# Semantic Web and Formal Design Methods

**WANG HAI**

*(B.Sc.(Hons). NUS )*

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2004

# Acknowledgement

I am deeply indebted to my supervisor, Dr. DONG Jin Song, for his guidance, insight and encouragement throughout the course of my doctoral program and for his careful reading of and constructive criticisms and suggestions on drafts of this thesis and other works.

I owe thanks to Dr. SUN Jing, other office-mates and friends for their help, discussions and friendship.

I would like to thank the numerous anonymous referees who have reviewed parts of this work prior to publication in journals and conference proceedings and whose valuable comments have contributed to the clarification of many of the ideas presented in this thesis. I would also like to thank Hugh Anderson for his helpful comments on the draft of the thesis.

This study received funding from the project "Integrated Formal Methods" supported by National University of Singapore Academic Research Fund and the project "Defence Innovative Research Project (DIRP) – Formal Design Methods and DAML" supported by Defence Science & Technology Agency (DSTA) of Singapore. The School of Computing also provided the finance for me to present papers in several conferences overseas. In addition, I have been encouraged by receiving the Dean's Graduate Award 2003. For all this, I am very grateful.

Finally, I sincerely thank my parents WANG Ling Zhang and ZHANG Jine for their love, encouragement and financial support in my years of study.

# Contents

# List of Figures

# List of Tables

# Summary

Semantic Web (SW), commonly regarded as the next generation of the Web, is an emerging area from the Knowledge Representation and the Web Communities. The Formal Methods (FM) community can also play an important role to contribute to SW development. For example, formal methods and tools can be used to facilitate the reasoning and consistency checking tasks for Semantic Web ontologies and services. Semantic Web ontologies can even be generated automatically from formal requirement models. It is hoped that SW will be a new novel application domain for formal methods. On the other hand, the diversity of various formal specification techniques and the need for their effective combinations require an extensible and integrated supporting environment. The success of the Semantic Web may have profound impact on the Web environment for formal methods, especially for extending and integrating different formalisms. This thesis demonstrates the latest investigations on the links between Semantic Web and Formal Methods. First, a Semantic Web (RDF/DAML+OIL) environment for supporting, extending and integrating many different formalisms was built. Such a *meta integrator* may bring together the strengths of various formal methods communities in a flexible and widely accessible fashion. The Semantic Web environment for formal specifications may lead to many benefits. One novel application which has been demonstrated in this thesis is the notion of specification comprehension based RDF query techniques. Since the SW builds on

the success of XML, as the preliminary work this thesis also presents the development of an XML based Web browsing environment for Z family notations. On the other hand, to apply formal methods to SW, formal methods and tools can be used to facilitate the reasoning and consistency checking tasks for semantic web ontologies. The semantics of the SW languages has been encoded into a formal language (in particular Alloy), so that Alloy can be used to provide automatic reasoning and consistency checking services for SW. At the same time, formal methods have been used to assist design Semantic Web service application and the translation rules and tools have been developed to extract the SW ontology and semantic markup for Web service from the formal model automatically. In summary, we believe that there is a close association between formal specification and Semantic Web, and the two can benefit from each other in many ways.

# Chapter 1

# Introduction and Overview

## 1.1 Motivation and goals

Most discussions related to “Web and Software Engineering” are centered around two main issues: how software engineering techniques facilitate Web applications and how Web technology assists software design and development. This thesis tries to address both issues within a specific context “Semantic Web (SW) [3] and formal software modelling techniques”.

In recent years, researchers have begun to explore the potential of associating Web content with explicit meaning so that the Web content becomes more machine-readable and intelligent agents can retrieve and manipulate pertinent information readily. The Semantic Web proposed by W3C is one of the most promising and accepted approaches. It has been regarded as the next generation of the Web. SW not only emerges from the Knowledge Representation and the Web Communities, but also brings the two communities closer together. We believe that there is also a close association between formal specification and Semantic Web. The Semantic Web has good support for automation, collaboration, extension and integration. However it is less expressive and there is no systematic design process for Web ontology and no mature reasoning tool support. On the other hand, Formal Specifications are expressive, diverse, can be combined effectively and have some mature tool supports. However, it is hard to link various methods for collaborative design. The two communities can benefit from each other in many ways.

This thesis will demonstrate the latest investigations on the links between Semantic Web and Formal Methods. First, the success of the Semantic Web may have profound impact on the Web environment for formal methods, especially for extending and integrating different formalisms. At the same time, there is a role for software engineering techniques and tools to play and make important contributions to the SW development.

Many formal languages, like Z, are closely related to data modelling. Many researchers investigated Z with database schemas [95, 14]. For example the Z schema calculus is extended to model the familiar relational algebra operations [57]. Besides of database, linking formal methods with SW is another novel and important research area for formal methods researchers.

### 1.1.1 Semantic Web for Formal Methods

Many formal specification techniques exist for modelling different aspects of software systems and it is difficult to find a single notation that can model all functionalities of a complex system clearly and precisely [68, 99]. For instance, B/VDM/Z are designed for modelling system data and states, while CSP/CCS/$\pi$-calculus are designed for modelling system behaviors and interactions. Various formal notations are often extended and combined for modelling large and complex systems. In recent years, *Formal Methods Integration* has been a popular research topic [2, 33, 10]. In the context of combining state-based and event-based formalisms, a number of proposals have been presented [9, 29, 31, 55, 81, 87, 89, 97, 69]. Our general observations on these works are that

> Various formal notations can be used in an effective combination if the semantic links between those notations can be clearly established. The semantic/syntax integration of those languages would be a consequence when the semantic links are precisely defined. Due to different motivations, there are possible different semantic links between two formalisms,

which lead to different integrations between the two.

Unlike UML [72], an industrial effort for standardizing diagrammatic notations, a single dominating integrated formal method may not exist in the near future. The reason may be partially due to the fact that there are many different well established individual schools, e.g., VDM forum, Z/B users, CSP group, CCS/$\pi$-calculus family etc. Another reason may be due to the open nature of the research community, i.e. FME (www.fmeurope.org), which is different from the industrial 'globalization' community, i.e. OMG (www.omg.org).

Regardless of whether there will be or there should be an ultimate integrated formal method (like UML), *diversity* seems to be the current reality for formal methods and their integrations. Such diversity may have an advantage, that is, different formal methods and their combinations may be effective for developing various kinds of complex systems[1]. The best way to support and popularize formal methods and their effective combinations is to build a widely accessible, extensible and integrated environment.

The World Wide Web provides an important infrastructure for a promising environment for various formal specification and design activities because it allows sharing

[1]In fact, one of the difficult tasks of OMG is to resist many good new proposals for extending UML — a clear consequence and drawback of pushing a single language for modelling all software systems.

of various design models and provides hyper textual links among the models. The success of the Semantic Web may have profound impact on the Web environment for formal specifications. Under this Meta integrating and intelligent Web environment, formalist can work in co-operation easily. Many formal tasks like model reusing and model refining can be achieved automatically or semi-automatically. This thesis only demonstrates an approach on how to build a Semantic Web environment for supporting, checking, extending and integrating various formal specification languages. Furthermore, based on this Semantic Web environment, specification comprehension (queries for review/understanding purpose) can be supported. Since the SW builds on the success of XML, as the preliminary work this thesis also demonstrates how the traditional Web techniques like XML can assist formal specification and design process. We present the development of a Web browsing environment for Z family notations.

### 1.1.2 Formal Methods for Semantic Web

After decades of research and development, some mature formal tools have been established successfully. This thesis addresses how the existing formal tools can be used to reasoning about the SW ontology.

From a different angle, the development of Semantic Web systems requires precise modelling techniques to capture ontology domain properties and application functionalities. However, the Semantic Web language itself is too low level to be used for systematically capturing ontology requirement and it is also not expressive enough for designing Semantic Web service/agents. The TCOZ notation [55] is an extension to Z, as a formal specification language based on set theory and predicate calculus. We believe that TCOZ as a specification technique can contribute to the Semantic Web-based system development in many ways. We demonstrate that TCOZ can capture various requirements of SW services including ontology and service functionalities. We also develop systematic translation rules and tools which can project TCOZ models to DAML+OIL ontology and DAML-S automatically.

## 1.2 Thesis outline and overview

The structure of the thesis is as follows:

### 1.2.1 Chapter 2

This chapter is devoted to an overview of the Semantic Web and some formal notations involved in this thesis. Following the success of eXtensible Markup Language (XML) [92], W3C's primary focus is on Semantic Web. Currently, one of the major Semantic Web activities at W3C is the work on Resource Description Framework (RDF) [47], which provides interoperability between applications that exchange machine-understandable information on the Web. RDF Schema [7] and DARPA Agent Markup Language (DAML) [91] provide the basic vocabulary to describe RDF vocabularies. They can be used to define properties and types of the Web resources. A fundamental component of the Semantic Web will be the markup of Web Services to make them computer-interpretable, use-apparent, and agent-ready. DAML-S [12] is a DAML+OIL ontology for Web service developed by a coalition[2].

Many formal specification techniques exist for modelling different aspects of software systems. The formal specification notations involved in this thesis include the Z notation [82], the Object-Z [24, 80], CSP [38], Alloy [44] and the TCOZ [55] etc. Z and CSP are two well known formal notations with their respective user groups. Recently there has been active investigation of the integration [29, 55, 81] of formal object-oriented methods (e.g. Object-Z) with process description languages (e.g. CSP). One such approach, the Timed Communicating Object Z (TCOZ) combines Object-Z's strengths in modelling complex data and state with TCSP's strengths in modelling real-time concurrency. Alloy [44] is a structural modelling language based on first-order logic, for expressing complex structural constraints and behavior. In this chapter we give a brief overview of these formal notations.

[2]DAML Service Coalition: A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, H. Zeng.

