Fraunhofer IOSB
Institut für Optronik,
Systemtechnik
und Bildauswertung

Comparing Ontologies and Situation Graph Trees in Cognitive Vision Systems

Diploma Thesis

by

Sebastian Bauer

30. September 2012

Supervisor: Prof. Dr.-Ing. Rainer Stiefelhagen[1]

Advisor: Dr. rer. nat. Michael Arens

Advisor: David Münch

Fraunhofer Institut für Optronik, Systemtechnik und Bildauswertung

Gutleuthausstr. 1, 76275 Ettlingen

[1]Computer Vision for Human-Computer Interaction Lab - Karlsruhe Institute of Technology

I declare that I have developed and written the enclosed Diploma Thesis completely by myself, and have not used sources or means without declaration in the text.

Ettlingen, 30. September 2012

Sebastian Bauer

Abstract

The recognition of situations by machines is an important and comprehensive field of research. The goal is to give interpretations automatically about what is going in the observed scene. In the SGT/Fmthl framework, situations can be recognized which previously were defined by use of a-priori expert knowledge. Situation Graph Trees (SGT) provide a representation of this expert knowledge in a coherent way for humans as well as for machines. With the rise of the so called *Semantic Web*, another universal and exchangeable knowledge base establishes in the form of ontologies. This diploma thesis examines the capability to describe an SGT with an ontology to provide a general interface for knowledge provided by the rest of the world.

First, an ontology is designed that can represent an SGT. Based on this ontology, an application was written to automatically transform an existing SGT into an ontology. For the ability to use situation definitions from the rest of the world, the application was extended to provide the transformation in the backwards direction. The ontology to be converted is expected to describe an SGT. Third-party ontologies may be processed as long as there is a transformation ontology that maps it to an SGT. To achieve the support for such third-party ontologies, the application searches the class hierarchy until corresponding concepts in the SGT domain were found. Then, the transformation to an SGT is applied.

This thesis proves the equivalence of SGTs to the ontological SGTs. It is shown that the application proposed above does not lose informations during the transformations and that it terminates. The existence of such a program provides the evidence of equivalence.

The contribution of this thesis allows the replacement of the actual proprietary representation format for the more general ontologies. Further, third parties can use our situation definition by only using standard tools. Future work is the design of transformation ontologies which goes hand in hand with the comparison of different theories about the representation of situations.

Zusammenfassung

Das maschinelle Erkennen von Situationen ist ein umfassendes Forschungsgebiet. Das Ziel hierbei ist es, eine Maschine verwertbare Aussagen darüber treffen zu lassen, was in einer aktuell beobachteten Szene gerade passiert. Aktuell können Situationen erkannt werden, welche a-priori mittels Expertenwissen definiert wurden. Sowohl für Maschinen als auch für Menschen leicht verständliche Form der Repräsentation dieses Expertenwissens bieten Situationsgraphenbäume (SGT). Mit dem Aufkommen des sogenannten *Semantic Web*, etablieren sich derzeit allgemeine, tauschbare Wissensbasen in Form von Ontologien. Diese Diplomarbeit untersucht die Möglichkeit, SGTs in einer Ontologie abzubilden, um eine allgemeine Schnittstelle für verwandte Arbeiten zu bieten.

Zunächst wurde eine Ontologie entworfen, in welcher SGTs dargestellt werden können. Darauf aufbauend wurde ein Programm zur automatisierten Transformation von existierenden SGTs in Ontologien geschrieben. Damit auch Situationsbeschreibungen verwandter Arbeiten verwendet werden können, wurde das Programm um die Transformation der entsprechenden Rückrichtung erweitert. Es erwartet dabei, dass die einzulesende Ontologie einen SGT beschreibt. Dabei wurde darauf geachtet, dass es auch möglich ist fremde Ontologie einzulesen, solange sie mittels einer Transformationsontologie auf einen SGT abgebildet worden ist. Umgesetzt wurde dies indem das Programm die Klassenhierachie der Ontologie durchsucht, bis entsprechende Begriffe aus dem Diskursbereich SGT gefunden wurden und wendet dann darauf die Transformation in einen SGT an.

Diese Arbeit beweist schließlich die Äquivalenz von SGTs mit denen in der Ontologie. Der Beweis wird unter Zuhilfenahme des oben beschriebenen Programmes geführt, indem aufgezeigt wird, dass zum einen bei der Transformation keine Informationen verloren gehen, sowie zum anderen es terminiert. Die Existenz eines solchen terminierenden Programmes genügt dem zu Zeigenden.

Der Beitrag dieser Arbeit erlaubt das Ablösen des bisher proprietären Repräsentationsformats von SGTs durch die allgemeineren Ontologien, sowie das damit verbundene Nutzen unserer Situationsdefinitionen von Dritten unter Zuhilfenahme von Standardwerkzeugen. Für die Zukunft noch offene Punkte sind unter anderem das Erstellen oben beschriebener Transformationsontologien, welcher einher geht mit dem Vergleich unterschiedlicher Theorien zur Repräsentation von Situationen.

Contents

Chapter 1

Introduction

Since the development of the internet in the early 1980s, it became more and more popular and thus larger from year to year. Nowadays, one can barely imagine living without internet in daily life. From a scientific view the amount and the exchange of information offers lots of possibilities. One example is giving semantics to the informations in the internet or trying to extract semantics from the internet. The long-term aim is about machines gaining the ability to process and to understand these informations. An approach towards this aim gives the World Wide Web Consortium by its definition of the Web Ontology Language Owl (McGuinness and Harmelen, 2004) which helps to conceptualize the world in a machine-readable way.

An other part about understanding the world besides natural language is visual cognition. Knowing the context of people in a scene is a base requirement in robotics. To give an example, a good autonomous service robots helps human people in situations where they are needing assistance but does not bother them otherwise. Situation Graph Trees (SGT) are developed to describe the knowledge about the expected behaviour of agents in a situation. The SGTs are used to recognize situations thus machines are capable of reacting to them. In contrast to Owl, the file format for SGTs is highly domain specific and beyond that, restricted to applications dealing with SGTs.

There is a lot of other work on situation awareness. Of course only few of them uses SGTs as the format of their definitions of situations. Situations has to be made at every group that does not use the same knowledge representation. By combining SGTs and ontologies, we want to move a step closer to the semantic exchange of information.

Ontologies are not only about the exchange of informations on the internet. As mentioned in the first place, they enhance plain informations with semantics. At this point, it is of interest whether the ontology of the SGT may be enhanced by additional informations. This information must not matter in the field of SGTs but could be useful in other semantic analysis of SGTs.

1.1 Aim and Contribution

This thesis compares ontologies with Situation Graph Trees. In other words, we analyse the ability of representing an SGT in an ontology and vice versa. The comparison is achieved by providing an application that has the ability to convert an SGT into an ontology and vice versa. Having the definition of a situation in a standardized and commonly accepted file format for knowledge representation, other research groups may implement the import of SGTs without the necessity to deal with the SGT-specific knowledge representation format – SIT++, the current file format for situation graph trees.

We have also in mind the usage of foreign knowledge about situation, too. Thus, ontologies about situation awareness are processable in the SGT environment. For this goal, we extend the application for converting SGTs into an ontology by the way backwards. We show that the informations which are encoded in the SGT ontology are equivalent to the content of an SGT and we proof that the application works correctly.

Having both ways of the conversion, the ontology serves us as an interface for knowledge exchange.

1.2 Contents

This thesis is structured as follows: Chapter 2 covers the foundations. Beginning with an introduction to *Description Logic*, we describe ontologies in general and give some examples where they are applied actually. The *Web Ontology Language* is presented as well as a graphical editor Protégé for it. The chapter continues with the introduction of situation recognition: *Situation Graph Trees* for the description of the expected behaviour of agents and Fuzzy Metric Temporal Horn Logic (Fmthl) as the language for its reasoning. The chapter closes by embedding this thesis in the context of a *Cognitive Vision System*. The following Chapter 3 presents related work. It is divided in two main parts: first, it presents situation awareness by other groups. Second, several existing approaches about using ontologies in situation recognition are introduced.

We describe the design of our ontology in Chapter 4. Its first section is about the first attempt of encoding an SGT to an ontology. A discussion about the first attempt and the current design of our SGT ontology is proposed in the section afterwards. This chapter also shows a way of merging other ontologies with the SGTs, thus we are capable of using other ontologies in the domain of situation recognition. The implementation in Java is described in Chapter 5 where the used design patterns and the program procedures are proposed. The Chapter 6 shows the evidence that situation graph trees can be transformed to an ontology without loss of information. The argumentation is divided in the direction from a situation graph tree to an ontology and in the direction backwards from

an ontology to a situation graph tree. It is provided by giving a program that implements the transformation. This thesis finishes in Chapter 7 by giving a prospect of future work and a conclusion.

Chapter 2

Foundations

2.1 Description Logic

As this thesis is about the comparison of two different knowledge representation languages, we introduce *Description Logic* (DL) as the foundation of both of them. Although, there is previous work on knowledge representation languages, the beginning of research on DL is regarded to be in 1985 with the first work that addresses the trade-off between the expressiveness of Kl-One-like languages (Brachman and Schmolze, 1985) and the computational complexity of reasoning (Baader *et al.*, 2007, p. xv). This section of the thesis presents a basic overview based on (Baader *et al.*, 2007).

2.1.1 Syntax and Basic Description Language

Beginning with the syntax of DL, we use $P \sqcap W$ as an example, where the unary predicate symbols P and W denote *concepts*. Concepts can be interpreted as a set of individuals fulfilling the defined properties and are — in contrast to first order logic — variable-free. The operation symbol $\sqcap$ represents the intersection of two concepts, further operators are union ($\sqcup$) and complement ($\neg$). For example, if we state, that P represents "parent" and W "women", their intersection implies the concept of "mothers". To map concepts to each other, you use *roles*, represented by a binary predicate symbol, for example $\mathsf{hasChild}(P, C)$. Here the role is called $\mathsf{hasChild}$, while the concepts are P (parent) and C (child). This leads to a basic construct called *value restriction*, which allows establishing relationships between concepts. In its syntactical representation, value restrictions consist of a quantifier ($\forall$, $\exists$) followed by the relationship R, a dot and a concept C: i.e. $\forall R.C$. Let us explain the semantics of a value restriction with the following example:

$$\forall \mathsf{hasChild}.\mathsf{Female}$$

The concept Female is used as a *role filler* for the second argument of the role hasChild, meaning that it is restricted to, or filled with concepts of the value Female. Thus, the whole value restriction can be translated to "individuals all of whose children are female" or less formal "parents, which only have daughters".

This basic syntax leads to a formal definition of a description language. The basic description language is $\mathcal{AL}$, the attributive language. Allowed concept descriptions are as follows:

$$
\begin{array}{lll}
C, D \longrightarrow & A \mid & \text{(atomic concept)} \\
& \top \mid & \text{(universal concept)} \\
& \bot \mid & \text{(bottom concept)} \\
& \neg A \mid & \text{(atomic negation)} \\
& C \sqcap D \mid & \text{(intersection)} \\
& \forall R.C \mid & \text{(value restriction)} \\
& \exists R.\top & \text{(limited existential quantification)}
\end{array}
$$

This minimal language is of practical interest (Schmidt-Schauß and Smolka, 1991). It serves as basis for the family of $\mathcal{AL}$-languages, which can be extended by several properties, see Table 2.1:

Table 2.1: Overview of the $\mathcal{AL}$-language family.

Indicator	Description	Syntax
$\mathcal{U}$	Union of concepts	$C \sqcup D$
$\mathcal{E}$	Full existential quantification	$\exists R.C$
$\mathcal{N}$	Number restrictions	$\geqslant n\ R$ (at-least restriction) and $\leqslant n\ R$ (at-most restriction), where n ranges over the non-negative integers
$\mathcal{C}$	Negation	$\neg C$

2.1.2 TBox and ABox

The TBox is the terminology in the DL knowledge base for classifying concepts. In general, classification takes place by the *definition* of new concepts in terms of other previously defined concepts and in the further *specialization* of concepts. The following

example defines a woman as a female person:

$$\text{Woman} \equiv \text{Person} \sqcap \text{Female}$$

whereas this example specifies a woman as a subsumption of a person:

$$\text{Woman} \sqsubseteq \text{Person}$$

The TBox is the set of concept definitions where every concept is defined once. Figure 2.1 shows such a simple TBox with concepts about family relationships.

$$\begin{aligned} \text{Woman} &\equiv \text{Person} \sqcap \text{Female} \\ \text{Man} &\equiv \text{Person} \sqcap \neg\text{Woman} \\ \text{Mother} &\equiv \text{Woman} \sqcap \exists\text{hasChild.Person} \\ \text{Father} &\equiv \text{Man} \sqcap \exists\text{hasChild.Person} \end{aligned}$$

Figure 2.1: A TBox with concepts about family relationships.

A cycle in the terminological box occurs by defining a concept with the concept itself directly or in its expansion. An example for a cycle is the definition of a human by an animal, whose parents all are humans:

$$\text{Human}^{\text{I}} \equiv \text{Animal} \sqcap \text{hasParent.Human}^{\text{I}}$$

The expansion of a TBox is the recursive evaluation of the right side of each concept definition. The expansion of the family relationship TBox can be seen in Figure 2.2.

$$\begin{aligned} \text{Woman} &\equiv \text{Person} \sqcap \text{Female} \\ \text{Man} &\equiv \text{Person} \sqcap \neg(\text{Person} \sqcap \text{Female}) \\ \text{Mother} &\equiv (\text{Person} \sqcap \text{Female}) \sqcap \exists\text{hasChild.Person} \\ \text{Father} &\equiv (\text{Person} \sqcap \neg(\text{Person} \sqcap \text{Female})) \sqcap \exists\text{hasChild.Person} \end{aligned}$$

Figure 2.2: The expansion of the family relationship TBox from Figure 2.1.

A TBox is called cyclic if the concept definitions contain one cycle in minimum, acyclic otherwise. A TBox containing specializations is called general, specializations are not subject to the requirement of being acyclic. Having a cyclic TBox leads to powerful description logic as well as increasing effort in reasoning. The definitions and specializations in the TBox are not bound to concepts only. You can replace them by roles, for example $\mathsf{hasChild} \sqsubseteq \mathsf{hasRelative}$.

The ABox contains TBox-compliant *assertions* about the domain of interest. The assertions itself are subdivided in concept and role assertions. The ABox introduces concrete individuals and represents the extensional knowledge in the description logic. To give an example,

$$\mathsf{Female} \sqcup \mathsf{Person}(\mathsf{ANNA})$$

states that the individual ANNA is a female person and is of type concept assertions. An example for a role assertions is the statement, that the individual ANNA has a child called PAUL:

$$\mathsf{hasChild}(\mathsf{ANNA}, \mathsf{PAUL})$$

Later in this thesis we present various types of TBoxes and ABoxes.

2.1.3 Reasoning and Complexity

Corresponding to the operators allowed in a description logic, it has a different expressiveness. The more you want to express, the more difficult the DL becomes to reason. Reasoning, in the context of the description logic, is defined as logical inferences. Satisfiability of $\mathcal{ALCN}$-concept descriptions is PSpace-complete in acyclic terminologies (Schmidt-Schauß and Smolka, 1991), therefore it is decidable but a touring machine needs polynomial account of space while solving it. Consistency of $\mathcal{ALCN}$-ABoxes is PSpace-complete, w.r.t. general inclusion axioms ($\sqsubseteq$) ExpTime-complete (Baader *et al.*, 2007, Theorem 2.26, 2.27, and 2.28). The PSpace-completeness of $\mathcal{ALC}$ with transitive and inverse roles is shown in (Baader *et al.*, 2008). Decision problems in ExpTime can be decided by a deterministic touring machine in time of $O(2^{p(n)})$ where $p(n)$ is a polynomial function of n. PSpace and ExpTime are complexity classes like P and NP. If in (currently assumed) circumstances $P \neq NP$ holds, then the following inclusion should bring the complexity of reasoning in mind: $P \subset NP \subset PSpace \subset ExpTime$, Figure 2.1.3 gives a graphical representation of that proportions.

2.2 Ontology

Computer Science adopted the word ontology from the philosophy. The word has its origins in Greek, composed of the two words “onto” which means being and “logos” – theory. According to its translation, ontology is the theory about the nature of being

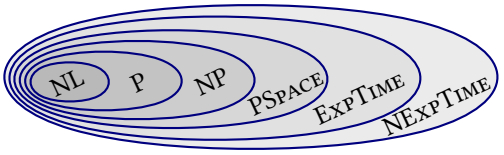

Figure 2.3: Diagram about the relationship between complexity classes. The set NL of decision problems solvable by a non-deterministic touring machine using logarithmic amount of memory is a subset of P which itself is a subset of NP under the assumption P $\neq$ NP. The complexity class PSpace containing the decision problems solvable with polynomial amount of memory is superset of them and itself subset of ExpTime. Decision problems in ExpTime can be decided by a deterministic touring machine in time of $O(2^{p(n)})$. The superset NExpTime uses a non-deterministic touring machine for deciding problems, the time requirements are the same as in ExpTime.

and existence by conceptualizing entities. To merge the ontology from philosophy into computer science, we give the definition of an ontology by (Liu and Özsu, 2009): "In the context of computer and information sciences, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application."

A shorter definition gives (Gruber, 1993): "an explicit specification of a conceptualization". From this Sharman *et al.*, 2007 infers that an "ontology is based on the idea of conceptualization: a simplified view of the world that we want to represent. Conceptualization is the process by which the human mind forms its idea about part of the reality".

The following examples show existing ontologies, its format and how they are used. The Friend of a Friend (FOAF) Ontology (Brickley and Miller, 2010) provides definitions for persons and groups in the machine readable format RDF. It is used in three kinds of networks: social networks of human collaboration, friendship and association. The NCI-Thesaurus (NCIt) provided online at http://ncit.nci.nih.gov/ by the National Cancer Institute (NCI), can be seen as a comprehensive ontology. The NCIt is available in different formats, including Owl (refer to Section 2.3). The NCI offers a web interface,

the "NCI Thesaurus Browser", to browse and query the ontology. But you also have the choice to download and install the ontology and query the ontology by using for example Protégé (see Section 2.3.2). The NCI itself offers a version of Protégé, bundled with some NCI specific plug-ins. Another ontology in the medical domain is the Gene Ontology (GO). The GO is a free database with information about the gene product attributes and aims to standardise the terms used in the bioinformatics. The Gene Ontology can be found online at http://www.geneontology.org/. Like the NCIt, the GO brings its own tools to browse through an query the ontology. But, as the GO ontology itself is stored in a MySQL database, you are free to install a copy of the GO on your local machine.

2.3 Web Ontology Language

The Web Ontology Language (Owl) was created by the World Wide Web Consortium to provide a standardised knowledge representation language. The primary intention of Owl was to build a inter-operational language for the semantic web. Currently, the only languages that are standardized and commonly supported by popular software tools and systems are the languages of the Semantic Web[1]: the Resource Description Framework and the Web Ontology Language, which is based on RDF (Kokar *et al.*, 2007, p. 84).

The definitions of operations and objects in Owl are very similar to description logics. The counterpart of concepts in description logics are classes in Owl. Classes can be organized in a hierarchical manner by a *SubClass* relation, semantically equivalent to a *is-a* relation – $\sqsubseteq$ in description logics – or by a *EquivalentClass* relation which is outlined by the $\equiv$ operator on concepts in description logic.

The Owl-equivalent of roles are *object properties*. Object properties may be organized in a hierarchy as well as classes. A domain and a range can be specified for an object property. The domain defines the classes, the object property may be assigned to, and the range defines the allowed classes of the filler object. A inverse object property can be defined: the inverse object property of hasChild is hasParent, for example. Further, property chains can be specified for an object property. A property chain is the set of object properties, that implies an other object property. I.e. the property chain hasParent $\circ$ hasBrother implies hasUncle.

Some characteristics can be added to object properties in Owl: functional, inverse functional, transitive, symmetric, asymmetric, reflexive and irreflexive. A description of functional and transitive properties can be found in the Section 2.3.1. An example for symmetric properties is the "sibling" relation: if for individuals ANNA and PAUL the symmetric relation hasSibling(ANNA, PAUL) is asserted, then hasSibling(PAUL, ANNA) infers.

[1] http://www.w3.org/2001/sw/

A reflexive property relates everything to itself, for example everyone has himself as a relative.

Object properties are restricted to individuals of classes as arguments. To assert relations from individuals to literals there exists *data properties*. With the exception of inverse properties and property chains, data properties can be specified with the same descriptions as object properties, the only specifiable characteristic is "functional".

```
<owl:Class rdf:about="&my;Person">
  <rdfs:subClassOf rdf:resource="&owl;Thing"/>
</owl:Class>
```

Figure 2.4: Definition of a subclass axiom in RDF/XML.

