both with the quality stereotype, introduced only recently in (ALBUQUERQUE; GUIZZARDI, 2013), and as attributes within their entities with a respective datatype as their types. Both representations are mapped to the same *AttributeReference* metaclass in SML, depicted as a white ellipse.

Nevertheless, modes, which are also intrinsic moments, cannot be evaluated in a value space as qualities can, making it impossible to measure or compare them to some specific value or to other entities of the same nature. For instance, consider a conceptualization in which we consider a person's "headache" as a mode. In this setting, someone's headache cannot be directly compared to a value literal. However, we may still want to compare particular characteristics of modes, such as their types, which are regardless of value spaces. For instance, in the context model of Figure 14 we have represented Disease as a mode that characterizes an ill Person and can be further classified as Infectious Disease, turning an ill Person into an Infected Person. Disease has an attribute named "type" to model the classification of the particular mode and allow comparison². In order to represent modes explicitly (in this case *Infectious Disease*), we introduce a novel construct in the notation (a red rectangle with rounded top borders) as shown in Figure 19. The situation depicted

² This is a common a workaround to avoid employing *powertypes* (OMG, 2011), which could have been an ideal representation in this situation (with different subclasses of Disease representing the various diseases). This was avoided here as support for powertypes in OntoUML is currently ongoing work.

address a risky state where a person is infected with a serious disease named “Ebola” and need immediate quarantine.

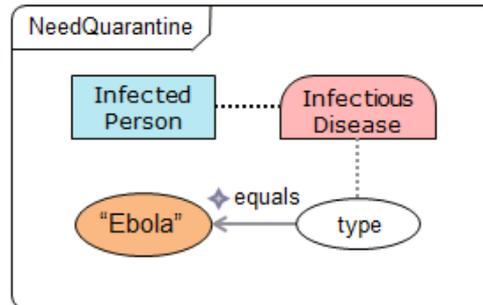


Figure 19. Risky NeedQuarantine situation illustrating the use of modes.

3.2 OTHER EXTENSIONS/MODIFICATIONS

When revising the SML metamodel, we identified other opportunities for improvement beyond the integration with OntoUML. Some of those other modifications are related to semantic problems on the representation of elements in the original SML, while others are based on an approach that consists of a reification of situation types in OntoUML models. This approach was inspired in the one presented in (SANTOS JÚNIOR, 2008), which proposed a representation for Events alongside OntoUML entities. In the referred work, Event types were represented as classes in the very same way as regular OntoUML entities. Similarly, we represent situation types as classes using the situation stereotype and their relations with their composing objects using the participation stereotype. Figure 20 depicts how a Fever situation type is represented along with a kind for a Person participant (in the left), and how a situation (Fever) can be also a participant of another situation (Intermittent Fever), in the right. It's important to mention that we do not aim to provide another representation for situation types besides SML, but we use this representation solely to analyze the integration between situation types in SML and other types in OntoUML. Whenever appropriate, insights obtained with this representation are used to suggest changes to SML's metamodel.

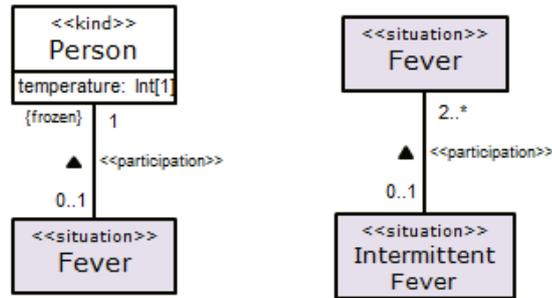


Figure 20. Fever situation reified in an OntoUML model.

The representation of reified situations in UML allows us to investigate the elements' meta-properties and the relations between a situation and its participants. With this, we intend to identify some element's and relation's properties necessary for the representation of situation types but which SML does not support or which are harder to observe in this language since it prizes for simplicity and readability. For instance, one may notice at a glance at the left of Figure 20 the multiplicities of the participation relation, which indicates that a Fever situation is linked to one person only, and a person may have none or at most one fever at a time. This interpretation of the first example is called *current semantics* (GUERSON, 2015), i.e. the cardinalities refer to the possible combinations at a *specific point in time*. OntoUML models are usually interpreted in current semantics, which is what we use throughout this thesis.

On the other hand, *lifetime semantics* interpretation is one in which the cardinalities refer to the possible combinations *throughout time*. For example, a person may have only one marriage at a time, but many throughout its life. At the right of Figure 20, an Intermittent Fever is connected to "two or more" instances of Fever. Since the same person cannot have two fevers at the same time, this representation indicates lifetime semantics, i.e. an intermittent fever is characterized by at least two fevers but may be "linked to", cumulatively in time, an *unbound* number of (intermittent) fevers. This characterizes a composite situation and, again, is solely for the purpose of analyzing the relations between OntoUML and SML, since we always interpret OntoUML models in current semantics. This unbound top limit, which is not representable in SML, represents all the possible *past fevers* that together characterize the instance of intermittent fever. The interpretation is derived from the Growing Block Universe theory (SIDER, 2006), which states that the past and present exists in the present, and only the future does not (e.g. Beethoven exists as a deceased person).

Therefore the worlds (and possibly some entities such as intermittent fever) are cumulative, as a block that always “grows”.

Next, we present the extensions that were identified when analyzing this representation, as well as the mentioned semantic related ones. We use the terms “*situation end*” and “*participant end*”, to identify respectively, the situation side of the participation association, i.e. the side which the “reading arrow” (►) points from, and the participant (sometimes also a situation, but in the role of a participant) side of the association, i.e. the side which the “reading arrow” (►) points to.

3.2.1 Cardinality of Participants

In our reified representation of situation types one can see that the participant end defines what we call the participant *cardinality*. In the example of Figure 20 we notice that this end’s cardinality is one, but it may not be so for every participant in every situation. If we take for instance an *Intermittent Fever* situation, defined as *one to many* occurrences of past fever situations and *one* current occurrence (analogous to the one in Figure 12), we would have a situation in a UML-like representation similar to the one depicted in Figure 21.

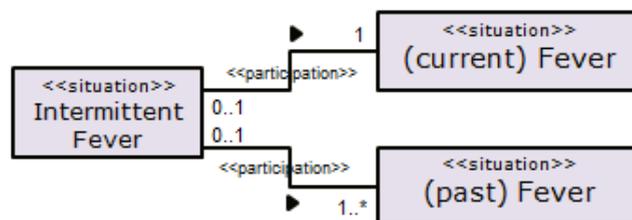


Figure 21. Intermittent Fever situation reified in an OntoUML model.

In the example given, the cardinality of the past fever situation is one to many. Although SML provides the *ExistsSituation* metaclass, which is equivalent to the logical existential quantifier, it does allow explicitly specifying a minimum and maximum number of a particular participant. Besides, although the *ExistsSituation* indicates that at least one instance of the situation participant exists (one to many cardinality), it also adds to the participant the property of being mutable, meaning that any instance of the respective participant’s type will validate the situation without

compromising its continuity. However, we argue that multiplicity and mutability should be addressed separately.

Furthermore, every extra participant of the situation needs to be represented with an exclusive instance of a situation participant construct. If we suppose a situation that requires a minimum number of participants of type A of 'n', we would have to create 'n' instances of participant A inside the situation type. It is not difficult to see that the situation definition would unnecessarily grow in complexity and become very hard to represent as the number of a particular participant increases. Added to that, we understand that the semantics of having different participants is that of having different participations, which is not necessarily true every time. Although modeling multiple instances of a participant is still possible, we have included the possibility of providing a minimum and maximum number when necessary. Figure 22 illustrates both possibilities and the respective interpretation with reified situation types. In (A) two situation participants indicate two participations. In turn, (B) introduces the multiplicity of the participant ([2..2]), which states a minimum and maximum of two and one single participation. The absence of the cardinality is interpreted as [1..1].

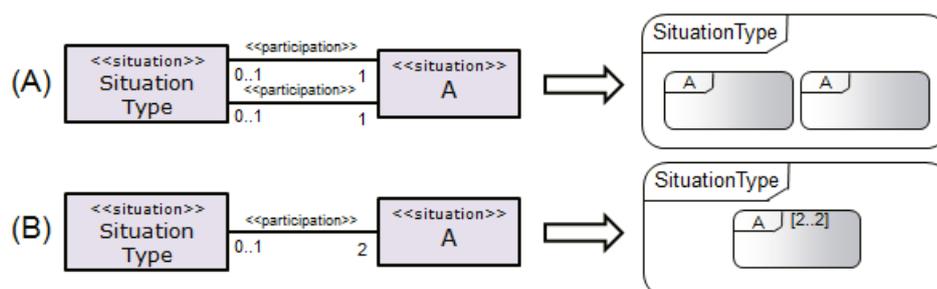


Figure 22. Different possibilities of representing multiple participants and the respective reified representation.

Essentially, multiplicity is used the same way as many instances of the respective participant in the situation type, meaning that a [2..2] participant represents basically the same as two [1..1] participants. The difference is that using multiplicity one can talk about an unbound number of instances. Figure 23 illustrates, in the left side, a situation where six or more people are inside an elevator, exceeding its capacity and characterizing an *Overload* situation.

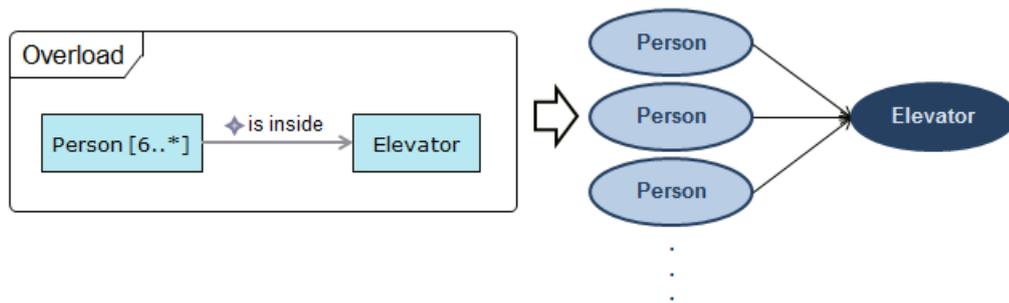


Figure 23. Overload situation and respective interpretation.

The situation of Figure 23 is interpreted, as it is reasonable, as if each person has a relation “is inside” with elevator, as demonstrated in the right side of image. However, it could be the case, although not in this particular example, where the multiple instances are required to be connected in a sequential manner. For instance, suppose a conceptualization where an intermittent fever situation is a sequence of fevers that occurs indefinitely with a time space of at most 2 hours from each other. This situation is depicted in Figure 24 at the top of the image while the respective sequential interpretation is depicted at its bottom.

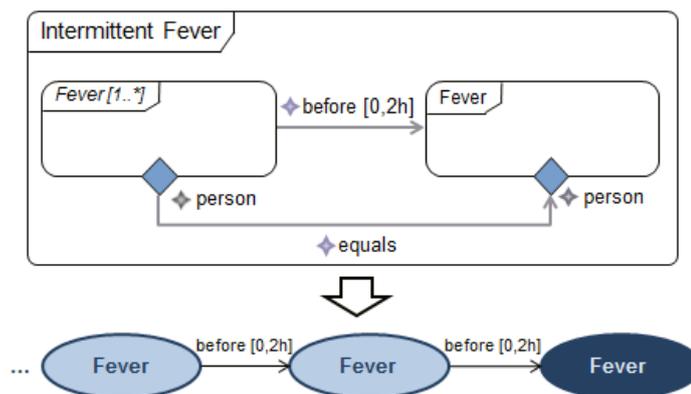


Figure 24. Intermittent Fever situation and respective interpretation.

To address this particular interpretation, which we consider that it is not the default one, we have included the *isImageOf* property of a participant. This property indicates that the representation is only an image of another participant. It is not interpreted as a genuine participant but as a replication (all its properties also have the same values as the ones from the participant they replicate) so that relations between the many instances (such as before and equals in the example) can be applied. In Figure 24 the situation participant in the left side is an image of the

situation participant in the right side (illustrated by an italic name) such that its replication is as depicted.

3.2.2 Mutability of Participants

In contextual models such as OntoUML, mutability is addressed apart from cardinality. An immutable association end is represented by the keyword *frozen* (see Figure 20), while its absence indicates a mutable one. In the original SML metamodel, mutability was addressed by the metaclass *ExistsSituation*, which was applied only to situation participants and stated that any occurrence of that composing participant may validate the situation constraints. Unlike the cardinality, who needed to be addressed separately for the purpose of specifying minimum and maximum numbers of instances, mutability is properly addresses by the *exists* logical quantifier, even though it apply the “at least one” instance rule. This happens because, usually, the combination of elements in the situation itself will restrict the number of entities to one, one or more and so on. Therefore, we map the former concept of the *ExistsSituation* to the *immutable* meta-property in the new metamodel, extending it to other participants as well, namely entity and relator participants.

As an example of the use of the immutable meta-property, one can consider a conceptualization where one wants to monitor whether a person is having a treatment or not. While an *Is Being Treated* situation occurs for every different treatment instance, the situation we want to create, which we will name *Has Any Ongoing Treatment*, would be satisfied by any occurrence of the treatment relator for the same person. Thus, the relator treatment would be mutable (the hospital could be mutable too or not, depending if the modelers wants to talk about treatments in different hospitals or in the same one only), as depicted at the left of Figure 25. An example timeline for this situation is illustrated at the right of the same figure.

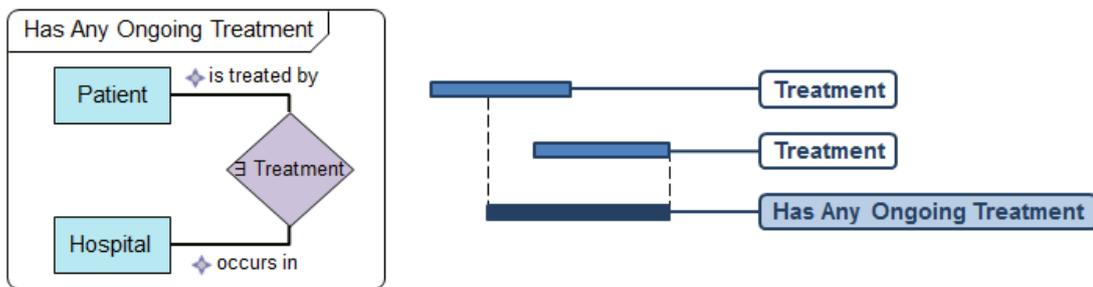


Figure 25. Has Any Ongoing Treatment situation and example timeline.

3.2.3 Attribute Links

Essentially, attributes are class' properties derived from the context model. In SML they are represented by the Attribute Reference metaclass and linked to its owning participant by means its *attribute* meta-property. The owning participant in turn is a reference to the respective attribute's owner in the context model and may be either an Entity Participant or a Relator Participant.

An example of usage of attributes is given in (COSTA et al., 2012), where the authors present the situation type model of Figure 26, representing a *Suspicious Faraway Login*. A situation of this type is instantiated when two accesses to the same account (depicted by the *Logged In* situation participants) occur in a period shorter than 2 hours, each access being done by one different device in a distance greater than 500km from each other. To establish the distance constraint, two new entity devices are represented and linked by the "equals" relation to the respective ones from the Logged In participants, so that the attribute location can be referred to.

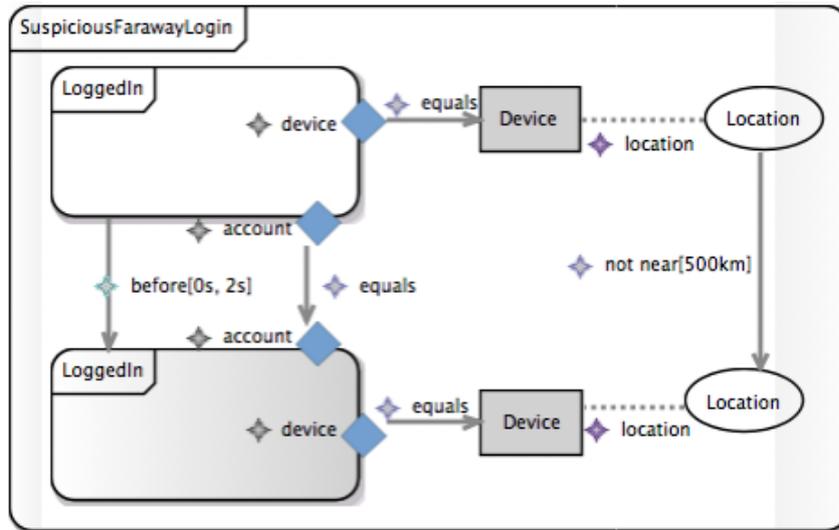


Figure 26. Suspicious Faraway Login from (COSTA et al., 2012).

This choice in order to refer to attributes has a side-effect, due to language's semantics: the Device entities participate now in the Suspicious Faraway Login situation, i.e., they are bound by the temporal space in which the composite situation occurs. Therefore, the situation type depicted constrains that, by the time the Suspicious Faraway Login situation is instantiated, two devices exist. We consider that this may be an undesired side-effect since the device from the past Logged In situation may no longer exist. Added to that, the particular devices where the logins happened are irrelevant to the situation being defined, since only their location matter. Thus, representing again the device's entities overpopulates the situation and may induce the reader to wrongly suppose that the presence of the device is important. Therefore, we propose a representation of the same situation type that would not include explicit entities, not requiring the presence of the entities in the composite situation timeframe. With this we may refer only to the properties that are relevant to the situation type, at the time they are relevant (in the former example the *past loggedin* location may be mistaken with the *actual* device's location, since, for example, a mobile's location changes frequently), allowing the designer to abstract from all other properties besides those. This proposed representation is illustrated in Figure 27, and requires changing the language's metamodel.