### 1.2.2 Chapter 3

This chapter presents the development of a Web browsing environment for Z family notations – ZML. The World Wide Web (WWW) is a promising environment for software specification and design because it allows sharing design models and providing hyper textual links among the models [46]. It is important to develop links and tools from FM to WWW so that FM technology transfer can be successful. In this chapter, we demonstrate the use of the eXtensible Stylesheet Language (XSL) [93] to develop a Web environment that provides various browsing and syntax checking facilities for Z family languages.

The achievement presented in this chapter does not use the Semantic Web related techniques. This work was done during the early stage of the PhD program. It was the first attempt to investigate how the Web technology assists a formal design and development process. ZML provides a nice environment for browsing the Z families formal models on the Web. However under this environment, it is difficult to extend and integrate the formalisms. In fact, this motivates us to investigate how the SW can be used to build a flexible environment for different formalisms (The details about this flexible environment will be presented in Chapter 4).

### 1.2.3 Chapter 4

The best way to support and popularize formal methods and their effective combinations is to build a widely accessible, extensible and integrated environment. In this chapter we first use Z [98] and CSP [38] as examples to demonstrate how a Semantic Web environment for formal specification languages can be developed. After that we show these environments can be further extended and integrated easily. Furthermore we illustrate how specification comprehension can be supported by RDF queries.

### 1.2.4 Chapter 5

This chapter presents the development of a reasoning environment for SW ontology using formal techniques and tools, in particular, Alloy. In the development of Semantic Web there is a pivotal role for ontology, since it provides a representation of a shared conceptualization of a particular domain that can be communicated between people and applications. Reasoning can be useful at many stages during the design, maintenance and deployment of ontology. Because autonomous software agents may perform their reasoning and come to conclusions without human supervision, it is essential that the shared ontology is consistent. However, since the Semantic Web technology is still in the early stage, the reasoning and consistency checking tools are primitive.

The software modelling language Alloy [44] is suitable for specifying structural properties of software. Alloy is a first order declarative language based on relations. We believe SW is a new novel application domain for Alloy because relationships between Web resources are the focus points in SW. Furthermore, Alloy specifications can be analyzed automatically using the Alloy Analyzer (AA) [45]. Given a finite scope for a specification, AA translates it into a propositional formula and uses SAT solving technology to generate instances that satisfy the properties expressed in the specification. This chapter presents a Alloy semantics for the SW languages and shows how Alloy can be used to provide automatic reasoning and consistency checking services for SW. Various reasoning tasks can be supported effectively by AA.

### 1.2.5 Chapter 6

This chapter tries to demonstrate that the formal technology can be used to assist in the design of Semantic Web service applications. Complex Semantic Web (SW) services may have intricate data state, autonomous process behavior and concurrent interactions. The design of such SW service systems requires precise and powerful modelling techniques to capture not only the ontology domain properties but also the services' process behavior and functionalities. On the other hand, the formal method is the use of notations and languages with a defined mathematical meaning to enable specifications (that is statements of what the proposed system should do) to be expressed with precision and no ambiguity. We illustrate how TCOZ can be used as high level design language to design SW services. Furthermore, the chapter presents the development of the systematic translation rules and tools which can automatically extract the SW ontology and services semantic markup from the formal TCOZ design model. The online talk discovery system is used as a demonstrating case study.

### 1.2.6 Chapter 7

Chapter 7 concludes the thesis with a summary of the main contributions of this thesis, and some suggestions for further research.

### 1.2.7 Thesis's theme and relations between the main chapters

This thesis centers on one theme – the linkage between the Semantic Web and Formal Methods. This linkage can be illustrated in two directions: how FM techniques facilitate SW applications and how SW assists FM. Each of the main chapters in this thesis demonstrates that these two techniques can assist each other. Since each main chapter demonstrates a different aspect of the main theme and has been published as a full paper, the correlation between chapters may not be evident to the reader at first glance. This section provides a detailed explanation on how the different chapters are related.

This thesis includes four main chapters from Chapter 3 to Chapter 6. Chapter 3, Z family Markup Language – ZML, presents a nice interchange format for the formal notations. This chapter serves two purposes. Firstly, it builds the foundation for the subsequent chapters. All the tools developed in this thesis will use ZML as the underlying encoding format. Secondly, as one such a way on FM contributing SW, our research group are proposing to use the formal language Z as a Semantic Web language (on top of the Semantic Web ontology layer). The following reasons make the Z as a good candidate to be used as SW language:

- As a prestigious formal method, Z has a wide user group.
- After more than twenty years of development, many relatively mature supporting tools have been setup.
- Z is very expressive.

However according to W3C's requirement, to use a language as a SW language, it must have the XML syntax. Therefor ZML will be the first important step to achieve our goal.

The Chapter 4: Semantic Web for Extending and Linking Formalisms, demonstrates how one aspect the Semantic Web can assist the formal methods, which is how Semantic Web techniques can assist integrating the formalisms. The Semantic Web can contribute to formal methods in many other areas, like formal model reuse, model refinement, etc.. There will be some other PhD theses from the research group giving more details.

Chapter 5 and Chapter 6 show that formal techniques can also contribute to Semantic Web. Chapter 5 presents how the formal tools can be used to check and reason over a Semantic Web ontology. This assumes that the Semantic Web ontology has already been built up, e.g., extracted from a natural language document using NLP techniques or merged from two different existing ontologies. One natural question people may ask is what happens if the Semantic Web ontology has not been developed yet? Can formal techniques assist the process of the Semantic Web ontology and system design and developing? If we have a formal model, can we get the ontology easily? All those questions will be answered in Chapter 6: TCOZ Approach to Semantic Web Service Design. In this chapter, we demonstrate that a integrated formal method – TCOZ is very suitable to be used as a high level design language for the Semantic Web service system. Moreover, not only the ontology but also the semantic markup information for the SW service can be automatically extracted from the TCOZ formal design model by the tool we developed.

## 1.3 Related works

To our knowledge, we are the first research group working on the linkage between Semantic Web and Formal Methods. There is no much related works being done before. One of the early work by Bicarregui and Matthews [4] has proposed ideas to integrate SGML (earlier version of XML) and EXPRESS for documenting control systems design. Z notation on the web based on HTML and Java applets has been investigated by Bowen and Chippington [5] and Cinancarini, Mascolo and Vitali [11]. HTML has been successful in presenting information on the Internet, however the lack of content information has made the retrieval and exchange of resource more difficult to perform, and different formalism hard to be extended and integrated.

## 1.4 Publications

Most chapters of the thesis have been accepted in international refereed journals or conference proceedings.

Chapter 3 has been published in the thirteenth volume of the *Annals of Software Engineering journal (ASE, June 2002)* [86]. Chapter 4 was presented at *The Eleventh International Formal Methods Europe Symposium (FME'02, July 2002, Copenhagen)* [20]. Chapter 5 has been presented at *The Twelfth International Formal Methods Europe*

*Symposium (FM'03, Sep 2003, Pisa)* [22]. The technique/tool presented in Chapter 5 was successfully applied to a military case study and was presented at *The 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03, July 2003, San Francisco)* [23]. Chapter 6 has been presented at *The 4th International Conference on Formal Engineering Method (ICFEM'02, Nov 2002, Shanghai)* [21].

I also made partial contributions to other publications [17, 18, 84, 85] which are although related to this thesis, they can be considered as side-stories or pre-thesis/follow-up work.

# Chapter 2

# Background

This chapter reviews the vision of Semantic Web and some supporting techniques, and then reviews the related formal notations and tools.

## 2.1 Semantic Web overview

As a huge information space, the Web should be useful not only for human-human communication, but also allows machines to participate and help. However, nowadays most information on the Web is designed for human consumption and the structure of the data is not evident for a robot browsing the Web. There are two distinct approaches to enable the machine to automatically manipulate the information in the Web. One approach which comes from artificial intelligence is machine learning. The machine is trained to behave like a person. However this approach is domain-dependent and requires a huge training process. The Semantic Web [3] approach instead develops language for expressing information in a machine processable form. The W3C gives the following definition for the Semantic Web:

> The Semantic Web is an extension of the current Web in which information is given a well-defined meaning, better enabling computers and people to work in cooperation.

SW is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners. With the SW, the machine can do many complicate tasks which currently can only be performed manually. For example, user can directly send the following request to web agent –"Book me a holiday next weekend somewhere warm, not too far away, and where they speak Chinese or English.". The

Web agent will be able to 'understand' the request and perform it for the users.

A series of technologies has been proposed to realize the vision of the Semantic Web as the next generation Web. It extends the current Web by giving the Web content a well-defined meaning and representing the information in a machine-understandable form. HTML, the current Web data standard, is aimed at delivering information to the end user for human-consumption (e.g. display this document). XML is aimed at delivering data to systems that can understand and interpret the information. XML is focused on the syntax (defined by the XML schema or DTD) of a document and it provides essentially a mechanism to declare and use simple data structures. However there is no way for a program to actually understand the knowledge contained in the XML documents.