In its first version (McGuinness and Harmelen, 2004), OWL was built as an extension of the Resource Description Format (RDF) (Horrocks *et al.*, 2003) with XML as the underlying representation language. Figure 2.4 illustrates the declaration of the class Person as well as the axiom Person $\subset$ Thing in RDF/XML. As XML is well readable and writeable by machines, humans may have problems to create, maintain or debug complex ontologies. As a consequence of the latter, the current version Owl2 proposed by (Bao *et al.*, 2009) provides amongst others the new Owl Functional-Style Syntax (Horridge and Patel-Schneider, 2009). The Functional-Style Syntax is more human-readable and used in this thesis in examples and as storage format. In Figure 2.5 is displayed the same example as in Figure 2.4, but in Owl functional-style syntax.

```
Declaration(Class(my:Person))
SubClassOf(my:Person owl:Thing)
```

Figure 2.5: Definition of a subclass axiom in Owl functional-style syntax.

2.3.1 Expressiveness and Complexity of Owl

To focus expressiveness of Owl, we first present in Table 2.2 some extensions to the basic $\mathcal{AL}$-languages.

Table 2.2: Overview of extensions to the $\mathcal{AL}$-language family.

Indicator	Description	Syntax
$\mathcal{S}$	abbreviation for $\mathcal{ALC}$ including transitively closed roles	$\mathcal{ALC}$: see Section 2.1.1 and Table 2.1, transitivity: $R \in \mathbb{R}_+$
$\mathcal{H}$	role hierarchy	$R \sqsubseteq S$
$\mathcal{O}$	nominal	o
$\mathcal{I}$	inverse role	R^-
$\mathcal{Q}$	qualifying number restrictions	$\geqslant n\ P.C$ $\leqslant n\ P.C$
$\mathcal{F}$	functional properties	$\geqslant 1\ P.C$
(D)	datatype properties, data values or datatypes	

The symbol P, R, and S represent roles, o is the name of an individual, n a natural number, and C stands for a role. The indicator $\mathcal{S}$ is an abbreviation for $\mathcal{ALC}$ including transitively closed roles. An example for transitivity is the role hasSibling: if the axioms hasSibling(ANTON, MARKUS) and hasSibling(MARKUS, ANNA) hold and hasSibling is declared transitively, then hasSibling(ANTON, ANNA) infers. Role hierarchy ($\mathcal{H}$) as partly introduced in Section 2.1.2 allows the declaration of a sub-role-relation. To give an example, notice the role subsumption hasChild $\sqsubseteq$ hasRelative, which means that from the role axiom hasChild(CAESAR, BRUTUS) also infers hasRelative(CAESAR, BRUTUS).

By using nominals ($\mathcal{O}$) you can add enumerated classes to your ontology, for example a class CardinalDirection does only contain the enumerated individuals NORTH, EAST, SOUTH, and WEST. They can used in conjunction with the object value restrictions oneOf or hasValue. An example for the inverse role ($\mathcal{I}$) of hasChild is hasParent.

Number restrictions ($\mathcal{N}$, see Table 2.1) are a special case of qualified number restrictions ($\mathcal{Q}$), where the class C is the top class Thing/$\top$. Qualified number restrictions are used to further restrict the definition of object used as parameters for roles. Let us state the role wearClothes, which range is already set to the class Clothes. So, subclasses of Clothes like Trousers and Socks may be set as role fillers for wearClothes. By combining the declaration of a object property with number restrictions, you get for example $\geqslant 2$ wearClothes.Socks as a role restrictions for the default uniform of CIA special agents (unconfirmed). Please note, that qualified number restrictions are not allowed in Owl-DL, although there already are reasoners which can handle qualified number restrictions.

The declaration of a property being functional ($\mathcal{F}$) does not allow to specify two different values for the same object. The indicator (D) denotes the usage of datatype properties, data values or datatypes.

Owl itself is parted in three different subsets with different expressiveness: Owl Lite, Owl-DL and Owl Full. In Owl Lite, the allowed kind of declarations are $\mathcal{SHIF}$, Owl-DL allows $\mathcal{SHOIN}$ and Owl Full allows every concept that is realizable with XML/RDFS. That also includes classes of instances, for example. The downside of Owl Fulls maximum expressiveness is the undecidability and the absence of reasoning algorithms at the moment. While it is doubted that such reasoning algorithms for Owl Full will exist (McGuinness and Harmelen, 2004), there already are various reasoning engines for Owl-DL like Pellet, FaCT++ or HermiT. Owl-DL is decidable (Horrocks *et al.*, 2006), but the reasoning of concept satisfiability with respect to general TBoxes is in NExpTime ($\supset$ ExpTime) – an upper bound is shown by (Tobies, 2001, Corollary 6.31). The definition of NExpTime corresponds to the definition of ExpTime but with the usage of a non-deterministic touring machine. Owl-DL is named due to its correspondence to description logics (McGuinness and Harmelen, 2004). Although Owl-DL was designed to give maximum expressiveness while retaining computational completeness, reasoners can handle the more expressive $\mathcal{SHOIQ}$ in the meantime (Horrocks and Sattler, 2007). Easier to reason but with even less expressiveness is Owl Lite, designed to give users basic functionality to create ontologies with classification hierarchy and simple constraints. Owl Lite supports $\mathcal{SHIF}$, reasoning is in ExpTime (Tobies, 2001, Corollary 6.29,6.30).

In this thesis, we decided to use Owl-DL, as Owl Lite does not suffice and we wanted to avoid computational incompleteness. The next section introduces the Owl editor of our choice.

2.3.2 Protégé: an Editor for Owl

Since 2008, the Stanford University is developing a tool called Protégé[2] for editing Owl ontologies. The development of Protégé is concentrated on two different versions in parallel, 3 and 4. The lower version still is maintained, as its features are not all ported to the newer version for example editing of SWRL-rules. Further, the newer version does only support editing Owl-ontologies with the expressiveness of $\mathcal{SHOIQ}$ or less. As the ontology to be created in this thesis is meant to retain expressiveness of Owl-DL without the need of rules, the newer version matches our requirements perfectly. As an additional benefit, the newer version uses the framework Owl Api (Horridge and Bechhofer, 2009) instead of an own implementation as an abstraction for editing ontologies and therefore has support for Owl 2.0.

[2] http://protege.stanford.edu/

A manual how to use is proposed in (Horridge, 2011). Figure 2.6 shows a screenshot of Protégé 4.1.0 with the Pizza Ontology[3] loaded. The user interface is divided in the tabs "Active Ontology", "Entities", "Classes", "Object Propertie", "Data Properties", "Individuals", "OWLViz", "DL Query", and "OntoGraf", the tab "Etc" is custom configured. In the screenshot, the class hierarchy is on the left side. The class IceCream is painted red, as the reasoner inferred the equivalence of IceCream to Nothing/$\perp$ – in other words, it has detected an inconsistency in the declaration of the class IceCream or its properties. The right side of the editor shows annotations in the top and in the bottom descriptions of the currently selected class CheesyPizza. In the bottom left area of the "Entities" tab there is embedded a view for the hierarchy of object, data, and annotation properties, as well as an overview of asserted individuals by type and datatypes. The tab "Classes" is composed of the same views as in the figure but without the bottom left area. The tabs "Object Propertie", "Data Properties", and "Individuals" are very similar constructed as the yet proposed view. In the view "OWLViz" is displayed a selected class in the hierarchy of subclass axioms by GraphViz in a graphical manner. The "OntoGraph" view also prints a graph of the ontology but has also the ability to display object properties linking classes and is highly interactively. In comparison to the result of GraphViz, the graph may not be very clear. DL queries to the ontology can be asked in the corresponding view.

2.4 Situation Graph Tree

A Situation Graph Tree (SGT) (Arens, 2004) is an extension of the idea of situation graphs by (Krüger, 1991). Situation graph trees are introduced with the objective to describe the knowledge of the expected behaviour of agents. An agent is the current object of interest such as a car in the domain of traffic monitoring or a person in a surveillance scenario. Other objects the agent interacts with are called patients. To obtain such a description, the SGT uses some concepts that are described in the following sections. Later, in the Chapter 4.1, they are compared to the features an ontology is able to provide. The following sections explain the basic concepts of a situation scheme, situation graph, and situation graph tree in a bottom-up manner. The visualization of a situation graph tree and parts of it follow the visual representation of situation graph trees in the SGT-Editor, a tool to create and manipulate situation graph trees (Arens, 2004).

2.4.1 Situation Scheme

A situation scheme describes a situation. A situation scheme consists of three different parts, as Figure 2.7 depicts. First, the name of the situation scheme, than the state scheme

[3] http://www.co-ode.org/ontologies/pizza/pizza.owl

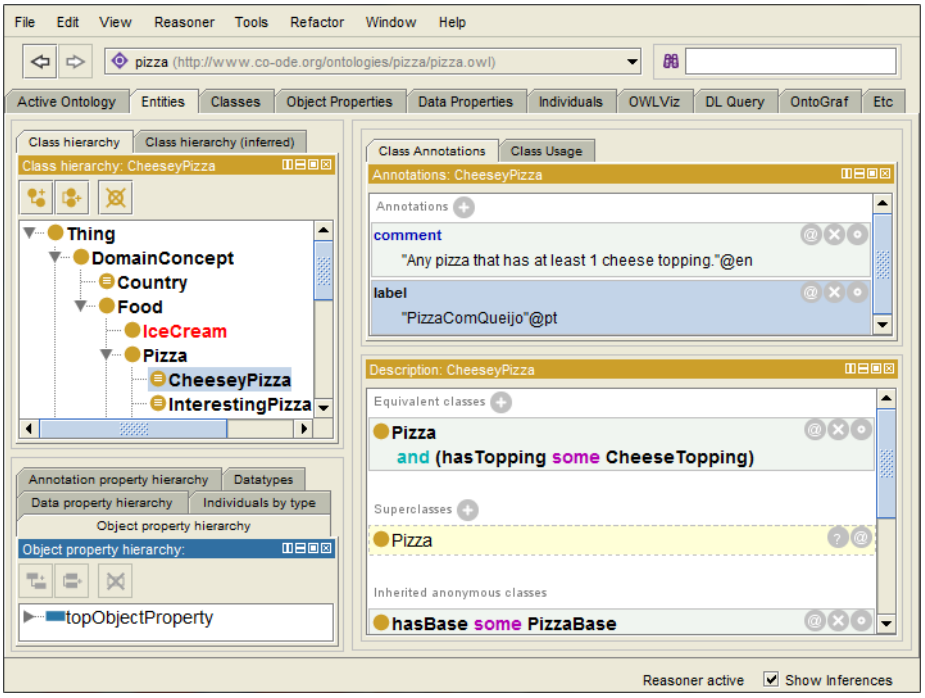

Figure 2.6: Screenshot of the Ontology-Editor Protégé 4.1.0 (Build 239). The image shows the entities view which is divided in 4 parts: the class hierarchy at the top left, the object property hierarchy on the bottom left, annotations at the top right, and descriptions on the bottom right.

and last the action scheme. The situation scheme has an unique name located in the top area, which is used to identify that situation scheme in later processing. The two squares in the top left and top right corner mark the situation scheme as start or end situations and are optional in the situation scheme. Their semantics are described in the next Section 2.4.2. The state scheme describes the requirements an agent has to fulfil to be located in the actual state. The requirements are framed as logical predicates. If all of the predicates are satisfiable, the situation can be instantiated. The action scheme is the description of an agent's course of actions. If the situation was instantiated, the logical predicates — as used in the state scheme — of the action scheme are evaluated. In this step, the variables were allocated with the fulfilling values of the state scheme.

2.4.2 Constructing Situation Graphs

The situation graph models the temporal sequence of situations. Figure 2.4.2 outlines

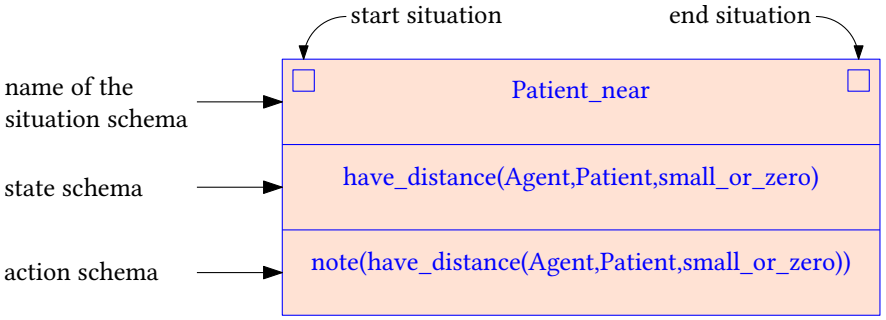

Figure 2.7: Visual Representation of a situation scheme consisting of the name, flags marking the situation as a start and an end situation, a state and an action scheme which both contain a predicate.

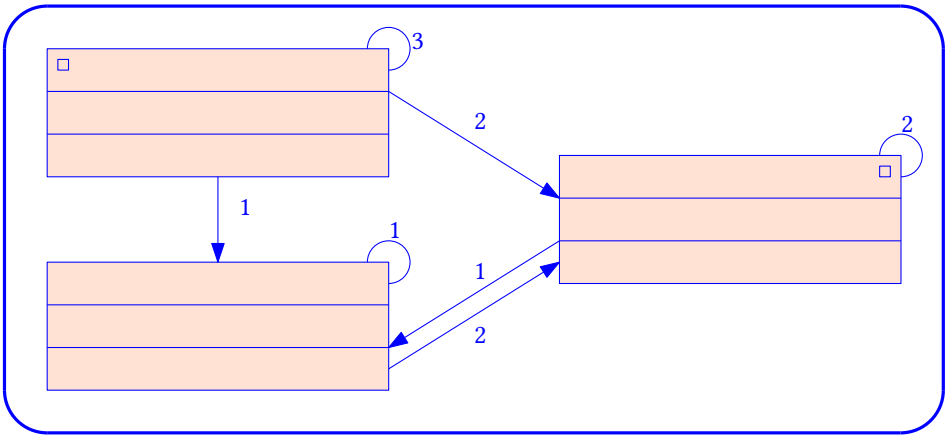

Figure 2.8: Situation graph containing three situation schemes, prediction edges between the situation schemes and prediction loops at each situation scheme.

the visual representation of a situation graph. We omit the textual containments of the situation schemes for better readability. The situation schemes represent the nodes in the situation graph. These nodes are connected to other nodes by directed edges called predication edges. Cycles are permitted in the situation graph as well as reflexive edges from a situation scheme to itself.

The procedure of walking through a situation graph is as follows: it begins with the instantiation of a start situation – the top left situation in our example, as that one is marked with the start situation square in the top left corner. If there are multiple start situations, they are instantiated simultaneously in case of the preconditions are

fulfilled (Münch *et al.*, 2011b, 2012c). The situation then is specified temporally by the instantiation of subsequent situations according to its prediction edges. A situation is detected successfully if the procedure ends up in an ending situation. An ending situation is marked by the end situation flag in the upper right corner of the corresponding situation scheme. In the example, the right situation scheme is an ending situation.

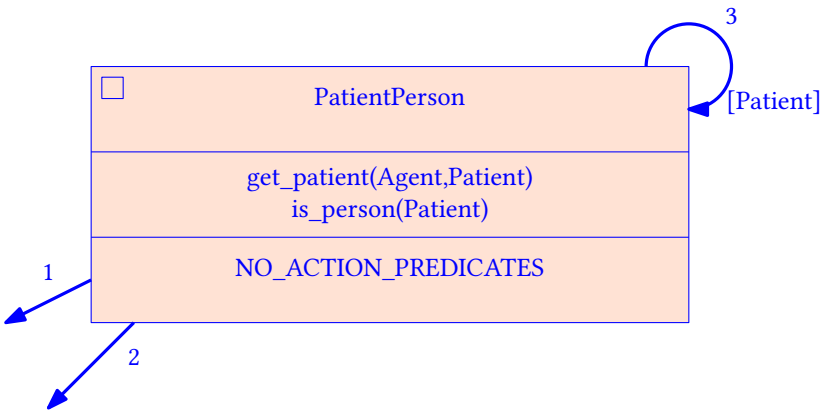

Figure 2.9: A single situation scheme embedded in a situation graph. The situation scheme has two prediction edges (1,2) and a prediction loop (3) with a binding that releases the variable Patient.

An edge can also be labelled with bindings. A binding specifies the variables you want to re-assign or allocate newly. Figure 2.9 demonstrates an example for its usage: the prediction edge marked with 3 has a binding of the variable "Patient". At the instantiation of this situation, the situation scheme chooses a value for Patient which has to be a Person according to requirements denoted in the state scheme. While the temporal edge 1 and 2 do not have bindings, the edge 3 releases the variable Patient so it is able to be allocated newly. If this binding would be omitted, always the same Patient would be chosen. Please note, that the label in Figure 2.9 is not visualized in the SGT-Editor and can be only accessed by a extra properties window.

2.4.3 Building Situation Graph Trees from Situation Graphs

As we have described the basic ingredients in the previous sections, we conclude the situation graph tree by merging situation graphs and specialization edges. Within the graph representation, a specialization edge is directed, ordered, must start in a situation scheme, and has to end in a situation graph.

The semantics of such a specialization edge forms a tree of situation graphs: a specialization edge either specializes a situation scheme in a temporal or in a conceptual manner. To give an example of the conceptual specialization, you can imagine two agents walking together. Then, "walking fast together" or "walking slowly together" is a conceptual specialization of "walking together". The temporal specialization can be seen as a temporal *is-part-of* relation. A situation, in which an agent walks through a door, can be split into several time steps "walks at the door", "opens the door", "goes to the other side of the door", optionally "closes the door", and "walks away from the door". At each step of a specialization, the general situation remains valid and must be identified uniquely. Therefore, a situation graph cannot specialize more than one situation scheme. Further, specialization edges from a specialized situation to a more general one are not permitted. The combination of these two restrictions place the situation graphs in a tree, the situation graph tree. The situation graph in this tree, that does not specify any other situation, is called root graph.

Figure 2.10 gives an example of the visual representation of an arbitrary situation graph tree. At the top of the figure is the root graph. The root graph has two conceptual specializations, the left one contains a situation scheme, that is specialized conceptually in two different ways.

2.5 Fuzzy Metric Temporal Horn Logic

Having situation graph trees as a description of the expected behaviour of agents is quite well, but it actually does not detect a situation in a video stream. This task is managed by the inference engine F-Limette[4]. The situation graph tree is translated by the SGT-Editor to fuzzy metric-temporal Horn-logic predicates, which F-Limette can handle. Fmthl allows you to phrase logic formulas, where the truth values may be in the interval $[0, 1]$ and a predicate class with a temporal context. In this section, we want to give a short introduction to Fmthl.