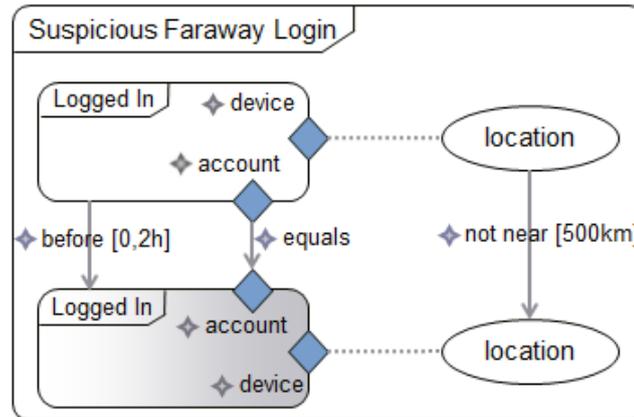


Figure 27. Proposed Suspicious Faraway Login representation.

3.2.4 Primitive, Formal and Allen Comparative Relations

In the current SML metamodel the *Comparative Relation* metaclass encompasses primitive (equals, greater than, less than), Allen (before, overlaps, meets, etc.) and domain-specific formal relations. This requires that primitive and Allen relations are defined in the context model, since a comparative relation must have a respective reference to it. However, when using OntoUML as a context model we have to be aware of the ontological implications of such a restricted model. Relations such as *equals* or temporal relations such as *before* and *overlaps* cannot be directly defined in an OntoUML model since the former, in theory, can be established between any classes and the latter can only be defined between situation types. In both cases, a general solution to define them as “context elements” would be to consider a pre-defined structure that comprises those definitions and which encompasses the user-defined model. So, for example, each modeled class would specialize *Thing*, which would then have a formal relation *equals* to itself. Similarly, this structure would include situation types as higher-level entities (such as a *Kind* or *Relator* for example), since Allen relations represent restrictions exclusively over those types. Besides going out of scope (it would need an investigation over powertypes in OntoUML or an extension of the language to include the Situation Type stereotype), we consider that a simpler approach such as to include those elements as primitives in the SML metamodel satisfactorily attends our needs.

Thus, we defend that primitive and Allen comparative relations should be specified in SML itself, so that there is no need to define them in the context model. This enables including syntactical constraints in the SML metamodel to restrict the elements that can be connected by those relations, which is not possible if they are defined in the OntoUML model. For example Allen relation could be constrained to only connect situation types and greater than/less than relations could be constrained to only connect structured properties (qualities). With this, we avoid making changes in the OntoUML metamodel and also seclude the former formal comparative relations and guarantee that they are exclusively references to the domain-specific formal relations defined in the context model. By separating and specializing each type of relation we can create specific constraints for each type and restrict the creation of wrong combinations of elements. An equals or other comparative relation could still be defined in the context model and used in SML as a “context defined” reference.

Furthermore, the distinction that existed between comparative and qualitative formal relations is dropped, as OntoUML does not distinguish one from the other. The former SML’s qualitative relations are challenging since they are not present in UFO’s and neither in OntoUML’s formalizations. However, we have decided to maintain references to this concept since it represents an important element in SML models. Thus, while comparative formal relations representation remains the same, we have defined two ways of addressing qualitative ones: the first one relies on the same representation for comparative formal relations, only that the reference in SML would require a manual parameterization of the relation, which would then be interpreted as a qualitative one; the second representation requires a modification of the OntoUML model to represent the qualitative formal relation as a ternary association with both the original elements and a third one being a datatype that stands for the result of the comparison, so that when referred in SML it would represent the parameter of the relation. Both approaches are depicted in Figure 28.

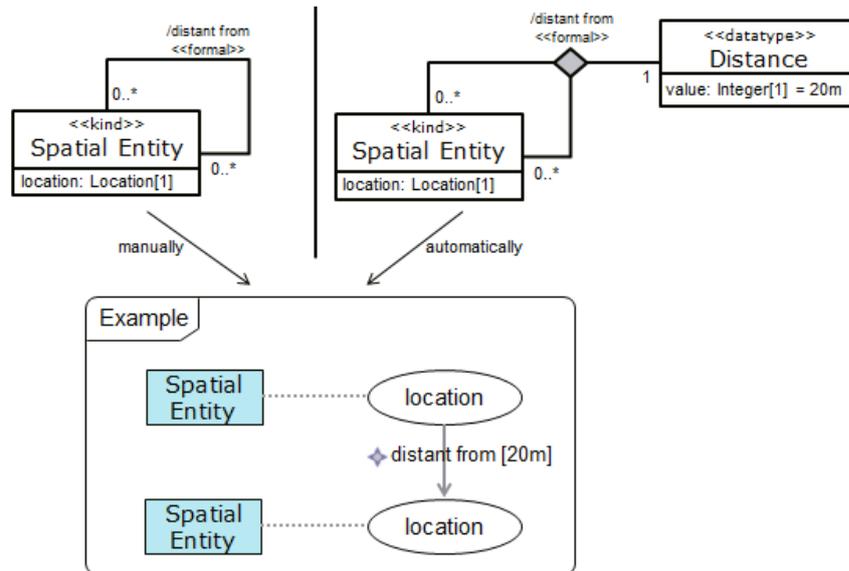


Figure 28. Approach for Qualitative Formal Relations.

3.2.5 Functions

Formerly references to the so-called Qualitative Formal Relations, *Functions* are re-designed as user-defined operations on the situation type elements. Not every relation between elements in the SML model is necessarily established in the context model. Take for example the *sum* of two numbers. Although in the real world, given two numbers there exists a formal sum relation between them, usually the ontology designer (in our case the context-model designer) leaves out those subtleties that may not aggregate much in the description of a domain. This means establishing a modeling scope so that the model does not grow indefinitely in complexity, as technically the possibilities of talking about things are infinite. Thus, relations like *sum* may always be defined in the context model if wanted, but we also leave open the possibility of defining them as functions (without linking to any context model elements) in the SML model itself. It is important to mention, however, that functions must be manually implemented if some automatic process is to be used (such as simulation in Alloy) or some code is to be generated from the models. This happens because, since functions are user-defined, it is impossible to establish a pattern for each possible function. Figure 29 depicts an example of a *High Body Mass Index (BMI)* situation using functions.

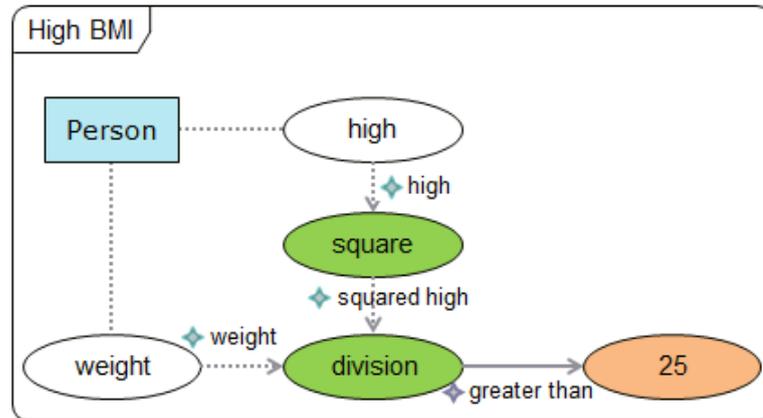


Figure 29. Function example.

3.2.6 Self-Reference Node

Sometimes we may want to constrain the time in which a past situation participant occurred without having a present situation to establish a temporal relation. Formerly, as shown in (MIELKE, 2013), a past situation that occurred at some point in the past 30 days, for example, needed a specific relation named *within the past*, such as demonstrated in Figure 30. This situation type model establishes that an instance of *Account Under Observation* will occur during a period of 30 days, starting right after an instance of *Ongoing Suspicious Withdrawal* ceases to occur. Looking for a more generic way to address this constraint (without the use of specific formal relations), we provide that situation participants may be temporally constrained with respect to the situation type being defined.

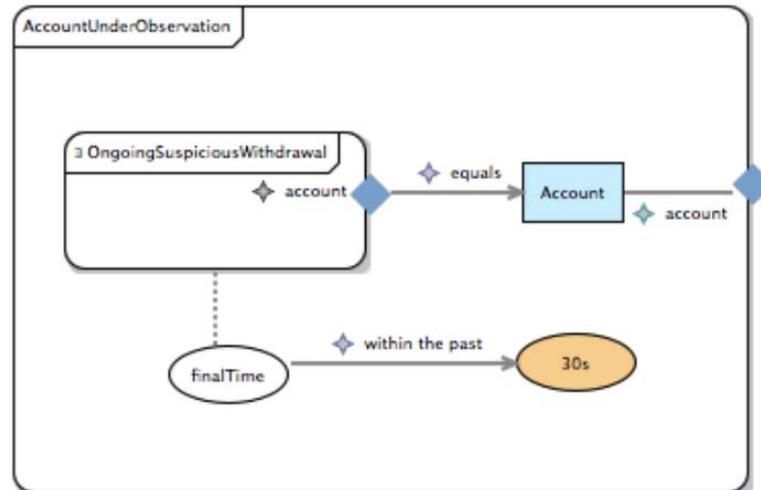


Figure 30. AccountUnderObservation from (MIELKE, 2013).

The relation *within the past* cannot be specified at the context model since it is a specific relation for situation types and the context model does not support them. Thus it needs to be defined at the situation model as a formal relation, which we consider a workaround. Since Allen relations are the base in SML to constrain the timing between situations, we propose a solution that uses those relations when a specific constrain with the present time must be established. The situation type being defined is reified as a situation participant represented by the keyword *self*. The Allen relation would then be established between the past participant and this *self-participant* and parameterized to indicate the desired time space, situation that is illustrated in Figure 31. Since *self* is always the current situation, this would mean a temporal relation between the past situation and the current time.

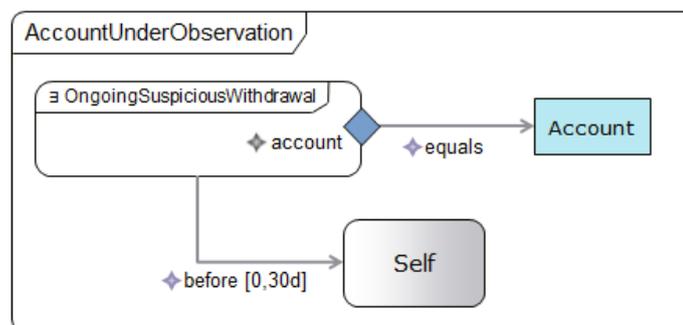


Figure 31. AccountUnderObservation with the self-reference node.

3.3 THE NEW SML METAMODEL

Here we will present the altered SML metamodel, considering the improvements discussed previously. We have used the Eclipse EMF Ecore language to create the metamodel following the choice of language of the previous metamodel. Besides, we have included some syntactical constraints to restrict the creation of situation type models. This allows the user to verify the models for construction mistakes, an approach that is similar to the one used for OntoUML models. Thus, we aimed at increasing SML models' reliability. We have also defined operations to facilitate the navigation of the model; these operations are used in the transformation of SML models.

A situation type is defined by the entities that participate in it, the associations involving those entities (in the form of extrinsic contexts, i.e. Relators), by the restrictions over the values associated to intrinsic contexts (in the form of comparative relations), restrictions over the type of entities (in the form of instantiation relations) and by some user defined functions. The SML metamodel takes this definition as foundation and is a base to the code generation for simulation in Alloy, which will be presented in chapter 4. The SML metamodel presented here makes direct references to the concepts in the OntoUML metamodel, binding the situation elements to the ones previously defined in the context model.

Figure 32, Figure 33, Figure 34 and Figure 35 show partial views of the SML metamodel. The classes depicted will be explained subsequently to the illustration of the model fragments.

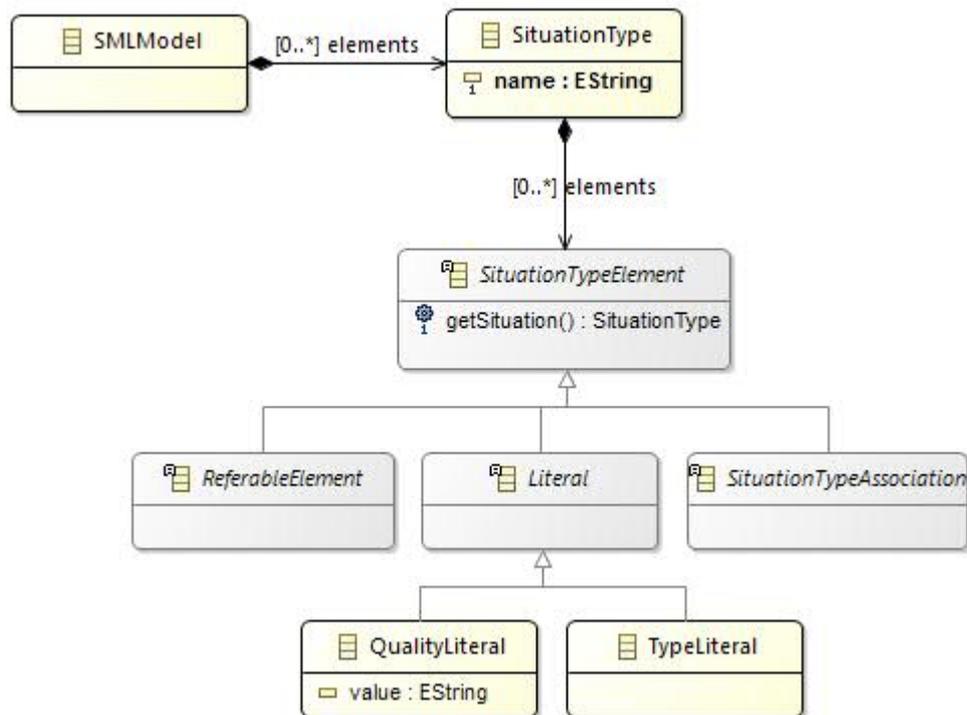


Figure 32. Fragment of the SML metamodel depicting the main classes.

A *SMLModel* is SML's top-level container class and represents the situation model. It is composed by zero-to-many *SituationTypes*. Those, in turn, represent the specification of a single situation type by means of its composing *SituationTypeElements*. A *SituationTypeElement* is every element that appears in a situation type and that are directly owned by a *SituationType*. They are further specialized in: *ReferableElement*, *Literal* and *SituationTypeAssociation*.

ReferableElement is an abstract metaclass that encompasses the elements that can be referred to by composite situation types. This metaclass will be detailed in Figure 33. A *Literal* either represents a Quality's value (*QualityLiteral*, but can also represent a Datatype's value) such as \$1000.00 or 37°, or some Type (*TypeLiteral*) used in instantiation relations to indicate whether an entity is or is not an instance of that class. Furthermore, *SituationTypeAssociation* is an abstract class that represents the connections that can be made between the nodes of a situation type. It will also be detailed in Figure 34. The *ReferableElement* metaclass' specializations will be explained next.

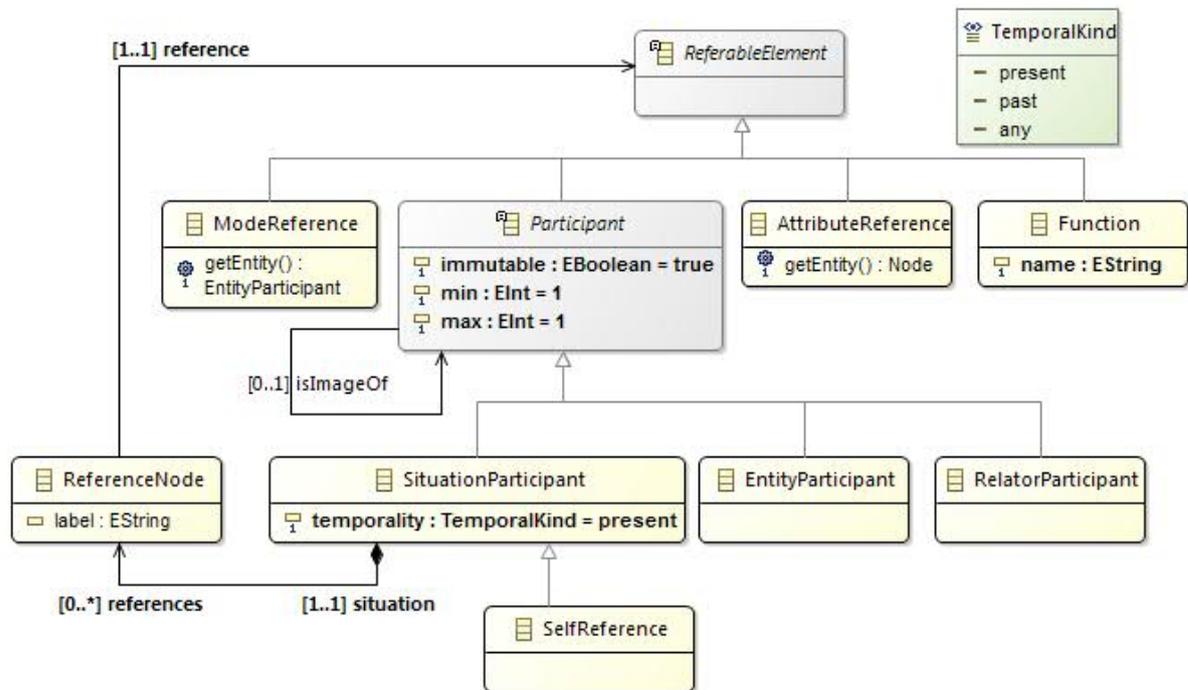


Figure 33. Fragment of the SML metamodel showing referable elements and participants.

A *Participant* is the main element inside a situation type. Participants can be multiple (by setting their *min* and *max* values) and/or immutable. When a participant is multiple it can be duplicated in a situation type (by setting the *isImageOf* property) so as to allow the creation of constraints over all the multiple instances. A *Participant* can be specialized in the following types:

- *SituationParticipant*, which represents other situation types used as a composing situation such as the fever in an intermittent fever situation. Situation participants have references to the situation types they represent. They can be past situations or current ones and are composed of *ReferenceNodes*. A *SelfReference* is a special kind of participant that represents the situation type being defined so as to allow the use of Allen relations between it and other situation participants.
- *EntityParticipant*, which represents a class from the context model, such as a person, a car, a client, a building and every other class that is not represented by a moment stereotype (Relator, Quality and Mode).
- *RelatorParticipant*, which represents the external contexts of the object classes, i.e. the relators. They are called participants since they provide

identity to the situation type, but they are not connected to them by *participation* relations, in the ontological sense.