Resource Description Framework (RDF) [47] is a foundation for processing metadata; it provides interoperability between applications that exchange machine-understandable information on the Web. RDF uses XML to exchange descriptions of Web resources and emphasizes facilities to enable automated processing. The RDF descriptions provide a simple ontology system to support the exchange of knowledge and semantic information on the Web. RDF Schema [7] provides the basic vocabulary to describe RDF documents. RDF Schema can be used to define properties and types of the Web resources. In a similar fashion to XML Schema which gives specific constraints on the structure of an XML document, RDF Schema provides information about the interpretation of the RDF statements. The DARPA Agent Markup Language (DAML) [91] is an AI-inspired description logic-based language for describing taxonomic information. DAML currently combines Ontology Inference Layer (OIL) [8] and features from other ontology systems. It is now called DAML+OIL and contains richer modelling primitives than RDF schema. The DAML+OIL language builds on top of XML and RDF(S) to provide a language with both a well-defined semantics and a set of language constructs including classes, subclasses and properties with domains and ranges, for describing a Web domain. DAML+OIL can further express restrictions on membership in classes and restrictions on certain domains and ranges values. Semantic Web is highly distributed, and different parties may have different understandings of the same concept. Ideally, the program must have a way to discover the common meanings from the different understandings. It is central to one important concept in Semantic Web system – ontology. The ontology for a Semantic Web system is a document or a file that formally defines the relations among terms. The most typical kind of ontology for the Web has taxonomy and a set of inference rules. Ontology can enhance the functioning of the Web in many ways. RDFS and DAML+OIL supply the language to define the ontology. For example, the following DAML+OIL code specifies a concept 'talk' which hold in a certain place. A 'talk' (a DAML+OIL class) has a property 'talk_place', having only one value 'place' (also a DAML+OIL class).

```
<daml:class rdf:ID="talk">
```

| **DAML+OIL constructs** | **Description** |
|---|---|
| *DAML_class* | classes |
| *DAML_property* | properties |
| *DAML_subclass*[*C*] | subclasses of C |
| *DAML_subproperty*[*P*] | subproperties of P |
| *instanceof*[*C*] | instances of the DAML+OIL class C |

Table 2.1: DAML+OIL constructs (partial)

```
  <rdfs:label>Talk</rdfs:label>
</daml:class>
<daml:class rdf:ID="place">
  <rdfs:label>Place</rdfs:label>
</daml:class>
<daml:ObjectProperty rdf:ID="talk_place">
  <rdf:type rdf:resource=
       "http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdf:domain rdf:resource="#talk"/>
  <rdf:range rdf:resource="#place"/>
</daml:ObjectProperty>
```

We summarize some essential DAML+OIL constructs in Table 2.1.

## 2.2 Semantic markup for Web service: DAML-S

A fundamental component of the Semantic Web will be the markup of Web Services to make them computer-interpretable, use-apparent, and agent-ready. DAML-S [12] is a DAML+OIL ontology for Web services developed by a coalition[1]. DAML-S was

[1]DAML Service Coalition: A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, H. Zeng.

expected to enable the following tasks automatically:

- Web service discovery,
- Web service invocation,
- Web service composition and interoperation,
- Web service execution monitoring.

DAML-S consists of three main parts: the profile, the process model and the grounding. The DAML-S `profile` describes `what the service does`. Thus, the class SERVICE `presents` a `SERVICEPROFILE`. The service `profile` is the primary construct by which a service is advertised, discovered and selected. The DAML-S `process` model tells `how the service works`. Thus, the class SERVICE is `describedBy` a `SERVICEMODEL`. It includes information about the service inputs, outputs, preconditions and effects. It also shows the component processes for a complex process and how the control flows between the components. The DAML-S `grounding` tells `how the service is used`. It specifies how an agent can access a service.

SW services(DAML-S) may have intricate data state, complex process behavior and concurrent interactions. The design of such SW service systems requires precise and powerful modelling techniques to capture not only the ontology domain properties but also the services' process behavior and functionalities. It is desired to have a powerful formal notation to precisely design the Web system. TCOZ is such a good candidate. In this thesis, we focus on the connection between the TCOZ model and the DAML-S process model (Chapter 6).

### 2.2.1 DAML-S process

The DAML-S process model is intended to provide a basis for specifying the behavior of a wide array of services. It is influenced by the work in AI on standardizations of planning languages [26], work in programming languages and distributed systems [61, 62], emerging standards in process modelling and workflow technology such as the NIST's Process Specification Language (PSL) [74] and the Workflow Management Coalition effort (http://www.aiim.org/wfmc), work on modelling verb semantics and event structure [64], work in AI on modelling complex actions [49], and work in agent communication languages [59, 28].

There are two chief components of a DAML-S process model – the process, and process control model. The process describes a Web Service in terms of its input, output, precondition, effects and, where appropriate, its component subprocess. The process model enables planning, composition and agent/service inter-operation. The process control model – which describes the control flow of a composite process and shows which of various inputs of the composite process are accepted by which of its subprocesses – allows agents to monitor the execution of a service request. The con-

| **Constructs** | **Description** |
|---|---|
| *damls_Process* | Describes service which includes atomic, composite and simple process |
| *damls_input* | Specifies one of the inputs of the service. |
| *damls_output* | Specifies one of the outputs of the service. |
| *damls_precondition* | Specifies one of the preconditions of the service. |
| *damls_effect* | Specifies one of the effects of the service. |
| *damls_AtomicProcess* | Process which is directly invocable, have no subprocess and execute in a single step. |
| *damls_CompositeProcess* | Process which is composed from other process. |
| *damls_SimpleProcess* | Process which is used as elements of abstraction. |
| *damls_Sequence*$[P_1, P_2, ...]$ | Executes a list of processes in order |
| *damls_Split*$[P_1, P_2, ...]$ | Execute a bag of processes concurrently |

Table 2.2: DAML-S constructs (partial)

structs to specify the control flow within a process model include `Sequence`, `Split`, `Split+Join`, `If-Then-Else`, `Repeat-While` and `Repeat-Until`. We will use the following table (Table 2.2) to summarize some of the constructs available in DAML-S process.

## 2.3 Description Logic

From a formal point of view, SW ontology language DAML+OIL can be seen to be equivalent to a very expressive Description Logic (DL) [39]. Before discussing the technical issues like automatically generating and reasoning about DAML+OIL ontology in the later chapters, this section presents an introduction to DL.

### 2.3.1 DL history

The Description Logic (DL) [40] is an important powerful class of logic-based knowledge representation languages. The DL is used to represent and to reason about terminological knowledge and it was evolved from two knowledge representation formalisms Frames and Semantic Networks. Frames developed by Minsky [63] are record-like data structures for representing stereotyped situations and objects. Attached to each frame is all the information necessary for treating a situation, which may include information about how to use the frame, information about what one can expect to happen next and information about what to do if these expectations are not confirmed and etc. Semantic Network, develop after the work of Quillian (1967) [71], is a graph-based representation formalism to capture the semantics of nature language. The common problem of both Frames and Semantic Networks is the lack of formal semantics. This may lead to the result that every system behaved differently from the others. In response to this problem, the researchers tried to develop knowledge representation languages equipped with a formal semantics to precisely capture its meaning independently of the underlying inference machine.

### 2.3.2 Knowledge representation in DL

A DL-system consists of two components. The first component, known as the knowledge base, provides a precise characterization of the type of the knowledge to be specified to the system. The second is the reasoning engine, which provides various inference services. The knowledge base in DL can further be divided into the TBox and the ABox.

#### TBox

A TBox stores the conceptual knowledge of an application domain. It defines the intentional knowledge in the form of a terminology (reason for the term "TBox"). The terminology consists of *concepts*, which denote sets of individuals, and *roles*, which denote binary relations between individuals. The DL systems can build atomic concepts and roles (concept and role names) and can also build complex descriptions of concepts and roles. The different DL systems are distinguished by their description language used for building complex concepts and roles. For example, $\mathcal{AL} - language$, introduced in [75] as a minimal language that is of practical interest, has the following syntax rule:

$$\begin{aligned} C, D \rightarrow\ & A \mid (atomicconcept) \\ & \top \mid (universalconcept) \\ & \bot \mid (bottomconcept) \\ & \neg A (atomicnegation) \\ & C \sqcap D (intersection) \\ & \forall\, R.C (valuerestriction) \\ & \exists\, R.\top (limitedexistentialquantification) \end{aligned}$$

The DL language $\mathcal{FL}^-$ is a sublanguage of $\mathcal{AL}$ by disallowing atomic negation. $\mathcal{FL}_0$ is a sublanguage of $\mathcal{FL}^-$ by disallowing existential quantification. The $\mathcal{AL}$ can be extended to $\mathcal{ALU}$, $\mathcal{AL}\varepsilon$, $\mathcal{ALN}$ and $\mathcal{ALC}$ if the union of concepts, full existential qualification, number restriction and negation of arbitrary concepts is allowed accordingly. Allowing more concept constructs makes a DL language more expressive, but more difficult and complex to reason about. TBoxes allow introducing names for concept descriptions.

### ABox

An ABox contains extensional knowledge about the domain of interest. It introduces the assertional knowledge (reason for the term "ABox") (world description). Whereas TBoxes restrict the set of possible words, ABoxes allow us to describe a specific state of the world by introducing individuals (or instances) together with their properties. In the Abox, knowledge can be divided into a concept assertion, which states an individual is a member of concept (in the form `C(a)`), and a role assertion with a pair of individuals (in the form `R(a, b)`). When we say an ABox `A` is defined with respect to a Tbox, the concept description in `A` may contain defined names of TBox.

### 2.3.3 Description Logic and FOL

The basic DL is considered as a fragment of first-order logic. We use $L^k$ to denote first order predicate logic over unary and binary predicates with at most $k$ variables and we use $C^k$ to denote first order predicate logic over unary and binary predicates with at most $k$ variables and counting quantifiers $\exists^{\geqslant n}$,$\exists^{\leqslant n}$. The basic DL concepts can be translated into $L^2$ formulae or $C^2$ if the number restriction is allowed. $L^2$ and $C^2$ are known to be decidable and NExpTime-complete, so the basic DL is decidable and NExpTime-complete. Both $L^2$ and $C^2$ are far more expressive than basic DL.

Different DL languages can be extended from the basic DL language. Some of the extension can be as expressive as $L^2$ and some can be as expressive as $L^3$. For the latter case, the DL language becomes undecidable.

Besides increasing the number of variables in the predicates, a certain extension of the DL makes it go beyond first order logic, e.g. including transitive closure of roles.

On the whole, the DL can be considered as a subset of FOL. The reason FOL is not directly used to represent knowledge without additional restrictions is that:

- the expressive power is too high for obtaining decidable and efficient inference problems;
- the inference power may be too low for expressing interesting, but still decidable theories.