The first-order logic (Fol) was extended incrementally by (Schäfer, 1996). Beginning with adding support for a temporal structure $(\mathcal{T}, t_0, \prec)$ and temporal operators $(\circ, \bullet, \Box, \Diamond, \mathcal{S}_S, \mathcal{U}_S)$, the metric-temporal logic (Mtl) was created. $\mathcal{T}$ is a set of points in time, not necessarily an continuous interval, t_0 a reference point in time and $\prec$ an ordering relation in $\mathcal{T}$.
The operators $\circ$ and $\bullet$ represent the next point in time and the previous point in time, respectively. The operator $\Box$ states a formula being valid always, $\Box_S$ means "always" within the set $S \subseteq \mathbb{Z}$ of points in time. The following operator $\Diamond$ restricts the validity to "sometimes", analogously, the extension $\Diamond_S$ can be translated to sometimes within the

[4] http://cogvisys.iaks.uni-karlsruhe.de/Vid-Text/f_limette/index.html

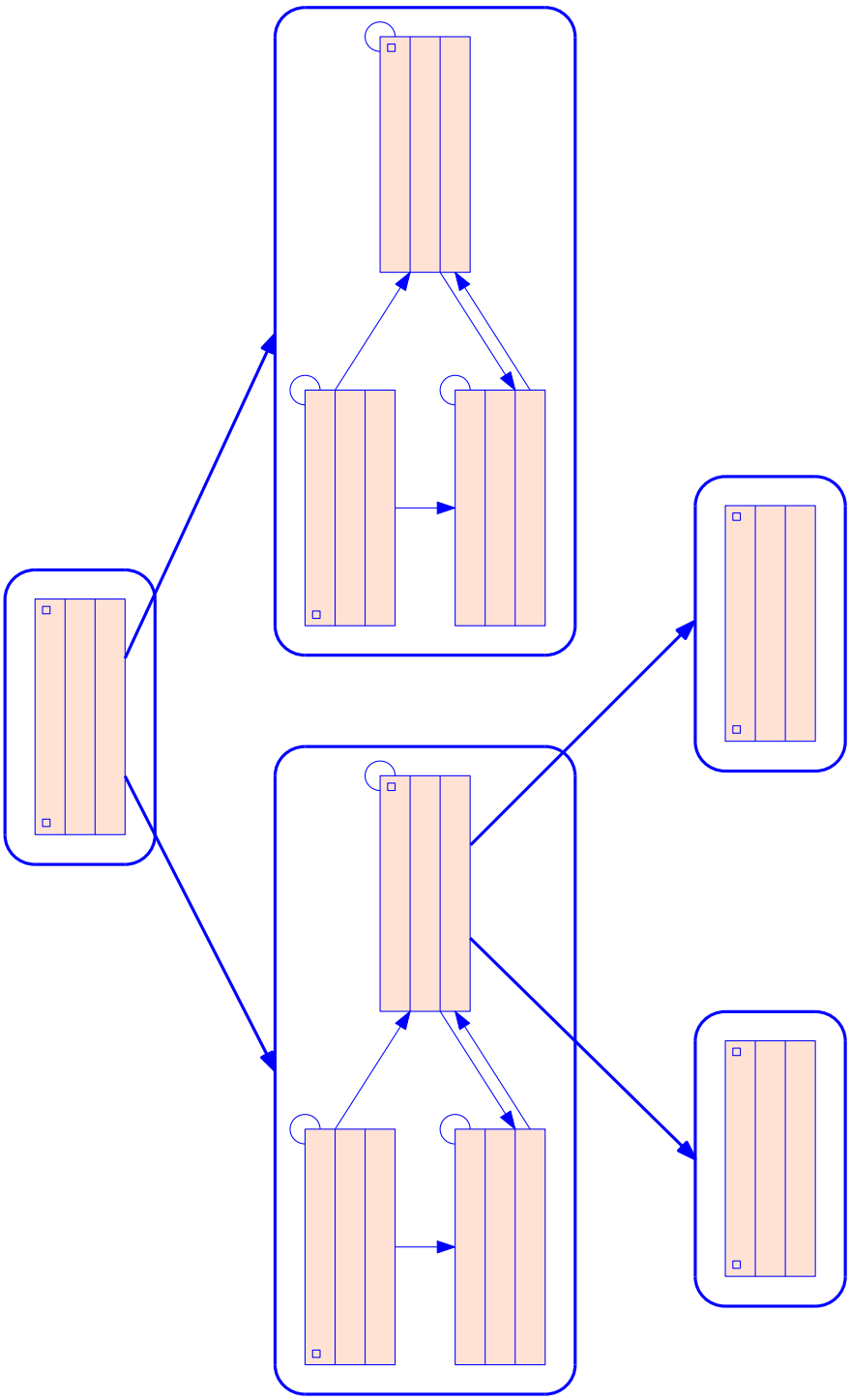

Figure 2.10: Situation graph tree with temporal and conceptual specialization edges.

set $S \subseteq \mathbb{Z}$ of points in time. The operators $\mathcal{S}$ and $\mathcal{U}$ are taken for the concept "since" and "until". The expression $\mathcal{F}_1 \mathcal{S}_S \mathcal{F}_2$ means in natural language "$\mathcal{F}_1$ always since $\mathcal{F}_2$ within S, the procedure with $\mathcal{U}$ is analogously.

In parallel, Fl1 extends the Fol by truth-values in the interval $[0, 1]$ instead of the values in $\{0, 1\}$. In Fl1, there are the operator $\downarrow_\kappa$, called fuzzy weakening and defined as the truth value of $\mathcal{F} \geq \kappa \Rightarrow \downarrow_\kappa$ is absolutely true, as well as the operator $\uparrow_\lambda$, called fuzzy strengthening and defined as $\mathcal{F}$ absolutely true $\Rightarrow \uparrow_\lambda \mathcal{F}$ has the truth value λ. Furthermore, extensions of the logical operators $\wedge$, $\vee$, $\neg$ and $\leftarrow$ are introduced as you can see in Table 2.3.

Table 2.3: Semantics of fuzzy operators for conjunction ($\wedge_w, \wedge_m, \wedge_s$), disjunction ($\vee_w, \vee_m, \vee_s$), negation ($\neg_w, \neg_m, \neg_s$) and subjunction ($\leftarrow_w, \leftarrow_m, \leftarrow_s$).

$v \in$	$\{w, m, s\}$	weak	medium	strong
Konjunktion	$x \wedge_v y$	$\min\{x, y\}$	$x * y$	$\max\{0, x + y - 1\}$
Disjunktion	$x \vee_v y$	$\min\{1, x + y\}$	$x + y - x * y$	$\max\{x, y\}$
Negation	$\neg_v(x)$	$1 - x^2$	$1 - x$	$1 - \sqrt{x}$
Subjunktion	$y \leftarrow_v x$	$\neg_m x \vee_w y$	$\neg_m x \vee_m y$	$\neg_m x \vee_s y$

Finally, the Mtl and Fl1 including the restriction on the Horn-Fragment form the Fmthl. Horn-formulas are conjunction of clauses, which contain a single positive literal at the maximum.

Figure 2.11 shows an example situation graph tree. There are two situation graphs, the upper one contains the root situation scheme, specialized by the bottom situation graph containing the following situation schemes: the starting schemes *SituationApproach* and *InAgentArea* as well as the ending scheme *SituationSplit*. The Fmthl representation of this situation graph tree is shown in Appendix B. An example for the usage of SGTs and Fmth in surveillance and smart environments provides (Münch *et al.*, 2011a), (IJsselmuiden *et al.*, 2012) depicts a different scenario.

2.6 Cognitive Vision System

In the past decades, several different cognitive architectures have been proposed, for example the Cognitive Vision System (CVS) firstly proposed by (Nagel, 2000) and slightly extended in (Nagel, 2004). The first application of CVS was the semantic understanding of vision-based environments, e.g. traffic control. To rely on a cognitive architecture is shaping up well towards an Artificial General Intelligence.

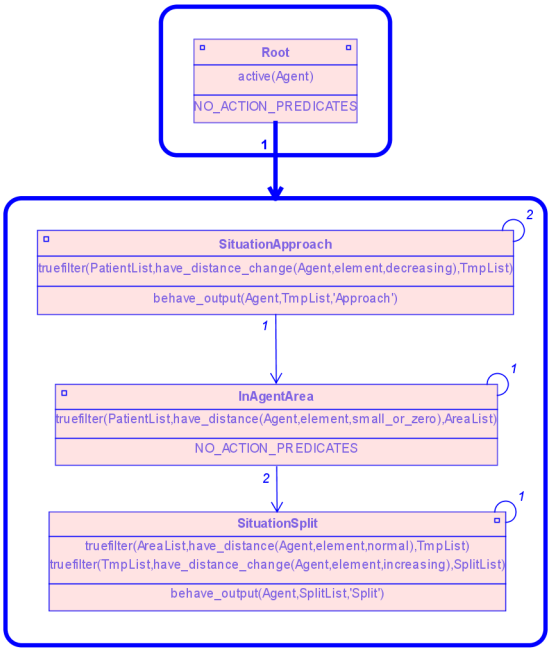

Figure 2.11: Example of a situation graph tree. The SGT consists of two situation graphs, the root graph at the top and the graph at the bottom which specializes the situation scheme in the root graph.

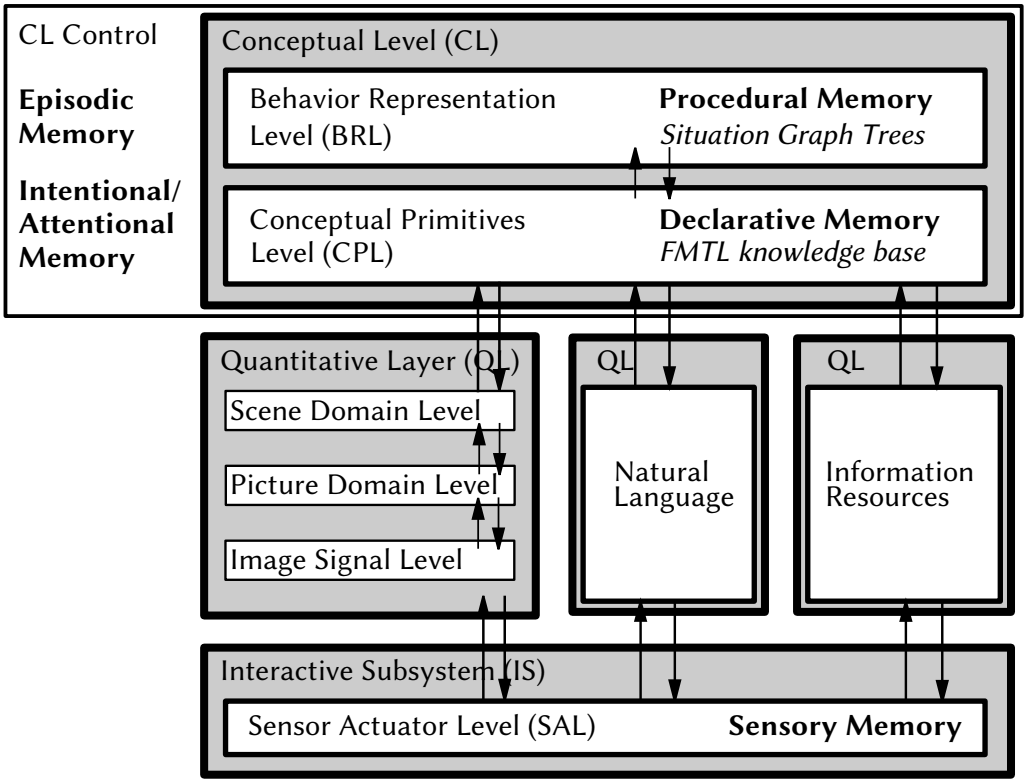

Figure 2.12: Comprehensive overview on the Cognitive Vision System. The architecture can be seen as three-layered: the Conceptual Layer at the top, the Quantitative Layer in the middle, and the Interactive Subsystem at the bottom. Arrows depict the flow of information, boxes with white background represent information.

The development on CVS includes effort in universal knowledge representations as well as the extension of the environment with understanding natural language capabilities. Multiple hypothesis inference and consistent truthvalue propagation throughout the whole inference process has been developed which was limited to Boolean single hypothesis results previously. Multiple hypothesis inference and reliable truthvalues allow the CVS to master more sophisticated domains of the real world. Theoretical limitations of the inference process have been overcome in an engineering perspective with sophisticated parallelization and knowledge sharing leading to ensured real-time properties of the whole system. The issue of combining noisy input data with uncertainty and fuzziness is addressed and a proposition of the calculation of truthvalues is raised. There are requirements to be met while working towards a general intelligent system. Comprehensive work about concrete requirements needed in a cognitive architecture for general intelligence is presented in (Laird and Wray III, 2010).

Figure 2.12 displays a comprehensive overview of the CVS. In the coarse seen, three-layered architecture the bottom is the *Interactive Subsystem* (IS). All information that is gathered by sensors and actuators is located there. Informations can also be sent to the IS e.g. steering informations. In the IS, the memory is the **Sensory Memory**. Several task-specific modules can be plugged in the *Quantitative Layer* (QL) on the middle and out of it. For example a vision-based module which gathers person tracks from image data, or a language-based module which transforms audio signal into text, or an information-based module which collects knowledge from public databases on the web.

The semantic gap is bridged at the *Conceptual Layer* (CL). The CL is located in the top layer. Quantitative informations from QL modules are turned into semantically meaningful concepts. They are first passed to the *Conceptual Primitives Level* (CPL). The CPL consists of basic knowledge expressed in Fuzzy Metric Temporal Logic (FMTL). Thus, quantitative numbers including their uncertainty can be mapped to concepts with a consistent truthvalue expressed in fuzzy terms. The memory in the CPL is the **Procedural Memory**.

Chapter 3

Related Work

By comparing situation graph trees with ontologies, there is no such work in the past. Of course, other work about situation recognition or situation awareness exists, respectively, modelling situations with ontologies, and the translation from an ontology into other forms of knowledge representation languages.

This Chapter starts with a paper about situation recognition in Section 3.1, which also describes the utilization of the inference engine F-Limette, we are using, too. Section 3.2.1 introduces a way to combine situation theory with ontologies. Section 3.2 concerns related work about ontologies in service of situation recognition, which itself is divided in four parts. We describe an approach to exclusively use ontologies and ontology reasoners in the domain of situation awareness in the Sections 3.2.2 and 3.2.3. The Section 3.2.4 summarizes a way to translate an ontology into Jess-rules, another inference engine. This Chapter closes by a summary about the related work in Section ??.

3.1 Situation Recognition

A way to track persons in real-time shows (Bellotto *et al.*, 2012). The area of surveillance, the floor of an atrium in an office building, is observed by a static wide-angle camera from the top. On each side of the floor is a Pan-Tilt-Zoom (PTZ) camera installed, also called tracker active camera (TAC). The theme of the paper is to present the system they use to steer the PTZ camera according to the movements of persons who move across the floor.

Figure 3.1 gives an overview about their system architecture. On the left side, the Visual Level represents the cameras in use, the grey box in the right side of the visual level signifies the supervisor tracker (SVT). The supervisor tracker is composed of submodules according to the figure, and responsible for the data fusion, reasoning and camera control

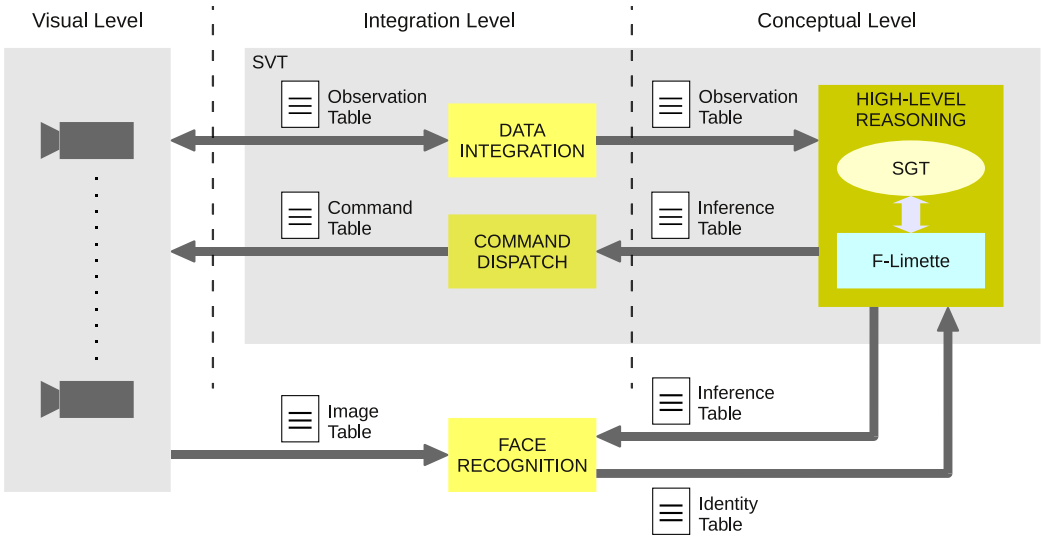

Figure 3.1: Overview of the system architecture used by (Bellotto *et al.*, 2012). Figure from: (Bellotto *et al.*, 2012).

strategy. A central SQL-Server implements the communication between the visual level, the supervisor tracker, and its submodules asynchrony.

To actually track persons, the program starts by processing the images from the static camera. Background removal with the Lehigh Omnidirectional Tracking System (LOTS) algorithm (Boult *et al.*, 2001; Hall *et al.*, 2005) serves to detect potential human targets in the surveillance area. The data integration module estimates the 3D-position and velocity of the targets' heads with the help of Kalman filters with a constant-velocity model and nearest-neighbour data association (Bellotto and Hu, 2010). The *a-priori* knowledge about the architectural environment including the embedding of the cameras is encoded in an Situation Graph Tree (see Chapter 2.4). The inference engine F-Limette for FMTHL (see Chapter 2.5) processes that data and computes the best tracker active camera to follow the target, depending on the targets moving direction and the area it is heading. Hereby, F-Limette produces instructions, that the command dispatch module sorts and delivers to the destination cameras like "track target" or "acquire face image".

To track a person with an active camera, they first steer and zoom the view of the camera on its estimated position. If there is a face detected in that area, the person gets an ID and the active camera then is instructed to track the person's head according to its estimated position and velocity until another target has to be tracked. The face recognition module has to deal with images, that are not optimal for the identification (blur as a consequence of person moving or camera altering; no frontal view). So, an image filter is used to select only the best images for the further face recognition algorithm by (Apostoloff and

Zisserman, 2007; Everingham *et al.*, 2006).

Besides problem, they have to face, such as the behaviour in a multiple target situation or stopping the tracking of an active camera before the target leaves the view of the cameras, there is a major lack in re-usability and therefore, in interchangeability. As the architectural setting is hard-coded into the situation graph tree, that situation graph tree is highly particular and has to be rewritten on an alternate surveillance area. Although it seems to be elegant coding camera instructions into the situation graph tree, so the inference engine initiates the active camera moving, it has its downsides from the knowledge representation side of view. There is no compelling reason to code environmental-specific software events into the knowledge representation.

3.2 Situation Recognition with Ontologies

In this section, we present related work that uses Ontologies as the knowledge representation resource for situations.

3.2.1 Ontology-based Situation Awareness

Another approach to model an ontology for situation awareness gives (Kokar *et al.*, 2007). Their main idea is to formalize the situation theory by (Barwise and Perry, 1984), which later was extended by (Devlin, 1991) and later summarized in (Devlin, 2006), using a language that is both processable by computer and commonly supported. As their ontology directly is built upon a formalization of Barwise's situation semantics, they call their resulting ontology Situation Theory Ontology (STO). In opposite to the other related work, their situation awareness ontology currently does not have an implementation in a software, but aims to be the base of future ontologies in the domain of situation awareness.

The design of the ontologys model is similar to the model of human situation awareness by (Endsley, 2000). Figure 3.2 shows the main classes and properties of the STO, in which classes are rectangles, blue arrows represent object properties, and the black arrow a subclass axiom. The object properties marked by an asterisk (*), must be set at least once for each corresponding object. The relation between *infons* and situations is called *support*, whereby an infon in situation theory consists of a n-place *relation* R, $a_1, \ldots, a_n$ *objects* appropriate for the R and a *polarity* $p \in \{0, 1\}$. If $p = 1$ states, the objects stand in the relation R, otherwise they does not. Infons may be recursively combined by using conjunction, disjunction and situation-bounded quantification. An infon that was not combined is an *elementary* infon. The class Individual represents entities perceived by an agent, Attribute signifies locations and time instants. The classes Rule, Value, and Dimensionality are claimed to be self-describing. Further details of the STO are omitted.

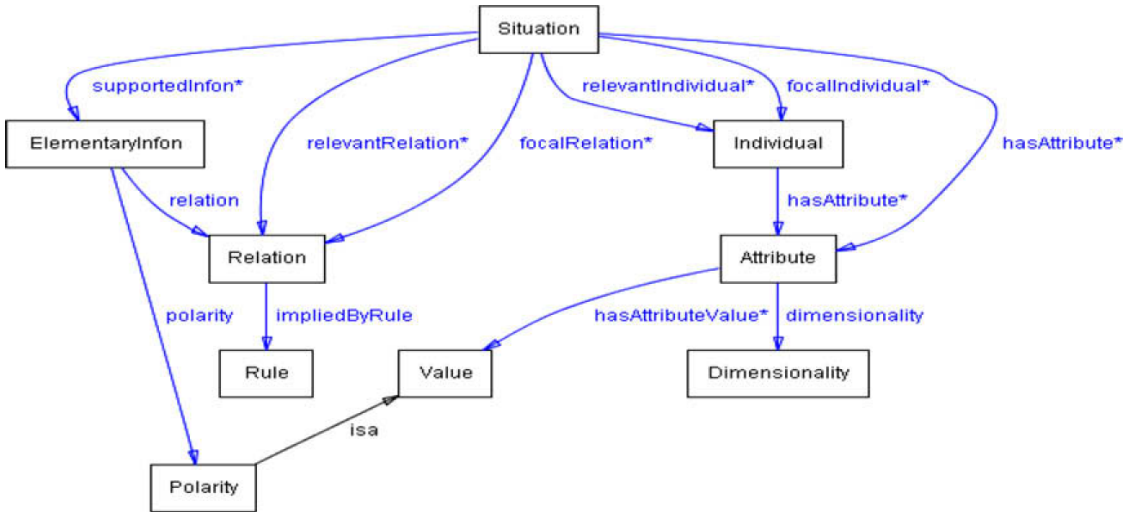

Figure 3.2: Main classes and properties of STO, from: (Kokar *et al.*, 2007).

As this ontology is designed primarily to build the base of other ontologies, it has been evaluated to realize a situation graph tree in this format. Apart from the easier comprehension of the structure of an SGT, at least in the opinion of this thesis' author, the ontology has a mayor disadvantage. As individuals are classes as instances, it is necessary to use the highest OWL level, OWL Full, which has maximum expressiveness, but is not decidable. Further, their suggested rule language BaseVISor is no official extension to OWL, so the interchangeability suffers.