A *ReferenceNode* is a node that represents a referable element from a composite situation type. It is used as a proxy of the element it refers to, such that it can be used in any relation that the referred element can. Reference nodes are used to compose *SituationParticipants*. An *AttributeReference* is a reference to a quality (also can be a reference to an attribute) defined in the context model. Analogously, a *ModeReference* is a reference to a mode defined in the context model. Finally, *Functions* materialize relations between elements that may not be defined in the context model because of scope limitation. They can represent any operation between the situation type elements such as a calculus or a derivation function in which some information is returned and used for comparison.

Next we detail the *SituationTypeAssociation* metaclass, which represents the connections used to relate the elements of a situation type model. Situation type associations are always binary, i.e. they have a source and a target, but the respective ends depend on the type of association. *Nodes* are elements that can be connected by situation type associations (detailed in Figure 35) and at first every association connects nodes, but constraints and operations may set specific sources and targets.

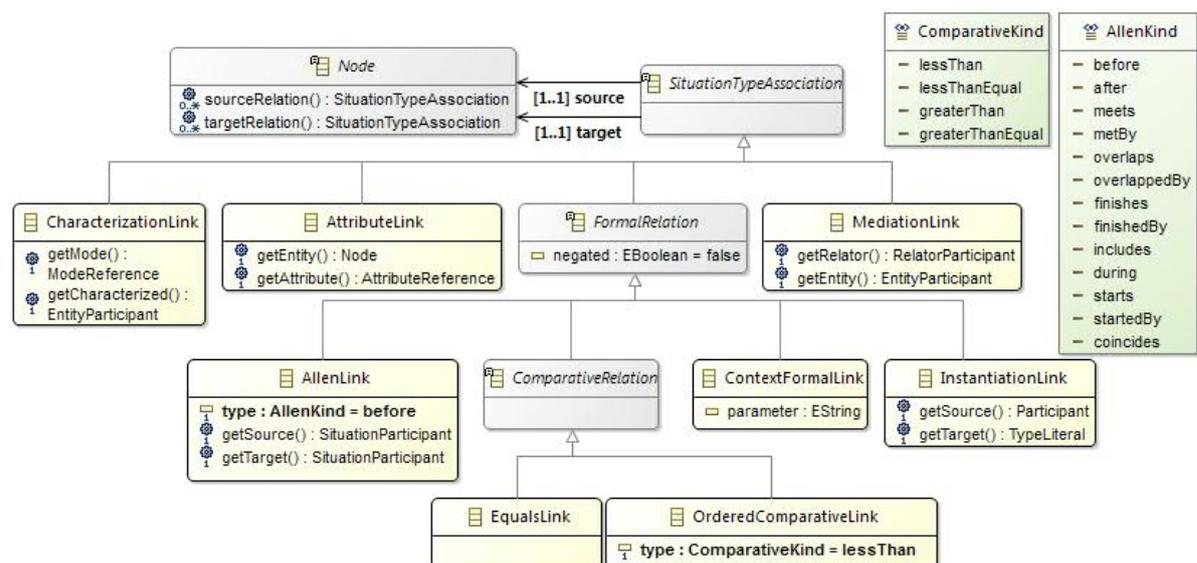


Figure 34. Fragment of the SML metamodel showing situation type associations.

A *CharacterizationLink* represents the occurrence of a connection between a mode and its respective characterized entity (or a node representing that entity). An *AttributeLink* is a connection between an *AttributeReference* and any node that represents a participant that may have an attribute. Those nodes are restricted to *EntityParticipants*, *RelatorParticipants* and *ReferenceNodes* that references one of the former. Furthermore, *FormalRelations* represent a relation that can be established between two elements and that depends only on the intrinsic properties of those elements. Formal relations can be negated or nor and are divided into the following specific connections:

- *AllenLink*, which represents a connection specifically between *SituationParticipants* that represents one of the Allen temporal relations, defined by the *AllenKind* enumeration.
- *ComparativeRelation*, which represents a comparison connection between elements and are divided into: *EqualsLink*, which may be established between any elements and indicates whether two elements are equal; and *OrderedComparativeLink*, which may only be established between qualities that have a value order, such as *high* or *temperature*, and are defined by the *ComparativeKind* enumeration.
- *ContextFormalLink*, which is a formal relation derived from the context model and is used to connect only the respective elements connected by it in this model.
- *InstantiationLink*, which represents the connection that restricts the type of a particular entity or relator participant. This connection is derived from the generalization relations from the context metamodel.

Finally, a *MediationLink* represents the occurrence of an association between a relational context and an entity, e.g. the relations between Treatment and Patient and Treatment and Hospital in the situation *Is Being Treated* from Figure 7.

Figure 35 shows all the subclasses of the *Node* metaclass. It depicts all the classes that can be related by some situation type association, although they may not be instances of *SituationTypeElement* and may not be directly children of situation types (the case of *ReferenceNode*). Figure 35 also introduces the *FunctionParameter* element, which represents the input parameters of a particular *Function*. Those

parameters may refer to any node in the situation type and are represented in the concrete syntax as connections between the node and the function.

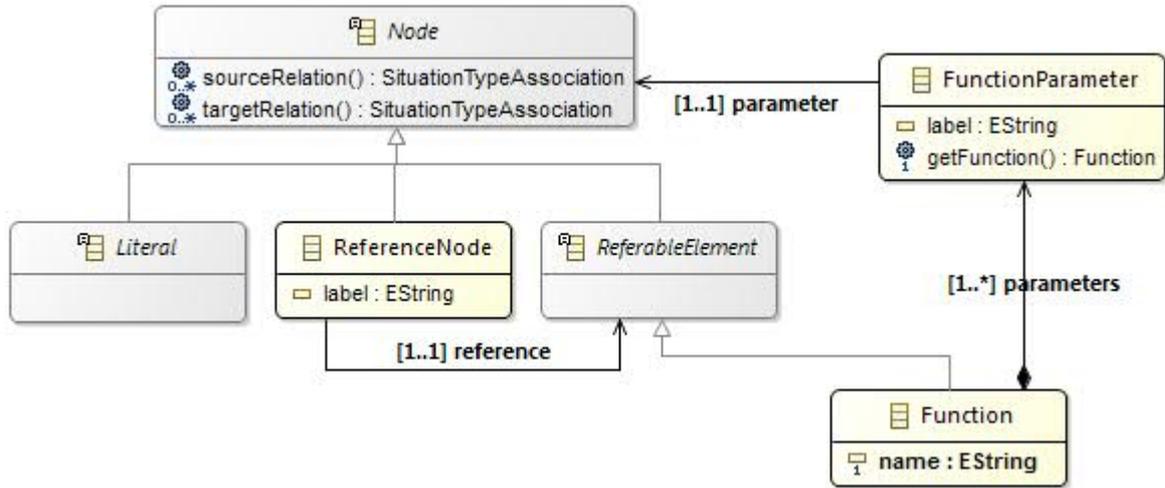


Figure 35. Fragment of the SML metamodel detailing nodes.

Besides the relations depicted, some classes of the SML metamodel are connected to classes of the context metamodel, i.e. the OntoUML metamodel. Those connections are Ecore references to the respective context classes. The relations between these metamodels are listed in Table 1.

Table 1. Relations between the SML and OntoUML metamodels.

Element in the SML metamodel	Reference name in the metaclass	Elements in the OntoUML metamodel
SMLModel	<i>contextModel</i>	Model
AttributeReference	<i>type</i>	Quality/Property
ModeReference	<i>type</i>	Mode
TypeLiteral	<i>type</i>	Class
ContextFormalLink	<i>type</i>	FormalAssociation
QualityLiteral	<i>type</i>	ReferenceStructure/DataType
EntityParticipant	<i>type</i>	<i>ObjectClass</i>
CharacterizationLink	<i>type</i>	Characterization
MediationLink	<i>type</i>	Mediation
RelatorParticipant	<i>type</i>	Relator

3.4 CONSTRAINTS

This section contains the set of constraints that restrict the ways the elements can be related and provide conformance to the structure established in the context metamodel. The goal is to facilitate the situation modeling activity so that the modeler can verify his model syntactically to detect structural errors and provide correction. Those constraints are written in the Object Constraint Language (OCL) (OMG, 2012).

Listing 1 presents a constraint that binds the creation of Allen relations between situation participants so that only compatible relations can be established. For instance, a past participant cannot occur after a present participant and two present participants cannot coincide (since at the present they are not finished yet).

Listing 1. AllenLink's type must be compatible with the participants' temporality.

```

1  context AllenLink inv
2
3  if (self.getSource().temporality = TemporalKind.present)
4  then if (self.getTarget().temporality = TemporalKind.present)
5      then self.type = AllenKind.overlaps or self.type =
AllenKind.overlappedby or self.type = AllenKind.includes or self.type =
= AllenKind.during or self.type = AllenKind.starts or self.type =
AllenKind.startedby
6      else self.type = AllenKind.after or self.type = AllenKind.metby
or self.type = AllenKind.overlappedby or self.type =
AllenKind.includes or self.type = AllenKind.startedby
7      endif
8  else if (self.getTarget().temporality = TemporalKind.present)
9      then self.type = AllenKind.before or self.type =
AllenKind.meets or self.type = AllenKind.overlaps or self.type =
AllenKind.during or self.type = AllenKind.starts
10     else true
11     endif
12 endif

```

Listing 2 binds that a CharacterizationLink connects only the elements (the ModeReference and EntityParticipant) that its respective characterization relation does (respective Mode and Class) in the context model.

Listing 2. A CharacterizationLink must connect the same entities as its Characterization does.

```

1  context CharacterizationLink inv
2
3  self.type.oclAsType(RefOntoUML::Characterization).characterizing() =
self.source.oclAsType(ModeReference).type and

```

```

4 self.type.oclassType(RefOntoUML::Characterization).characterized() =
  self.target.oclassType(EntityParticipant).type

```

Listing 3 binds that a `ContextFormalLink` connects only the elements that its respective Formal Association does in the context model. In this case the types of the elements are not established since formal associations connect classes of any type.

Listing 3. A ContextFormalLink must connect the same entities as its FormalAssociation does.

```

1 context ContextFormalLink inv
2
3 self.type.oclassType(RefOntoUML::FormalAssociation).memberEnd-
  >exists(x,y | x = self.source.type and y = self.target.type)

```

Listing 4 binds that a `MediationLink` connects only the elements (the `RelatorParticipant` and `EntityParticipant`) that its respective Mediation relation does (respective `Relator` and `Class`) in the context model.

Listing 4. A MediationLink must connect the same entities as its Mediation does.

```

1 context MediationLink inv
2
3 self.type.oclassType(RefOntoUML::Mediation).relator() =
  self.source.oclassType(RelatorParticipant).type and
4 self.type.oclassType(RefOntoUML::Mediation).mediated() =
  self.target.oclassType(EntityParticipant).type

```

Listing 5 binds the creation of multiple participants such that the minimum and maximum numbers are compatible (e.g. minimum not greater than maximum).

Listing 5. The maximum number of a Participant's instances must be greater than or equal the minimum.

```

1 context Participant inv
2
3 if(self.max <> -1)
4 then self.max >= self.min and self.min <> -1
5 else self.min <> -1
6 endif

```

Listing 6 establishes that an image participant, used when talking about many instances of a same participant, is only used for participants that are not unique (minimum and maximum only 1).

Listing 6. Only a multiple participant (max > 1) may have an image.

```

1 context Participant inv
2
3 if(self.isImageOf <> null)
4 then self.isImageOf.max > 1
5 else false
6 endif

```

The next constraints are restrictions over the types of the new metamodel association's source and target. This is necessary because of our modeling choice of creating a general SituationTypeAssociation element that connects two Node (most general class) elements, thus providing a source and target references for all the classes that specializes it. Therefore, to avoid creating relations between wrong types, those constraints are necessary.

Listing 7. The source of an AllenLink must be a SituationParticipant.

```

1 context AllenLink inv
2
3 self.source.ocIsKindOf(SituationParticipant)

```

Listing 8. The target of an AllenLink must be a SituationParticipant.

```

1 context AllenLink inv
2
3 self.target.ocIsKindOf(SituationParticipant)

```

Listing 9. The source of an AttributeLink must be a Participant, a ModeReference or a ReferenceNode, which in this case must be a reference to a Participant or a ModeReference.

```

1 context AttributeLink inv
2
3 self.source.ocIsKindOf(Participant) or
  self.source.ocIsKindOf(ModeReference) or
  (self.source.ocIsKindOf(ReferenceNode) and
   (self.source.ocAsType(ReferenceNode).reference.ocIsKindOf(Par
    ticipant) or
4   self.source.ocAsType(ReferenceNode).reference.ocIsKindOf(Mode
    Reference)))

```

Listing 10. The target of an AttributeLink must be an AttributeReference.

```

1 context AttributeLink inv
2
3 self.target.ocIsKindOf(AttributeReference)

```

Listing 11. The source of a CharacterizationLink must be a ModeReference.

```

1 context CharacterizationLink inv
2
3 self.source.ocIsKindOf(ModeReference)

```

Listing 12. The target of a CharacterizationLink must be an EntityParticipant or a ReferenceNode, which in this case must be a reference to an EntityParticipant.

```

1 context CharacterizationLink inv
2
3 self.source.ocIsKindOf(EntityParticipant) or
  (self.source.ocIsKindOf(ReferenceNode) and
4   self.source.ocAsType(ReferenceNode).reference.ocIsKindOf(Enti
  tyParticipant))

```

Listing 13. The source of a FunctionParameter must be a Function.

```

1 context FunctionParameter inv
2
3 self.target.ocIsKindOf(Function)

```

Listing 14. The source of an InstantiationLink must be an EntityParticipant, a RelatorParticipant or a ReferenceNode, which in this case must be a reference to an EntityParticipant or RelatorParticipant.

```

1 context InstantiationLink inv
2
3 self.source.ocIsKindOf(EntityParticipant) or
  self.source.ocIsKindOf(RelatorParticipant) or
  (self.source.ocIsKindOf(ReferenceNode) and
4   (self.source.ocAsType(ReferenceNode).reference.ocIsKindOf(Ent
  ityParticipant) or
5   self.source.ocAsType(ReferenceNode).reference.ocIsKindOf(Rela
  torParticipant)))

```

Listing 15. The target of an InstantiationLink must be a TypeLiteral.

```

1 context InstantiationLink inv
2
3 self.target.ocIsKindOf(TypeLiteral)

```

Listing 16. The source of a MediationLink must be a RelatorParticipant.

```
1 context MediationLink inv
2
3 self.source.ocIsKindOf(RelatorParticipant)
```

Listing 17. The target of a MediationLink must be an EntityParticipant.

```
1 context MediationLink inv
2
3 self.target.ocIsKindOf(EntityParticipant)
```

4 AUTOMATIC TRANSFORMATION

In order to simulate situation type models in Alloy the modeler must be able to represent the situation axioms in the language, requiring knowledge in logics and in the tool's particularities. To avoid the necessity of this learning step and prevent human flaws in the translation we have developed an automatic *model-driven* transformation from SML model to specifications in Alloy. The transformation uses as basis the SML metamodel described in chapter 3 and is further detailed in this chapter. Later, in chapter 5, we will demonstrate how the generated Alloy specifications can be used to validate situation type models and simulate worlds with situation type instances in order to assess those models. We present an assessment approach that requires no further knowledge in logics, as the transformation does all the translation work. The transformation was developed in Java, since all the infrastructure of Eclipse EMF is based on this language, and it is fully automated and implemented in the OntoUML Lightweight Editor (OLED)³, a tool originally created to facilitate the creation of OntoUML models but that is being extended to provide support for situation types in SML as well.

The resulting Alloy specification is divided in two parts: a *structural module* and a *situation module*. The structural module mostly contains the world structure, entities declarations and their properties along with some accessibility functions. The structural module is derived from the OntoUML validation framework from (SALES, 2014) and we will refrain from explaining it besides presenting the base world structure in section 4.1 and the parts that need to be included in order to support situation types in section 4.2. The situation module contains the rules that give identity, uniqueness and continuity to the situation types and will be explained in section 4.3.

³ <https://github.com/nemo-ufes/ontouml-lightweight-editor>

4.1 WORLD STRUCTURE

To be able to check the possibilities of instantiation and destruction of entities, the transformation we use considers a frame-based structure based on the Kripke semantics (KRIPKE, 1963). This world structure is inherited from the OntoUML's Alloy validation framework proposed in (BENEVIDES et al., 2010) and is in accordance with the formal semantics of SML presented in (COSTA et al., 2012), which considers that a frame or world represents a possible instantiation of the model (possible *state-of-affairs*) in a given moment, i.e. a model snapshot, according to the formalization described. The world structure is entirely described in the OntoUML part (the structural module) of the Alloy model description and has the following properties:

- Instances of classes *exist in* a world and can be related to other instances through the instantiation of associations. We refer to the set containing all individuals that exist in a world as its *population*.
- The exists in relation is non-empty, i.e. every world must have at least one instance of a model's type in order to avoid empty world instances. Conversely, every top-level type must exist in at least one world so that each type is validated against the model's predicates in every simulation.
- Worlds are accessible from each other through succession (*next*) relations, which are *asymmetric*, *intransitive* and *irreflexive*. It means that a successor world is one that, from a given state of affairs, identified by the first world, a sequence of events can occur leading to the second world. In the branching structure used in this work, every world can have at most one predecessor, but any number of successors, in order to capture the idea that the future may unfold in various ways, and allow for counterfactual analysis.
- A world branch is a set of sequential worlds. Every world in a branch either is accessible or can access any other world in the same branch, directly or indirectly. However, it is not admissible for worlds to access past states (time cannot go back).
- It is forbidden for an instance to exist again in a branch if it ceases to exist at some past point. This means that every instance's existence is always continuous.