## 2.4 Spectrum of formalisms

In this section, we will use a simple stack system to give a brief introduction to the Z, Object-Z, TCSP and TCOZ notations etc.

### 2.4.1 Z

Z notation [82] is a state-oriented formal specification language based on set theory and predicate logic. A Z specification typically includes a number of state and operation schema definitions. A state schema encapsulates variable declarations and related predicates (invariants). The system state is determined by values taken by variables subject to restrictions imposed by state invariants. An operation schema defines the relationship between the 'before' and 'after' states corresponding to one or more state schemas. Complex schema definitions can be composed from the simple ones by schema calculus. Z has been widely adopted to specify a range of software systems (see [34]). Various tools, i.e. editors, type/proof checkers and animators, for Z have been developed.

Consider the Z model of a stack. Let the given type *Item* represent a set of items. The notation for this is:

$[Item]$ [item type]

The stack contains operations to pop items off and push items onto the stack. The total items in the stack cannot be more than *max* (say, a number larger than 100). The global constant *max* can be defined using the Z axiomatic definition as:

$$\begin{array}{|l}
max : \mathbb{N} \\
\hline
max > 100
\end{array}$$

The state, potential state change and initial state of the stack system can be specified in Z as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, Stack \,\_\_\_\_\_\_\_\_\_\_\_\_} \\
items : \operatorname{seq} Item \\
\hline
\#items \leq max \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\_\, StackInit \,\_\_\_\_\_\_\_\_\_\_} \\
Stack \\
\hline
items = \langle\ \rangle \\
\hline
\end{array}$$

The operations to push items on, and pop items off of the stack can be modelled as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, Push \,\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta Stack \\
item? : Item \\
\hline
items' = \langle item? \rangle \frown items \\
\#items < max \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\_\, Pop \,\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta Stack \\
item! : Item \\
\hline
items \neq \langle\ \rangle \\
items = \langle item! \rangle \frown items' \\
\hline
\end{array}$$

More complex operations can be constructed by using schema calculus, e.g., a new item which is pushed on and then popped off, say *Transit*, can be specified by using the sequential composition schema operator $\fatsemi$ as:

$$Transit \mathrel{\widehat=} Push \fatsemi Pop$$

which is an (atomic) operation with the effect of a *Push* followed by a *Pop*. Other forms of schema calculus include schema conjunction '$\land$', disjunction '$\lor$' implication '$\Rightarrow$', negation '$\neg$' and pipe '$\gg$', which have been discussed in many Z text books [82, 98]. Appendix A presents a glossary of the Z notation.

## 2.4.2 Object-Z

Object-Z [24] is an extension of the Z formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring. Object-Z has a type checker, but other tool support for Object-Z is limited in comparison to Z. The essential extension to Z in Object-Z is the *class* construct which groups the definition of a state schema with the definitions of its associated operations. A class is a template for *objects* of that class: for each such object, its states are instances of the state schema of the class and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class.

Consider the following specification of the *Stack* system in Object-Z:

*Stack*

$items : \mathrm{seq}\, Item$

$\# items \leq max$

INIT

$items = \langle\ \rangle$

*Push*

$\Delta(items)$

$item? : Item$

$items' = \langle item? \rangle ^\frown items$

*Pop*

$\Delta(items)$

$item! : Item$

$items \neq \langle\ \rangle$

$items = \langle item! \rangle ^\frown items'$

Operation schemas have a Δ-list of those attributes whose values may change. By convention, no Δ-list means no attribute changes value. The standard behavioral interpretation of Object-Z objects is as transition systems [79]. A behavior of a transition system consists of a series of state transitions each effected by one of the class operations. A *Stack* object starts with *items* empty then evolves by successively performing either *Push* or *Pop* operations. Operations in Object-Z are atomic, only one may occur at each transition, and there is no notion of time or duration. It is difficult to use the standard Object-Z semantics to model a system composed by multi-threaded component objects whose operations have duration.

Every operation schema implicitly includes the state schema in un-primed form (the state before the operation) and primed form (the state after the operation). Hence the class invariant holds at all times: in each possible initial state and before and after each operation.

In this example, operation *Push* pushes a given input *item*? to the existing set provided the sequence has not already reached its maximum size (an identifier ending in '?' denotes an input). Operation *Pop* outputs a value *item*! defined as one element of *items* and reduces *items* by deleting the first one from the original stack (an identifier ending in '!' denotes an output). Appendix B presents the concrete syntax of Object-Z.

### 2.4.3 TCSP

Timed CSP (TCSP) [76] extends the well-known CSP (Communicating Sequential Processes) notation of Hoare [38] with timing primitives. As indicated by its name, CSP is an *event* based notation primarily aimed at describing the sequencing of behavior within a process and the synchronization of behavior (or *communication*) between processes. Timed CSP extends CSP by introducing a capability to consider temporal aspects of sequencing and synchronization.

CSP adopts a symmetric view of process and environment. Events represent a cooperative synchronization between process and environment. Both process and environment may control the behavior of the other by *enabling* or *refusing* certain events or sequences of events.

**Process Primitives**

The primary building blocks for Timed CSP processes are sequencing, parallel composition, and choice.

A process which may participate in event $a$ then act according to process description $P$ is written

$$a@t \rightarrow P(t).$$

The event $a$ is initially enabled by the process and occurs as soon as it is also enabled by its environment. The event $a$ is sometimes referred to as the *guard* of the process. The (optional) timing parameter, $t$, records the time (relative to the start of the process) at which the event $a$ occurs and allows the subsequent behavior, $P$, to depend on its value.

The second form of sequencing is process sequencing. A distinguished event ✓ is used to represent and detect process termination. The sequential composition of $P$ and $Q$, written $P;\ Q$, acts as $P$ until $P$ terminates by communicating ✓ and then proceeds to act as $Q$. The termination signal is hidden from the process environment. The process which may only terminate is written Skip.

The parallel evolution of processes $P$ and $Q$, synchronized on event set $X$ is written

$$P \,|[\, X \,]|\, Q.$$

No event from $X$ is enabled in $P \,|[\, X \,]|\, Q$ unless enabled jointly by both $P$ and $Q$.

Other events occur in either $P$ or $Q$ separately.

Diversity of behavior is introduced through two choice operators. The external choice operator allows a process a choice of behavior according to what events are enabled by its environment. The process

$$a \rightarrow P \,\Box\, b \rightarrow Q$$

begins with both $a$ and $b$ enabled and performs the first to be enabled by its environment. Subsequent behavior is determined by the event which actually occurred, $P$ after $a$ and $Q$ after $b$ respectively. External choice may also be written in an intentional form,

$$\Box\, a : A \bullet P(a).$$

Internal choice represents variation in behavior determined by the internal state of the process. The process

$$a \rightarrow P \sqcap b \rightarrow Q$$

may initially enable either $a$, or $b$, or both, as it wishes, but must act subsequently according to which event actually occurred. Again an intentional form is allowed.

An important derived concept in CSP is the notion of *channel.* A channel is a collection of events of the form $c.n$: the prefix $c$ is called the *channel name* and the collection of suffixes the allowed *values* of the channel. When an event $c.n$ occurs it is said that *the value $n$ is communicated on channel $c$*. By convention, when the value of a communication on a channel is determined by the environment (external choice) it is called an *input* and when it is determined by the internal state of the process (internal choice) it is called an *output*. It is convenient to write $c?n : N \rightarrow P(n)$ to describe behavior over a range of allowed inputs instead of the longer $\Box\, n : N \bullet c.n \rightarrow P(n)$. Similarly the notation $c!n : N \rightarrow P(n)$ is used instead of $\sqcap\, n : N \bullet c.n \rightarrow P(n)$ to represent a range of outputs. Expressions of the form $c?n$ and $c!n$ do not represent events, the actual event is $c.n$ in both cases.

Recursion is used to given finite representations of non-terminating processes. The process expression

$$\mu\, P \bullet a?n \rightarrow b!f(n) \rightarrow P$$

describes a process which repeatedly inputs an integer on channel $a$, calculated some function $f$ of the input, and then outputs the result on channel $b$. CSP specifications are typically written as a sequence of simultaneous equations in a finite collection of process variables. Such a specification $\vec{X} == \vec{F}(\vec{X})$ is implicitly taken to describe the solution to the vector recursion $\mu\, \vec{X} \bullet \vec{F}(\vec{X})$.

In general, the behavior of a process at any point in time may be dependent on its internal state and this may conceivably take an infinite range of values. It is often not possible to provide a finite representation of a process without introducing some notation for representing this internal process state. The approach adopted by CSP is to allow a process definition to be intentionally parameterized by state variables.

Thus a definition of the form

$$P_{n:N} \mathrel{\widehat{=}} Q(n)$$

represents a (possibly infinite) family of definitions, one for each possible value of $n$. It is important to note that there is no inherent notion of process state in CSP, but rather that this intentional form of expression is a convenient way to provide a finite representation of an infinite family of process descriptions.

To the standard CSP process primitives, Timed CSP adds two time specific primitives, the delay and the timeout.

A process which allows no communications for period $t$ then terminates is written $\textsc{Wait}\,t$. The process

$$\textsc{Wait}\,t;\ P$$

is used to represent $P$ delayed by time $t$.

The timeout construct passes control to an exception handler if no event has occurred in the primary process by some deadline. The process

$$a \to P \triangleright\{t\}\ Q$$

will pass control to $Q$ if the $a$ event has not occurred by time $t$, as measured from the invocation of the process.

A *Leave* process of the *Stack* example in TCSP can be constructed as follows:

$$\begin{aligned} \mathit{Stack}_{\mathit{Leave}}(\mathit{items}) = {} & \mathit{out}!\mathit{head}(\mathit{items}) \to \\ & ((\mathit{ack} \to \mathit{Pop}) \triangleright\{5\}\ \mathit{Stack}_{\mathit{Leave}}(\mathit{items})) \end{aligned}$$

It states that the *Leave* process will output the first element in the stack every 5 time units until an acknowledge message *ack* is received.