3.2.2 A Software Architecture for Ontology-Driven Situation Awareness

The paper "A software architecture for ontology-driven situation awareness" is proposed by (Baumgartner *et al.*, 2008). The design of the ontologies used for representation of the knowleadge rely on the distinction of Terminological-Boxes (TBox) and Assertion-Boxes (ABox), as used in Description Logics (Baader *et al.*, 2007). The discourse of this work is about traffic control. The situation recognition works in a hierarchical approach, where the recognition of a complex situation is built upon the results of components, that recognises lesser complex situations.

Two problems are claimed, they have to deal with when using Ontologies: by design, an ontology crosses various levels – from persistence via business logic through to presentation layer – of multi-tier software architecture. As an ontology is geared tight to the application, the re-usability could suffer. Furthermore, an ontology is designed to mostly

be queried but to pipe through data. So, scalability problems could arise with a huge amount of data that situation awareness application produce. To contrast difficulties further, they propose two approaches for a software architecture. The first one uses a globally shared ABox, where you have to invest an huge effort to synchronise the components to avoid inconsistencies or to build unwanted dependencies between components by adding control informations to the ABox. The other approach is about layering the components in which each component operate with the results of the previous one. But this approach has a problem with scalability, as the amount of individuals grow with each layer.

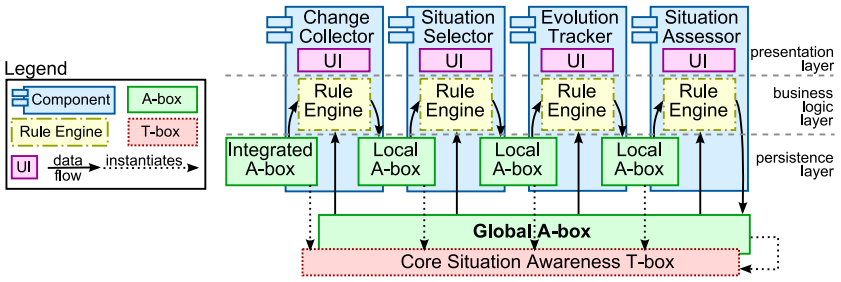

Figure 3.3: An architecture for ontology-driven situation awareness. Figure from: (Baumgartner *et al.*, 2008).

So they present an alternative solution using the pattern Pipes-and-Filters (Buschmann *et al.*, 1996) which is illustrated in Figure 3.3. They decided to share a global ABox and TBox, which each component accesses. A component presents its results to the user as well as writes them to a local ABox. The local ABox then is used to pipe the results to the next component. To implement the developed architecture, they used the Jena Semantic Web framework. They announce to reuse this architecture in the application BeAware! described in the next section.

This software architecture shows a way to use ontologies in the context of situation awareness. From an external side of view, their approach seems to accomplish its primal goal to be reusable and not limited to traffic control. So, it has to be discussed whether to use the design of this ontology for the transformation from an SGT into an ontology. In contrast to our SGT environment, this software architecture uses OWL-reasoners for the inference of situations. While an OWL-reasoner is well integrated, this architecture is bound to the reasoner's limits. Although they are able to perform some kind of temporal inferences, it is not possible by design to handle some kind of uncertainty and vagueness with OWL-reasoners.

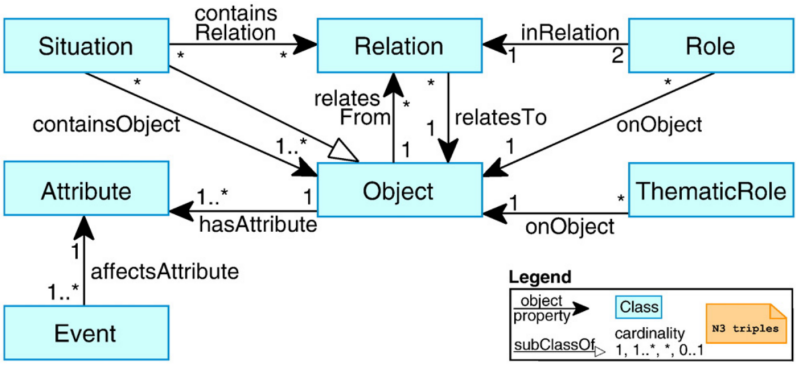

Figure 3.4: Schematic description of the SAW core ontology. Figure from: (Baumgartner *et al.*, 2010). The figure shows the basic entities in the SAW core ontology and its relations.

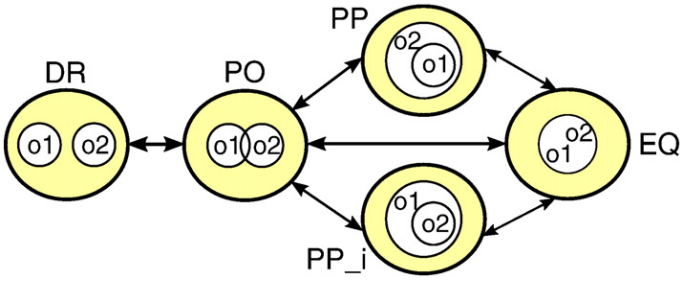

Figure 3.5: A conceptual neighbourhood graph for the region connection calculus, here the RCC-5. An explanation can be found in the text. Figure from: (Baumgartner *et al.*, 2010).

3.2.3 BeAware! – Situation Awareness, the Ontology-driven Way

The paper "BeAware! – situation awareness, the ontology-driven way" by (Baumgartner *et al.*, 2010) describes the development of a framework for ontology-driven information systems aiming at increasing the situation awareness of an operator, called BeAware!. The framework primarily bases on the work of (Baumgartner *et al.*, 2008) from the previous section. We introduce this paper, as it gives further insight into the design of their ontology (see Figure 3.4) and how they handle fuzziness in the context of situation recognition.

In comparison to their previous work, they implemented granular relations in time and space, according to the definition of situation awareness by (Endsley, 2000). To pre-

dict particular situations in advance of their actual occurrence, they use a Conceptual Neighborhood Graph (CGN) for each family of relations. Figure 3.5 shows an exemplary CGN for the RCC-5, the Region Connection Calculus (Randell *et al.*, 1992), relation family. From (Baumgartner *et al.*, 2010, p. 1118): "The CGN consists of five relations (exemplified with the objects o1 and o2 depicting the relation's meaning) and the possible evolutions in between. A relation between two objects evolves in the form of single-hop transitions with respect to the CNG of its corresponding family. For example, if the relation DR (discrete from) holds between o1 and o2, it can only evolve to EQ (equals) by traversing over PO (partly overlapping) [...] With this knowledge at hand, they are in a position to determine whether a situation may evolve into a critical one or just fuzzily matches a situation type definition."

The usage of CGNs in this work is the first approach in the category situation awareness with ontologies to handle situations that are not yet instantiated by a truth value of 1. Nevertheless, the expressiveness is very limited in comparison to what F-Limette is able to infer.

Further comparison against ontologies can be found in (Baumgartner and Retschitzegger, 2006).

3.2.4 Generation of Rules from Ontologies for High-Level Scene Interpretation

A generation of rules from ontologies for high level scene interpretation is introduced by (Bohlken and Neumann, 2009). As the approach in our work, they use OWL as storage format for the knowledge about situations, modelled by a domain expert, and use a transformation to another inference engine, as the reasoners for OWL does not suffice their needs. The inference engine of their choice is Jess. Their ontology mainly consists of four objects: conceptual objects like events and states, and physical objects like mobiles or zones. Every event, state and so on they want to detect is encoded as an subclass of these main objects. The spatio-temporal relations are coded into a bunch of high-level Swrl rules.

Figure 3.6 shows the architecture of their interpretation system: "In the initialisation (or offline) phase of the system, the concepts of the conceptual knowledge base are transformed to templates and aggregates are transformed to rules, all written to data files. An aggregate is formed by a concept and its parts together with the conceptual constraints. An initialisation file for the temporal constraint net is generated out of the SWRL rules. These files together form the Jess conceptual knowledge base. These data files are read in the working (or online) phase of the system, by the Java application with the embedded Jess engine. The templates and rules are added to the engine, a Temporal Constraint Net (TCN) is initialised and also added to the Jess engine as a shadow

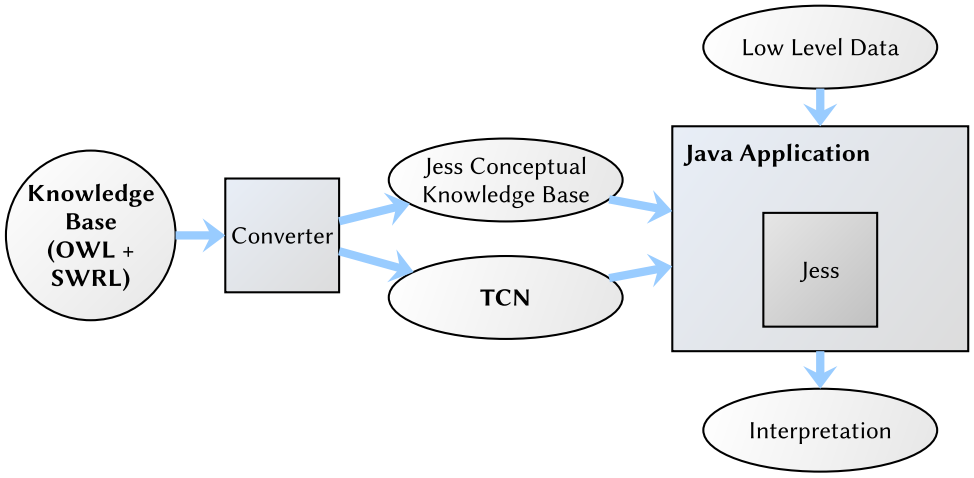

Figure 3.6: Overview of the architecture of the interpretation system. The figure shows the information flow through the components in their architecture. Informations are depicted as ovals, components as rectangles and the arrows represent the informations. Figure from: (Bohlken and Neumann, 2009).

fact. The temporal constraint net controls the activation of rules and stepwise aggregate instantiations, maintaining consistency of the temporal constraints" (Bohlken and Neumann, 2009).

Chapter 4

Ontology Design

The ontology to develop is subject to the two requirements: it has to contain definitions of situations and it must be possible that the ontology represent an SGT. This Chapter covers the design of this ontology. In the first Section 4.1, we show the approach to develop an ontology along the lines of the SGT structure. Step-by-step, it leads to the development of a more sophisticated ontology which is presented in Section 4.2 in detail. Section 4.3 pictures a way to apply additional semantics to an SGT by describing it in terms of simple graphs. This Chapter concludes in Section 4.4 with a discussion about our design in comparison to the ontologies proposed in Chapter 3.

4.1 Ontology Design: First Attempt

The first attempt reflects the initial contact with ontologies and its designing at all. The aim was to show the compatibility of situation graph trees and ontologies in general. The idea of this first attempt is about the representation of the tree-like SGT structure by the also tree-like Owl class hierarchy.

To translate the concept of the situation graph to an ontology's concept, we first reduce the situation graph to its role in the SGT's representation. For the moment, we hide the situation schemas and treat the situation graph as one single entity. Later, we break down the situation graph in its contents and discuss the SGT $\leftrightarrow$ ontology conversion in detail.

In the situation graph tree, a situation graph is a node. The parent of a situation graph is the situation, which the situation graph specializes. The children of a situation graph are situation schemes, that specifies the actual situation within the situation graph. This hierarchy resembles the ontology's class hierarchy much, but the latter is more general. In Owl, the class hierarchy additionally allows a class to have more than one parent.

In the final version of the ontology, situation graphs has to be restricted to only have one parent, as in SGT a situation graph may only be the specialization of exactly one situation at the maximum. In conclusion, we can map a situation graph to an Owl-class and use the built-in subclass axioms as connections to other objects.

The situation graph itself consists in detail of a cyclic, directed graph with situation schemas as nodes and ordered temporal edge. In comparison to specializations, the predication edges does not have a situation graph as end point but a situation schema within the same situation graph. This corresponds easily to the Class hierarchy of Ontologies. All situation schemas in one situation graphs are subclasses of their respective situation graph. To resolve the double occupancy of the ontology's "subclass"-relation — it is used for the hierarchy of situation graphs in the SGT as well as the "contains"-relation of situation schemas in a situation graph, hence we cannot determine the type of a subclass — we have multiple choices. The simplest way is to infer the type of the class by its depth in the hierarchy tree implicitly. An even number could represent a situation graph, an odd number a situation schema respectively. This variants drawbacks are obvious: a situation graph consisting of only one situation schema has to be modelled by two classes, the distinction, whether a class is a situation graph or a situation schema, is less intuitively, and you have to always satisfy the odd/even schema. A more intuitive way is to define the classes SituationGraph and SituationSchema and let the corresponding situation schema or Graph inherit from that classes. Thus leads to an initial meta-model for the definition of situations, which will be discussed in Section 4.2.

The edges in a situation graph has to be modelled in an ontology extra. The sibling-relation does not suffice the requirements of temporal edges, as on the one side, the sibling-relation is bidirectional, and on the other side, without further restrictions, it forms a clique with all of the other siblings. In the context of Owl, we consider siblings to be all subclasses of a given class. Although, you can identify a non existent edge by marking the classes disjoint, but the edges still are bidirectional. So, we make use of Object Properties, which, in short, have a name and put two Objects in relation to each other. Using a property called happensBefore, for example, that links two SituationSchemas is able to represent an SGT's temporal edge. But we still have to limit the range of that property to direct siblings, as in OWL each Class within the same property domain can be linked. While the temporal edges itself are easy to translate into an ontology, the ordering on the edges needs more effort as OWL only supports ternary relations (Object1 - ObjectProperty - Object2), but a quaternary Relation is needed (SituationSchema - orderIndex - happensBefore - SituationSchema). This purpose has been resolved by using the ordered list ontology (Abdallah, 2010). Individuals of SituationSchema with an extra property called hasIndexOrder.

That Individuals serves as proxies for the target situation schema. Additionally, these individuals can be used to model the bindings of a temporal edge by adding the Property hasBinding. To complete the requirements to model a situation graph as an ontology,

we have to mark situation schemas as starting or ending situations. Again, we add a property isStartSituation or isEndSituation.

We already touched the modelling of situation schemas, as far as it was necessary in the context of situation graphs. But apart of the bindings, markers for start and end situations, and the temporal edges, we have to model the state schema and the action schema. At the situation schema, the state schema and the action schema are lists of logical predicates, which can be represented in the ontology by an object property for each predicate. Such an object property can be named hasStatePredicate and hasActionPredicate, respectively.

The following section discusses the downsides of our first approach and introduces a better way to design an ontology that encodes a situation graph tree.

4.2 The SGT Ontology

The comparison of the concepts in Section 4.1 showed a straight way to represent an SGT in an ontology and showed the basic compatibility of these formats to represent definitions of complex situations. From the ontology's side of view, that translation is less-than-ideal. In this section, we discuss the disadvantages of the previous section and improve the ontology, which leads us to a well defined ontology for situation awareness.

Inheritance The representation of the SGT's tree-like data structure as a class hierarchy in the ontology is unfavourable. The main reason why that design is not favourable, is the diversity of situation graphs and situation schemas. If one declares a situation schema as the subclass of the corresponding situation graph, one declares a situation schema to *be* a situation graph, which in fact, is false. The idea of differentiating these concepts is a step in the right direction, but you still have to break that class hierarchy.

Re-usability The ontology we want to develop has to accomplish the requirement to be reusable. The development of a new situation graph tree from scratch is complicated and confusing compared to the first approach. As the current structure of such a situation graph tree relies on an existing one, it would be achieved by a program. Thus, a comfortable way to create an ontology would depend on a program. But depending on a special program that creates the ontology contradicts its idea.

Simplicity The further development of ontological situation definitions should be easy. The way, we developed the ontology in Section 4.1 requires expert knowledge both in the creation of a situation graph tree and in the development of the ontology presented

above. As we want to design the definition of a situation with an ontology, at least the former case should not be required. The long-term goal of this thesis is the possibility to define a situation in a few sentences of natural language. In this scenario, there is no space for SGT specific definitions, so a way to minimize that dependency should be provided.

4.2.1 Mapping of SGT Concepts to an Owl Ontology

As our first approach tried to map the hierarchy of an SGT onto the hierarchy of Owl subclass axioms, we now improve the ontology design by mapping every concepts of situation graph trees on the ontology. The idea emerges from the declaration of classes as situation graphs and situation schemes for distinguishing between them in the Owl class hierarchy. The following section explains how the features of situation graph trees are realized into the ontology.

As the following section deals with lots of concept names in the domain of situation graph trees as well as ontologies and we often switch between them, we introduce different semantic colour highlighting for each domain. Hereafter, concepts of the SGT domain are marked with a blue colour and the Owl counterparts in a purple colour.

4.2.1.1 Situation Graph Tree

The entry point of our ontology is the class SituationGraphTree, which is a subclass of our domain concept SGTElement. If nothing else is mentioned, all the classes introduced in the subsections of Section 4.2.1 are subclasses of SGTElement. The SGT has some attributes, that have to be carried to the ontology. The attributes come pairwise one of each attribute pair has to be set and are in terms: Depth or Breadth, Greedy or Nongreedy, Incremental or Nonincremental, Singular or Plural, and last Traversal or Occurrence. For each of the pairs exists an enumerated class with the respective attributes as members. The first attribute is chosen as the name of the class. To give an example: the Owl class for Depth is Depth, which itself is equivalent to an Attribute and one of {DEPTH, BREATH}.

The attributes were applied to SituationGraphTree by the functional object properties hasAttributeDepth, hasAttributeGreedy and so on. Each property is a sub-property of hasAttribute, so a query can list every attribute applied to the situation graph tree. Further, each of the attributes' domain is SituationGraphTree, its range the corresponding attribute.

Although it is called a tree, the SituationGraphTree does only have a connection to a single root situation graph. To bind the RootGraph, we use the functional object property hasRootGraph. The domain of the object property hasRootGraph is SituationGraphTree and its range is SituationGraph which is introduced in the following section.

4.2.1.2 Situation Graph

Following the naming scheme of the situation graph tree above, we name the class of an situation graph SituationGraph. As in the SGT a situation graph also can have the attribute Incremental, we extend the domain of Incremental by SituationGraph and add a NOTSET to its members. The introduction of the member follows the attribute inheritance behaviour of an SGT: if the attribute of Incremental is not set, the situation graph inherits its value from the corresponding situation graph tree.

The situation graph contains situation schemes, at least one. Nevertheless, situation schemes are bound to a situation graph by the object property hasSituationScheme with a value restriction of ≥ 0. The additional asserted object properties hasStartSituationScheme and hasEndSituationScheme allows us to use that value restriction, as these object properties have to be set at least once, so we implicitly have at least one situation scheme. Every object property has SituationGraph as domain and SituationScheme as range. Note that in the SGT model, the flags StartSituation and EndSituation refers to a situation scheme, but not a situation graph. We decided to deviate from the SGT model as the semantics of starting and ending situations fit semantically better to situation graphs. To be precisely, for the situation scheme itself it does not matter whether it has for example the starting flag. But as the traversal algorithm has to find the starting situations when instantiating a situation graph, you find them all directly.

4.2.1.3 Situation Schemes

Again, situation schemes are called SituationScheme in the Owl ontology. Like the previous introduced SGT concepts, this one has the attribute Incremental, too. This attribute is inherited from the situation graph, so the object property hasAttributeIncremental has to be filled with one of INCREMENTAL, NONINCREMENTAL or NOTSET.

A SituationScheme further must have exactly one StateScheme realized by the functional object property hasStateScheme. The classes on which this object property may be assigned to are restricted to SituationScheme objects and may be only filled with objects of the StateScheme class. The mapping of an ActionScheme works analogous with the StateScheme. Both of them are described in detail later as well as the other two classes linked to the situation scheme, ConceptualSpecialization and TemporalSpecialization.

Both, the ConceptualSpecialization and the TemporalSpecialization are fillers for the object properties hasTemporalSpecialization and hasConceptualSpecialization, respectively. The amount of assigned specializations is not restricted. The domain of that object properties is restricted to SituationScheme objects.

4.2.1.4 State and Action Schemes

As structure of the state and the action schemes are very similar, we introduce them simultaneously. The classes are named StateScheme and ActionScheme, their only asserted object property is hasPredicate. That object property's domain is StateScheme or ActionScheme, the range is restricted to Predicate objects. While the ActionScheme does not require a Predicate, the StateScheme requires at least one as we need a predicate to instantiate the situation but do not have to do nothing after the instantiation.

4.2.1.5 Conceptual Specializations

In words of graph theory, the conceptual specializations connect situation schemes to situation graphs. The Owl class name of conceptual specializations is ConceptualSpecialization. The obvious object properties are hasPreviousSituationScheme which has a domain of ConceptualSpecialization and a range of SituationScheme and hasSituationGraph with the same domain and range.