There are four world categories: Past, Future, Counterfactual and Current worlds. A *current world* stands for the current state of things, an analogy to the present time, and every generated branch in the simulation has exactly one current world, randomly chosen. *Future worlds* present possible state of affairs that can become true if we continue to move through time from the current world. *Past worlds*, conversely, are the ones were true and led to the current world. They present the outcomes of a series of events that lead to the current setting. Finally, the *counterfactual worlds* depict circumstances that could have happened if the past had unfolded differently. Figure 36 illustrates the structure discussed, where the sequence of circled worlds represents a possible branch.

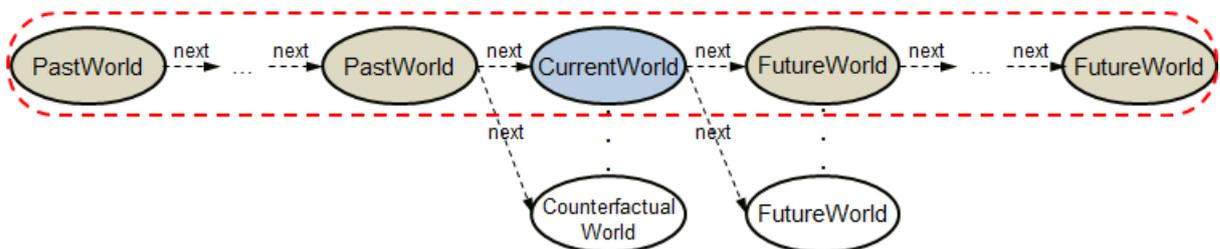


Figure 36. World Structure for simulation.

4.2 STRUCTURAL CHARACTERISTICS OF SITUATIONS

Before creating the situation type rules in Alloy we must first represent the structural aspects of those situations. This includes defining the situation types in Alloy, along with their meta-properties and the participation and derivation relations with its elements. This structure is important so that Alloy can identify what are the types and what are its relations and instantiate them properly, accordingly to the rules that we will define later. We use as basis the transformation from OntoUML to Alloy defined in (SALES, 2014). There the world structure and ontological properties such as rigidity are already included in the Alloy specification. Listing 18 shows the basic structure of a generated Alloy specification altered with situation type support. The lines that we have included are underlined.

Listing 18. Skeleton structure of a generated Alloy model including situations.

```

1  module Model
2
3  open world_structure[World]
4  open ontological_properties[World]
5  open util/relation
6  open util/sequniv
7  open util/ternary
8  open util/boolean
9  open situation_properties[World]
10
11 sig Object {}
12 sig Property {}
13 sig Situation {}
14 sig DataType {}
15
16 abstract sig World {
17     exists: some Object+Property+Situation,
18 }{}
19
20 fact additionalFacts {
21     continuous_existence[exists]
22     elements_existence[Object+Property+Situation,exists]
23 }
24
25 fun visible : World->univ { exists }
26
27 run { } for 10 but 3 World, 7 int

```

Mostly, the included lines basically introduces situation as a new main type, called signature in Alloy, in `sig Situation {}` and states that situations exist in a world along with objects and properties in `+Situation`. A situation definition in Alloy is a composition of patterns that together represents what we call the situation axiom. In addition to the specific facts that characterize each situation and will be discussed in section 4.3, a situation also admits general axioms that address situation *uniqueness* and *continuity*. The line `open situation_properties[World]` represents an import of an Alloy module which contains those general axioms as well as functions that represent the Allen relations used to facilitate the construction of the situation rules. For instance, the uniqueness axiom that states that *a situation is unique for a particular conjunction of entities in a world* is represented in Listing 19.

Listing 19. Situation uniqueness predicate in Alloy.

```

1  pred uniqueness[sit: univ->univ, parts: univ->univ->univ] {
2      all w:World | all s1,s2:w.sit | s1.(w.parts) = s2.(w.parts)
3      implies s1 = s2
4  }

```

Similarly, the situation continuity axiom is represented in Listing 20. It admits for every situation type that *if a conjunction of entities remains in a particular condition in two consecutive worlds, then the situations in both worlds are the same*. The entire *situation_properties* module is shown in Appendix A.

Listing 20. Situation continuity predicate in Alloy.

```

1  pred continuity[sit: univ->univ, parts: univ->univ->univ] {
2      all w1,w2:World | all s1: w1.sit, s2: w2.sit | w2 in (w1.next)
   and s1.(w1.parts) = s2.(w2.parts)
3      implies s1 = s2
4  }
```

Still, Table 2 shows the remaining *patterns* created from the situation type elements and which are applied to the structural module of the Alloy specification. The first column represents the SML elements and the second column the respective *pattern* it generates in the Alloy specification. The first and second lines represent the situation type declaration and the common situation predicates of uniqueness, continuity and rigidity, respectively. The third line represents the declaration of the participation relation between a situation type and its participants, with the respective cardinality of the relation. The fourth line represents accessibility functions and the fifth line an immutability predicate (only applied for participants not set as mutable). Finally, the line represented by “-” indicates singleton statements which depend on all situation types and state, respectively, that the Situation population is a composition of instances of each defined situation type and that all situation types are disjoint from each other. Assume that the orange words will change depending on each element’s type/name.

Table 2. Structural patterns in Alloy created from situation elements.

SML element	Alloy
SituationType	<pre> abstract sig World { (...) SitType: set exists:>Situation, }{} </pre>
	<pre> fact situationCommon { uniqueness[SitType, participation1 + ... + participationN] continuity[SitType, participation1 + ... + participationN] rigidity[SitType, Situation, exists] </pre>

	<pre>(...)</pre>
Participant (participation)	<pre>} abstract sig World { (...) participation: set Source set -> one/some Target, }{ -- if multiple all x: SitType # (x.participation) >= n all x: SitType # (x.participation) <= m }</pre>
participation ends	<pre>fun funName1 (x: World.Participant) World.SitType { (participation).x } fun funName2 (x: World.SitType) World.Participant { x.(participation) }</pre>
Immutable Participant	<pre>fact participationProperties { immutable_target[SitType, participation] }</pre>
-	<pre>-- additional facts abstract sig World {...} { (...) exists:> Situation in SitType1 + ... + SitTypeN disj[SitType1, ... ,SitTypeN] }</pre>

4.3 THE SITUATION MODULE

The specific facts that characterize each situation are a composition of what we call the situation module. Those facts are constructed by mapping each concept of the SML situation type metamodel to a respective *pattern* in Alloy and the union of these patterns represents the situation axiom. We create the situation axioms in Alloy accordingly to the formalization of SML presented in (COSTA et al., 2012) (in first-order logic).

Each situation axiom postulates the conditions for the existence of a situation of a particular type, i.e., those conditions that must be true for as long as the situation of the type exists. In Alloy we address this with a fact with two expressions: one that captures the *sufficient* conditions for the existence of the situation (which necessitates the creation of a situation of the type using the \Rightarrow /implies operator) and one that captures the *necessary* conditions. Since these facts are specific to a particular situation type, we present transformation rules that determine these facts from situation type definitions in SML. Listing 21 shows a skeleton of a situation

axiom as an Alloy fact. Lines 2 and 3 represent the sufficient condition while line 5 indicates the necessary one. Again, expressions in orange will change depending on the situation type.

Listing 21. Skeleton of a situation axiom represented as a fact in Alloy.

```

1 fact SitTypeRule {
2   all w1[,<worlds_quantification>]: World |
   <elements_quantification> | <elements_constraints>
3   implies one s: w1.SitType | <elements_binding>
4
5   all w1: World | all s: w1.SitType | <elements_quantification> |
   <elements_constraints>
6 }

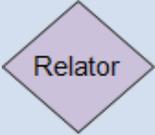
```

Firstly, every situation type will generate an isolate fact such as `fact SitTypeRule {}`, which will contain the situation axiom. In Alloy, the situation axiom is built by quantifying over the elements of the world, applying the constraints on them and then binding them to the respective situation. In the necessary part of the rule the binding is made while quantifying, since the condition is applied to an already existing situation, as we will show. The entities exist necessarily in some world and the conditions are applied taking them into consideration. For that reason, we must consider always at least one world (w_1) in the rule, which represents the *present* time, i.e. the world where the situation type being specified starts to exist (in the sufficient condition) and every world where it exists (in the necessary condition).

The tag `[,<worlds_quantification>]` indicates that other worlds can appear depending on the temporality of the situation type elements. For instance, a past situation participant represents a different world (w_2) in the past, thus it must be quantified and a specific constraint `before[w2,w1]` must be established. Later, in `<elements_quantification>`, every participant, including the relator, is quantified in their respective world and to them a variable is assigned. Table 3 shows examples of quantification for every type of participant.

Table 3. Examples of participant's quantification.

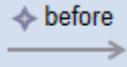
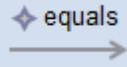
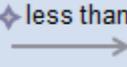
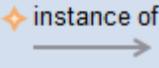
SML participant	Quantification in Alloy
Entity	<pre> all partN: w1.Entity all partN: s.entity{FUN}[w1] </pre>

	<pre>all partN: w1.Relator all partN: s.relator{FUN}[w1]</pre>
	<pre>all partN: w1.SitParticipant all partN: s.sitparticipant{FUN}[w1]</pre>
	<pre>all partN: wM.SitParticipant all partN: s.sitparticipant{FUN}[w1]</pre>

The first line in the second column indicates the quantification in the sufficient rule, while the second line indicates the quantification in the necessary rule already including the binding to the situation. Only the past situation participant in the sufficient rule is quantified in a world of its own, which depends on the number of past participants. The `{FUN}` tag indicates that we are referring to the respective participation end mapped as an Alloy function as shown in Table 2. When there are two participants of the same type and no “*equals*” relation exists between them, the keyword `disj` will appear after the keyword `all` and the variables will be separated by commas, indicating that those elements are different from each other.

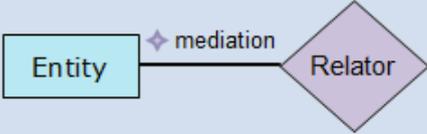
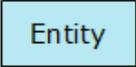
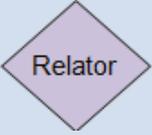
In `<elements_constraints>` the situation main constraints are established. Those constraints are guided by the existing *FormalRelations* and *MediationLinks* of the situation type. While *FormalRelations* define rules between the elements, the *MediationLinks* bind the situation type, the relator and the mediated entities so that they are always the same. Table 4 shows some examples of patterns for *FormalRelations*. *AllenLinks* are analogous to the *before* relation, changing the predicate name only. *ContextFormalLinks* are also defined as predicated with the respective name, but are not supported primitively in the transformation (since they are domain-specific user-defined relations) and thus must be defined manually. *OrderedComparativeLinks* have also a direct mapping to Alloy (`>` / `>=` / `<` / `<=`), and are analogous to *less than* relation. Finally, every *FormalRelation* can be negated, which is represented by the `[not]` before each example.

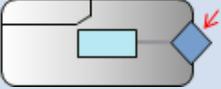
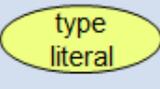
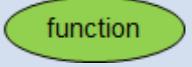
Table 4. Example of FormalRelation patterns.

SML association	Pattern in Alloy
	<code>[not] before[SourceWorld, TargetWorld, Source, Target, exists]</code>
	<code>[not] Source = Target</code>
	<code>[not] Source < Target</code>
	<code>[not] Source in World.TargetType</code>

The `Source` and `Target` will have different interpretations depending on the type of node they represent and, in some case, the elements they are connected to. Table 5 shows the patterns for the different kinds of nodes and connections. Here when we represent the participants in lowercase we mean it is going to be replaced by the respective variable assigned to it, and not its type/name. Again, if a `{FUN}` tag appears after the variable we are referring to the respective association end. Finally, `World` always means the world of the situation participant that is somehow connected to the node, when applicable (if a type literal is connected directly to an entity the world would be `w1`).

Table 5. Patterns for nodes representation in Alloy.

SML node	Pattern in Alloy
	<pre>relator.(w1.mediation) = entity {OR} relator.entity{FUN}[w1] = entity</pre>
	<code>entity</code>
	<code>relator</code>

	<code>sitparticipant</code>
	<code>sitparticipant.participant{FUN}[World]</code>
	<code>sitparticipant.participant{FUN}[World].(World.attribute)</code>
	<code>entity.(w1.attribute)</code>
	<code>sitparticipant.entity{FUN}[World].(World.attribute)</code>
	<code>entity.mode[w1]</code>
	<code>sitparticipant.entity{FUN}[World].mode[World]</code>
	<code>literalvalue</code> <code>{OR}</code> <code>"literalvalue" -- if string</code>
	<code>World.literaltype</code>
	<code>function[param1, ... , paramN]</code>

Finally, `<elements_binding>` is the binding made between the quantified and constrained elements and the situation. Since we quantify over all elements in the world and then apply the rules to them, we must state that the situation derived will be composed by the same elements that satisfy those rules. Thus, the binding is a statement that says the participation end that represents the participant (also an Alloy function `{FUN}`) and the elements used in the quantification are the same. This rule is always of the form `participant in s.participant{FUN}[w1]`. As an example, Listing 22 demonstrates how a complete fever situation rule looks like in Alloy. This situation occurs when a person's temperature is above 37 degrees Celsius.

Listing 22. Fever situation complete rule in Alloy.

```
1 fact Fever {
2     all w1: World | all part1: w1.Person | part1.(w1.temperature) >
3     37
4     implies one s: w1.Fever | part1 in s.person[w1]
5     all w1: World | all s: w1.Fever | all part1: s.person[w1] |
6     part1.(w1.temperature) > 37
}
```

5 ASSESSMENT APPROACH

This chapter presents the last step of our situation assessment approach, which is an extension of the work developed in (SOBRAL; ALMEIDA; COSTA, 2015). We use the new SML metamodel presented in section 3.3 and the automatic transformation to Alloy described in chapter 4 to provide systematic testing cases and examples on the simulation of situation type models and detect problematic scenarios. The context model used in this approach is the same healthcare model from Figure 14, and the situation types will be presented in the course of the simulation presentation.

5.1 QUALITY CRITERIA AND PROBLEMATIC SITUATION TYPES

We may say that situation type models should be adequate for its intended uses, since they are always created envisioning a subsequent purpose. To help systematically evaluate this quality of the models, i.e. its adequacy, one can use many different measurements or criteria which regard distinct aspects of it such as its syntax, semantics, usability, understandability and so on. We have addressed the syntax dimension by providing in chapter 3 an infrastructure to create SML models that encompasses syntactical constraints to verify the structure of situation type models. In this chapter we focus on the semantic dimension. That means that we are mostly concerned on how faithful a formalization is to the conception in the mind of the modeler.

To explain the relation between a model formalization and the modeler's intention, (GANGEMI et al., 2005) introduced the notions of precision and coverage. Those definitions are built upon the notions of intended, i.e. what the modeler wants to say, and possible instantiations, i.e. what the modeler actually said, of an ontology. We here borrow those definitions and extend them to ontology-based situation type models. Low model precision is usually related to under-constraining problems, allowing instantiations that were not originally intended. Low model coverage is usually related to over-constraining problems, not admitting instantiations that are

actually valid. Figure 37 illustrates those definitions, where the blue circles indicate the actual model definition and the grey circles indicate the modeler's intention.

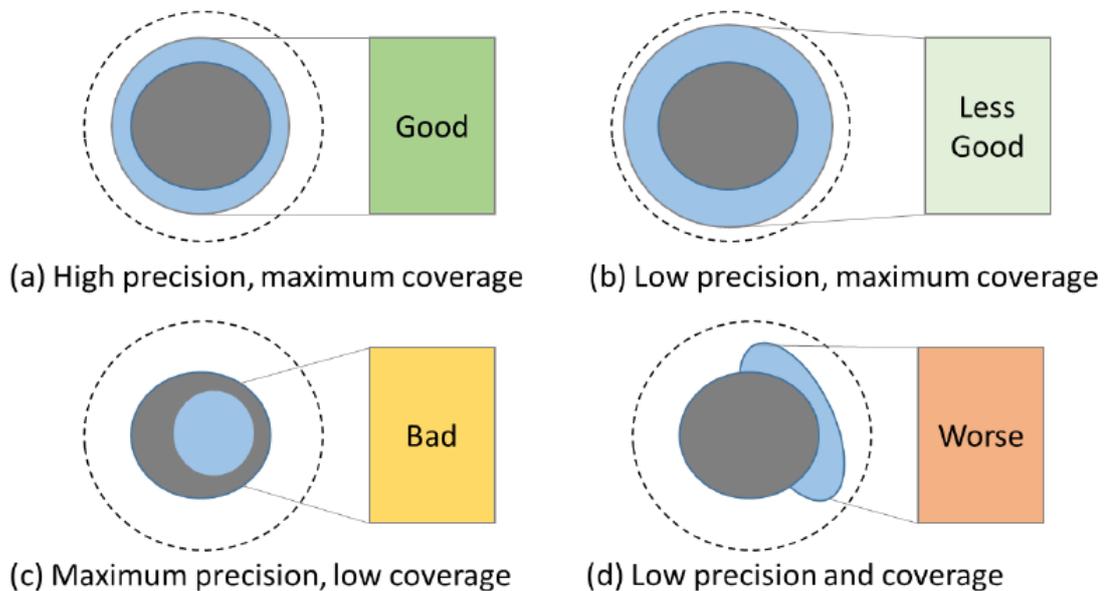


Figure 37. Intended and possible model instantiations adapted from (GANGEMI et al., 2005).

An appropriate model (a) is one that has high precision, allowing only some unintended states, and maximum coverage, meaning that all intended states are possible. When precision decreases, more unintended states are allowed, thus making the definition less appropriate (b). An inappropriate model definition (c) does not encompass every intended state (low coverage), but has maximum precision since every possible state is an intended one. Finally, the least appropriate case (d) is when both precision and coverage are low, meaning the definition allows many unintended states while not encompassing every intended one.