### 2.4.4 TCOZ

Timed Communicating Object Z (TCOZ) [55] is essentially a blending of Object-Z [25] with Timed CSP [76], for the most part preserving them as proper sub-languages of the blended notation. The essence of this blending is the identification of Object-Z operation specification schemas with terminating CSP processes. Thus operation schemas and CSP processes occupy the same syntactic and semantic category, operation schema expressions may appear wherever processes may appear in CSP and CSP process definitions may appear wherever operation definitions may appear in Object-Z. The primary specification structuring device in TCOZ is the Object-Z class mechanism.

In this section we briefly consider various aspects of TCOZ. A detailed introduction to TCOZ and its Timed CSP and Object-Z features may be found elsewhere [56]. The formal semantics of TCOZ is also documented [53].

**A model of time**

In TCOZ, all timing information is represented as real valued measurements in *seconds*, the SI standard unit of time [43]. We believe that a mature approach to measurement and measurement standards is essential to the application of formal techniques to systems engineering problems. In order to support the use of standard units of measurement, extensions to the Z typing system suggested by Hayes and Mahony [36] are adopted. Under this convention, time quantities are represented by the type

$$\mathbb{T} == \mathbb{R} \odot \mathsf{T},$$

where $\mathbb{R}$ represents the real numbers and $\mathsf{T}$ is the SI symbol for dimensions of time. Time literals consist of a real number literal annotated with a symbol representing a unit of time. All the arithmetic operators are extended in the obvious way to allow calculations involving units of measurement.

**Interface – channels, sensors and actuators**

CSP channels are given an independent, first class role in TCOZ. In order to support the role of CSP channels, the state schema convention is extended to allow the declaration of communication channels. If $c$ is to be used as a communication channel by any of the operations of a class, then it must be declared in the state schema to be of type **chan**. Channels are type heterogeneous and may carry communications of any type. Contrary to the conventions adopted for internal state attributes, channels are viewed as shared (global) rather than as encapsulated entities. This is an essential consequence of their role as communications interfaces *between* objects. The introduction of channels to TCOZ reduces the need to reference other classes in class definitions, thereby enhancing the modularity of system specifications.

As a complement to the synchronizing CSP channel mechanism, TCOZ also adopts a non-synchronizing shared variable mechanism. A declaration of the form $s : X$ **sensor** provides a channel-like interface for using the shared variable $s$ as an input. A declaration of the form $s : X$ **actuator** provides a local-variable-like interface for using the shared variable $s$ as an output. Sensors and actuators may appear either at the system boundary (usually describing how global analog quantities are sampled from, or generated by the digital subsystem) or else within the system (providing a convenient mechanism for describing local communications which do not require synchronization). The shift from closed to open systems necessitates close attention to issues of control, an area where both Z and CSP are weak [100]. We believe that TCOZ with the **actuator** and **sensor** can be a good candidate for specifying open control systems. Mahony and Dong [54] presented detailed discussion on TCOZ sensor and actuators.

### Active objects

Active objects have their own threads of control, while passive objects are controlled by other objects in a system. In TCOZ, an identifier Main (indicating a non-terminating process) is used to represent the behavior of active objects of a given class [19]. The Main operation is optional in a class definition. It only appears in a class definition when the objects of that class are active objects. Classes for defining passive objects will not have the Main definition, but may contain CSP process constructors. If $ob_1$ and $ob_2$ are active objects of the class $C$, then the independent parallel composition behavior of the two objects can be represented as $ob_1 \mid\mid\mid ob_2$, which means $ob_1.\textsc{Main} \mid\mid\mid ob_2.\textsc{Main}$

### Semantics of TCOZ

A separate paper details the blended state/event process model which forms the basis for the TCOZ semantics [53]. In brief, the semantic approach is to identify the notions of operation and process by providing a process interpretation of the Z operation schema construct. TCOZ differs from many other approaches to blending Object-Z with a process algebra in that it does not identify operations with events. Instead an unspecified, fine-grained, collection of state-update events is hypothesized. Operation schemas are modelled by the collection of those sequences of update events that achieve the state change described by the schema. This means that there is no semantic difference between a Z operation schema and a CSP process. It therefore makes sense to also identify their syntactic classes.

### Network topology

The syntactic structure of the CSP synchronization operator is convenient only in the case of pipe-line like communication topologies. Expressing more complex communication topologies generally results in unacceptably complicated expressions. In TCOZ, a graph-based approach is adopted to represent the network topology [52]. For example, consider that processes $A$ and $B$ communicate privately through the interface $ab$, processes $A$ and $C$ communicate privately through the interface $ac$, and processes $B$ and $C$ communicate privately through the interface $bc$. One CSP expression for such a network communication system is

$$\begin{aligned}&(A[bc'/bc] \,|[\, ab, ac \,]|\, (B[ac'/ac] \,|[\, bc \,]|\, C[ab'/ab]) \setminus ab, ac, bc)\\ &\quad [ab, ac, bc/ab', ac', bc']\end{aligned}$$

The hiding and renaming is necessary in order to cover cases such as $C$ being able to communicate on channel $ab$. The above expression not only suffers from syntactic clutter, but also serves to obscure the inherently simple network topology. This network topology of $A$, $B$ and $C$ may be described by

$$\|(A \xleftrightarrow{ab} B;\; B \xleftrightarrow{bc} C;\; C \xleftrightarrow{ca} A).$$

Other forms of usage allow network connections with common nodes to be run together, for example

$$\|(A \xleftrightarrow{ab} B \xleftrightarrow{bc} C \xleftrightarrow{ca} A),$$

and multiple channels above the arrow, for example if processes $D$ and $F$ communicate privately through the channel/sensor-actuator $df_1$ and $df_2$, then

$$\|(D \xleftrightarrow{df_1, df_2} F).$$

The syntactic implication of the above approach is that the basic structure of a TCOZ document is the same as for Object-Z. A document consists of a sequence of definitions, including type and constant definitions in the usual Z style. TCOZ varies from Object-Z in the structure of class definitions, which may include CSP channel and processes definitions. For instance, an active Stack can be derived from the previous (Object-Z) *Stack* model as:

*ActiveStack*

*Stack*

| | |
|---|---|
| $t_j, t_l : \mathbb{T}$ | [durations for Join/Leave operations] |
| $in, out : \mathbf{chan}$ | [channels for input and output] |

$Join \mathrel{\widehat=} [item : Item \mid \#items < max] \bullet in?item \rightarrow Push \bullet \textsc{Deadline}\ t_j$
$Leave \mathrel{\widehat=} [items \neq \langle\ \rangle] \bullet out!head(items) \rightarrow Pop \bullet \textsc{Deadline}\ t_l$
$\textsc{Main} \mathrel{\widehat=} \mu\, Q \bullet Join \,\Box\, Leave;\ Q$

where the TCOZ Deadline command is used to constraint the *Join* and *Leave* to be finished within their duration time.

As we can see that Object-Z and TCSP complement each other not only in their expressive capabilities, but also in their underlying semantics. Object-Z is an excellent notation for modelling data and states, but difficult for modelling real-time and concurrency. TCSP is good for specifying timed process and communication, but like CSP, cumbersome to capture the data states of a complex system. The combination of the two, TCOZ, treats data and algorithmic aspects in the Object-Z style and treats process control, timing, and communication aspects in the TCSP style. In addition, the object oriented flavor of TCOZ provides an ideal foundation for promoting modularity and separation of concerns in system design. With the above modelling abilities, TCOZ is potentially a good candidate for specifying composite systems in a highly constructed manner.

There is another well-known approach combining Object-Z and CSP developed by Smith and Derrick [81]. In this approach the Object-Z classes are given a CSP-style semantics in which operation calls become CSP events. Operation names take on the role of CSP channels, with input and output parameters being passed down the operation channel as values. In this approach any two operations with the same name and parameters will be modelled by identical events when their parameters have same values and hence will be able to synchronize. This view fits nicely with the Object-Z interpretation of operations being atomic, but is not well suited to considering multi-threading and real-time.

There are two main phases in specifying a concurrent system using Smith and Derrick's approach.

- The first phase is to decompose the complex system into components and specify each of these components using Object-Z.
- The second phase involves the specification of the system using CSP operators.

### 2.4.5 Alloy

Alloy [44][2] is a structural modelling language based on first-order logic, for expressing complex structural constraints and behavior. Z was a major influence on Alloy. Very roughly, Alloy can be viewed as a subset of Z. In any Alloy model the universe of atoms is partitioned into subsets, each of which is associated with a basic type. An Alloy model is a sequence of paragraphs that can be of two kinds: signatures, used for construction of new types, and a variety of formula paragraphs, used to record constraints.

#### Signature and fields

A signature paragraph introduces a basic type and a set of atoms drawn from that type. A signature declaration may include a collection of relations (that are called

[2]Version 2.0 is used in this thesis.

fields) in it along with the types of the fields and constraints on their values. Each field declaration introduces a relation whose left type is the signature type. For example,

```
sig S{f: T}
sig T{}
```

introduces S and T as an uninterpreted type (or a set of atoms). The field declaration for f introduces a relation from type of S to the type of T. Implicit in this declaration is that f is constrained to be a total function: it maps each atom in S to exactly one atom T. This constraint can be weakened by inserting the keyword option to say that each atom of S is mapped to at most one atom of T, or set to eliminate the constraint entirely.

A signature may inherit fields and constraints from another signature. This is called a subsignature. Declaring a subsignature doesn't introduce any new types. For example,

```
static part sig T, U extends S {}
```

declares T and U to be subsets of S and inherit the field f. To say that two subsignatures are mutually disjoint, the keyword disjoint is attached to each of their declarations. The keyword part declares these subsets to be disjoint and their union to be Class. To indicate that a signature contains exactly one atom, mark it as static.