The lesser obvious object property is hasOrderedIndex. Its domain is an OrderedListItem and the range a nonNegativeInteger. This object property assigns an integer value to the specializations indicating the order of instantiating the adjacent SituationGraph. Although such an object property is not needed any more since (Münch *et al.*, 2011b). The reader may have mentioned that the domain of this object property does not match the class name of conceptual specializations nor has anything to do with a situation graph tree. But it does not undermine the consistency of our ontology as reasoners now infer, that the ConceptualSpecialization is an OrderedListItem. The object property itself as well as its associated classes are imported by a self-written list ontology.

4.2.1.6 Temporal Specializations

An SGTs Prediction is realized in the ontology by the class TemporalSpecialization. It has assigned the object property hasOrderedIndex for the same backward compatibility reason as written in Section 4.2.1.5. The situation schemes connected to the Prediction are specified by the object properties hasPreviousSituationScheme and hasNextSituationScheme. As the first one is also used by the class ConceptualSpecialization, we extend the domain by TemporalSpecialization. The domain of hasNextSituationScheme is TemporalSpecialization, the filler class is restricted to SituationScheme objects. Temporal specializations does also have bindings where variables may be released or reassigned by other variables. The object property hasBinding with a domain of TemporalSpecialization objects and a range of Binding objects realizes this feature. While the previous introduced properties hasPreviousSituationScheme and hasNextSituationScheme has to be specified, hasBinding is optional and therefore has a value restriction of at least 0.

4.2.1.7 Predicates

Unlike the previous introduced mappings, Predicates should not be mapped in a straight forward way. In such a way, the predicate may be declared as a individual and gets a the content of the Predicates as string. By mapping a Predicate onto a string, you loose any semantic information such as the name of Variables and the possible connection to Bindings.

Before we propose our realization, let us discuss about the difficulties of predicates. A regular predicate looks as follows for example:

isPredicate(Argument1, Argument2, nestedPredicate(constant_argument))

The predicates name is isPredicate and has three arguments, the variables Argument1 and Argument2 and the predicate nestedPredicate(constant_argument). As Owl does not grant that declarations are in the same order as they are specified while reading, a mechanism for preserving the order of arguments has to be implemented. The first approach introduced has an argument list. But Owl does not have built-in support for lists, so the list ontology by (Drummond *et al.*, 2006) was used. One disadvantage of that ontology is an huge effort in reasoning – even with only a single list with three elements specified, the reasoner HermiT needed around 3 minutes to process the ontology. By writing a simpler list ontology or by using the ordered list ontology by (Abdallah, 2010) solved the reasoning problem but another disadvantage remained: querying the arguments of a predicate became unnecessary complex. First, the list of arguments has to be queried. Then, it has to be iterated over the elements of that list and for each element, its content has to be retrieved.

So a simpler, but not that flexible solution was implemented. Arguments are bound to a predicate by the object property hasArgument. To grant the order of arguments, sub properties hasArgumentX of hasArgument were introduced, where X is a non negative integer value representing the X-th argument of the predicate. An Argument is the subclass of Predicate or Variable, so nested predicates are realizable. While this implementation allows to retrieve the arguments of a predicate simply by querying the object property hasArgument, the generality of the object properties is lost, as for example there is no hasArgument4 of a 3-argument-predicate.

Needless to say, that the Predicate also has the object property hasAttributeIncremental which domain is extended again by the value Predicate.

4.2.1.8 Bindings

Bindings come in two variants: BindingRelease and BindingAssignment. For both, a class Binding was created with the two subclasses Release and Assignment. The functional

object property hasVariable has a domain of Release, a range of Variable objects and specifies the variable that has to be released in a BindingRelease.

The object properties specifying the variable to be assigned and the variable to be set in a Assignment are called hasVariableToBeAssigned and hasVariableToBeSet. Both are functional, the domain restricts to Assignment objects, the filler objects must be of the type Variable.

4.2.2 Expressiveness of the SGT ontology

The expressiveness of the ontology proposed above is $\mathcal{SHOIN}$. We use number restrictions ($\mathcal{N}$) to state that a situation scheme contains exactly one single state scheme, for instance. Inverse properties ($\mathcal{I}$) are declared for the purpose querying for object properties. The restriction of Attributes to be one of INCREMENTAL, NONINCREMENTAL or NOTSET for example leads to the utilization of nominals ($\mathcal{O}$). The declaration of the sub properties hasArgumentX of hasArgument leads to role hierarchies ($\mathcal{H}$). AL is the least requirement to a description logic, $\mathcal{C}$ results from using complex concept negation and together with the introduction of a transitive object property hasSGTProperty, that is the super object property of every declared property we get extension $\mathcal{S}$.

4.3 Embedding the SGT into a Graph Ontology

The semantics of a situation graph tree being a graph is lost in the proposed ontology so far. Informations like *"a situation scheme is a node"* or *"a prediction is a directed edge between situation schemes"* are of interest for example in graph drawing. This section presents the procedure of obtaining the graph informations of an situation graph tree by extending the asserted object properties. It reveals the higher flexibility of ontologies in comparison to SIT++.

4.3.1 The Graph Ontology

Before introducing the graph embedding mechanism of our ontology, we present the Graph-ontology itself. The ontology is a minimal example for representing graphs that only fits our needs. In graph theory, the definition of a graph G is: $G = (V, E)$, where V is a set of vertices and $E \subseteq (V, V)$ is a set of edges. In a directed graph, $E \subseteq V \times V$ is essential. The ontology contains four classes: Graph, Node, Edge and DirectedEdge. In our ontology, we define a Graph to be equivalent to everything that has at least one node and at least zero edges. To realize this definitions, we introduce the object properties hasEdge nad hasNode. They are sub-properties of hasGraphProperty that only isolates

properties of this domain. The property hasEdge may be assigned to Node objects and must be filled with objects of the type Edge. The other object property assigned to a Graph – hasNode – my be assigned to Graph objects and has to be filled with Node objects, respectively. The object property hasNode is declared to be inverse functional, meaning that if two graphs g1 and g2 has a connection to the same node via this property, g1 = g2 follows.

A Node is subset of things that has some edges. In the ontology, the object property hasEdge with the value restriction of at least zero edges realizes that expression. An Edge is equivalent to everything that is connected to at least one node. The object property isConnectedTo is introduced to express that statement. It has Edge objects as domain and Node objects as fillers. Note, that we allow edges to have only one adjacent Node. That is to shorten the definition of a loop from a node to itself.

As the situation graph tree mostly contains directed edges, we also introduce the class DirectedEdge. It is declared a subclass of Edge and equivalent to everything that has exactly one start and one end node. The object properties hasStartNode and hasEndNode, each a sub-property of isConnectedTo, a domain of DirectedEdge and a range of Node objects implement the equivalent-statement. We restrict a directed edge to only have one start and one end node and treat a situation graph as a node. In its current state, the graph ontology only supports simple graphs. Nonetheless, by treating situation graphs as node and a graph at the same time, we ignore the fact of an SGT being an hypergraph. The development and embedding of the SGT ontology into a hypergraph ontology is dedicated to the gentle reader.

4.3.2 Object Property Mapping

We presented the situation graph ontology as well as the graph ontology in the previous sections. How to bring them together is shown in the following paragraph. The process of bringing the ontologies together does not have to touch the definitions of classes, as they are well defined by object properties. We alter the object properties in the situation graph tree ontology so they match with object properties from the graph ontology and the classes of the situation graph tree ontology automatically inherits from classes of the graph ontology.

The situation graph tree has a root situation graph, symbolized by the object property hasRootGraph. As we treat situation graphs as nodes, we state this object property to be a sub-property of the graph ontology's object property hasNode. This sub-property axiom in combination with the declaration of a graph being equivalent to everything that has at least one node and at least zero edges leads the SituationGraphTree to be a Graph.

A situation graph contains situation schemes. So, we state the relation hasSituationScheme also to be a sub-property of the graph ontology's object property hasNode. Now, a situation graph is a Graph, too. Even if situation schemes are connected to the situation graph by only object properties of hasStartSituationScheme and hasEndSituationScheme, it is inferred the situation graph to be a graph, as these object properties are sub-properties of hasSituationScheme.

The predictions serves as a connection between situation schemes, thus, it obviously is an edge. As the start node of a prediction is asserted by the object property hasPreviousSituationScheme, we declare it to be a sub-property of hasStartNode. Analogously, we declare the object property hasNextSituationScheme to be a sub-property of hasEndNode. Now, a TemporalSpecialization is an edge and more particular, a DirectedEdge as we defined a start and an end node.

But the nodes, a prediction is adjacent to, are not already defined. The situation scheme has an object property hasTemporalSpecialization which establishes the connection to a directed edge of type TemporalSpecialization. The declaration of hasTemporalSpecialization to be a sub-property of hasEdge leads a situation scheme to b a Node.

Similar to the inference of a TemporalSpecialization to be a directed edge, we assert the object property hasConceptualSpecialization a sub-property of hasEdge connecting a SituationSchema to a SituationGraph. Unfortunately, the range of a object property does not affect the inference of its filler to be of the corresponding type. So currently, a SituationGraph only is inferred to be a graph but not a node as there is no hasEdge object property assigned to it. At this point, we extend the definition of a situation graph by adding the functional object property isConceptualSpecializedBy to its definition. This object property has a range of ConceptualSpecialization and a domain of SituationGraph, is an inverse property of hasSituationGraph, and is a sub-property of hasEdge.

4.4 Discussion

We proposed in Chapter 3.2.3 the ontology design of an other group and mentioned to use that eventually to encode situation graph trees in ontologies. Implementing a bidirectional transition from an situation graph tree to their ontology would allow us to directly use their definitions of situations and vice versa. Although, we decided against it for reasons described in the following section.

The ontology design in the current state allows a straight forward implementation of the transformation from an SGT into the ontology. The following Chapter 5 shows the implementation and substantiate this argument. The use of any another design than the current one not only leads to increased effort in the implementation but also puts semantics into the translation program. When operating with ontologies, hard-coding

semantics into a program is counterproductive. So, the realization of the transition from another description of situations takes place in the ontology. Section 4.3 shows the way, such transitions can be developed.

First, the SGT concepts have to be identified in the third-party ontology. So, by creating *is-a* relationships, the foreign concepts are mapped to the SGT counterparts. According to the procedure proposed above, the same must be done with object properties, too. But in contrast to the embedding of a graph, we assume that the development of such a mapping ontology can be very difficult if not impossible in some special cases. The ontology of (Bohlken and Neumann, 2009) for example describes agents and events very accurately, but a bunch of high-level Swrl-rules represent the relations between them which does not support an inheritance hierarchy such as classes or objects on the one side and we do not use them in our ontology on the other side. We assume that the situation awareness ontology (Baumgartner *et al.*, 2010) can be mapped to the situation graph ontology. But as this includes the comparison between different theories about the representation of situations – on the one hand see (Barwise and Perry, 1984) on the other hand (Schäfer, 1996) – it goes off this thesis' topic.

In this chapter, we proposed the way of designing the situation graph tree ontology in two ways. In Section 4.1 we firstly developed an ontology that resembles the tree-like structure of an SGT. But as we had to determine that this ends in an impasse, the design has been improved in Section 4.2 by porting the SGT model to the ontology. The Section 4.3 shows an example for the powerfulness of ontologies by embedding the SGT into graph theory.

Chapter 5

Implementation

In this Chapter, we show implementation details of our new developed software SGTowl (speak: [ez:dʒi:taʊl]). Section 5.1 points out the current implementation of the software-related model of the SGT. It states the disadvantages that implementation has and why it has to be re-developed. Beginning with Section 5.2, we present the design of SGTowl. It is stated how things were implemented and the motivation is explained to exactly do so. We proof in Section 6 that our software SGTowl works correctly.

5.1 Current State of the SGT-Editor

The model of a situation graph tree currently is implemented twice in the environment of the SGT-Editor. The background has a historical cause. First, there was the parser for Sit++ formatted data. This parser transforms an existing description of a situation graph tree into the SGT-model. Further, you are able to traverse this data structure into a F-Limette file. Having such a file, the detection of situations in an annotated video stream is possible with the support of the F-Limette inference engine. In the meantime, (Münch *et al.*, 2012b) adds machine learning to the former strict logic-based situation detection, (Münch *et al.*, 2012a) introduces the handling of incomplete data caused by loss or overlapping. As the Fmthl file is a terminological box of rules, it is very hard to read and write for a non-expert. The structured essence of an situation graph tree made it easier to model the expected behaviour of agents in situations. Even people were able to do it without being an F-Limette-expert. Lets call this model, the Sit-model.

One still has to create and edit textual data in the syntax of SIT++. So, a graphical user interface was born, the SGT-Editor (Arens, 2004). This graphical user interface offers its user a comfortable way to define situation graph trees without knowing anything about the syntax of SIT++, you only have to know about the structure and behaviour of an situ-

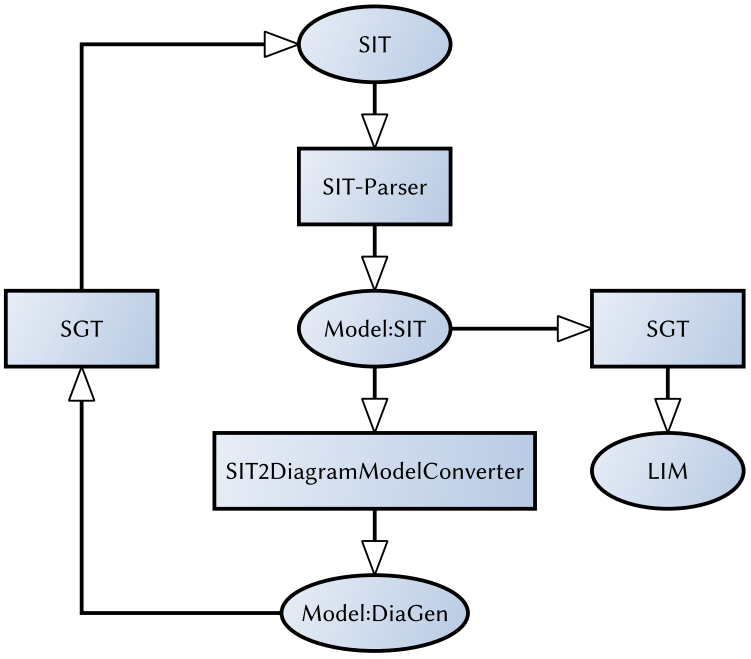

Figure 5.1: Data flow overview about the transitions of an SGT in the current SGT-Editor.

ation graph tree. The SGT-Editor extends DiaGen[1], an editor and a framework for graph editing applications, in the version 2.1. So a graphical representation was programmed, for every element of the SGT-Editor (see Section 2.4). Bound to the requirements of DiaGen and a graphical representation in general, a new model for situation graph trees was programmed. To distinguish it from the Sit-model, let us call it the Ed-model.

Unfortunately, the Ed-model is incompatible with the Sit-model. To give an example, we want to state the construction of visual elements. Some elements (a situation graph, a situation scheme, etc.) strongly requires an horizontal and a vertical coordinate in the visual embedding, but Sit++ does not support coordinates. Further, as you now have a graphical editor for a situation graph tree, you have to be able to write the in-memory data structure of an situation graph tree back to an Sit++ file – a functionality not needed in the past. As the Sit-model must not be broken, the development of the Ed-model was inevitable.

In this thesis, we also implement a program, that converts an SGT into an ontology and backwards. But the actual architecture of the SGT-Editor as described makes it very hard to extend it, as we want to meet the conditions of having well designed, maintainable

[1] http://www.unibw.de/inf2/DiaGen/

code, and being able to transform every existing model into an ontology and backwards. Figure 5.1 shows the current transformation of formats in the default procedure of the SGT-Editor: first, an Sit++ file is parsed by the SITParser, which emerges the Sit-model. Then, the Sit2DiagramModelConverter converts the Sit-model to the Ed-model. Now one is able to edit a situation graph tree with the graphical user interface. Before you get a F-Limette file, you first have to store the model in a Sit++ file, parse it again to get the Sit-model, and convert it to the resulting F-Limette file – there exists no other way to convert a Ed-model to a Sit-model or directly to a F-Limette file.

As you may have noticed, the current architecture of this part makes it hard to implement an additional conversion to an ontology in a maintainable and well designed way. Having in mind a future rework of the SGT-Editor, we wanted to build a common basis for both models. In this basis it should be possible to add additional functionality easily. The following Sections depicts the design of this basis, its implementation and the proof of equivalence.

5.2 Decisions about the Design in SGTowl

The application to implement – SGTowl – has to fulfil several requirements. This Section introduces the requirements and deals with its implementation. Before we will describe the translation between arbitrary models in arbitrary directions in Section 5.2.2, we show the software architecture and explain our design decisions in the Section 5.2.1.

5.2.1 Handling Multiple Representations

The main criteria is the support of multiple representations of a situation graph tree. As the definition and the behaviour of a situation graph tree is independent from an arbitrary model representation implementation, we want to encapsulate these features. The *Decorator* pattern (Gamma *et al.*, 1998) fulfils this and many other requirements stated later in this section. This pattern allows us to implement the basic functionality and decorates it with arbitrary features, i.e. positions for the visual representation. Figure 5.2.1 shows a UML-diagram of the decorator pattern. The class ConcreteComponent has a method called operation(), it is a realization of the interface Component and the class, that will be decorated by the abstract class Decorator. The Decorator class – a realization of the interface Component, too — holds a reference to the decorated Component, the ConcreteComponent in the simplest case. The exemplary method operation() in Decorator redirects method calls to the corresponding method of its referenced component. The ConcreteDecoratorA/B/... overwrites the method operation() and extend it with own functionality.

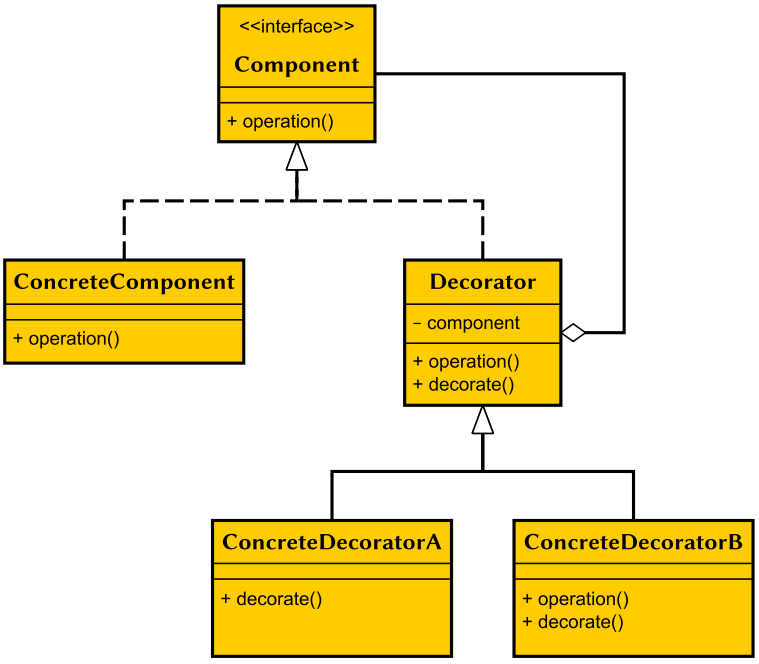

Figure 5.2: The Decorator Design Pattern in Uml. The Decorator as well as the ConcreteComponent implements the same interface Component. The function call operation() at the Decorator is redirected to the field component. In this case, a ConcreteDecorator that implements Decorator has to overwrite the method decorate().

The model of a situation graph tree does not only consists of a single class. Instead, each concept – as described in Section 2.4 – has its own class. Thus, the decorator pattern has to be applied to every concept. The highest abstraction layer, the interfaces of the concepts, counts eighteen classes. That makes 72 classes in total, when only one model representation is implemented. In comparison to the previous implementation with two separate model representations, there only were 45 classes. Although we now have more classes, we adhere to the decorator pattern, as the advantages prevail. Besides, we only need to maintain the behaviour and interaction of situation graph tree elements at *one* single model representation, we are in the beneficial situation to easily add new model representations with less lines of code.

5.2.2 Translation Between Different Representations

Using the decorator pattern allows the encapsulation of storage relevant code into several classes. This section describes the process of switching the representation, for example switching to the Sit++ representation.

As we intend to support a bunch of different representations of the SGT-model, we need an abstraction for object instantiation. The interface SGTModelFactory holds as a template to create new instances of a particular model representation. Implementations of that interface according to the *Factory* pattern (Gamma *et al.*, 1998) allows the instantiation of objects without knowing their types. Figure 5.2.2 shows a UML-diagram of the factory pattern. The interface Creator contains the method factoryMethod() for creating a Product. The implementation ConcreteCreator of the interface overwrites the factoryMethod() and returns a new instance of the implementation ConcreteProduct of the interface Product.

In SGTowl, we implemented for each model representation a factory of the type SGTModelFactory, that exclusively creates object of the corresponding model representations type. For every class of the model, there are two factory methods. One method for creating a whole new instance of an SGTElement and one method that creates a new decorated instance around an existing one. Such a method is required as we hide every constructor of a concrete SGTElement from outside the package. The intention of doing so is to force using the factory for the instantiation of classes. By this tactic, we promise ourself the prevention of a model mixture in implementations.