To assess situation type models regarding precision and coverage we use an lightweight formal approach that includes simulation and validation of those models in Alloy (JACKSON, 2006). The simulation consists of demanding automatic generation of model instances so that the modeler can check properties of the model. The assumption is that by visually inspecting possible instantiations, a modeler can sort out admissible and non-admissible model instantiations. In addition, in the validation, modelers can also “demand” the generation of particular model instances that are expected to hold or not. By recurrently performing simulation and validation analysis,

a modeler can capture under and over-constraining problems, thus improving both model precision and model coverage.

Situations consist of particular combinations of context elements and their combinations into complex situations may lead to problematic scenarios. We define *unintended situation type definitions* as definitions arising from the difference between the modeler's intention and the actual definitions he/she expresses in the language, i.e. low precision and/or low coverage definitions. This may be a result of lack of knowledge on the semantics of the language or simply the inherent difficulty in predicting all implications of a (complex) definition.

An *inconsistent situation type definition* specifies an impossible combination of conditions on context elements, and would probably be the result of a design error. A trivial example of inconsistent situation type in a healthcare setting would be a complex situation that is composed of hypothermia and fever simultaneously. Although such inconsistencies may be straightforward to detect, the composition of situations and temporal operators, i.e. Allen relations, on situations may lead to more subtle relations between situations that may go undetected by the modeler. An inconsistent situation type definition would have no practical purpose for situation-awareness. Since those definitions are never instantiated, they can be referred as zero coverage definitions.

Finally, we establish a special case of problematic definitions namely *redundant situation type definitions*, which neither regards precision nor coverage but are undesired situations that must be corrected. A redundant situation type may arise from different forms of specification that actually entail the very same context conditions. Redundant situations would violate parsimony in specifications and have the perverse effect that users would attempt to attribute different semantics to the (apparently) different (yet equivalent) situation types. Consider for example a *fever* situation type, and a *high body temperature* situation type, if both established as sole condition a bodily temperature of 38 degrees Celsius or higher.

5.2 VALIDATION/SIMULATION SCENARIOS

5.2.1 The Alloy Analyzer

The Alloy Analyzer is a tool that allows running simulations of model instances and checking for assertions regarding a model's constraints. The tool's main window is shown in Figure 38. The bar represented by (1) points the tool's menu. We are only interested in the *Execute* menu, which allows the modeler to choose the operation to be executed, e.g. a simulation, an assertion checking and so on. The buttons depicted by (2) represents shortcuts to common operations. Most of them are self-explanatory, while the *Execute* and *Show* buttons provide execution of the last used operation and visual display of the model instance found, respectively. The panel indicated by (3) comprises the model specification in Alloy, and is where the transformed model will be. Finally, (4) shows the result of the executed operations and errors found within the model.

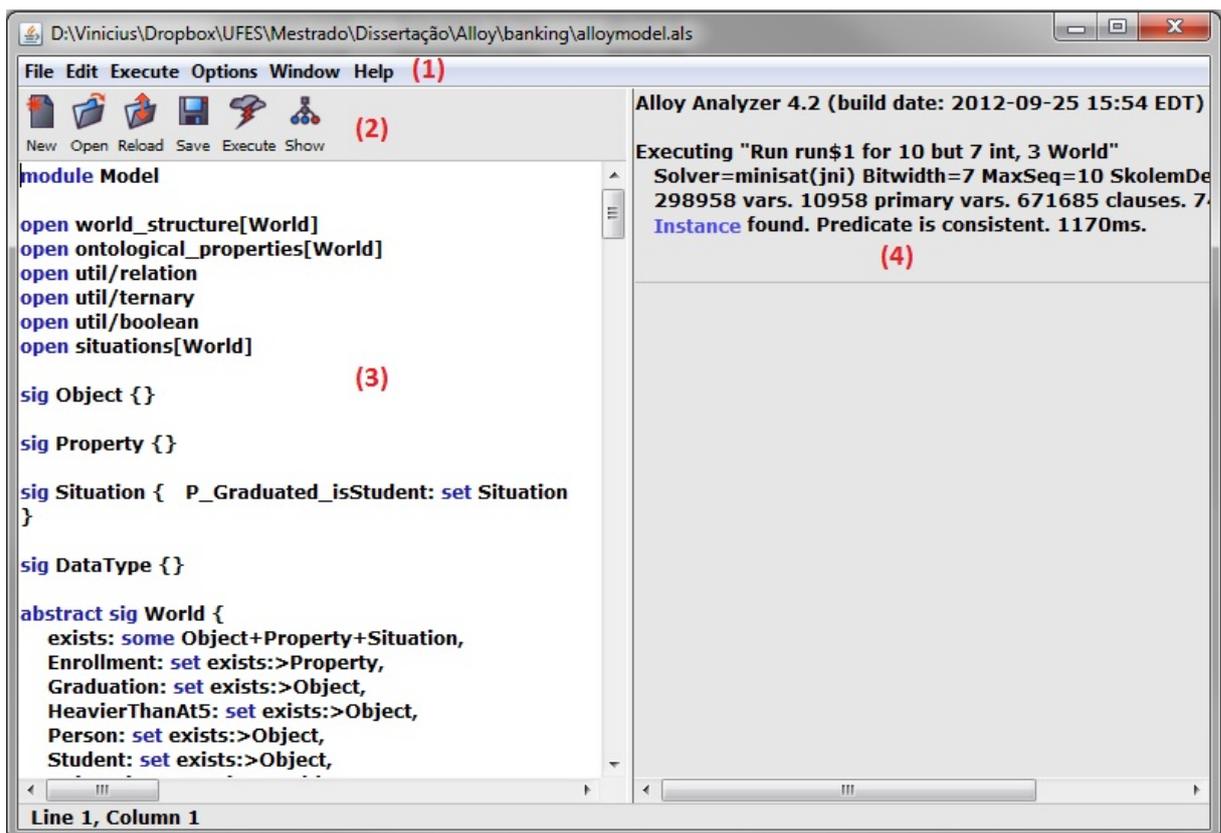


Figure 38. Alloy Analyzer.

A simulated model in Alloy is a representation of the instances of the defined classes, situation types and relations. The model instances are automatically generated by the Alloy Analyzer when we “execute” the model definition. The visualization of the instances is by default equal for every element, making the simulated model confusing. Nevertheless Alloy offers a theme menu where we may choose the layout of the represented objects and also which objects are going to appear in the visualization or not. We have modified the default representation through this theme menu to improve visualization and thus, in the subsequent illustrations, we will represent the classes as Table 6 shows.

Table 6. Types Representation in Alloy simulation model.

Class	Alloy representation
<i>Object Classes</i> (Kind, SubKind, Phase, Role, Category, RoleMixin, Mixin, Quantity and Collective)	
Relators and Modes	
Qualities (its associated values)	
Situation Types	

Figure 39 illustrates a simple example of a simulation in which there is an instance of a fever situation, whose occurrences are given when a person’s temperature is above 37 degrees Celsius. It shows a situation of type *Fever* which involves an object of type *Person* (also of type *Spatial Entity*, since person specializes this class, and *Healthy Person*) that has a temperature of 63 Degrees Celsius (although an absurd value, it is used with the purpose of exemplification only, since we didn’t established a boundary for it). Being such a simple example, we aren’t faced with any problematic scenario.

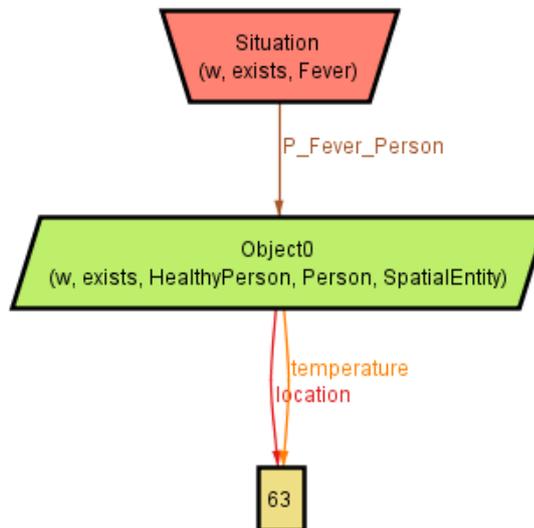


Figure 39. Simulation of fever situation.

In the example provided we have manipulated the execution so that at least one fever instance would appear. Nevertheless, one can run a simulation without any specific constraint and the analyzer will show a random instance. As the user hits *next* in the instance window, the analyzer will provide other instances that normally grow in complexity, allowing a random validation of the model. Besides, we must define a scope for the simulation (e.g. at most 10 instances of each concept), for which the analyzer checks every possible instance. This may seem as the validation done is not reliable (because of the limited scope), but Alloy's premise is that even a small scope can identify most of the problems within a model. Next we will present methods to manipulate the execution and detect specific problematic scenarios.

5.2.2 Inconsistency

Inconsistent situation types may arise as situation definitions grow in size and complexity. Inconsistency is related to impossible states of the elements, so that a situation type can never be instantiated. The situation type of Figure 40 illustrates an example where the modeler mistakenly matched a patient (role) and a treatment (relator). Since the “*equals*” relation may be set between any type of elements, this configuration is possible.

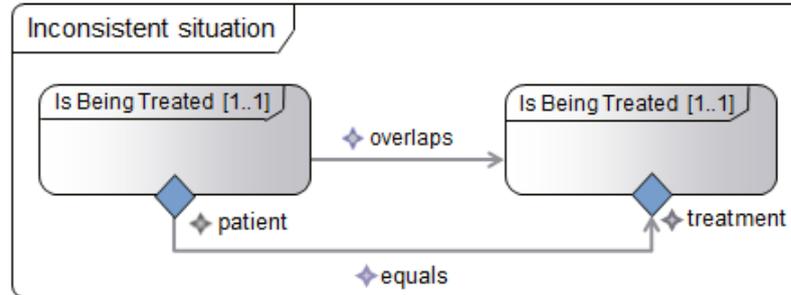


Figure 40. Example of an inconsistent situation.

As we know, object classes and moment classes are disjoint and, as we simulate, we notice that none of the worlds generated contain instances of our example situation. Alloy allows setting constraints over the worlds before running the simulation, thus we can assure that the example is inconsistent by running the command shown in Listing 23, which asks the analyzer to generate an example with at least one world with at least one *Inconsistent Situation* instance.

Listing 23. Run command example in Alloy.

```

1  run {
2    some w:World | #(w.InconsistentSituation) >= 1
3  } for 10 but 3 World, 7 int

```

After checking exhaustively all the possibilities within the defined scope, the tool present us the message depicted in Figure 41. It says that the predicate *may* be inconsistent since we have limited the number of instances it should generate in line `for 10 but 3 World, 7 int`. As mentioned, Alloy premise is that most of the problems arise from small scopes. In this case, we can easily see that no bigger scope is necessary since the number we used would be sufficient if there were to exist the situation.

```

Executing "Run run$1 for 10 but 7 int, 3 World"
Solver=minisat(jni) Bitwidth=7 MaxSeq=10 SkolemDepth=1 Symmetry=20
358370 vars. 14148 primary vars. 1039881 clauses. 4164ms.
No instance found. Predicate may be inconsistent. 321ms.

```

Figure 41. Alloy result example showing the situation is unsatisfiable/inconsistent.

Usually inconsistent models are related to overconstraining of its elements, such as setting that a person has fever and hypothermia concurrently. An efficient way to

check for this specific problem is simulating with rules in the run command. If no instance is found, the situation is probably inconsistent. To facilitate the systematic verification of inconsistent situations, we have included in the transformation a run command for each transformed situation type, analogous to the one in Listing 23. The user may then hit the “Execute” menu and select the desired inconsistency checking available, such as Figure 42 demonstrates.

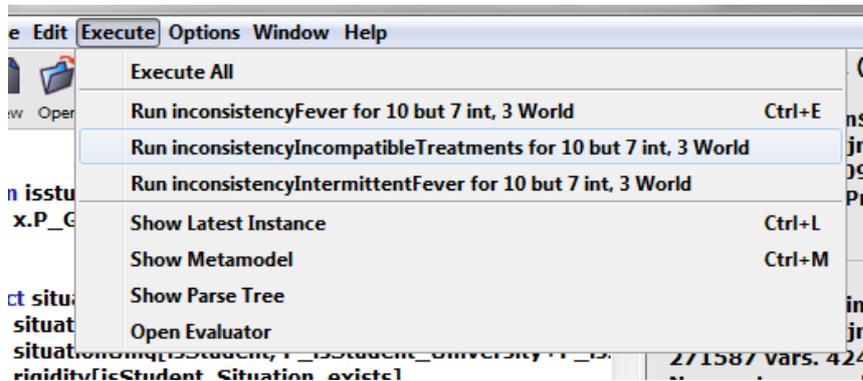


Figure 42. Run commands for detecting inconsistency.

5.2.3 Redundancy

The Alloy Analyzer can also be used to check whether different situation types are equivalent, which we call *redundant situation types*. In a large model, many types can be created to indicate a same situation, which is undesired since it overpopulates the model and does not aggregates semantics to it. For instance, we have a *Normal Fever* situation (Figure 43), which is defined as a person who has a temperature between 37 and 40 degrees Celsius, and a *Common Fever* situation (Figure 44), which is the composition of the *Fever* (temperature > 37°C) and *Under 40* (temperature < 40°C) situations definitions.

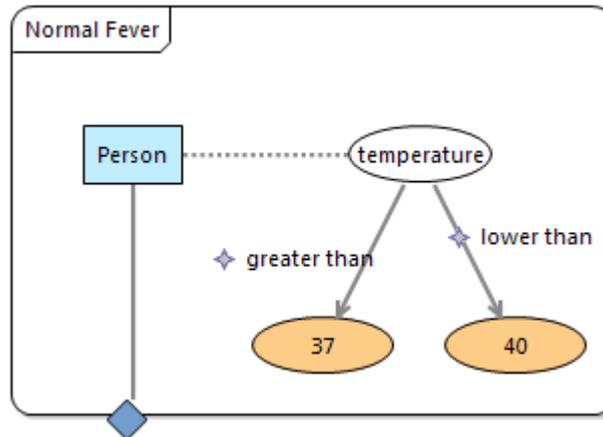


Figure 43. Normal Fever situation.

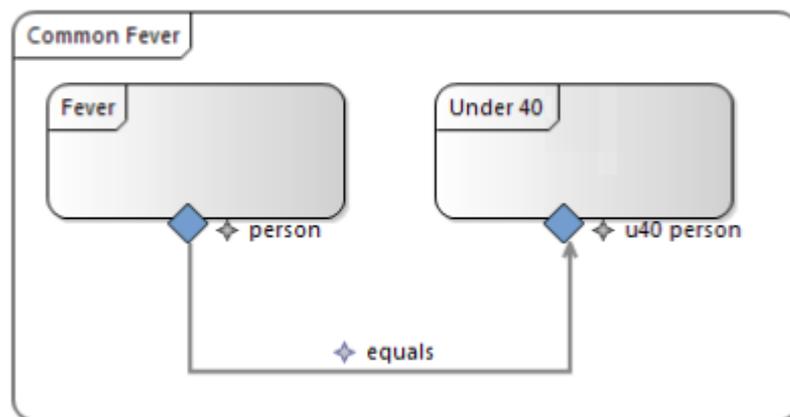


Figure 44. Common Fever situation.

Both of them should happen at the same temperature interval and, consequently, at the same time, as we see by running the simulation. Every world generated by the analyzer is similar to the one in Figure 45, where there is always a *Common Fever* situation (Situation3), connected (composed by) to a *Fever* and an *Under 40 C* situation, alongside a *Normal Fever* situation (Situation1), the two referring to the same person.

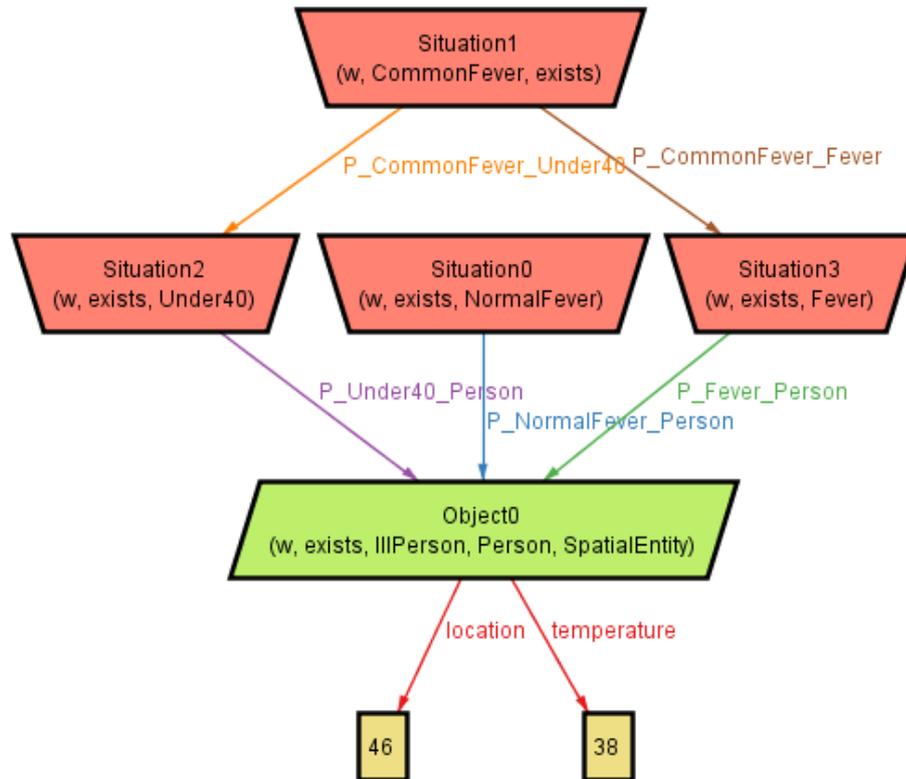


Figure 45. Alloy simulation for Common Fever and Normal Fever situations.