**Relational expressions**

In Alloy, every expression denotes a relation. There are no sets of atoms or scalars; they are all represented by relations. A relation is a structure that relates atoms, a collection of tuples of atoms. Each element of such a tuple is atomic and belongs to some basic type. A relation may have any arity greater than one and typed. Sets can be viewed as unary relations.

Relations can be combined with a set operator or relational operator to form expressions. For the set operators, the tuple structure of a relation is irrelevant; a relation might as well be a set of atoms. For the relational operators, the tuple structure is essential to the operator's definition.

The standard operators in ASCII form – union (+), intersection (&), and difference (-) are used on the set to combine two relations of the same type, viewed as sets of tuples. Their interpretation is standard: a tuple is in p+q for example if and only if it is in p or in q; a tuple is in p&q for example if and only if it is in p and in q; a tuple is in p-q for example if and only if it is in p but not in q.

The quintessential relational operator is composition, or join (dot operator). The join p.q of relation p and q is the relation derived from taking every combination of a tuple in p and a tuple in q, and including their join, if it exists. When p is a unary relation (i.e., a set) and q is a binary relation, p.q is standard composition; p.q can alternatively be written as q[p], but with lower precedence. The unary operators $\sim$ (transpose), $\wedge$ (transitive closure), and $*$ (reflexive transitive closure) have their standard interpretation and can only be applied to binary relations.

## Formulas

Formulas may have the value true or false. Formulas can be made using relational comparison operators: subset (: or in), equality (=) and their negations (!:, !in, !=). The formula p in q is true when every tuple of p is also a tuple of q. In other words, viewed as sets of tuples, p is a subset of q. Equality is just containment in both directions; p=q is true when both p in q and q in p are true. Larger formulas are made from smaller formulas by combining them with the standard logical operators, and by quantifying formulas that contain free variables. The formula no e is true when e denotes a relation containing no tuples. Similarly, some e, sole e, and one e are true when e has some, at most one, and exactly one tuple respectively. Alloy provides the standard logical operators: && (conjunction), || (disjunction), => (implication), and ! (negation).

Quantified formulas consist of a quantifier, a comma separated list of declarations, and a formula. Table 2.3 shows the various quantifiers in Alloy.

In a declaration, part specifies partition and disj specifies disjointness; they have their usual meaning.

| **Quantifiers** | **Description** |
|---|---|
| all x: e — F | universal: F is true for every x in e |
| some x: e — F | existential: F is true for some x in e |
| no x: e — F | F is true for no x in e |
| sole x: e — F | F is true for at most one x in e |
| one x: e — F | F is true for exactly one x |

Table 2.3: Quantifiers in Alloy

The expression can be prefixed with a set of keywords scalar, set or option. The keyword scalar adds the side condition that the variable denotes a relation containing a single tuple; set says it may contain any number of tuples; option says it contains at most one tuple. The default marking is set, except when the comparison operator is the colon(:) or negated colon (!:), and the expression on the right is unary, in which case it is scalar.

### Functions, facts, and assertions

A function (fun) is a parameterized formula that can be applied by instantiating the parameters with expressions whose types match the declared parameter types. A fact (fact) is a formula that constrains the values of the sets and relations. Fact takes no arguments and need not be invoked explicitly and it is always true. An assertion (assert) is a formula that is intended to be valid: in other words, it is a consequence that is supposed to follow from the facts.

**Alloy Analyzer**

The Alloy Analyzer (AA) is a tool for analyzing models written in Alloy. Given a formula and a scope – a bound on the number of atoms in the universe – it determines whether there exists a model of the formula (that is, an assignment of values to the sets and relations that makes the formula true) that uses no more atoms than the scope permits, and if so, returns it. Since first order logic is undecidable, AA limits its analysis to a finite scope. If no model is found, the formula may still have a model in larger scope. Nevertheless, the analysis is useful, since many formulas that have models have small scope.

AA works by translating constraints to boolean formulas, and then applying state-of-art SAT solvers. It can analyze billions of states in seconds.

AA provides two kinds of analysis, addressing the two principal risks of declarative modelling. The first risk is that the constraints given are too weak. Flaws of this sort are found by AA by checking assertions, in which a consequence of the specification is tested by attempting to generate a counterexample. The second risk is that the constraints given are too strong; in the worst case, the constraints contradict one another and all states are ruled out. Flaws of this sort are found in simulation in which the consistency of a fact or function is demonstrated by generating a snapshot showing its invocation.

# Chapter 3

# ZML: Browsing Z Family Documents On the Web

This chapter presents the development of a Web browsing environment for ZML.

## 3.1 Introduction

One reason for the slow adoption of formal methods (FM) is the lack of tool support and connections to the current industrial practice. Recent efforts and success in FM have been focused on building 'heavy' tools, such as theorem provers and model checkers. Although those tools are essential and important in supporting applications of formal methods, they are usually less used in practice due to the intrinsic difficulty involved in the technology. In order to achieve wider acceptance of formal methods, it is necessary to develop 'light' weight tools, such as easy-access browsers for formal specifications and projection/transformation tools from formal specifications to industry popular graphical notations. The World Wide Web (WWW) is a promising environment for software specification and design because it allows sharing design models and providing hyper textual links among the models [46].

Object-Z [24, 80], the object-oriented extension to Z, has an active research community but lacks tool support. TCOZ [55, 54] integrates Object-Z with process algebra Timed-CSP [76, 77]. In this chapter, we use XML and the eXtensible Stylesheet Language (XSL) [93] to develop a Web environment that provides various browsing and syntax checking facilities for Z family languages.

The SW builds on the success of XML and uses XML as the foundation technique. The achievement presented in this chapter uses only the traditional Web techniques like XML and does not use the Semantic Web related techniques. This work has been done during the early stage of the PhD study. It is a first attempt to investigate how the Web technology assists a formal design and development process. ZML provides a nice environment for browsing the Z families formal models on the Web. However under this environment, the formalisms are difficult to extend and integrate. This motivates us to investigate how the SW can be used to build a flexible environment for different formalisms, and details about this flexible environment will be presented in the next chapter. ZML is joint work done by Dr. SUN Jing and myself. Dr. SUN Jing contributed more to the design of the ZML schema. The details of the ZML schema design have been presented at [83]. I focus on the XSL translation between ZML and HTML, and some extensive browsing facilities for schema calculus and class inheritance expansion etc. We continue this work with involving in the definition of a standard markup language [90] for the ISO Z standard [1], contributed to the Community Z Tools (CZT) initiative [58]. Hopefully it will become part of the ISO Z standard in the future.

The remainder of the chapter is organized as follows. Section 3.2 gives a brief introduction to the requirements for browsing Z family notations on the Web. Section 3.3 presents the implementation issues of the Web environment and browsing facilities for Z family languages. Section 3.4 concludes the chapter.

## 3.2 Z family languages requirements

In this section, we will outline some requirements for browsing Z family specifications on the Web. The differences among Z, Object-Z and TCOZ notations are illustrated and Z schema calculus and Object-Z/TCOZ inheritance expansions (which is the challenge of the ZML development) are explained. Note that the essential requirements of building ZML are highlighted in **bold** fonts.

### 3.2.1 Schema inclusion and calculus

Z specifications consist of schema inclusion and schema calculus, which are important constructs for composing complex schema definitions. Consider the Z model of a stack in Chapter 2. The expansions from the schema inclusion of the *Stack* and *StackInit* definitions are illustrated as below in $\Delta Stack$ and $StackInit_e$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\Delta Stack} \\
\hline
items : \mathrm{seq}\, Item \\
items' : \mathrm{seq}\, Item \\
\hline
\#items \leqslant max \\
\#items' \leqslant max \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{StackInit_e} \\
\hline
items : \mathrm{seq}\, Item \\
\hline
\#items \leqslant max \\
items = \langle\ \rangle \\
\hline
\end{array}$$

The expanded form of the schema calculus in *Transit* is:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Transit}_e} \\
\hline
\Delta \textit{Stack} \\
\textit{item}?, \textit{item}! : \textit{MSG} \\
\hline
\exists\, \textit{items}'' : \operatorname{seq} \textit{Item} \bullet \textit{items}'' = \langle \textit{item}? \rangle \frown \textit{items} \\
\qquad \wedge\ \textit{items}'' \neq \langle\ \rangle \wedge \textit{items}'' = \langle \textit{item}! \rangle \frown \textit{items}' \\
\hline
\end{array}
$$

The schema calculus expansions such as $\textit{Transit}_e$ are useful for analysis, review and reasoning about Z specifications. ZML should support all schema inclusion and calculus expansions automatically.

### 3.2.2 Inheritance

Inheritance is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes. Active classes can be defined by inheriting passive classes. TCOZ is a superset of Object-Z and all Object-Z classes are treated as passive classes (without MAIN operation) in TCOZ. For instance, the expanded form of the active stack example in Chapter 2 is as follows:

$ActiveStack_e$

$items : \mathrm{seq}\, Item$
$t_i, t_j : \mathbb{T};\ in, out : \mathbf{chan}$
$\#\, items \leq max$

$\mathrm{INIT}$
$items = \langle\ \rangle$

$Push$
$\Delta(items)$
$item? : Item$
$items' = \langle item? \rangle ^\frown items$

$Pop$
$\Delta(items)$
$item! : Item$
$items \neq \langle\ \rangle$
$items = \langle item! \rangle ^\frown items'$

$Join \mathrel{\widehat{=}} [item : Item \mid \#items < max] \bullet in?item \rightarrow Push \bullet \mathrm{DEADLINE}\ t_j$
$Leave \mathrel{\widehat{=}} [items \neq \langle\ \rangle] \bullet out!head(items) \rightarrow Pop \bullet \mathrm{DEADLINE}\ t_l$
$\mathrm{MAIN} \mathrel{\widehat{=}} \mu\, Q \bullet Join \mathbin{\square} Leave;\ Q$

Essentially, all definitions are pooled with the following provisions. Inherited type and constant definitions and those declared in the derived class are merged. The state and initialization schemas of derived classes and those declared in the derived class are conjoined. Operation schemas with the same name are also conjoined.