To transform a representation to another one, we introduced the utility class SGTModelConverter. This class has two functions, setModelFactory(SGTModelFactory) returning the SGTModelConverter itself and convert(SituationGraphTree) which returns a SituationGraphTree according to the chosen representation by calling the function createSituationGraphTree(SituationGraphTree) of the assigned SGTModelFactory. At the moment, only the object SituationGraphTree is converted to another class – SitSituationGraphTree in

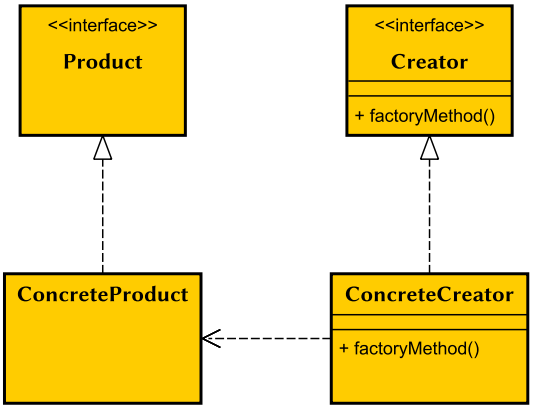

Figure 5.3: The Factory Design Pattern. The Creator is an interface for an ConcreteCreator. The method factoryMethod() creates a new instance of a ConcreteProduct which is an implementation of Product.

our example. The tail of the model remains in its original state. As the behaviour of the model is independent of the underlying serialization, it does not matter that only the SituationGraphTree has an altered model. So, the switch actually is made up of the instantiation of a new object and the return of the reference to that object which makes it really fast.

The actual effort relies in the serialization of the situation graph tree to a file. The different implementations of DecoratorSituationGraphTree have to overwrite the function writeTo(OutputStream) of the interface Writeable. So, every class that inherits SitSituationGraphTree and can be used to write the data structure of the situation graph tree to an output stream, for example to a FileOutputStream.

The serialization of a situation graph tree highly depends on the kind of representation, but the procedure is mostly similar. The call writeTo(outputStream) of the object SitSituationGraphTree writes a string representation of the current situation graph tree to the output stream and calls the function SitSituationGraph.writeTo(...). As after this second function call the situation graph tree still may write data to the output stream, the potential output result is a context free grammar. Note, that the second call has to create a new object SitSituationGraph that decorates the (unconverted) root situation graph of SitSituationGraphTree. In this way, the actual instance of the referenced situation graph does not matter – it could be possible, that we decorate a SitSituationGraph by itself, but as we discard the reference to all created objects after the serialization, the model does not blow up by decorated objects of decorated objects and so on.

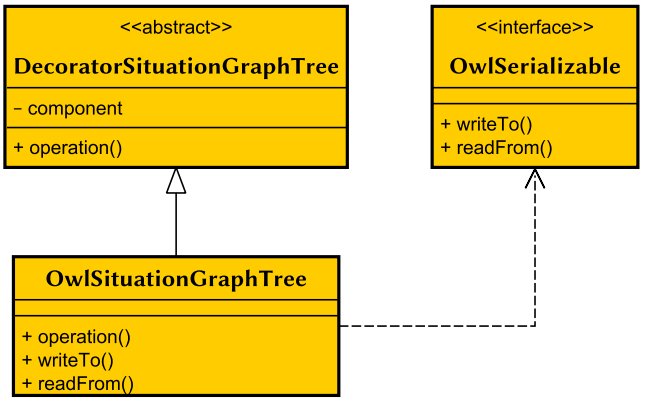

Figure 5.4: Extract of the Uml-Diagram about the interaction among the Decorator and the Owl-model. Classes in the Owl-model like OwlSituationGraphTree inherits from its DecoratorSituationGraphTree and implement OwlSerializable. The interface OwlSerializable contains the two methods writeTo() and readFrom().

When visiting the writeTo(...) function of the SitSituationGraph, we do the same for every associated situation scheme in a recursive manner: create a new Sit-object around the associated objects and call their writeTo(...) function. The recursion terminates in situation schemes that does not have conceptual specializations. The following Section describes the recursion in detail.

Chapter 6

Equivalence of SGTs and the SGT Ontology

In the previous chapters, we described the design and the implementation of the transformation from an SGT to Owl and the handling of different representation formats of the SGT. This chapter extends Chapter 5 by the proof that a situation graph tree can be described in an ontology. The proof shows two directions. First, Section 6.1 describes the way a situation graph tree is written into an ontology. This proof is backed by pseudo code which represents the corresponding part of the program SGTowl in a very abbreviated manner. Then, Section 6.2 shows the other way: a situation graph tree is read from an Owl ontology into our data structure. Although the writing to and reading from the ontology is nearly symmetric, challenges arise – for instance the Owl Api does not stick to the order of asserted axioms.

We limit our proof to the program code, we have written. A proof of the correctness of third-party tools and libraries is omitted. Such a third party library are the Owl Api in the first place. So, we assume these libraries and tools (i.e. the Java JRE) work correctly.

6.1 SGT Concepts in Owl Representation

To prove that we can express all concepts of a situation graph tree in an Owl ontology, we show a program that does the job. That program has to fulfil several requirements which are shown one by one.

(i) The program has to terminate. In fact, we can show our program terminates in $O(n)$ where n is the amount of concepts in the situation graph tree to transform.
(ii) All informations an SGT provides have to be represented in the ontology.
(iii) The program must ensure the structure of an SGT.

6.1.1 Termination of the Transformation

Algorithm 6.1: SGTwalkThrough()

```
   input : SituationGraphTree
1  begin
       // See init(): Procedure 6.3
2      context ← OwlSituationGraphTree.getRootGraph() ;
       // visit OwlSituationGraph, see Procedure 6.4
3      foreach situation ∈ context.getSituationSchemes() do
           // visit OwlSituation, see Procedure 6.5
4          situation.getActionScheme().visit() ;          // see Procedure C.1
5          situation.getStateScheme().visit() ;           // see Procedure C.2
6          foreach specialization ∈ situation.getSpecializations() do
               // visit OwlSpecialization, see Procedure 6.6
7              context ← specialization.getSpecializationGraph() ;
8              go to line 3 ;
9          foreach prediction ∈ situation.getPredicitons() do
10             prediction.visit() ;                       // see Procedure C.3
```

The structure of a situation graph tree transformed into an Owl ontology by walking through its structure recursively. The Algorithm 6.1 shows pseudo code about the way SGTowl walks through the structure. For reasons of increased readability, we chose using the expression *visit* instead of the function call writeTo(...). The algorithm takes a situation graph tree as parameter and visits it. There, it sets the root graph to the current context and visits every situation within that context (line 3). At the situation, the action scheme as well as the state scheme are visited. These steps always return and does not change the way of walking through the structure as it is shown later in this section. The visiting of every specialization in the current situation influences the walk through the SGT structure massively. As the destination of a specialization is a situation graph like the root graph, we change the current context to that situation graph and continue in line 3. So, we recursively visit all descendent SGT elements in the tree-like hierarchy below the current situation scheme. Returning from the recursion, we finally visit the situation schemes predictions. Like action or state schemes, predictions do not affect the way we walk through the data structure.

The recursion in Algorithm 6.1 always terminates. The numbers of iterations of the **foreach** loop are fixed for each context in line 3 as changing the context creates a new

loop that does not affect the loop of the previous context. The same argument holds for the loop in line 6.

Every situation graph is visited once in maximum. The specialization of a situation scheme by situation graphs containing situation schemes that are ancestors in the tree hierarchy is forbidden (Arens, 2004). The program checks whether the specializing situation graph is in the ancestors of the situation scheme while adding specializations. Thus, loops are avoided and (i) holds with respect to the termination of the program.

The rest of this section shows that every visit terminates and that he translates every SGT concept to the ontology. The following procedures' pseudo code needs an explanation in some places. The function `getOwlIndividual(SGTelement)` converts an SGT concept deterministic to an unique Owl individual. Individuals in the ontology are identified by its name, if two individuals are declared with the same name, they are the same individual. So the function mostly creates Owl individuals with a name of a combination of the SGT concepts, its internal identifier, and a number that represents for example the position of an element in a list. The function `createAxiom(subject,property,object)` creates an axiom that applies the expression property.object to the individual subject.

Procedure 6.2: addOption

```
   input : OWLIndividual this; OWLObjectProperty property; Attributes opt1, opt2
1  begin
2      if isAttribute (opt1) then
3          individual ← getOwlIndividual(opt1) ;
4      else
5          individual ← getOwlIndividual(opt2) ;
6      axiom ← createAxiom(this,property,individual) ;
7      ontology.add(axiom) ;
```

The Procedure 6.3 shows the visit of a situation graph tree element. It is called `init()` as it is the entrance to the walk through the data structure. In line 5 the context changes to the root graph and returns a reference to that root graphs individual. As previously shown, this function call returns under the circumstances that the visiting of every object terminates. The reference is connected to the situation graph tree with the object property hasRootGraph in line 6. That axiom is added to the ontology in line 7. From line 8 to line 14 including, individuals of all situation graph trees attributes are created,

Procedure 6.3: init

```
output: OWLIndividual SituationGraphTree
1  begin
2      this ← getOwlIndividual(OwlSituationGraphTree);
3      ontology.add(this) ;
4      rootGraph ← OwlSituationGraphTree.getRootGraph();
5      rootGraph ← rootGraph.visit();
6      axiom ← createAxiom(this,hasRootGraph,rootGraph) ;
7      ontology.add(axiom) ;
       // Add the attributes
8      addOption(this,hasAttributeDepth,DEPTH,BREADTH) ;
9      addOption(this,hasAttributeGreedy,GREEDY,NONGREEDY) ;
10     addOption(this,hasAttributeSingular,SINGULAR,PLURAL) ;
11     addOption(this,hasAttributeTraversal,TRAVERSAL,OCCURENCE) ;
12     individual ← getOwlIndividual(getDefaultIncremental()) ;
13     axiom ← createAxiom(this,hasAttributeIncremental,individual) ;
14     ontology.add(axiom) ;
```

the attributes are added to the ontology in O(5).

Procedure 6.4: OwlSituationGraph.visit

```
output: OWLIndividual SituationGraph
1  begin
2      this ← getOwlIndividual(OwlSituationGraph);
3      ontology.add(this) ;
4      individual ← getOwlIndividual(getDefaultIncremental()) ;
5      axiom ← createAxiom(this,hasAttributeIncremental,individual) ;
6      ontology.add(axiom) ;
7      foreach element situation of getSituationSchemes() do
8          individual ← situation.visit() ;
9          axiom ← createAxiom(this,hasSituationScheme,individual) ;
10         ontology.add(axiom) ;
           // Add situation schemes marked with the start flag
11         if situation.isStartSituation then
12             axiom ← createAxiom(this,hasStartSituationScheme,individual) ;
13             ontology.add(axiom) ;
           // Add situation schemes marked with the end flag
14         if situation.isEndSituation then
15             axiom ← createAxiom(this,hasEndSituationScheme,individual) ;
16             ontology.add(axiom) ;
17     return this;
```

The visit of the situation graphs shows Procedure 6.4. At the beginning, the individual for the current visited situation graph is created and added to the ontology. As a situation graph is marked with the flag defaultIncremental, this flag is added to the ontology beginning with line 4. Until now, the operations need time in O(1) each. In line 7 we loop through the list of assigned situation schemes and visit them in the following line. The checks and subsequent assertions of starting and ending situations in the lines 11 and 14 finish in constant time. So procedure 6.4 needs computational time in $O(n \times O(m))$ where n is the length of the situation scheme list and O(m) is the time needed for the visit of a situation scheme. It can be easily seen, that every operation terminates, the termination of the situation schemes visit shows the initial proof as well as the description of the following procedures.

The visitation of a situation scheme in Procedure 6.5 terminates if all operations and subsequent visits terminate. Again, we start with the declaration of the situation individual itself (Line 2) and add the incremental flag. The action and the state scheme are added in Lines 7 and 10 respectively. The Procedures C.1 and C.2 show the evidence that the "visit" function calls for action and state schemes return in nearly constant time. The visitation of each specialization in line 13 triggers the context change to a specializing situation graph. If there are no specializations, the recursion ends in this situation and the stack of function calls is processed. The procedure of visiting situation schemes ends in the visit of every prediction. Procedure C.3 shows the termination of visiting predictions. An upper bound for the amount of predictions for a situation scheme is the amount of situation schemes in the parent situation graph O(m): a situation scheme may have one prediction edge to any other situation scheme at the maximum, including itself. A prediction edge to a situation scheme outside the current situation graph is prohibited. Only outgoing edges are considered, so every prediction edge is visited exactly once.

The recursion in the walk through the situation graph tree structure starts in the visitation of a Specialization. Procedure 6.6 outlines the transition of specializations to Owl. The required features to translate are the connection to the situation graph that specializes the situation and the index number in the order of the other specializations of that situation. In Line 7, we visit the connected situation graph. After that function call returns, we get the reference to the situation graphs Owl individual and assign it to the specialization by the hasSituationGraph object property. The ordered index is assigned afterwards. Although it is not an essential feature of an SGT, we additionally store the specialized situation as we use that link in our SGT data structure. The termination of the function call is obvious, the amount of time is constant apart from the visit of the specializing situation graph.

The procedures for the StateScheme and the ActionScheme can be found in the Appendix C, the only feature they contain is a list of predicates. Thus, it is obvious that their transformation to Owl terminates, if the visitation of a Predicate terminates. Procedure 6.7 outlines the transition of predicates into Owl. According to the description in

Procedure 6.5: OwlSituation.visit

```
output: OWLIndividual Situation
1  begin
2      this ← getOwlIndividual(OwlSituation) ;
3      ont.add(this) ;
4      ind ← getOwlIndividual(getDefaultIncremental()) ;
5      axiom ← createAxiom(this,hasAttributeIncremental,ind) ;
6      ont.add(axiom) ;
       // Visit the action scheme
7      ind ← getActionScheme().visit() ;
8      axiom ← createAxiom(this,hasActionScheme,ind) ;
9      ont.add(axiom) ;
       // Visit the state scheme
10     ind ← getStateScheme().visit() ;
11     axiom ← createAxiom(this,hasStateScheme,ind) ;
12     ont.add(axiom) ;
       // Visit all the specializations
13     foreach element e of getSpecializations() do
14         ind ← e.visit() ;
15         axiom ← createAxiom(this,hasConceptualSpecification,ind) ;
16         ont.add(axiom) ;
       // Visit all the predictions
17     foreach element e of getPredicitons() do
18         e.situation ← this;
19         ind ← e.visit() ;
20         axiom ← createAxiom(this,hasTemporalSpecification,ind) ;
21         ont.add(axiom) ;
22     return this;
```

Procedure 6.6: OwlSpecialization.visit

output: OWLIndividual Specialization

```
1  begin
2      this ← getOwlIndividual(OwlSpecialization);
3      ont.add(this) ;
       // Link the source of this specialization
4      ind ← getOwlIndividual(getSpecializedSituation()) ;
5      axiom ← createAxiom(this,hasPreviousSituationScheme,ind) ;
6      ont.add(axiom) ;
       // Link the destination of this specialization
7      ind ← getSpecializationGraph().visit() ;
8      axiom ← createAxiom(this,hasSituationGraph,ind) ;
9      ont.add(axiom) ;
       // Add the ordered index
10     literal ← getOwlLiteral(getSpecializationIndex()) ;
11     axiom ← createAxiom(this,hasOrderedIndex,literal) ;
12     ont.add(axiom) ;
13     return this;
```

Procedure 6.7: OwlPredicate.visit

output: OWLIndividual Predicate

```
1  begin
2      this ← getOwlIndividual(OwlPredicate);
3      ont.add(this) ;
4      size ← #getArguments() ;
5      for i = 0 to size do
           // Create object property axioms
6          axiom ← subProperty(hasArgument + i, hasArgument) ;
7          ont.add(axiom) ;
           // Add the arguments
8          ind ← getArguments(i).visit() ;
9          axiom ← createAxiom(this,hasArgument + i, ind) ;
10         ont.add(axiom) ;
11     return this;
```

Section 4.2.1.7, for the X-th argument of the predicate, an object property hasArgumentX must be created. Line 6 handles the declaration of this object properties, in Line 9 the Arguments are applied correctly to the predicate. The visit of an argument in Line 8 simply redirects the function call to the visitation of a Predicate, if it is a ComplicatedArgument or to the visitation of a Variable. As the predicate is finite, we cannot run into an endless loop by visiting a ComplicatedArgument. In Procedure C.6 can be seen that Variable only is turned into an individual, so this visit always terminates. As there are not any further features to add, we can state that every Predicate is mapped correctly to the ontology.

The left SGT concepts to show the equivalence are Prediction, BindingAssingment, and BindingRelease. The visit of these classes is similar to the procedures presented so far thus, we skip a detailed description. They can be found in C.3, C.4, and C.5, respectively. It can be seen easily that they map all the features of their SGT description and that the procedures terminate. The computational time of the Prediction is linear in its amount of bindings which need constant time for the transformation.

As described above, the transformation of all the elements need constant time plus the computational time of objects connected to it. Every element in the SGT is visited once, after leaving an element, it is never visited again. Thus, in a SGT with n elements, the transformation needs O(n) time.

6.2 Owl transformation to SGT

Procedure 6.8: loadowl

```
1 begin
2     buildMapIndividual2OwlClass();
3     buildClassForest();
4     buildMapIndividual2SGTClass();
5     buildMapIndividual2SGTElement();
6     assembleSGT();
```

The transformation of situation graph trees the way backwards from an Owl ontology works as depicted in the Procedure 6.8. At first, we have to know the type of individuals. We assert individuals may have a single type only (sgt2owl only applies one type to an individual). Next, we create a ClassForest that represents the subclass relation in a tree-like structure. The ClassForest allows us to map the type of an individual to a known SGT class by searching the tree upwards until the root or an SGT class is reached. As we

now know the type of an individual that corresponds to an SGT concept, we create SGT objects for each Owl individual and map it to the object. The final step is the assembling of the SGT objects according to the assertions in the ontology.

6.2.1 Map Owl-Individual

The transformation to the SGT model begins with building a map that stores the individuals of an ontology and the corresponding Owl-class. The process primarily is realized within the Owl Api framework by implementing an OWLOntologyWalkerVisitor and using an OWLOntologyWalker. The walker walks through the whole ontology and calls for example visit(OWLClassAssertionAxiom desc) when he meets an OWLClassAssertionAxiom. The visitor stores every visit of such a OWLClassAssertionAxiom into a list. The proof of these functions correctness is out of this thesis' scope.

The implementation of the visitor has additional functionality: it builds up an OwlClassForest when visiting OWLSubClassOfAxiom. The evidence that the class tree represents the subclass axioms of an ontology in a tree-like data structure is shown in the following section.

6.2.2 Of Subclasses, Trees, and Forests

With an eye towards "non-SGT" ontologies, we have to provide a technique to handle individuals that do not have an SGT class as type. Further, we want to allow a semantic refinement of variables and predicates in the ontology thus classes are defined that have no representative in the situation graph tree model. But we require that one of its super types is an SGT class so we can translate it to the model. There are no data structures in an ontology, there only are axiom and declarations etc. So we have to read all the subClassOf-relations and build up a data structure that is searchable effectively. The OwlClassForest is a helper class that always holds a forest of trees of subclass axioms.

A forest is a set of trees. We say a tree is a data structure of at least one connected objects of the type GraphNode, thus every GraphNode is also a tree. A GraphNode is connected to another one by the parent or child property. While the parent of a GraphNode always is an unique GraphNode, it may has multiple children. A GraphNode must not have a parent as a child.

The Procedure 6.9 constructs a forest of Owl classes by adding OWLSubClassOfAxioms as parent-child relations to the existing forest. During the processing of the asserted OWLSubClassOfAxiom in the ontology, the current forest is subject to the following invariants which holds at the end of every procedures call:

(i) an Owl class without super classes is always the root of a tree.

Procedure 6.9: addSubClassOfAxiom

```
   input  : OWLSubClassOfAxiom subClassAxiom
1  begin
2      child ← subClassAxiom.getSubClass() ;
3      child ← new GraphNode(child) ;
4      foreach non-anonymous class parent ∈ subClassAxiom.getSuperClass() do
5          if parent == owl:Thing then
6              go to 4 ;
7          parent ← new GraphNode(parent) ;
8          foundParent ← forest.search(parent) ;
9          if foundParent then
10             foundChild ← forest.search(child) ;
11             if foundChild then
12                 add ← foundChild ;
13                 if empty(add.parent) then
14                     forest.remove(add) ;
15             else
16                 add ← GraphNode(child);
17             foundParent.addChild(add) ;
18             go to 4 ;
19         foreach GraphNode tree ∈ forest do
20             if tree.getRoot() == child then
21                 forest.remove(tree) ;
22                 forest.add(parent) ;
23                 parent.addChild(child) go to 4 ;
24         parent.addChild(child) ;
25         forest.add(parent) ;
```

(ii) an Owl class with super classes is the child of every non-anonymous superclass.
(iii) an Owl class with super classes is never the root of a tree.