As a definitive test for this case, we can create assertions and ask Alloy to verify whether it holds. An assertion is a proposition which the analyzer tries to contradict. A successful contradiction means a false assertion and is supported by a counterexample to the user, while an unsuccessful one means that the assertion is true (or more precisely, it means that there are no counterexamples for the scope defined). Consequently, we created an assertion that affirms that whenever a *Common Fever* exists (the number of its instances is > 0), a *Normal Fever* also exists and vice versa, as shown in Listing 24. One can check the result obtained, which indicates that the assertion is valid, in Figure 46.

Listing 24. Assertion for checking redundancy of situations.

```

1  assert redundancy {
2      all w:World | #(w.NormalFever) > 0 implies #w.CommonFever > 0
3      all w:World | #(w.CommonFever) > 0 implies #w.NormalFever > 0
4  }
5
6  check redundancy for 5 but 1 World, 7 int

```

```

Executing "Check redundancy for 5 but 7 int, 1 World"
Solver=minisat(jni) Bitwidth=7 MaxSeq=5 SkolemDepth=1 Symmetry=20
22833 vars. 1781 primary vars. 60294 clauses. 422ms.
No counterexample found. Assertion may be valid. 66ms.

```

Figure 46. No counterexample found. Valid assertion.

Similar to what we have done for inconsistent situation types, the transformation also includes assertions to check redundant types, as shown in Figure 47. Those assertions, which are generated for each situation type, validate them generically for redundancy against each other type in the model. Since redundancy must be checked in pairs, with each specific class, our generic approach requires that at least two assertions are valid (no counterexample found) to characterize redundancy. This happens because a single valid assertion may indicate a one-way dependency, e.g. since CommonFever depends on Fever the assertion for CommonFever would be valid, although they are not redundant. In this case, the assertion for Fever would be invalid and not characterize the redundancy. Two valid assertions, however, is a strong indicative that the respective situation types are redundant, e.g. both the CommonFever and NormalFever assertions would be valid, indicating that they are redundant. Listing 25 shows the generic assertion used to verify the one-way redundancy for each situation type (one can later prove the redundancy by using the rule from Listing 24 with the specific classes).

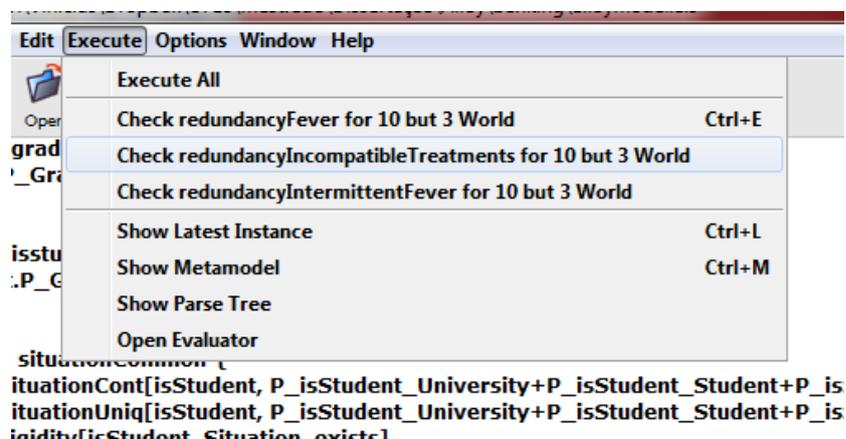


Figure 47. Run commands for detecting redundancy.

Listing 25. Generic assertion to verify redundancy.

```

1  assert redundancySitType {
2      all w:World | #(w.SitType) > 0 implies #((w.exists & Situation)
   - w.SitType) > 0
3  }

```

```

4
5 check redundancySitType for 5 but 1 World, 7 int

```

5.2.4 Unintended states

Finally, we can use the analyzer to detect *unintended situation type definitions*, resulting from states-of-affairs that were not originally intended by the modeler. Unintended states can be challenging to identify since we cannot evaluate them automatically as we did for inconsistency and redundancy. They represent domain-specific scenarios that are related to the intended semantics, i.e. they are in the modeler's mind only. The easiest way to assess a model in Alloy regarding unintended states is recurrently simulating this model and evaluating the generated model instances, looking for wrong relations, strange outcomes of entities/situations and so on. Running random underconstrained model instances, however, might not be very efficient, since users would have to analyze every possible instance that the tool generates. Thus, providing constraints in the *run* command such as restricting the number of entities or the possible outcomes may result in a faster detection of unintended states.

Figure 48 introduces some situations that are used to build a more complex one named *Possible Contagion*, illustrated in Figure 49. In Figure 48 we have defined a situation in which a person is healthy (*Healthy* situation), one in which a person is infected (*Infected* situation), characterized by an infectious disease, a situation in which a patient is having a treatment in a hospital (*Is Being Treated*) and finally, a situation in which a person *Becomes Infected*, i.e. it was a healthy person and turns into an infected person in a subsequent stage. The Possible Contagion situation establishes that two people have had a treatment in the same hospital at the same time period, one being healthy and the other one being infected. Right after that, at the present time, the person which was healthy also becomes infected with the same disease of the previously infected person.

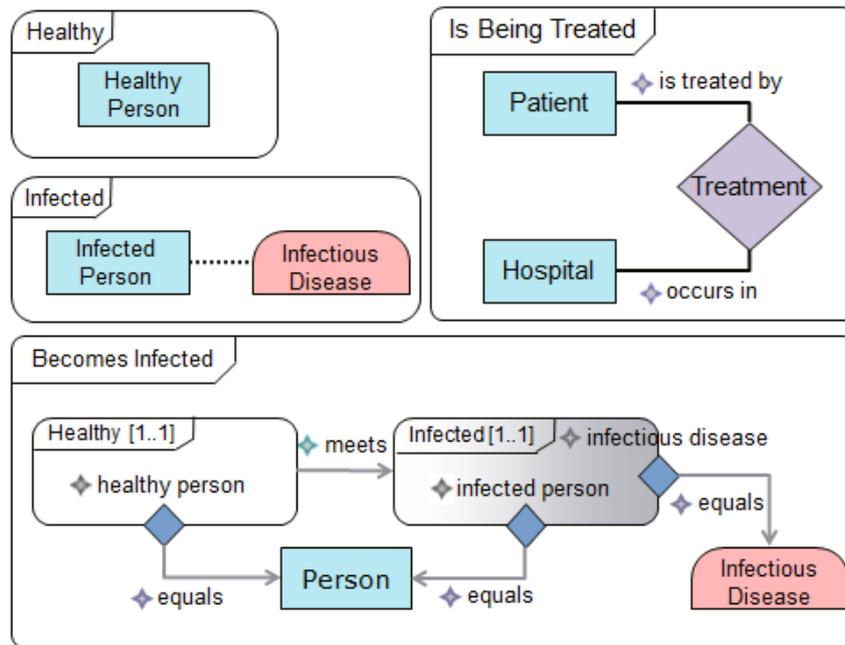


Figure 48. Healthy, Infected, Is Being Treated and Becomes Infected situations.

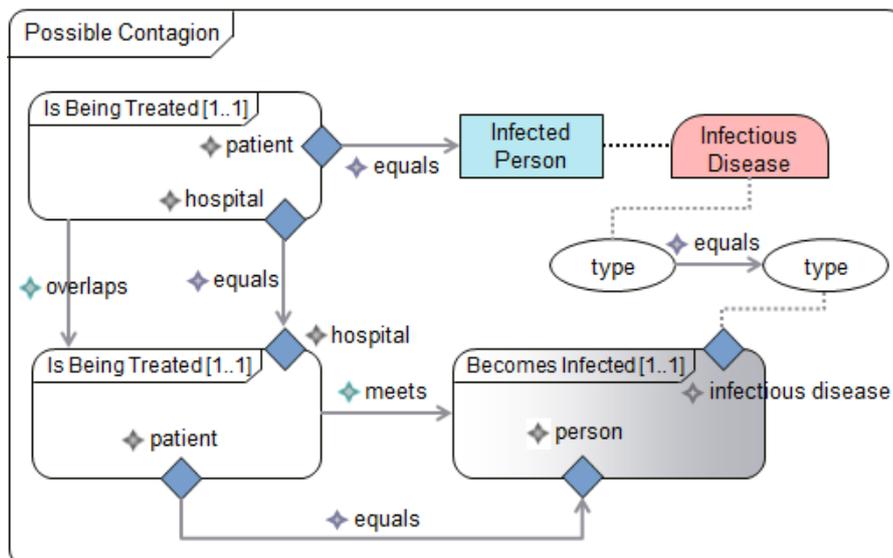


Figure 49. Possible Contagion situation.

As we ask Alloy to generate instances of the modeled situation, we notice in Figure 50 (world 1) and Figure 51 (world 2) that we have underconstrained our definition since it is feasible that the same person alone (Object0) generates a *Possible Contagion* situation (Situation3) by turning from a *Healthy Person* in world 1 into an *Infected Person* in world 2 (Situation2 as the *Becomes Infected* situation), thus validating the situation constraints. Situation 0 and Situation1 represents both *Is Being Treated* situations and the objects that only have the instance's name are

past/future objects, which mean they are connected to the entities but do not exist at the same time instant. We have also omitted the representation of the *Healthy* and *Infected* situation since they are simply defined as a direct relation (one to one) to their respective entities (healthy person and infected person).

This situation happens for some reasons: the same *Patient* can have different treatments in the same hospital at the same time (defined by our context model, which is reasonable) and thus can be the Patient of both *Is Being Treated* situations; we didn't explicitly said that the originally infected person should be infected *while* having the treatment, just that it had a treatment and is infected (this would require another composite situation); and finally we also didn't explicitly said that the Patient from the first treatment must be different from the Patient from the second treatment (and consequently from the infected person).

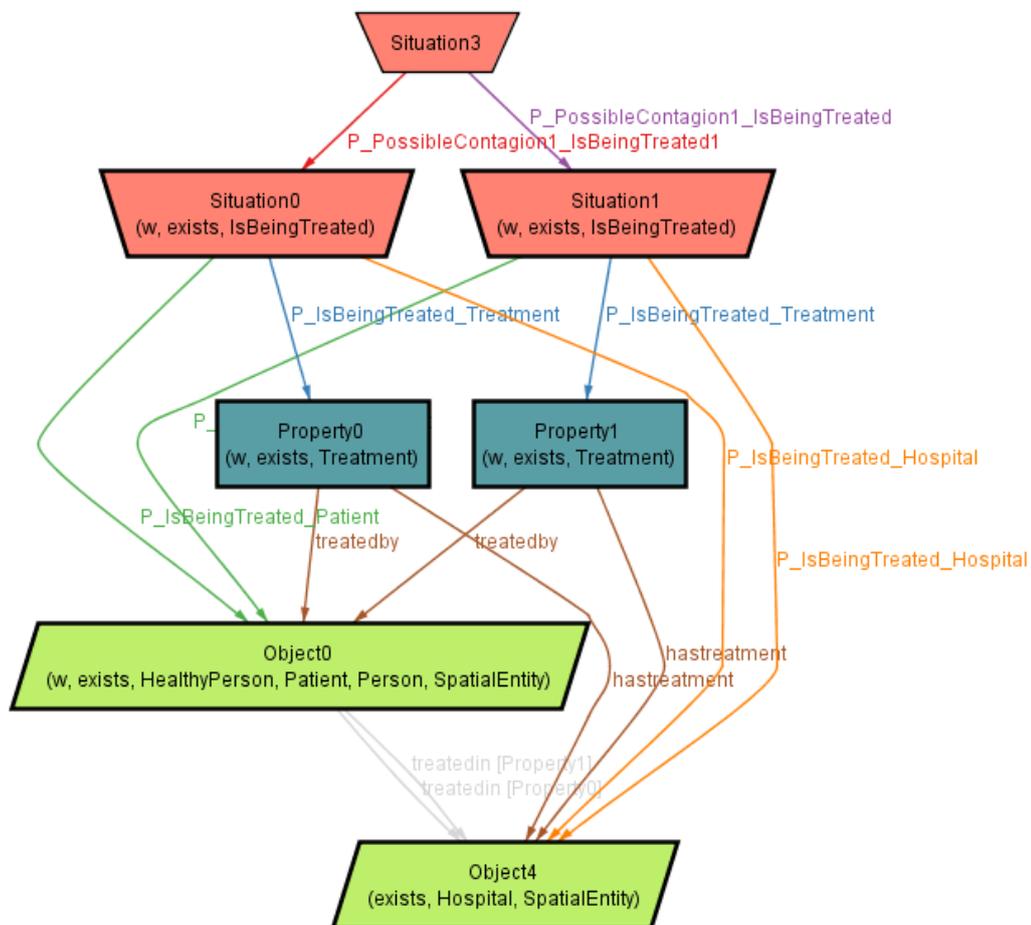


Figure 50. Possible Contagion situation simulation (World 1).

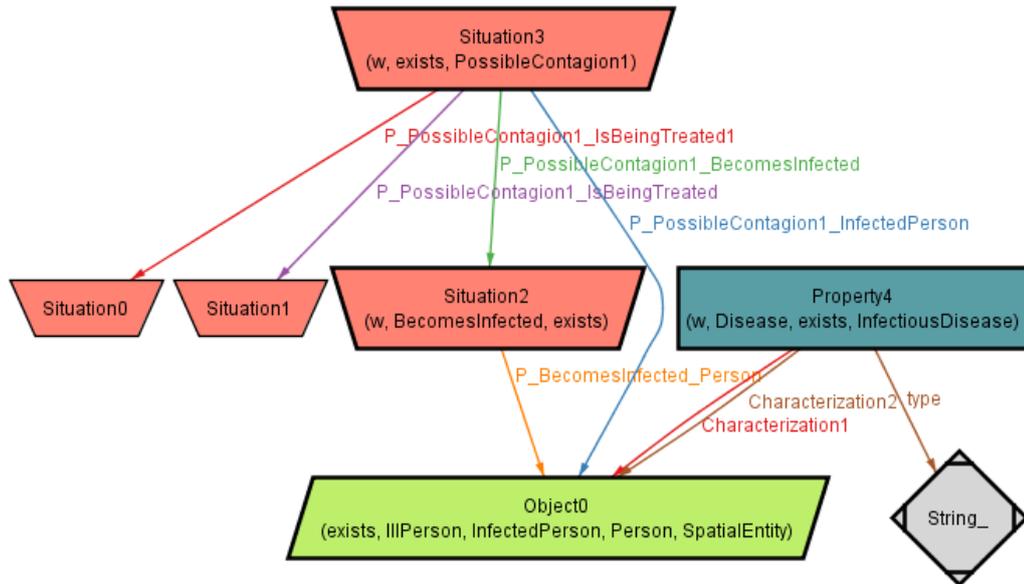


Figure 51. Possible Contagion situation simulation (World 2).

As one can see, problematic scenarios tend to appear as soon as models grow in size and complexity. Thus, validating those models is important and Alloy provides a powerful way of doing so by visual model simulation and checking. Since unintended worlds can appear in various forms, systematically performing this validation is difficult. In (SALES, 2014) the author provides many simulation scenarios and the respective Alloy rules to evaluate different world and entity outcomes. By the usage of those pre-defined scenarios, users partially know what to expect from the simulation, diminishing the cognitive work and facilitating one's analysis. For example, if one requires all generated worlds to contain the same individuals, he/she would not need to keep track of object creation and destruction when moving throughout worlds. We present those simulation scenarios in Appendix B.

Finally, we present in Figure 52 a correct situation type model for our Possible Contagion case example, in which we added the negated (not) *equals* relation between the person originally infected and the one that becomes so and also changed the first Is Being Treated definition to one that indicates that the person is infected during the treatment.

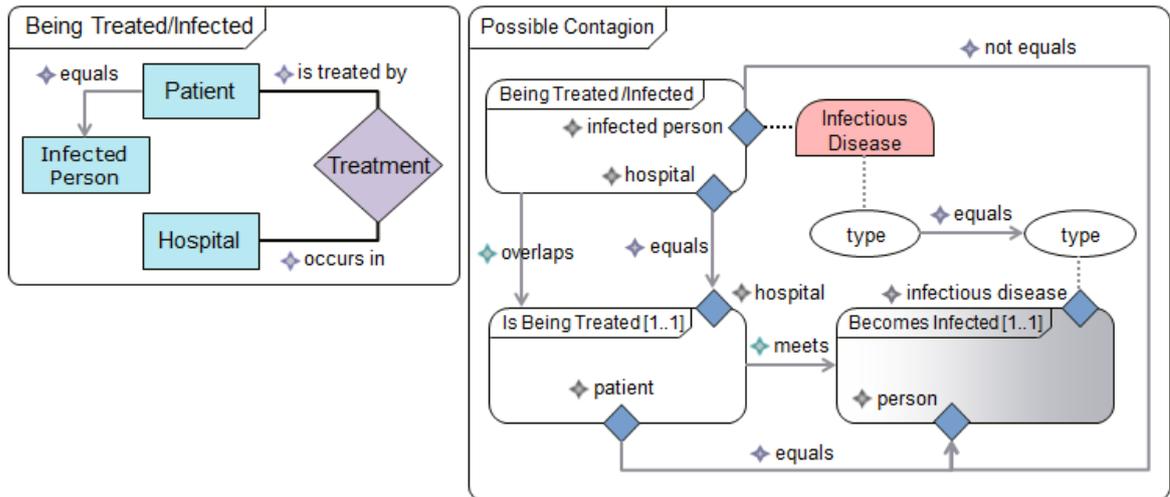


Figure 52. Being Treated/Infected and correct Possible Contagion situation.

6 CONCLUDING REMARKS

Throughout this thesis we have studied, evaluated and improved the situation modeling activity, in the context of situation-aware applications, using the SML language. Our contributions and conclusions are summarized in this chapter and at the end we provide our vision on possible future works.