**We believe the browsing facilities are particularly useful to Object-Z/TCOZ since the notations support cross references and various inheritance techniques for large specifications. It is necessary to view a full expanded version of an inheriting class for the purpose of reasoning and reviewing the class in isolation. It is desirable for ZML to automatically support the inheritance zoom-in/out features.**

### 3.2.3 Instantiation and composition

Let $C$ be the name of a class. The identifier $C$ semantically denotes a collection of objects of the class. Objects may have object references as attributes, i.e. conceptually, an object may have constituent objects. Such references may either be individually named or occur in aggregates. For example, the declaration $c : C$ declares $c$ to be a reference to an object of the class described by $C$. The term $c.att$ denotes the value of attribute $att$ of the object referenced by $c$, and $c.Op$ denotes the evolution of the object according to the definition of $Op$ in the class $C$. Both Object-Z and TCOZ support object composition, e.g., two stacks and two active-stacks classes can be constructed based on Chapter 2's examples in Object-Z and TCOZ respectively as:

*TwoStack*

$q_1, q_2 : Stack$

$Join \mathrel{\widehat=} q1.Push$
$Leave \mathrel{\widehat=} q2.Pop$
$Transfer \mathrel{\widehat=} q1.Pop \parallel q2.Push$

*TwoActiveStack*

$q_1 : ActiveStack[talk/out]$
$q_2 : ActiveStack[talk/in]$

$\textsc{Main} \mathrel{\widehat=} q_1 \mid[\, talk \,]\mid q_2$

The Object-Z parallel operator '$\|$' used in the definition of *Transfer* (in *TwoStack*) achieves inter-object communication: the operator conjoins constraints and equates variables with the same name and also equates and hides any input variable to one of the components of $\|$ with any output from the other component that has the same base name (i.e. the inputs and outputs are denoted by the same identifier apart from ? and ! decorations).

The CSP parallel operator '$|[\, talk \,]|$' used in the definition of MAIN (in TwoActiveStack) captures the concurrent and synchronization behavior of the two communicating active processes $q_1$.MAIN and $q_2$.MAIN.

The models of *TwoStack* and *TwoActiveStack* appear to have similar behavior. However, the behavior of *TwoStack* is purely sequential. For example, *Join* ($q_1.Push$) and *Leave* ($q2.Pop$) cannot concurrently operate or partially overlap (even assuming the duration of Object-Z operations can be explicitly modelled). This limitation is overcome in the (TCOZ) *TwoActiveStack* (since two active stacks have their own threads of control, only synchronizing through the *talk* channel).

**Object-Z/TCOZ Models of complex systems may involve complex composition hierarchies, it is useful to have hyper links for all defined types (particularly the class types) automatically created in the design document – a clear requirement for the ZML tool.**

## 3.3 Web environment for Z family languages

### 3.3.1 Syntax definition and usage

Firstly, a customized XML document for Z family language is defined according to its syntax formal definitions. This document is used for checking the syntax validity of the user input specifications in XML. The World Wide Web Consortium (W3C) has provided two mechanisms for describing XML structures: Document Type Definition (DTD) and XML Schema. The former originated from the SGML Recommendations and had a total different syntax. XML Schema is a kind of XML file itself and is going to play the role of the DTD in defining customized XML structure in the future. It is consistent with XML syntax and simpler to write than the DTD. We use XML Schema to define our ZML structure syntax for the Z family notations. Part of the XML Schema (for defining a class and its operation schema) is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
  xmlns:dt="urn:schemas-microsoft-com:datatypes">
 ...
  <ElementType name="op" content="eltOnly" order="seq">
    <element type="name" minOccurs="1" maxOccurs="1"/>
    <element type="delta" minOccurs="0" maxOccurs="1"/>
    <element type="decl" minOccurs="0" maxOccurs="*"/>
    <element type="st" minOccurs="0" maxOccurs="1"/>
    <element type="predicate" minOccurs="0" maxOccurs="*"/>
      <AttributeType name="layout" dt:type="enumeration"
         dt:values="simpl calc" default="simpl"/>
      <attribute type="layout"/>
  </ElementType>
  <ElementType name="classdef" content="eltOnly">
  ...
   <element type="op" minOccurs="0" maxOccurs="*"/>
  ...
  </ElementType> ...
</Schema>
```

It states that the `op` tag is an element of `classdef` and consists of one `name`, a $\Delta - delta$ list, a number of declarations `decl`, a horizontal line `st` and some `predicate` definitions. An attribute `layout` is defined to distinguish between vertical layout schemas `simpl` and horizontal layout schemas `calc`.

Z family languages consist of a rich set of mathematical symbols. Those symbols can be presented directly in Unicode that is supported by XML. We have defined all entities in the DTD so that users do not have to memorize all the Unicode numbers when authoring their ZML documents. Part of the entity declaration DTD is defined as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<!ENTITY emptyset "&#x2205;">
<!ENTITY mem "&#x2208;">
<!ENTITY pset "&#x2119;">
<!ENTITY nem "&#x2209;">
<!ENTITY uni "&#x222a;">
...
```

As most existing Z specifications were constructed in LaTeX, translating them to our format can be a trivial task as each entity is given a Z LaTeX compatible name. The

DTD is chosen to define our entity declaration because XML Schema does not support entity declaration at the moment. When authoring ZML files, the user simply declares the name space of the XML schema and Entity DTD file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<!DOCTYPE unicode
   SYSTEM "http://nt-appn.comp.nus.edu.sg/fm/zml/unicode.dtd">
<objectZnotation xmlns="x-schema:
  http://nt-appn.comp.nus.edu.sg/fm/zml/objectZschema.xml"
  xmlns:HTML="http://www.w3.org/Profiles/XHTML-transitional">
...
</objectZnotation>
```

With the above namespace links, the XML editing tools can check the validity of the file via XML Schema definition and the DTD entity declarations. Any unspecified structures and entity symbols would be reported as a syntax error. The following is the Web browsing environment for the `Stack` class (of the stack specification example) in our ZML format.

```
<classdef layout="simpl" align="left">
  <name>Stack</name>
  <state>
    <decl>
      <name>items</name>
      <dtype>&seq; <type>Item</type></dtype>
    </decl>
    <st/>
    <predicate># items &leq; max</predicate>
  </state>
  <init>
    <predicate>items=&emptyseq;</predicate>
  </init>
  <op layout="simpl">
    <name>Push</name>
    <delta>items</delta>
    <decl>
      <name>item?</name>
      <dtype><type>Item</type></dtype>
    </decl>
    <st/>
    <predicate>items'=&lseq;item?&rseq; &cat; items </predicate>
  </op>
  <op layout="simpl">
    <name>Pop</name>
    ...
  </op>
</classdef>
```

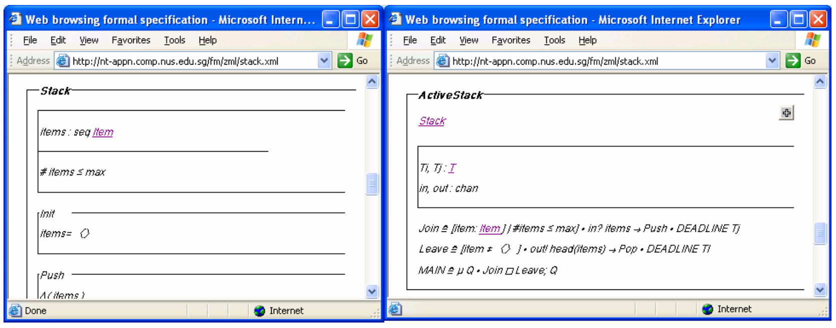
### 3.3.2 XSL transformation

With a valid XML file in hand, the next step is to transform the XML file into HTML format and display it on the Web. XSL is a stylesheet language to describe rules for matching and transforming XML documents. An XSL file is an XML document itself and it can perform the transformation from XML to HTML, XML to XML, XSL to XSL and so on. This kind of transformation can be done on the server side or the client side. Since Internet Explorer 5 (IE5) has already supported XSL technology, the current ZML environment is based on client side (browser) transformation (server side transformation will be discussed later). A partial XSL stylesheet segment for displaying operation *op* and class definition `classdef` is defined below.