As a reminder, an example of a OWLSubClassOfAxiom could be as follows:

$$\text{subclass} \sqsubseteq \top \sqcap \text{supclassA} \sqcap \text{objectpropA.classB} \sqcap \text{supclassC} \sqcap \leqslant 4\ \text{objectpropB.classD}$$

In this example, the left side represents the subclass and the whole right side the superclass in the OWLSubClassOfAxiom. The Owl Api handles the right side as an OWLSubClassOfAxiom while the objectpropA.classB and $\leqslant$ 4 objectpropB.classD are anonymous classes. Such anonymous classes as well as the $\top$-class owl:Thing are of no interest, as the first does not represent an *is-a* relationship and the latter are implicit for every class. The Lines 4 and 5 excludes these classes.

The evidence of Procedures 6.9 correctness is shown by induction on the invariants (i)-(iii). In the first call of `addSubClassOfAxiom`, the forest is empty. Without loss of generality we state that the axiom to add only holds a single non-anonymous superclass SC which is unequal to owl:Thing, so the **foreach**-loop (Line 4) terminates after the first loop. As the forest is empty, no parent of SC are found and the procedure continues in Line 24. At the end of the function call, we have a forest containing a single tree which has SC as root node and the subclass as its only child, the invariants (i)-(iii) hold. Let us show that they also hold at the nth call of the procedure.

We assume, that the invariants hold at the nth call and without loss of generality that the axiom to add only holds a single non-anonymous superclass SC unequal to owl:Thing so we can skip to line 9. At this point there are three cases we handle separately:

1. The parent (superclass) already is in the forest.
2. The child is the root of a forest.
3. Everything else.

The algorithm searches for the superclass (parent) in the forest and – in case of success – afterwards for the subclass (child) which covers case 1. If the child is already in the graph, a reference to its corresponding node is added as child to the parent (Line 17), otherwise a new node is used for that. In line 13 is checked whether the child already is the root of a graph. As this entity now has a superclass, we remove that graph from the forest so (iii) holds. The invariant (ii) holds as the procedure so far is performed for every anonymous superclass. If the class does not have a superclass, the **foreach**-loop is not executed and the assumption for (i) holds. As then in line 18 the loop ends and therefore the function call, the invariants stay valid.

If the parent was not found in the forest, we search for the child to add in the root of the graphs. This case (2) occurs when the current child previously was declared to be a superclass. Then, we have to remove that graph from the forest and attach it to the list

of the parents children. As the child now has a superclass and it is removed from the forest (21), the invariants (i) and (iii) hold, (ii) follows as written before. In case 3 is the situation the same as in the previous one but the child is not the root of a graph, so we do not have to remove that graph from the forest. The search for elements in the forest handles Procedure C.7 and C.8 respectively. They are standard implementations of the depth-first search and thus not be proven here.

6.2.3 Prerequisites for the Assembling

With the prerequisite of having the Owl classes in a tree-like data structure we continue the Procedure 6.8 by building a map of Owl individuals to SGT elements (line 4). This process iterates over the list of individuals built in Section 6.2.1 and maps their classes to an SGT class. The mapping is achieved by identifying the individuals type within the OwlClassForest class (Section 6.2.2) is found or the root has been passed. In the last case, where no corresponding SGT class was found, we can ignore this individual. Such situations may appear by working with foreign ontologies. As this process is trivial and heavily relies on third-party software, the prof is omitted in this thesis.

We also skip the proof of correctness for the procedure in line 5 for the same reason. In that procedure, we iterate through the map of Owl individuals to SGT classes and instantiate SGT objects according to the class. These objects are mapped to the Owl individuals in a look-up-table. At this moment, the created SGT object instances only consist of a reference but does not have any SGT-semantic content, yet. The next sections shows that every SGT feature we put into the ontology is restored correctly.

6.2.4 Assembling the SGT

Finally, we reached the last step of the transformation Procedure 6.8 in line 6. At this moment, we have built a lookup-table that maps every Owl individual to an unique instance of its corresponding SGT class. In the final assembling, we iterate over this table and call the assemble function for each class. This function gets the loops current Owl individual as an argument, so the instantiated object knows the data it has to reconstruct from the ontology. The lookup-table can be seen as a global variable. Procedure 6.10 shows an exemplary function call, here for an SituationGraphTree. At first, the root graph individual is read from the ontology (Line ch:proof:sec:owl2sgt:sgtassemble:sgt:getroot). Second, this individual is searched in the lookup-table. The result is a reference to the root-SituationGraph-object which is applied to the SituationGraphTree as its root graph. Beginning with the line 4, the attributes of the situation graph tree are extracted and applied. As they are simple enumerations instead of objects, their string value

Procedure 6.10: OwlSituationGraphTree.assmeble

```
   input  : OWLIndividual individual
1  begin
2      rootGraph ← individual.getObjectProperty(hasRootGraph) ;
3      setRootGraph(lookUpTable.get(rootGraph));
4      depth ← individual.getObjectProperty(hasAttributeDepth) ;
5      setAttribute(depth);
6      greedy ← individual.getObjectProperty(hasAttributeGreedy) ;
7      setAttribute(greedy);
8      singular ← individual.getObjectProperty(hasAttributeSingular) ;
9      setAttribute(singular);
10     traversal ← individual.getObjectProperty(hasAttributeTraversal) ;
11     setAttribute(traversal);
12     incremental ← individual.getObjectProperty(hasAttributeIncremental) ;
13     setDefaultIncremental(incremental);
```

Procedure 6.11: restoreListOrder

```
   input  : OWLIndividual individual, OWLObjectProperty hasProp, List list
1  begin
2      index ← 0 ;
3      indexMap ← new Map ;
4      foreach OWLIndividual ind ∈ individual.hasProp  do
5          sgtelem ← SGTlookUpTable.get(ind) ;
6          index ← ind.getIndexFromName() ;
7          indexMap.put(index - 1, sgtelem) ;
8      size ← #indexMap ;
9      for i = 0 to size − 1  do
10         list.add(indexMap.get(i));
```

can be applied directly. At the end of this procedure, all informations belonging to the SituationGraphTree are completely restored.

We propose another procedure 6.12 as it introduces the function call `restoreListOrder`. Although we already mentioned that the order of predictions or specializations is irrelevant (Münch *et al.*, 2011b) for the recognition of situations, it counts on this evidence. We additionally wrote an unit test, that loads an SGT from a Sit++ file, stores the SGT into an ontology, loads the SGT from the ontology, and write it back to an Sit++ file. Then, the original Sit++ file is compared line by line with the resulting file. The only lines that may differ are comments otherwise the test fails. As we want to support this proof by such a test, we also have to restore the order of predictions or specializations, even the order of predicates in a state scheme is affected. As we encoded the index into the name of an individual, we now restore this index. Procedure 6.11 begins with the initialization of the index and a map that servers as a sparse array. The **foreach**-loop in line 4 iterates over the finite list of individuals that are connected by the given object property hasProp. In a loop, we get the SGT element that corresponds to the current individual, extract its index, and add them to the map. We need the latter as in Java, this list does not support the insertion of elements at an arbitrary position i if the element at the position (i − 1) is empty. A second loop (line 9) is needed that iterates from 0 to the length of the map and adds the SGT element with the index i to the list. Thus, the order of the resulting list is accordingly to the indices of the individuals.

Procedure 6.12: OwlSituationGraph.assmeble

```
   input  : OWLIndividual individual
1  begin
2      incremental ← individual.getObjectProperty(hasAttributeIncremental) ;
3      setDefaultIncremental(incremental);
4      restoreListOrder(individual,hasSituationSchema,sitList) ;
5      getSituations().addAll(sitList) ;
6      foreach OWLIndividual ind ∈ individual.hasStartSituationSchema  do
7          getSituations().get(ind).setStartSituation(true) ;
8      foreach OWLIndividual ind ∈ individual.hasEndSituationSchema do
9          getSituations().get(ind).setEndSituation(true) ;
```

Procedure 6.12 shows the assembling of a situation graph. It shows that all concepts belonging to a situation graph are restored, namely the value defaultIncremental and the list of situations. As in the SGT ontology, the information about situation schemes being a start situation or an end situation is stored in the situation graph, we extract

this information and apply it to the situation schemes in this procedure beginning with Line 6.

The Procedure 6.13 for assembling a situation scheme is a representative for the remaining procedure. The schematic about reading informations from the ontology and applying to the corresponding SGT element apply to them all in a very similar way. They can be found in Appendix C.2.

Procedure 6.13: OwlSituation.assmeble

```
   input  : OWLIndividual individual
1  begin
2      ind ← individual.getObjectProperty(hasAttributeIncremental) ;
3      setDefaultIncremental(ind);
4      ind ← individual.getObjectProperty(hasActionScheme) ;
5      setActionScheme(ind);
6      ind ← individual.getObjectProperty(hasStateScheme) ;
7      setStateScheme(ind);
8      restoreListOrder(individual,hasConceptualSpecification,list) ;
9      getSpecializations().addAll(list) ;
10     restoreListOrder(individual,hasTemporalSpecification,list) ;
11     getPredictions().addAll(list) ;
```

Having showed that every feature is restored correctly and as the evidence of the termination of each proposed procedure is obvious, we successfully have proven that the transformation of an SGT ontology to an SGT works correctly. Thus, in combination with the evidence of the other transformation direction in Section 6.1, the equivalence of the two representation formats is proven.

Chapter 7

Summary

7.1 Conclusion

As a discussion about the thesis already takes place in Chapter 4.4, we focus upon the conclusion in this Chapter. In Chapter 2, we presented the foundations for this thesis. This includes an introduction to Description Logic, the Web Ontology Language as well as an overview on Situation Graph Trees. Further, we outlined the Fuzzy Metric Temporal Horn Logic and embedded this work in a Cognitive Vision System. Related work was presented in Chapter 3.

The comparison of SGTs with ontologies took place in Chapter 4. We analysed the ability of representing an SGT in an ontology. As a result, we got a well designed and reusable ontology that represents an SGT perfectly. The developed ontology has expressiveness in $\mathcal{SHOIN}$, thus we support the maximum expressiveness in Owl-DL. The definition of SGTs allowed us not to need the rules extension Swrl as done before in e.g. (Baumgartner *et al.*, 2010; Bohlken and Neumann, 2009).

The equivalence between SGTs and the developed ontology was proven by giving a program that implements the transformation from an SGT to an ontology and vice versa. The transformation performs without any loss of information. The application was presented in Chapter 5, the comprehensive proof in Chapter 6. With the work of this thesis, a representation of SGTs is available in a standardized and commonly accepted file format for knowledge representation and sharing.

As a further benefit, this thesis opens up new possibilities. A technique was proposed for the development of a transformation ontology that allows importing foreign situation awareness ontologies without any adjustments to the developed application.

7.2 Future Work

In this thesis we developed the foundation for further research. As we proposed a way to enhance the situation graph tree ontology with additional information and to create a mapping for other situation awareness ontologies in the same way, the comparison of situation graph trees with other representations for situations are of interest. Comparing situation graph trees with other representations leads to interesting questions about the conversion of spatial relations, for example. In a situation graph tree, the spatial relations between agents are coded into Fmthl predicates, thus this part is shifted into the inference engine. Other ontologies have classes for such relations, so a conversion from predicates to classes has to be emerged.

A helping strategy may be the refinement of the current realization of predicates in the ontology. Currently, the Fmthl predicates are represented in the ontology by its name and its arguments. Thus leads to absence of semantics about the predicates itself. As there are some predicates that are used very often, for example to specify the distance between an agent and a patient. These predicates could be refined further by ontological concepts. So, the exchange of situation definition across research groups can be simplified. The so far presented improvements for the handling of predicates may be enhanced further by encoding Fmthl predicates itself into an ontology.

Apart from arising questions about semantics in the ontology, the proposed program SGTowl can be developed further, too. In this thesis, we encode the situation graph tree in individuals. The only place where we actually require individuals is the encoding of ordered indices of outgoing edges. But as they are deprecated and supported only for backwards compatibility, this requirement easily could be dropped. Then, a concrete situation graph tree could also be represented as classes instead of individuals in the ontology which increases the flexibility in creating a mapping to other situation awareness ontologies.

A further improvement would be the ability of handling individuals that have multiple types. In the scope of this thesis, such a feature was not needed but the requirement of supporting it could arise by importing third-party ontologies.

The development of SGTowl extracted the current model of the situation graph trees from the SGT-Editor. During the thesis it was decided to switch the visual representation framework of the SGT-Editor to a newer version. To avoid twice the effort or breaking the developed model to support the soon-to-be-outdated SGT-Editor, we omitted the back-port of SGTowl into the editor. So, the integration of SGTowl as standard knowledge format in the next generation SGT-Editor is still an outstanding task.

Appendix A

Timetable

A.1 Planned Timetable

Time	Task
4 weeks	Review of literature, discussion of various ontology representations, representation of concepts (time, hierarchy etc.)
3 weeks	Development of an extensive ontology.
3 weeks	Derivation of requirements
2 weeks	Introduction to the api of the chosen ontology representation. If there is no existing API, this step will take more time.
2 weeks	Design the transformation from ontology to SGT and vice versa.
5 weeks	Implementation of the transformation from an ontology to SGT.
2 weeks	Implementation of the transformation SGT to an ontology.
3 weeks	Evaluation of the implementation.
2 weeks	Finalization the diploma thesis and preparation of the presentation.

A.2 Final Timetable

Time	Task
02.04.2012- 07.05.2012	Review of literature, discussion of various ontology representations, representation of concepts (time, hierarchy etc.)
07.05.2012- 20.05.2012	Porting the Virat Situation Graph Tree into a basic ontology. Discussion about the pros and cons of that first solution. Aim: Show a proof of concept, that is actual possible to represent a Situation Graph Tree in an ontology.
21.05.2012- 08.06.2012	Design an meta-model for SGT in an ontology. Port the Virat Situation Graph Tree to the meta-model. Aim: Provide a consistent ontology to use for later use.
11.06.2012- 15.06.2012	Setup the new project called sgtowl with maven. Aim: Provide a state of the art development environment, be independent of the development of the SGT-Editor
18.06.2012- 22.06.2012	Training in usage of the framework owlapi, as well as the automated development of situation graph trees Aim: Be ready to translate situation graph trees from Owl to the SGT-Editor and vice versa
25.06.2012- 06.07.2012	Development of the SGT model. Aim: Providing a extensible and future-proof software architecture, that can handle multiple representation formats
09.07.2012- 20.07.2012	Implementing the transformation from an SGT to an ontology Aim: Having a full-featured SGT encoded into an ontology for the backwards transformation.
23.07.2012- 22.07.2012	Preparing the half-time presentation
25.07.2012	Presenting the intermediate result
30.07.2012- 24.08.2012	Implementing the transformation from an ontology to an SGT Aim: Having a bidirectional transformation. From now on, the ontology may serve as a replacement for the Sit++ representation format.

Time	Task
30.07.2012- 24.08.2012	Implementing the transformation from an ontology to an SGT Aim: Having a bidirectional transformation. From now on, the ontology may serve as a replacement for the Sit++ representation format.
27.08.2012- 31.08.2012	Refining the transformation application Aim: Maintain readability of the application and fixing bugs.
03.09.2012- 14.09.2012	Writing the proof of equivalence.
17.09.2012- 28.09.2012	Finalizing the diploma thesis.

Appendix B

Fmthl Examples

We introduced an example situation graph tree in Section 2.5. The following Fmthl predicates represent the situation graph tree of Figure 2.11:

```
always (sgt_vars([Agent, SplitList, TmpList, AreaList, PatientList])).
always (sgt_root_graph(gr_ED_SITGRAPH0)).
always (sgt_graph(gr_ED_SITGRAPH0)).
always (sgt_graph(gr_ED_SITGRAPH1)).
always (sgt_situation(lim_ID_SIT8)).
always (sgt_situation_name(lim_ID_SIT8, sit_Root)).
always (sgt_situation(lim_ID_SIT9)).
always (sgt_situation_name(lim_ID_SIT9, sit_SituationSplit)).
always (sgt_situation(lim_ID_SIT10)).
always (sgt_situation_name(lim_ID_SIT10, sit_SituationApproach)).
always (sgt_situation(lim_ID_SIT11)).
always (sgt_situation_name(lim_ID_SIT11, sit_InAgentArea)).
always (sgt_sit_of_graph(lim_ID_SIT8, gr_ED_SITGRAPH0)).
always (sgt_sit_of_graph(lim_ID_SIT9, gr_ED_SITGRAPH1)).
always (sgt_sit_of_graph(lim_ID_SIT10, gr_ED_SITGRAPH1)).
always (sgt_sit_of_graph(lim_ID_SIT11, gr_ED_SITGRAPH1)).
always (sgt_start_sit(lim_ID_SIT8)).
always (sgt_start_sit(lim_ID_SIT10)).
always (sgt_start_sit(lim_ID_SIT11)).
always (sgt_end_sit(lim_ID_SIT8)).
always (sgt_end_sit(lim_ID_SIT9)).
always (sgt_prediction_edge(prededge10)).
always (sgt_prediction_edge(prededge11)).
always (sgt_prediction_edge(prededge12)).
always (sgt_prediction_edge(prededge13)).
always (sgt_prediction_edge(prededge14)).
always (sgt_binding(prededge10, prededge10_BIND)).
always (sgt_binding(prededge11, prededge11_BIND)).
always (sgt_binding(prededge12, prededge12_BIND)).
```

```
always (sgt_binding(prededge13, prededge13_BIND)).
always (sgt_binding(prededge14, prededge14_BIND)).
always (sgt_prediction(prededge10,lim_ID_SIT9,lim_ID_SIT9)).
always (sgt_prediction(prededge11,lim_ID_SIT10,lim_ID_SIT11)).
always (sgt_prediction(prededge12,lim_ID_SIT10,lim_ID_SIT10)).
always (sgt_prediction(prededge13,lim_ID_SIT11,lim_ID_SIT11)).
always (sgt_prediction(prededge14,lim_ID_SIT11,lim_ID_SIT9)).
always (sgt_specialization_edge(specedge2)).
always (sgt_specialization(specedge2,lim_ID_SIT8,gr_ED_SITGRAPH1)).
always (sgt_state(lim_ID_SIT8, Vars) :-(
        nth_elem(Vars, Agent, 1),
        active(Agent))).
always (sgt_incr_state(lim_ID_SIT8, Vars) :-(
        nth_elem(Vars, Agent, 1),
        active(Agent))).
always (sgt_state(lim_ID_SIT9, Vars) :-(
        nth_elem(Vars, Agent, 1),
        nth_elem(Vars, SplitList, 2),
        nth_elem(Vars, TmpList, 3),
        nth_elem(Vars, AreaList, 4),
        truefilter(AreaList,have_distance(Agent,element,normal),TmpList),
        truefilter(TmpList,have_distance_change(Agent,element,increasing),SplitList))).
always (sgt_incr_state(lim_ID_SIT9, Vars) :-(
        nth_elem(Vars, Agent, 1),
        nth_elem(Vars, SplitList, 2),
        nth_elem(Vars, TmpList, 3),
        nth_elem(Vars, AreaList, 4),
        active(Agent),
        truefilter(AreaList,have_distance(Agent,element,normal),TmpList),
        truefilter(TmpList,have_distance_change(Agent,element,increasing),SplitList))).
always (sgt_state(lim_ID_SIT10, Vars) :-(
        nth_elem(Vars, Agent, 1),
        nth_elem(Vars, TmpList, 3),
        nth_elem(Vars, PatientList, 5),
        truefilter(PatientList,have_distance_change(Agent,element,decreasing),TmpList))).
always (sgt_incr_state(lim_ID_SIT10, Vars) :-(
        nth_elem(Vars, Agent, 1),
        nth_elem(Vars, TmpList, 3),
        nth_elem(Vars, PatientList, 5),
        active(Agent),
        truefilter(PatientList,have_distance_change(Agent,element,decreasing),TmpList))).
always (sgt_state(lim_ID_SIT11, Vars) :-(
        nth_elem(Vars, Agent, 1),
        nth_elem(Vars, AreaList, 4),
        nth_elem(Vars, PatientList, 5),
        truefilter(PatientList,have_distance(Agent,element,small_or_zero),AreaList))).
always (sgt_incr_state(lim_ID_SIT11, Vars) :-(
        nth_elem(Vars, Agent, 1),
        nth_elem(Vars, AreaList, 4),
```

```
        nth_elem(Vars, PatientList, 5),
        active(Agent),
        truefilter(PatientList,have_distance(Agent,element,small_or_zero),AreaList))).
always (sgt_incr_action(lim_ID_SIT8, Vars) :-(
        true)).
always (sgt_incr_action(lim_ID_SIT9, Vars) :-(
        true)).
always (sgt_incr_action(lim_ID_SIT10, Vars) :-(
        true)).
always (sgt_incr_action(lim_ID_SIT11, Vars) :-(
        true)).
always (sgt_nonincr_action(lim_ID_SIT8, Vars) :-(
        true)).
always (sgt_nonincr_action(lim_ID_SIT9, Vars) :-(
        nth_elem(Vars, Agent, 1),
        nth_elem(Vars, SplitList, 2),
        behave_output(Agent,SplitList,'Split'))).
always (sgt_nonincr_action(lim_ID_SIT10, Vars) :-(
        nth_elem(Vars, Agent, 1),
        nth_elem(Vars, TmpList, 3),
        behave_output(Agent,TmpList,'Approach'))).
always (sgt_nonincr_action(lim_ID_SIT11, Vars) :-(
        true)).
always (sgt_process_binding(prededge10_BIND, Vars, VarsNew) :- (
        VarsNew := Vars)).
always (sgt_process_binding(prededge11_BIND, Vars, VarsNew) :- (
        reset_nth_elem(Vars ,3 ,VarsNew))).
always (sgt_process_binding(prededge12_BIND, Vars, VarsNew) :- (
        VarsNew := Vars)).
always (sgt_process_binding(prededge13_BIND, Vars, VarsNew) :- (
        reset_nth_elem(Vars ,3 ,VarsNew))).
always (sgt_process_binding(prededge14_BIND, Vars, VarsNew) :- (
        reset_nth_elem(Vars ,3 ,VarsNew))).
```