6.1 CONTRIBUTIONS AND CONCLUSIONS

We have addressed the activity of situation modeling by proposing: an extension of the SML language, which was defined in previous works; an automatic transformation from this language to a logic-based language called Alloy; an assessment approach for situation models in a lightweight formal method using the transformation created. The first step was accomplished by integrating the SML language with an ontologically well-founded conceptual modeling language, called OntoUML. This integration allowed the improvement of the expressivity of the situation type models by using OntoUML as a language for creating context models to be used with SML. It also aggregated to the situation modeling language a conjunction of tools and models created for and with the OntoUML language, resulting from continuous works focused on the language and also its successful application in industrial and commercial projects.

In order to accomplish the assessment of situation models in a manner that is transparent to the user, we have proposed an automated transformation from SML to Alloy and analysis of the result using the Alloy Analyzer. We used the OntoUML validation framework developed in (SALES, 2014), but also extended it including a *Situation Module* that enables one to validate situation type models developed with SML. As we have demonstrated, our approach allows us to identify from simple inconsistencies (solved by adding a single element or constraint), to more sophisticated semantic and equivalence problems, which are hard to notice without the help of an automated tool.

6.2 RELATED WORKS

We address the related works by separating them into *ontology-based situation specification* approaches and *situation validation* approaches. The first one situates our work regarding the specification of situations using ontologies as a means to define situational and domain knowledge for situation-aware applications. The second discusses existing validation approaches for situation definitions in general.

6.2.1 Ontology-based Situation Specification

In the words of Ye (2012), “ontologies have been and will still be a preferred choice to translate, represent, and instantiate the (domain and situational) knowledge” in the so-called specification-based techniques for situation identification. Therefore, many are the examples of the use of ontologies for situation-aware applications, such as (KOKAR; MATHEUS; BACLAWSKI, 2009), (YAU; LIU, 2006) and (STEVENSON et al., 2009), which use the Web Ontology Language (OWL), (ROMÁN et al., 2002) for the combination of the DARPA Agent Markup Language (DAML) and Ontology Interchange Language (OIL), or simply DAML+OIL, and (COUTAZ et al., 2010) that relies on the Extensible Markup Language (XML) to create its ontologies. (KOKAR; MATHEUS; BACLAWSKI, 2009) formalizes the main concepts in the situation-awareness field by means of an OWL ontology; (YAU; LIU, 2006) provides a situation ontology that allows to model situations in a hierarchical manner; (STEVENSON et al., 2009) presents Ontonym, a collection of OWL upper ontologies for developing pervasive systems; (ROMÁN et al., 2002) introduces Gaia, an infrastructure for smart spaces, which relies on ontologies as a way to manage the diversity and complexity of describing resources (e.g. devices and services); finally (COUTAZ et al., 2010) presents the GLObal Smart Space (GLOSS), whose ontologies describe a small set of concepts that provide an understanding of how services are used and how users interleave various contexts at run time, allowing reusing in different services implementation and abstraction over specific details of technologies.

In a nutshell, languages like OWL, DAML+OIL (which was superseded by OWL) and XML are computer oriented in the sense that they have diminished expressivity with the purpose of being machine processable. As described by (KOKAR; MATHEUS; BACLAWSKI, 2009), there exist many cases where OWL is not sufficiently expressive to capture every desired concept, idea that can be extended to DAML+OIL and XML. In the process of situation-aware applications engineering, we argue that expressivity must be addressed in the highest level at first, when modeling the system, so that the subtleties of the elements are captured. In a second stage (runtime) the high-expressive models can be used to generate less-expressive ones in a model-driven fashion, such that the generated models can be processed by computers. By using OntoUML as our ontology language we attend the expressivity issue since the language is grounded by a foundational ontology. Besides, OntoUML can be automatically transformed to OWL (BARCELOS et al., 2013) so that designers can profit from inference and processability inherent to the latter language.

Furthermore, we divide the situation type modeling in two parts, i.e. context modeling and situation modeling, using languages that are more suitable to capture the elements properties in each case, such as elements hierarchy and relations with OntoUML and situations composition and temporal relations with SML, instead of general purpose languages like OWL and XML. Still, as visual and higher abstraction languages OntoUML and SML are easier to use and communicate by modelers and domain specialists and are not bound to a specific implementation platform.

Finally, (COSTA et al., 2012) and (MIELKE, 2013) proposed a situation type specification that served as basis for this thesis. As demonstrated throughout this work, the context model language used, although based on ontological foundations, left aside important distinctions such as dynamic classification, which restrict the creation of some situation types. By using OntoUML, we were able to improve the quality of the context specification, also expanding and increasing the expressivity of the situation type models.

6.2.2 Situation Validation

Although context and situation specification is a recurrent subject in the situation-awareness community, validating those specifications is a task that hasn't gained much attention, especially if done in a conceptual level. In a broader view, techniques to help assessing situations at runtime are proposed more often, but usually for very specific scenarios such as in (FISCHER; BEYERER, 2013) for the maritime domain and (SCHUBERT; SCHULZE; WANIELIK, 2010) for drive-assistant systems, unlike the more general domain-independent approach we take in this thesis. Meanwhile, the only approach found that specifically refers to "situation type validation" is the one in (SALFINGER et al., 2014), which is closest to our approach. The referred work proposes a tool suite that supports the knowledge management in situation-awareness systems from the specification phase to runtime, also addressing evolving environments and user needs. Its validation includes syntactical and semantic checking, such as our proposal, although not specifying if it can guarantee the detection of inconsistent (or contradictory) and redundant situation, i.e. if the checking is made against every possible model instance, such as the Alloy Analyzer does. Moreover the ontology used in the referred work is very simple, not addressing many important aspects of elements representation, as extensively mentioned throughout this thesis, such as entity dynamics, intrinsic and relational properties, among others.

Finally, we should mention the work of Sales (2014), which evolved from (BENEVIDES et al., 2010) and provided a validation framework for OntoUML which served as basis for the assessment proposal of this thesis. Although not including support for situations, the elements and world structured present in the referred work were essential to simulate the situation type's dynamics and flexible enough to allow the inclusion of this novel concept.

6.3 LIMITATIONS AND FUTURE WORK

Although we have addressed the integration of SML with OntoUML in almost its entirety, whole-part relations were not studied and may be subject to future works.

Including those elements in SML may allow one to express in a well-founded manner constraints regarding composition (a car which is composed by 4 or 6 wheels), membership and other *part of* relations. Besides, UFO is an extremely rich set of theories that describe many elements besides the structural ones that we used in this work, such as events and social aspects of the world (e.g. actors, objectives and social commitments). Future research could investigate those other “slices” of UFO, even deepening in the study of the relation between the situation concept and UFO-events and which brings situation into existence. This would increase even more SML expressivity as a conceptual modeling language and even making it an ontologically well-founded reference (what we may call a *core ontology*) for describing situation types.

Regarding other necessities, SML could be improved to incorporate other features such as disjointness between elements, undefined temporality (e.g. a situation participant that may be either past or present) and value accumulators, e.g. to enable the definition of a situation where the average temperature in the last 24h is greater than 38 degrees Celsius. Still, since context information is obtained mostly from sensors, its accuracy may not be perfect, which may cause problems in the identification of situation. For this reason, studies may be realized in order to include ways to express the subtleties of the acquisition of context information (including quality of context) in the language. Addressing quality of context in the modeling phase poses a challenge that is dealing with uncertainty in a higher level. This could significantly change the way context-aware systems are modeled since many problems related to sensor data would be identified at an earlier phase. In any case, those improvements will require extending the approach presented in this work.

Furthermore, the extension of the language created a gap in the concrete syntax of the language and in the runtime support of SML, namely in the SCENE platform. One of the main subjects of the original work, the concrete syntax was not the focus in this thesis and was addressed only superficially with the only purpose of providing examples to support the studies developed. With a concrete syntax for the extended metamodel provided, the editor created for the original metamodel should be extended to support the new elements and characteristics. Regarding the runtime support, it should be revisited and extended to also support the revised metamodel.

Although many elements remained the same, the metamodel structure was significantly changed which will require a deep modification in the runtime support.

Finally, other possibilities of future work are the improvement of the simulation visualization for assessing situation models and usability studies for SML and the simulation. Although we have used so far the visualization tool provided with the Alloy Analyzer, we believe that a richer tool with explicit support for the situation concept may be more appropriate, allowing us to explore richer graphical patterns. Diagrams generated by the Analyzer would then be used to communicate with domain experts. It is important, though, to evaluate both the language and the simulation regarding their usability in real, larger projects and among many users. So far SML had been treated in the academia only and requires this feedback from industrial/commercial projects.

BIBLIOGRAPHY

- ALBUQUERQUE, A.; GUIZZARDI, G. **An ontological foundation for conceptual modeling datatypes based on semantic reference spaces** Proceedings - International Conference on Research Challenges in Information Science. **Anais...**2013
- ALLEN, J. F. **Maintaining knowledge about temporal intervals** **Communications of the ACM**, 1983.
- BARCELOS, P. P. F. et al. **Ontological evaluation of the ITU-T Recommendation G.805** 2011 18th International Conference on Telecommunications, ICT 2011. **Anais...**2011
- BARCELOS, P. P. F. et al. **An Automated Transformation from OntoUML to OWL and SWRL**. ONTOBRAS. **Anais...**2013
- BARWISE, J. **The Situation in Logic**. [s.l.] Cambridge University Press, 1989.
- BASTOS, C. A. M. et al. **Building Up a Model for Management Information and Knowledge: The Case-study for a Brazilian Regulatory Agency** Proceedings of the 2nd International Workshop on Software Knowledge (SKY). **Anais...**2011
- BAUMGARTNER, N. et al. BeAware! - Situation Awareness, the Ontology-Driven Way. **Data & Knowledge Engineering**, v. 69, n. 11, p. 1181–1193, 2010.
- BENEVIDES, A. et al. Assessing Modal Aspects of OntoUML Conceptual Models in Alloy. In: HEUSER, C.; PERNUL, G. (Eds.). . **Advances in Conceptual Modeling - Challenging Perspectives**. Lecture Notes in Computer Science. [s.l.] Springer Berlin / Heidelberg, 2009. v. 5833p. 55–64.
- BENEVIDES, A. B. et al. Validating Modal Aspects of OntoUML Conceptual Models Using Automatically Generated Visual World Structures. **J. UCS**, v. 16, n. 20, p. 2904–2933, 2010.
- CAROLO, F.; BURLAMAQUI, L. **Improving Web Content Management with Semantic Technologies** Semantic Technology Conference (SemTech). **Anais...**San Francisco: 2011
- CARRARETTO, R. **A Modeling Infrastructure for OntoUML**. Vitória, Brazil: [s.n.]. Disponível em:
<<http://rcarraretto.googlecode.com/files/aModelingInfrastructureForOntoUML.pdf>>.
- COSTA, P. D. et al. **Situations in Conceptual Modeling of Context** 2006 10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW'06). **Anais...**ieee, 2006 Disponível em:
<<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4031266>>

COSTA, P. D. **Architectural Support for Context-Aware Applications: From Context Models to Services Platforms**. Enschede, The Netherlands: Universiteit Twente, 2007.

COSTA, P. D. et al. **A Model-Driven Approach to Situations: Situation Modeling and Rule-Based Situation Detection** 2012 IEEE 16th International Enterprise Distributed Object Computing Conference. **Anais...** Beijing, China: IEEE, set. 2012
Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6337246>.

COSTA, P. D.; ALMEIDA, J. P. A. Situation specification and realization in rule-based context-aware applications. **Distributed Applications and Interoperable Systems, 7th IFIP WG 6.1 International Conference, DAIS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings**, p. 32–47, 2007.

COUTAZ, J. et al. Working document on gloss ontology. **arXiv preprint arXiv:1006.5661**, 2010.

DEY, A. Understanding and using context. **Personal and ubiquitous computing**, 2001.

FISCHER, Y.; BEYERER, J. **Ontologies for probabilistic situation assessment in the maritime domain** Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2013 IEEE International Multi-Disciplinary Conference on. **Anais...** 2013

GANGEMI, A. et al. Ontology evaluation and validation. **An integrated formal model for the quality diagnostic task**, v. 30, p. 36, 2005.

GUARINO, N.; OBERLE, D.; STAAB, S. What Is an Ontology? 2009.

GUERSON, J. **Representing Dynamic Invariants in Ontologically Well-Founded Conceptual Models**. [s.l.] UFES, 2015.

GUERSON, J.; ALMEIDA, J. P. A. **Representing Dynamic Invariants in Ontologically Well-Founded Conceptual Models** 20th EMMSAD, CAiSE 2015. **Anais...** Stockholm, Sweden: 2015

GUIZZARDI, G. **Ontological foundations for structural conceptual models**. Enschede: CTIT, Centre for Telematics and Information Technology, 2005.

GUIZZARDI, G. et al. The Role of Foundational Ontologies for Domain Ontology Engineering. **International Journal of Information System Modeling and Design**, v. 1, n. 2, p. 1–22, 2010.

HANSMANN, U. **Pervasive Computing: The Mobile World**. [s.l.] Springer, 2003.

JACKSON, D. **Software Abstractions - Logic, Language, and Analysis**. [s.l.] MIT Press, 2006.

KOKAR, M. M.; MATHEUS, C. J.; BACLAWSKI, K. Ontology-based situation awareness. **Information Fusion**, v. 10, n. 1, p. 83–98, jan. 2009.

KRIPKE, S. A. Semantical Analysis of Modal Logic I. Normal Propositional Calculi. **Zeitschrift fur mathematische Logik und Grundlagen der Mathematik**, v. 9, n. 5{6}, p. 67–96, 1963.

MIELKE, I. T. **Uma Abordagem Baseada em Modelos para Especificação e Detecção de Situações em Sistemas Sensíveis ao Contexto**. [s.l.] Universidade Federal do Espírito Santo, 2013.

MOODY, D. L. et al. Evaluating the quality of information models: empirical testing of a conceptual model quality framework. **25th International Conference on Software Engineering, 2003. Proceedings.**, 2003.

MYLOPOULOS, J. Conceptual Modelling and Telos 1. **Information Systems Journal**, p. 1–20, 1992.

OLIVÉ, A. Taxonomies and Derivation Rules in Conceptual Modeling. In: [s.l: s.n.]. p. 417–432.

OLIVÉ, A. **Conceptual Modeling of Information Systems**. [s.l: s.n.].

OMG. **UML 2.4.1 Superstructure Specification**. [s.l: s.n.].

OMG. **OMG Object Constraint Language (OCL), Version 2.3.1**, 2012. Disponível em: <<http://www.omg.org/spec/OCL/2.3.1/>>

PEREIRA, I. S. A.; COSTA, P. D.; ALMEIDA, J. P. A. **A rule-based platform for situation management** 2013 IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support (CogSIMA). **Anais...IEEE**, fev. 2013 Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6523827>>.

RAYMUNDO, C. R. et al. **An infrastructure for distributed rule-based situation management** 2014 IEEE International Inter-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support (CogSIMA). **Anais...IEEE**, mar. 2014 Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6816563>>

ROMÁN, M. et al. A middleware infrastructure for active spaces. **IEEE pervasive computing**, v. 1, n. 4, p. 74–83, 2002.

ROSEMANN, M.; RECKER, J. C. **Context-aware Process Design: Exploring the Extrinsic Drivers for Process Flexibility** (T. Latour, M. Petit, Eds.) 18th International Conference on Advanced Information Systems Engineering. Proceedings of Workshops and Doctoral Consortium. **Anais...Namur University Press**, 2006 Disponível em: <<http://eprints.qut.edu.au/4638/>>

SALES, T. P. **Ontology Validation for Managers**. [s.l.] Universidade Federal do Espírito Santo (UFES), 2014.

SALES, T. P.; GUIZZARDI, G. **Detection, Simulation and Elimination of Semantic Anti-patterns in Ontology-Driven Conceptual Models**International Conference on Conceptual Modeling (ER). **Anais...**2014

SALFINGER, A. et al. **SEM 2 suite - Towards a tool suite for supporting knowledge management in situation awareness systems**Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on. **Anais...**2014

SANTOS JÚNIOR, J. C. O. **Enriquecendo a Semântica de Modelos de Processos de Negócio com Modelos Conceituais em OntoUML**. [s.l.] Universidade Federal do Espírito Santo (UFES), 2008.

SCHUBERT, R.; SCHULZE, K.; WANIELIK, G. Situation Assessment for Automatic Lane-Change Maneuvers. **IEEE Transactions on Intelligent Transportation Systems**, v. 11, n. 3, p. 607–616, set. 2010.

SIDER, T. Quantifiers and temporal ontology. **Mind**, p. 75–97, 2006.

SOBRAL, V. M.; ALMEIDA, J. P. A.; COSTA, P. D. **Assessing situation models with a lightweight formal method**2015 IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision. **Anais...**Orlando, FL: IEEE, mar. 2015Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7108173>>

STEVENSON, G. et al. **Ontonym: a collection of upper ontologies for developing pervasive systems**Proceedings of the 1st Workshop on Context, Information and Ontologies. **Anais...**2009

WEISER, M. The Computer for the 21st Century. **SIGMOBILE Mob. Comput. Commun. Rev.**, v. 3, n. 3, p. 3–11, 1999.

YAU, S. S.; LIU, J. **Hierarchical situation modeling and reasoning for pervasive computing**Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop on. **Anais...**2006

YE, J.; DOBSON, S.; MCKEEVER, S. Situation identification techniques in pervasive computing: A review. **Pervasive and mobile computing**, v. 8, n. 1, p. 36–66, 2012.

ZAMBORLINI, V.; GUIZZARDI, G. **On the Representation of Temporally Changing Information in OWL**.EDOCW. **Anais...**IEEE, out. 2010Disponível em: <<http://dblp.uni-trier.de/db/conf/edoc/edoc2010w.html#ZamborliniG10>>. Acesso em: 22 out. 2012

APPENDIX A - ALLOY MODULE WITH SITUATIONS

COMMON PREDICATES

```

1 module situations[World]
2
3 open world_structure[World]
4
5 //Situation Continuity
6 // States that a situation continues through time, i.e. it is the
7 // same if its participants remain the same in consecutive worlds
8 pred situationCont[sit: univ->univ, parts: univ->univ->univ,
9 partsTemp: univ->univ] {
10     all w1,w2:World | all s1: w1.sit, s2: w2.sit | w2 in (w1.next)
11     and s1.(w1.parts) = s2.(w2.parts) and s1.partsTemp = s2.partsTemp
12     implies s1 = s2
13 }
14
15 pred situationCont[sit: univ->univ, parts: univ->univ->univ] {
16     all w1,w2:World | all s1: w1.sit, s2: w2.sit | w2 in (w1.next)
17     and s1.(w1.parts) = s2.(w2.parts) implies s1 = s2
18 }
19
20 //Situation Uniqueness
21 // States that a situation is unique for a particular conjunction
22 // of entities in a world
23 pred situationUniq[sit: univ->univ, parts: univ->univ->univ,
24 partsTemp: univ->univ] {
25     all w:World | all s1,s2:w.sit | s1.(w.parts) = s2.(w.parts) and
26     s1.partsTemp = s2.partsTemp implies s1 = s2
27 }
28
29 pred situationUniq[sit: univ->univ, parts: univ->univ->univ] {
30     all w:World | all s1,s2:w.sit | s1.(w.parts) = s2.(w.parts)
31     implies s1 = s2
32 }
33
34 //situation s1 happens before situation s2 (w3 and w4 assures they do
35 // not overlap)
36 // s1 |-----|
37 //                                     s2 |-----|
38 pred before[w1: World, w2: World, s1: univ, s2: univ, exists: univ-
39 >univ] {
40     some w3,w4:World |
41     w3 in w1.next and w2 in w4.next and w1 != w2 and w2 in
42     w1.^next and
43     (s1 in w1.exists) and not(s1 in w3.exists) and (s2 in
44     w2.exists) and not(s2 in w4.exists)
45 }
46
47 pred before[w1: World, w2: World, s1: univ, exists: univ->univ] {
48     w1 != w2 and w2 in w1.^next and
49     (s1 in w1.exists) and not(s1 in w2.exists)
50 }
51
52 //situation s1 happens after situation s2
53 // s2 |-----|
54 //                                     s1 |-----|
55 pred after[w1: World, w2: World, s1: univ, s2: univ, exists: univ-
56 >univ] {

```

```

40     w1 != w2 and w1 in w2.^next and
41     (s1 in w1.exists) and not(s1 in w2.exists) and (s2 in
w2.exists) and not(s2 in w1.exists)
42 }
43
44 //situation s1 ends right before situation s2 starts
45 // s1 |-----|
46 //           s2 |-----|
47 pred meets[w1: World, w2: World, s1: univ, s2: univ, exists: univ-
>univ] {
48     w1 != w2 and w2 in w1.next and
49     (s1 in w1.exists) and not(s1 in w2.exists) and (s2 in
w2.exists) and not(s2 in w1.exists)
50 }
51
52 //situation s1 begins right after situation s2 ends
53 // s2 |-----|
54 //           s1 |-----|
55 pred metby[w1: World, w2: World, s1: univ, s2: univ, exists: univ-
>univ] {
56     w1 != w2 and w1 in w2.next and
57     (s1 in w1.exists) and not(s1 in w2.exists) and (s2 in
w2.exists) and not(s2 in w1.exists)
58 }
59
60 //situation s1 overlaps(occurs at the same time as) situation s2
61 // s1 |-----|
62 //           s2 |-----|
63 pred overlaps[w: World, s1: univ, s2: univ, exists: univ->univ] {
64     (s1 in w.exists) and (s2 in w.exists)
65 }
66
67 //situation s1 overlaps(occurs at the same time as) situation s2
(identical to overlaps)
68 // s2 |-----|
69 //           s1 |-----|
70 pred overlappedby[w: World, s1: univ, s2: univ, exists: univ->univ] {
71     (s1 in w.exists) and (s2 in w.exists)
72 }
73
74 //situation s1 ends at the same time as situation s2
75 //           s1 |-----|
76 // s2 |-----|
77 pred finishes[w1: World, w2: World, s1: univ, s2: univ, exists: univ-
>univ] {
78     w1 != w2 and w2 in w1.next and
79     (s1 in w1.exists) and not(s1 in w2.exists) and (s2 in
w1.exists) and not(s2 in w2.exists)
80 }
81
82 //situation s2 ends at the same time as situation s1 (identical to
finishes)
83 //           s2 |-----|
84 // s1 |-----|
85 pred finishedby[w1: World, w2: World, s1: univ, s2: univ, exists:
univ->univ] {
86     w1 != w2 and w2 in w1.next and
87     (s1 in w1.exists) and not(s1 in w2.exists) and (s2 in
w1.exists) and not(s2 in w2.exists)
88 }
89

```

```

90 //situation s1 exists through the entirety of situation s2
91 // s1 |-----|
92 //           s2 |-----|
93 //w1->w2/w3->w4, w2 and w3 are not necessarily different
94 pred includes[w1: World, w2: World, w3: World, w4: World, s1: univ,
95   s2: univ, exists: univ->univ] {
96     w1 != w2 and w1 != w3 and w1 != w4 and w2 != w4 and w3 != w4
97     and
98     w2 in w1.next and w3 in w2.*next and w4 in w3.next and
99     (s1 in w1.exists) and (s1 in w2.exists) and (s1 in
100    w3.exists) and (s1 in w4.exists) and
101    not(s2 in w1.exists) and (s2 in w2.exists) and (s2 in
102    w3.exists) and not(s2 in w4.exists)
103 }
104
105 //situation s2 exists through the entirety of situation s1
106 // s2 |-----|
107 //           s1 |-----|
108 //w1->w2/w3->w4, w2 and w3 are not necessarily different
109 pred during[w2: World, w1: World, w3: World, w4:World, s1:
110   univ, s2: univ, exists: univ->univ] {
111     w1 != w2 and w1 != w3 and w1 != w4 and w2 != w4 and w3 !=
112     w4 and
113     w2 in w1.next and w3 in w2.*next and w4 in w3.next and
114     (s2 in w1.exists) and (s2 in w2.exists) and (s2 in
115     w3.exists) and (s2 in w4.exists) and
116     not(s1 in w1.exists) and (s1 in w2.exists) and (s1
117     in w3.exists) and not(s1 in w4.exists)
118 }
119
120 //situation s1 starts at the same time as situation s2
121 // s1 |-----|
122 // s2 |-----|
123 pred starts[w1: World, w2: World, s1: univ, s2: univ, exists:
124   univ->univ] {
125     w1 != w2 and w2 in w1.next and
126     not(s1 in w1.exists) and (s1 in w2.exists) and
127     not(s2 in w1.exists) and (s2 in w2.exists)
128 }
129
130 //situation s2 starts at the same time as situation s1
131 // s2 |-----|
132 // s1 |-----|
133 pred startedby[w1: World, w2: World, s1: univ, s2: univ,
134   exists: univ->univ] {
135     w1 != w2 and w2 in w1.next and
136     not(s1 in w1.exists) and (s1 in w2.exists) and
137     not(s2 in w1.exists) and (s2 in w2.exists)
138 }
139
140 //situation s1 coincides with situation s2
141 // s1 |-----|
142 // s2 |-----|
143 pred coincides[w1: World, w2: World, w3: World, w4:World, s1:
144   univ, s2: univ, exists: univ->univ] {
145     w1 != w2 and w1 != w3 and w1 != w4 and w2 != w4 and w3 !=
146     w4 and
147     w2 in w1.next and w3 in w2.*next and w4 in w3.next and
148     not(s1 in w1.exists) and (s1 in w2.exists) and (s1
149     in w3.exists) and not(s1 in w4.exists) and

```

```
135         not(s2 in w1.exists) and (s2 in w2.exists) and (s2
    in w3.exists) and not(s2 in w4.exists)
136     }
```

APPENDIX B - SIMULATION SCENARIOS FROM (SALES, 2014)

In (SALES, 2014) the author proposes a number of simulation scenarios, which users can parameterize and combine in order to validate OntoUML models in Alloy. We replicate some of those scenarios here that are also interesting for situation type models validation (one should check the referred work if other scenarios are required, especially if dealing with dynamics of ontology classes, since we here focus only on situation types). With this we intend to provide a more efficient option to validate situation type models than running unconstrained simulations such as our example of section 5.2.4. The parameterization of a scenario in the provided sentences is identified by using brackets ([]). It has two uses: first, to indicate the need to specify a numeric value, like “at least [n] instances of class”. Second, it can detail alternative options, like “[every / no / at least / at most / exactly] worlds must have”.

Next we present the scenarios. For each scenario we will provide a description of it and later a table containing a respective natural language sentence (as if demanded by the modeler) and alloy expression that a user should add to the Alloy specification (within either a fact, a predicate or in the *run* command before running the simulation).

Linear Branch

This scenario defines that every simulation will generate a linear world structure, i.e., a branch in which exactly one world does not have a successor and exactly one does not have a predecessor. All the others worlds must have a predecessor and a successor. This structure is how we commonly think of things, a linear sequence of events. Therefore, it relieves users from the cognitive work of understand the world order.

Table 7. Linear Branch - scenario description.

Sentence
<i>I want to see a linear story.</i>
Alloy Expression
<code>one w:World no w.next</code> <code>one w:World no next.w</code>

Alternative Futures

This scenario defines a world branch composed by a unique world that leads to alternative futures. It does not generate counterfactual or past worlds. Branches fitting this pattern are useful to analyze what can happen to an individual after a given setting. For instance, if a couple is married in a world, some possible futures are: they can either continue to be married, break up or even break up and marry other people. This can be used to check different behaviors of situation types from a single point in time.

Table 8. Alternative Futures - scenario description.

Sentence
<i>I want to see different outcomes for a same setting.</i>
Alloy Expression
<code>one w:World no next.w && all w2:World w!=w2 implies w2 in w.next</code>

Counterfactual Worlds

Counterfactual worlds exemplify alternative possibilities in the past, i.e., alternative future worlds from a past world. This scenario is specified as a world branch that contains at least two distinct worlds, w1 and w2, which share a common past world and either w1 and w2 have a next world. This type of scenario is also useful to analyze alternative turn of events.

Table 9. Counterfactual Worlds - scenario description.

Sentence
<i>I want to see that things may have taken a different outcome in the past.</i>
Alloy Expression
<code>some w1,w2:World w1 != w2 && next.w1 = next.w2 && (some w1.next or some w2.next)</code>

Branch Depth

The depth of a branch corresponds to the number of consecutive worlds it has, i.e., a set of worlds within a branch that characterize a linear branch by themselves. The Alternative Futures scenario implies an exact world depth of two, since all futures come directly from the same world.

Table 10. Branch Depth - scenario description.

Sentence
<i>I want to see a story composed [at least / at most / exactly] of [n] consecutive worlds.</i>
Alloy Expression
<pre>-- minimum_world_depth some w1,w2:World w1 != w2 && next.w1 = next.w2 && (some w1.next or some w2.next) -- maximum world depth no w1,w2,w3:World w2 in w1.next and w3 in w2.next</pre>

Content Constraints

Content Constraint scenarios regard restricting the contents of worlds, instead of their branch structure. These scenarios are useful to help modelers customize the simulation and facilitate the generation of particular settings of entities. Moreover, it provides users with some upfront knowledge about the worlds that the Alloy Analyzer

will generate, facilitating the cognitive task of understanding the simulation results. Those scenarios are especially useful for validating situation types since one can manipulate the settings and check in whether circumstances a situation will be instantiated or not.

Population Size

The population of a world corresponds to the set of individuals that exist within that World, regardless if it is an entity (Person, Student), property (Marriage, Disease) or situation (Fever). The population size scenario allows imposing upper and/or lower bounds for the size of a population. For instance, one may instruct the analyzer to generate worlds with at least four and at most eight individuals. This scenario is mostly useful for validating situations if combined to check whether, for example, a situation of some type can be instantiated with a limited population size.

Table 11. Population Size - scenario description.

Sentence
<i>I want to see a story with [at least / at most / exactly] [n] individuals.</i>
Alloy Expression
<code>all w: World #w.exists = n</code>

Population Variability

This scenario regards defining the variability of world population throughout the branch. One can define it as constant, where every world contains the same individuals, although they can instantiate different types. Conversely, one can set it as variable, forcing the generation of branches composed by worlds with necessarily different populations.

Although a constant population will always have the same size, it is not true that a variable population must have different sizes. Two populations are different if they do not have the same elements, and they can still do that having the same number of

individuals. Thus, one can combine this scenario with the population size without generating any inconsistencies.

The main cognitive advantage of defining a constant population is that one does not need to be concerned with the dynamics of object creation and destruction. When inspecting a world, one can focus exclusively on the instantiation of situations that, for example, relies on instantiation of anti-rigid types. It is interesting to keep a variable situation population, since their instantiation means actually creation and destruction of situation objects. For this reason, one can optionally apply partial rules that restrict the population of entities (technically called objects, such as a Kind, a SubKind, Roles, etc.) and leave property classes (Relators, Qualities and Modes) and situation classes variable. Combining a complete variable population rule with partial constant population rule (only applied for entities, for example), necessarily changes the population as a whole but keeping the wanted population constant.

Table 12 provides a straightforward natural language description of this scenario with the two customizable points: the first regarding if the population varies or not; the second regarding which part of the world population the modeler wants to apply the constraint – the whole population, only entities, properties, or situations. We provide the respective Alloy expression for each possible combination.

Table 12. Population Variability - scenario description.

Sentence
<i>I want to see a story where every moment [has the same / has different] [objects / properties / individuals].</i>
Alloy Expression
<pre> -- variable population all w1,w2:World w2 != w1 implies w1.exists != w2.exists -- variable entity population all w1,w2:World w2 != w1 implies w1.exists:>Object != w2.exists:>Object -- variable property population all w1,w2:World w2 != w1 implies w1.exists:>Property != w2.exists:>Property </pre>

```

-- variable situation population
all w1,w2:World | w2 != w1 implies w1.exists:>Situation !=
w2.exists:>Situation

-- constant population
all w1,w2:World | w1.exists = w2.exists

-- constant entity population
all w1,w2:World | w1.exists:>Object = w2.exists:>Object

-- constant property population
all w1,w2:World | w1.exists:>Property = w2.exists:>Property

-- constant situation population
all w1,w2:World | w1.exists:>Situation = w2.exists:>Situation

```

Population Growth

The Population Growth scenario defines constraints between worlds that are directly accessible. In the incremental scenario, no individual ceases to exist in the future. For instance, if x exists in world w_0 , then in all worlds that follow it, x must also exist. That does not exclude the possibility of new things being created in future worlds, but also does not require.

Conversely, in decrement branches, we reverse this constraint: if an individual does not exist in the initial world of the branch (the one that has no predecessor) it will never come to life. From a world to its next individuals can keep existing, but none “comes to life”. Notice that there is intersection between incremental and decremental branches: the one where every world has the same individuals. Thus, specifying a scenario as both incremental and decremental is equivalent as defining it as constant.

Table 13. Population Growth - scenario description.

Sentence

I want to see a story exclusively composed by individuals [coming to existence / ceasing to exist].

Alloy Expression

```
-- incremental_worlds
all w1,w2:World | w2 in w1.next implies w1.exists in w2.exists
-- decremental_worlds
all w1,w2:World | w2 in w1.next implies w2.exists in w1.exists
```

Extension Size

The extension of a class in a given world corresponds to the set of individuals that are currently instantiating it. This scenario proposes the definition of lower and/or upper bounds to the size of a class' extension. Users can choose to enforce this constraint in all worlds or just a subset of them. For instance, one can require that the class Person always have exactly three objects instantiating it in every world in the generated branch.

One can check the possibility of instantiating situation types if some class remains with only a limited number of individuals, for example. In situation types that require more than one world to happen, for instance, this scenario is especially useful to check inconsistencies when the number of individuals remains the same (the constraint must be applied to every world).

Table 14. Extension Size - scenario description.

Sentence

I want to see a story composed [only / at least / at most] by worlds with [at least / at most / exactly] [n] instances of [Class].

Alloy Expression

```
all w: World | #w.Class = n
```

Temporal Extension Size

This scenario is very similar to Extension Size. The difference is that instead of defining the number of individuals that instantiate a class within a world, one can define the lower and/or upper bounds for the set containing all individuals that

instantiate it throughout time. For example, if one defines a temporal extension of three for the class Person, there can only be three distinct individuals in all worlds that instantiate Person. Notice that this is not the sum of the class' extension in each world, but the sum of distinct individuals.

Table 15. Temporal Extension Size - scenario description.

Sentence
<i>I want to see a story with [at least / at most / exactly] [n] instances of [Class].</i>
Alloy Expression
<code>#World.Class = n</code>

Extension Variability

In the Extension Variability scenario, instead of setting the variability nature of the entire world population, one sets it only on the extension of a class. On one hand, enforcing a variable extension for a class implies that the individuals that instantiate it will be different for any two worlds in every generated branch. On the other hand, a constant class extension means that the same set of individuals will instantiate it in every world of the branch.

Requesting a variable extension for a rigid class or situations types implies a variable population, because extensions of rigid types and situations only change with instance creation or destruction. Thus, one cannot request variable extension for those classes and constant population.

Table 16 provides the representation of this scenario in natural language, alongside with the Alloy expressions that characterize it. The first expression requires "Class" to have different extensions in all worlds, whilst the second requires all extensions to be equal.

Table 16. Extension Variability - scenario description.

Sentence
<i>I want to see a story where the set of instances of [Class] [always / never] change from world to world.</i>
Alloy Expression
<pre>-- variable extension all w1,w2:World w2!=w1 implies w1.Class != w2.Class -- constant extension all w1,w2:World w1.Class = w2.Class</pre>