Appendix C

Additional Procedures for Chapter 6

C.1 SGT Concepts in Owl Representation

Procedure C.1: OwlActionScheme.visit

```
   output: OWLIndividual ActionScheme
1  begin
2      this ← getOwlIndividual(OwlActionScheme);
3      ont.add(this) ;
       // Visit all the predicates
4      foreach element e of getPredicates() do
5          ind ← e.visit() ;
6          axiom ← createAxiom(this,hasPredicate,ind) ;
7          ont.add(axiom) ;
8      return this;
```

Procedure C.2: OwlStateScheme.visit

```
output: OWLIndividual StateScheme
1  begin
2      this ← getOwlIndividual(OwlStateScheme);
3      ont.add(this) ;
       // Visit all the predicates
4      foreach element e of getPredicates() do
5          ind ← e.visit() ;
6          axiom ← createAxiom(this,hasPredicate,ind) ;
7          ont.add(axiom) ;
8      return this;
```

Procedure C.3: OwlPrediction.visit

```
output: OWLIndividual Prediction
1  begin
2      this ← getOwlIndividual(OwlPrediction);
3      ont.add(this) ;
       // Link the source of this prediction
4      ind ← getOwlIndividual(getPredictedSituation()) ;
5      axiom ← createAxiom(this,hasPreviousSituationScheme,ind) ;
6      ont.add(axiom) ;
       // Link the destination of this prediction
7      ind ← getOwlIndividual(getNextSituation()) ;
8      axiom ← createAxiom(this,hasNextSituationScheme,ind) ;
9      ont.add(axiom) ;
       // Add the ordered index
10     literal ← getOwlLiteral(getPredictionIndex()) ;
11     axiom ← createAxiom(this,hasOrderedIndex,literal) ;
12     ont.add(axiom) ;
       // Add the bindings
13     foreach element e of getBindings() do
14         ind ← e.visit() ;
15         axiom ← createAxiom(this,hasBinding,ind) ;
16         ont.add(axiom) ;
17     return this;
```

Procedure C.4: OwlBindingAssignment.visit

output: OWLIndividual BindingAssignment

```
1  begin
2      this ← getOwlIndividual(OwlBindingAssignment);
3      ont.add(this) ;
       // Add the variable to be set
4      ind ← getVariableToBeSet().visit() ;
5      axiom ← createAxiom(this,hasVariableToBeSet,ind) ;
6      ont.add(axiom) ;
       // Add the variable to be assigned
7      ind ← getVariableToBeAssigned().visit() ;
8      axiom ← createAxiom(this,hasVariableToBeAssigned,ind) ;
9      ont.add(axiom) ;
10     return this;
```

Procedure C.5: OwlBindingRelease.visit

output: OWLIndividual BindingRelease

```
1  begin
2      this ← getOwlIndividual(OwlBindingRelease);
3      ont.add(this) ;
       // Add the variable to be released
4      ind ← getVariableToBeSet().visit() ;
5      axiom ← createAxiom(this,hasVariableToBeSet,ind) ;
6      ont.add(axiom) ;
7      return this;
```

Procedure C.6: OwlVariable.visit

output: OWLIndividual Predicate

```
1  begin
2      this ← getOwlIndividual(OwVariable);
3      ont.add(this) ;
4      return this
```

C.2 Owl Transformation to SGT

Procedure C.7: forest.search(node)

```
   input  : GraphNode node
   output: GraphNode foundNode
1  begin
2      foundNode ← null ;
3      foreach GraphNode tree ∈ forest do
4          foundNode ← tree.search(node) ;
5          if foundNode then
6              return foundNode ;
7      return foundNode ;
```

Procedure C.8: search(tree,node)

```
  input  : GraphNode tree,node
  output: GraphNode foundNode
1 begin
2     if isMarked(node) then
3         return null ;
4     if node == tree then
5         return tree;
6     mark(node) ;
7     foundNode ← null ;
8     foreach GraphNode child ∈ tree do
9         foundNode ← search(child,node) ;
10        if foundNode then
11            return foundNode ;
12    return null ;
```

Procedure C.9: OwlPrediction.assmeble

```
  input  : OWLIndividual individual
1 begin
2     ind ← individual.getObjectProperty(hasPreviousSituationScheme) ;
3     setPredictedSituation(ind);
4     ind ← individual.getObjectProperty(hasNextSituationScheme) ;
5     setNextSituation(ind);
6     restoreListOrder(individual,hasBinding,list) ;
7     getBindings().addAll(list) ;
```

Procedure C.10: OwlBindingAssignment.assmeble

```
  input  : OWLIndividual individual
1 begin
2     ind ← individual.getObjectProperty(hasVariableToBeAssigned) ;
3     setVariableToBeAssigned(ind);
4     ind ← individual.getObjectProperty(hasVariableToBeSet) ;
5     setVariableToBeSet(ind);
```

Procedure C.11: OwlBindingRelease.assmeble

```
  input  : OWLIndividual individual
1 begin
2     ind ← individual.getObjectProperty(hasVariable) ;
3     setVariableToBeReleased(ind);
```

Procedure C.12: OwlSpecialization.assmeble

```
  input  : OWLIndividual individual
1 begin
2     ind ← individual.getObjectProperty(hasPreviousSituationScheme) ;
3     setSpecializedSituation(ind);
4     ind ← individual.getObjectProperty(hasSituationGraph) ;
5     setSpecializationGraph(ind);
```

Procedure C.13: OwlScheme.assmeble

```
  input  : OWLIndividual individual
1 begin
2     restoreListOrder(individual,hasPredicate,list) ;
3     getPredicates().addAll(list) ;
```

Procedure C.14: OwlPredicate.assmeble

```
  input  : OWLIndividual individual
1 begin
2     ind ← individual.getObjectProperty(hasAttributeIncremental) ;
3     setDefaultIncremental(ind);
4     setPredicateName(individual.name);
5     i ← 0 ;
6     prop ← hasArgument + i; while ind ← individual.getObjectProperty(prop) do
7         getArguments().add(ind);
```

List of Figures

List of Tables

List of Algorithms

Bibliography

Abdallah, Samer A.

2010 *The Ordered List Ontology 0.72*, English, http://smiy.sourceforge.net/olo/spec/orderedlistontology.html (visited on 09/29/2012). (Cited on pp. 34, 39.)

Apostoloff, N. and A. Zisserman

2007 "Who are you? - real-time person identification", in *Proceedings of the British Machine Vision Conference*, doi:10.5244/C.21.48, BMVA Press, pp. 48.1–48.10, ISBN: 1-901725-34-0. (Cited on p. 26.)

Arens, Michael

2004 *Repräsentation und Nutzung von Verhaltenswissen in der Bildfolgenauswertung*, PhD thesis, Universität Karlsruhe (TH), ISBN: 3-89838-287-7. (Cited on pp. 14, 45, 55.)

Baader, Franz, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider

2007 *The DescriptionLogic Handbook: Theory, Implementation and Applications*, ed. by Cambridge University Press, 2nd ed., Cambridge University Press, ISBN: 978-0521150118. (Cited on pp. 5, 8, 28.)

Baader, Franz, Jan Hladik, and Rafael Peòaloza

2008 "Automata can show PSpace results for description logics", *Inf. Comput.*, 206, 9-10 (Sept. 2008), pp. 1045–1056, ISSN: 0890-5401. (Cited on p. 8.)

Bao, Jie, Diego Calvanese, Bernardo Cuenca Grau, Martin Dzbor, Achille Fokoue, Christine Golbreich, Sandro Hawke, Ivan Herman, Rinke Hoekstra, Ian Horrocks, Elisa Kendall, Markus Krötzsch, Carsten Lutz, Deborah L. McGuinness, Boris Motik, Jeff Pan, Bijan Parsia, Peter F. Patel-Schneider, Sebastian Rudolph, Alan Ruttenberg, Uli Sattler, Michael Schneider, Mike Smith, Evan Wallace, Zhe Wu, and Antoine Zimmermann

2009 *OWL 2 Web Ontology Language*, English, W3C, http://www.w3.org/TR/owl2-overview/ (visited on 09/29/2012). (Cited on p. 11.)

Barwise, Jon and John Perry

1984 *Situations and Attitudes*, MIT Press, ISBN: 978-0262021897. (Cited on pp. 27, 43.)

Baumgartner, N. and W. Retschitzegger

2006 "A survey of upper ontologies for situation awareness", *Proc. of the 4th IASTED International Conference on Knowledge Sharing and Collaborative Engineering, St. Thomas, US VI*, pp. 1–9+. (Cited on p. 31.)

Baumgartner, Norbert, Werner Retschitzegger, and Wieland Schwinger

2008 "A Software Architecture for Ontology-Driven Situation Awareness", SAC 2008, pp. 2326–2330. (Cited on pp. 28–30.)

Baumgartner, Norbert, Wolfgang Gottesheim, Stefan Mitsch, Werner Retschitzegger, and Wieland Schwinger

2010 "BeAware! - Situation awareness, the ontology-driven way", *Data & Knowledge Engineering*, 69, 11, pp. 1181–1193. (Cited on pp. 30, 31, 43, 69.)

Bellotto, Nicola and Huosheng Hu

2010 "Computationally efficient solutions for tracking people with a mobile robot: an experimental evaluation of Bayesian filters", *Autonomous Robots*, 28 (42010), pp. 425–438, ISSN: 0929-5593. (Cited on p. 26.)

Bellotto, Nicola, Ben Benfolda, Hanno Harlandb, Hans-Hellmut Nagelb, Nicola Pirlob, Ian Reida, Eric Sommerladea, and Chuan Zhaoa

2012 "Cognitive Visual Tracking and Camera Control", *Computer Vision and Image Understanding*, 116, 3, pp. 457 –471, ISSN: 1077-3142. (Cited on pp. 25, 26.)

Bohlken, Wilfried and Bernd Neumann

2009 "Generation of Rules from Ontologies for High-Level Scene Interpretation", p. 15. (Cited on pp. 31, 32, 43, 69.)

Boult, T.E., R.J. Micheals, Xiang Gao, and M. Eckmann

2001 "Into the woods: visual surveillance of noncooperative and camouflaged targets in complex outdoor settings", *Proceedings of the IEEE*, 89, 10, pp. 1382 –1402, ISSN: 0018-9219. (Cited on p. 26.)

Brachman, Ronald J. and James G. Schmolze

1985 "An overview of the Kl-One Knowledge Representation System", *Cognitive Science*, 9, 2, pp. 171 –216, ISSN: 0364-0213. (Cited on p. 5.)

Brickley, Dan and Libby Miller

2010 *FOAF Vocabulary Specification 0.98*, http://xmlns.com/foaf/spec/ (visited on 09/29/2012). (Cited on p. 9.)

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal
1996 *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, ISBN: 978-0-471-95869-7. (Cited on p. 29.)

Devlin, Keith J.
1991 *Logic and information / Keith Devlin*, English, Cambridge University Press, Cambridge [England] ; New York, pp. I–XII, 1–307, ISBN: 0521410304. (Cited on p. 27.)
2006 "Situation theory and situation semantics", in *Logic and the Modalities in the Twentieth Century*, ed. by Dov M. Gabbay and John Woods, vol. 7, Handbook of the History of Logic, North-Holland, pp. 601 –664. (Cited on p. 27.)

Drummond, Nick, Alan Rector, Robert Stevens, Georgina Moulton, Matthew Horridge, Hai H. Wang, and Julian Seidenberg
2006 "Putting OWL in Order: Patterns for Sequences in OWL", English. (Cited on p. 39.)

Endsley, M. R.
2000 "Theoretical underpinnings of situation awareness: a critical review", in *Situation Awareness Analysis and Measurement*, ed. by M. R. Endsley and D. J. Garland, Lawrence Erlbaum Associates, Mahwah, NJ, USA. (Cited on pp. 27, 30.)

Everingham, Mark, Josef Sivic, and Andrew Zisserman
2006 "Hello! My name is... Buffy" – Automatic Naming of Characters in TV Video.", in *BMVC*, ed. by Mike J. Chantler, Robert B. Fisher, and Emanuele Trucco, British Machine Vision Association, pp. 899–908, ISBN: 1-904410-14-6. (Cited on p. 27.)

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides
1998 *Design Patterns: Elements of Reusable Object-Oriented Software*, ed. by Erich Gamma, Addison-Wesley Professional, Reading, Mass. [u.a.], ISBN: 0-201-63361-2. (Cited on pp. 47, 49.)

Gruber, Thomas R.
1993 "A Translation Approach to Portable Ontology Specifications", p. 27. (Cited on p. 9.)

Hall, D., J. Nascimento, P. Ribeiro, E. Andrade, P. Moreno, S. Pesnel, T. List, R. Emonet, R.B. Fisher, J.S. Victor, and J.L. Crowley
2005 "Comparison of target detection algorithms using adaptive background models", in *Visual Surveillance and Performance Evaluation of Tracking and Surveillance, 2005. 2nd Joint IEEE International Workshop on*, pp. 113 –120. (Cited on p. 26.)

Horridge, Matthew

2011 *A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools*, 1.3, The University Of Manchester, The University of Manchester Oxford Road Manchester M13 9PL. (Cited on p. 14.)

Horridge, Matthew and Sean Bechhofer

2009 "The Owl Api: A Java Api for Working with Owl 2 Ontologies", *OWLED 2009*, 6th OWL Experienced and Directions Workshop. (Cited on p. 13.)

Horridge, Matthew and Peter F. Patel-Schneider

2009 *OWL 2 Web Ontology Language Manchester Syntax*, http://www.w3.org/TR/2009/NOTE-owl2-manchester-syntax-20091027/ (visited on 09/29/2012). (Cited on p. 11.)

Horrocks, I., P. F. Patel-Schneider, and F. van Harmelen

2003 "From $\mathcal{SHIQ}$ and RDF to OWL: The Making of a Web Ontology Language", *Journal of Web Semantics*, 1, 1. (Cited on p. 11.)

Horrocks, Ian and Ulrike Sattler

2007 "A Tableau Decision Procedure for $\mathcal{SHOIQ}$", English, *Journal of Automated Reasoning*, 39 (32007), pp. 249–276, ISSN: 0168-7433. (Cited on p. 13.)

Horrocks, Ian, Oliver Kutz, and Ulrike Sattler

2006 "The Even More Irresistible $\mathcal{SROIQ}$.", in *KR*, ed. by Patrick Doherty, John Mylopoulos, and Christopher A. Welty, AAAI Press, pp. 57–67, ISBN: 978-1-57735-271-6. (Cited on p. 13.)

IJsselmuiden, Joris, Ann-Kristin Grosselfinger, David Münch, Michael Arens, and Rainer Stiefelhagen

2012 "Automatic Behavior Understanding in Crisis Response Control Rooms", in *Proc. of International Joint Conference on Ambient Intelligence*. (Cited on p. 20.)

Kokar, Mieczyslaw M., Christopher J. Matheus, and Kenneth Baclawski

2007 "Ontology-based situation awareness", *Information Fusion*, 10, 83 – 98, ISSN: 1566-2535. (Cited on pp. 10, 27, 28.)

Krüger, Wolfgang

1991 *Begriffsgraphen zur Situationsmodellierung in der Bildfolgenauswertung*, PhD thesis, Fakultät für Informatik der Universität Karlsruhe (TH). (Cited on p. 14.)

Laird, J.E. and R.E. Wray III

2010 "Cognitive Architecture Requirements for Achieving AGI", in *Proc. of the Third Conference on Artificial General Intelligence*, pp. 79–84. (Cited on p. 23.)

Liu, Ling and M. Tamer Özsu

2009 *Encyclopedia of Database Systems*, Springer US, ISBN: 978-0-387-35544-3, 978-0-387-39940-9. (Cited on p. 9.)

McGuinness, Deborah L. and Frank van Harmelen

2004 *OWL Web Ontology Language*, English, W3C, http://www.w3.org/TR/owl-features/ (visited on 09/29/2012). (Cited on pp. 1, 11, 13.)

Münch, David, Joris IJsselmuiden, Michael Arens, and Rainer Stiefelhagen

2011a "High-level Situation Recognition Using Fuzzy Metric Temporal Logic, Case Studies in Surveillance and Smart Environments", in *ICCV Workshops*, pp. 882–889. (Cited on p. 20.)

Münch, David, Kai Jüngling, and Michael Arens

2011b "Towards a Multi-purpose Monocular Vision-based High-Level Situation Awareness System", Anglais, in *International Workshop on Behaviour Analysis and Video Understanding (ICVS 2011)*, Sophia Antipolis, France, p. 10. (Cited on pp. 17, 38, 66.)

Münch, David, Joris IJsselmuiden, Ann-Kristin Grosselfinger, Michael Arens, and Rainer Stiefelhagen

2012a "Rule-Based High-Level Situation Recognition from Incomplete Tracking Data", in *Rules on the Web: Research and Applications*, ed. by Antonis Bikakis and Adrian Giurca, vol. 7438, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, pp. 317–324, ISBN: 978-3-642-32688-2. (Cited on p. 45.)

Münch, David, Eckart Michaelsen, and Michael Arens

2012b "Supporting Fuzzy Metric Temporal Logic Based Situation Recognition by Mean Shift Clustering", in *KI 2012: Advances in Artificial Intelligence*, ed. by Birte Glimm and Antonio Krueger, vol. 7526, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, pp. 233–236, ISBN: 978-3-642-33346-0. (Cited on p. 45.)

Münch, David, Stefan Becker, Wolfgang Hübner, and Michael Arens

2012c "Towards a Real-time Situational Awareness System for Surveillance Applications in Unconstrained Environments", in *Proc. of the 7th Future Security Research Conference, Bonn, Germany*. (Cited on p. 17.)

Nagel, H.-H.

2000 "Image sequence evaluation: 30 years and still going strong", in *Pattern Recognition, 2000. Proceedings. 15th International Conference on*, vol. 1, 149 –158 vol.1. (Cited on p. 20.)

Nagel, Hans-Hellmut

2004 "Steps toward a cognitive vision system", *AI Mag.*, 25, 2 (June 2004), pp. 31–50, ISSN: 0738-4602. (Cited on p. 20.)

Randell, David A., Zhan Cui, and Anthony G. Cohn

1992 "A spatial logic based on regions and connection", in *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, Morgan Kaufmann, pp. 165–176. (Cited on p. 31.)

Schäfer, Karl H.

1996 *Unscharfe zeitlogische Modellierung von Situationen und Handlungen in Bildfolgenauswertung und Robotik*, German, vol. 135, Dissertationen zur Künstlichen Intelligenz (DISKI), infix, Sankt Augustin, ISBN: 3-89601-135-9. (Cited on pp. 18, 43.)

Schmidt-Schauß, Manfred and Gert Smolka

1991 "Attributive concept descriptions with complements", *Artificial Intelligence*, 48, 1, pp. 1–26. (Cited on pp. 6, 8.)

Sharman, Raj, Rajiv Kishore, and Ram Ramesh

2007 *Ontologies: a handbook of principles, concepts and applications in information systems*, 9th ed., Integrated series in information systems ; 14, Springer, New York, NY, ISBN: 0-387-37019-6. (Cited on p. 9.)

Tobies, Stephan

2001 "Complexity Results and Practical Algorithms for Logics in Knowledge Representation", *CoRR*, cs.LO/0106031. (Cited on p. 13.)