```
<xsl:template match="op[@layout='simpl']">
 <html>
  <tr>
    ...
    <td height="24" valign="middle" align="left" nowrap="true">
      <i><xsl:value-of select="name"/></i>
      ...
    </td>
    ...
  </tr>
  <xsl:for-each select="delta | decl">
    <xsl:apply-templates select="."/>
  </xsl:for-each>
  <xsl:apply-templates select="st"/>
  <xsl:for-each select="predicate">
    <xsl:apply-templates select="."/>
  </xsl:for-each>
  ...
 </html>
</xsl:template>

<xsl:template match="classdef[@layout='simpl'] |
                     classdef[@layout='gen']">
 <html>
  ...
  <a><xsl:attribute name="name">
    <xsl:value-of select="name"/>
    </xsl:attribute></a>
  ...
  <xsl:apply-templates select="state"/>
  <xsl:apply-templates select="init"/>
  <xsl:apply-templates select="op"/>
  ...
 </html>
</xsl:template>
```

The XSL stylesheet defines a `match` method for each tag in the XML structure and describes the corresponding HTML codes. From the example above, in matching the


Web browsing formal specification - Microsoft Intern...
File Edit View Favorites Tools Help
Address http://nt-appn.comp.nus.edu.sg/fm/zml/stack.xml Go
Done Internet
Web browsing formal specification - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://nt-appn.comp.nus.edu.sg/fm/zml/stack.xml Go
Internet

Stack

items : seq Item

# items ≤ max

Init

items= ⟨⟩

Push

Δ( items )

ActiveStack

Stack

Ti, Tj : T

in, out : chan

Join ≙ [item: Item ] | #items ≤ max] • in? items → Push • DEADLINE Tj

Leave ≙ [item ≠ ⟨⟩ ] • out! head(items) → Pop • DEADLINE Tl

MAIN ≙ μ Q • Join □ Leave; Q

Figure 3.1: Stack specification on Web

'op' tag the XSL will display the operation name, Δ-list, declaration and predicates accordingly; in matching the 'classdef' tag the XSL will first convert the class name into an HTML bookmark for the type reference usage and then apply the templates of drawing state schema, initiation schema, operations and so on. To apply a template in XSL is similar to making a function call in a programming language, and each template will perform its own transformation. When authoring Z family specifications in our ZML format, the users only need to construct their ZML files and add a URL to the defined XSL stylesheet location as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"
 href="http://nt-appn.comp.nus.edu.sg/fm/zml/objectzed.xsl"?>
```

With this link, the browser will automatically transform a ZML document into the desired HTML output via the built-in XML parser. This process is totally user transparent and much faster than the Java applet approaches [5, 11]. For example, the *Stack* and *ActiveStack* classes in ZML format specified previously is transformed into HTML as in Figure 3.1.

Note that by clicking the 'plus' button the expanded version of class "*ActiveStack*" will be displayed. A full demonstration of the *Stack* specification example is available at

`http://nt-appn.comp.nus.edu.sg/fm/zml/xml-web/stack.xml.`

### 3.3.3 Extensive browsing facilities

In the previous section we showed how the Z family notations can be elegantly and statically presented on the Web. To make the environment more powerful and user friendly, some advanced functionalities are developed. This section discusses the extensive browsing facilities for type reference, class inheritance expansion and schema calculus expansion.

#### Type referencing

When building a large formal model, which could include many type definitions and references, users often want to recall the definition of a particular type. Type referencing allows the user to browse back to the actual type definition and quickly access the corresponding type declarations. In a predicate or declaration, by clicking the name of the type, the user will be brought to the location where the type was declared. This is very useful for specification understanding.

This functionality is achieved in two steps. Firstly when a type definition node in XML is transferred to HTML, its name is converted into an HTML bookmark. Secondly, when the user needs to reference a type in a declaration or predicate, a hyper link that points to the defined bookmark was created. The XSL template for the `type` node is shown as follows:

```
<xsl:template match="type">
  <xsl:choose>
    <xsl:when test="//classdef[$any$ name=context(-1)] |
      //tydef[$any$ name=context(-1)] |
      //schemadef[$any$ name=context(-1)]">
    <a>
      <xsl:attribute name="href">#<xsl:value-of/>
        </xsl:attribute><xsl:value-of/>
    </a>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

It tests whether any name of class definition, basic type definition or schema definition is equivalent to the current type name. If such a name exists, a type hyper-link is established.

### Class inheritance expansion

Inheritance is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes.

Essentially, in Object-Z all definitions are pooled with the following provisions. Inherited type and constant definitions and those declared in the derived class are merged. The state and initialization schemas of derived classes and those declared in the derived class are conjoined. Operation schemas with the same name are also conjoined. Name clashes, which would lead to unintentional merging or conjunction, can be resolved by renaming when inheriting. The inheritance in TCOZ is similar to inheritance in Object-Z except there are some slight differences in inheriting the MAIN operation.

The aim of the class inheritance expansion is to allow a user to view the full definition of a derived class. In the *ActiveStack* class case (in the right hand side of Figure 3.1), when a user clicks the button '+', the full definition of the class of *ActiveStack* will be shown. Clicking button '−' is for going back to the un-expanded version.

The challenge to achieve this facility is in how the completed definition of an inherited class can be automatically constructed from individual class definitions and how to control the presentation of different expanded and non-expanded definitions of a class.

To construct the full expanded version of an inheriting class, it is necessary to access and analyze each of the related XML class definition nodes, and some manipulation is done based on the class inheritance definition. It is a complicated process which is hard to achieve from the simple XSL commands, such as <xsl:match> and <xsl:select>. Fortunately Microsoft has extended two XSL elements <xsl:script> and <xsl:eval> to help the user to perform some complex calculations (As an extension from Microsoft, these two XSL commands are not supported by browsers from non-Microsoft platforms, such as Netscape. Section 3.3.4 will focus on the transformation for the non-Microsoft platforms.) The script node can hold a piece of script for function definitions or variable declarations. The 'eval' element allows a user to generate a text node in the destination document using script. <xsl:script> and <xsl:eval> will be used to automatically construct the full expanded definition of an inheriting class.

We also need to control the visibility of the two versions of definitions (expanded and non-expanded) based on the user requirements, which can be achieved by DHTML and JavaScript. For each of the expandable classes two blocks of HTML content are created using <div> or <layer> elements. Each of the blocks contains one version of the class definition. We swap the visibility of these two blocks based on user requirements by the following JavaScript.

```
<xsl:template match="/">
    <html>
        <head>
            <title>Web browsing Formal Specification</title>
            <SCRIPT language="JavaScript">
                <xsl:comment>
        <![CDATA[
                <!-- some definition omitted here -->
                //a, b are block id.
                function showclass(a, b) {
                    a.style.display="block";
                    b.style.display="none";
                    }
                function hide(a, b) {
                    b.style.display="block";
                    a.style.display="none";
                }
                <!-- some definition omitted here -->
                ]]></xsl:comment>
            </SCRIPT>
        </head>
        <body>
             <!-- some definition omitted here -->
        </body>
    </html>
</xsl:template>
```

**Schema inclusion and schema calculus expansion**

The purpose of extending schema inclusion and schema calculus is similar to class inheritance expansion, which is giving the user a full picture of a schema definition.

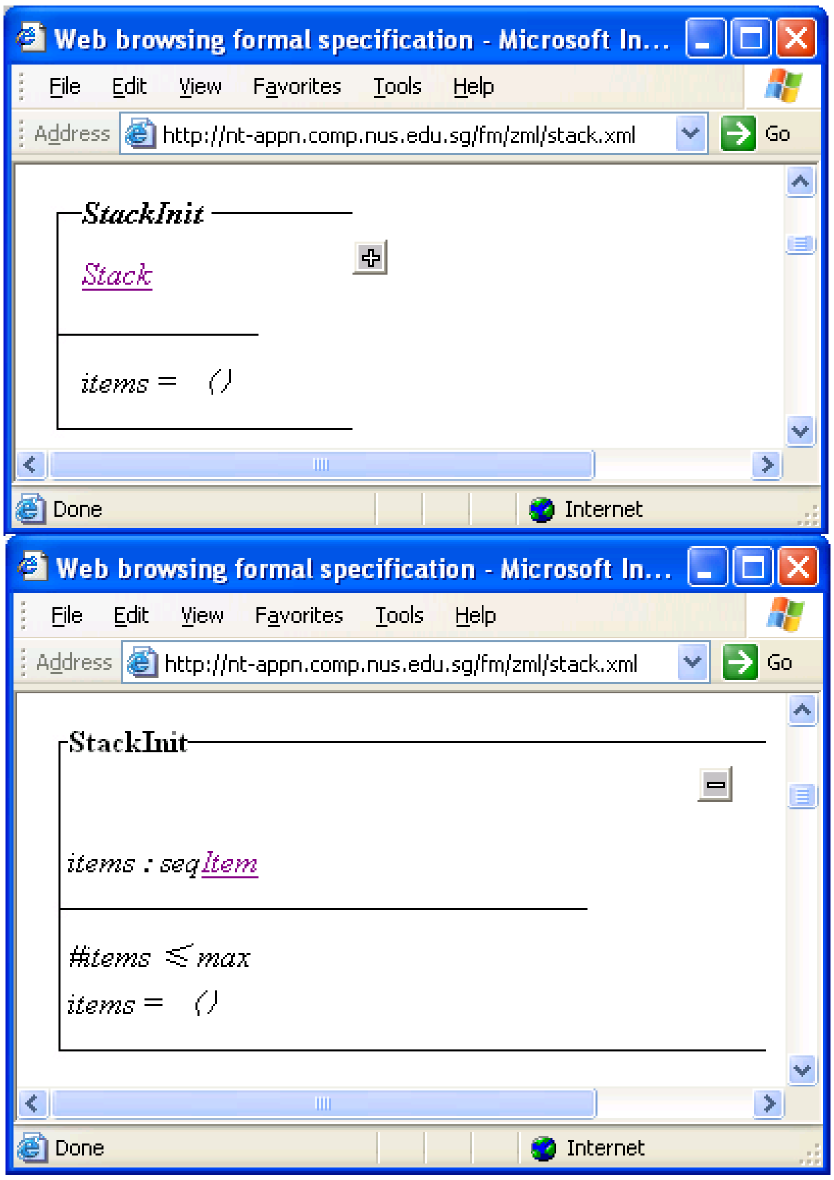
Strictly speaking, schema inclusion refers to including a schema in the declaration part of another schema. In this chapter we also include $\Delta$ and $\Xi$ declaration used in operation schema as one form of schema inclusion. The meaning for each form of inclusion is given here.

In general, including a schema in the declaration part of another schema means that the included schema has its declaration added to the new schema, and its predicate conjoined to the predicate of the new schema. Figure 3.2 shows how *StackInit* schema which includes *Stack* schema was expanded.

The $\Delta$ naming convention is an abbreviation for the schema that includes both the unprimed "before" and the primed "after" state. In $\Delta$ schemas the predicate is always repeated with the primed variables. Figure 3.3 shows how schema *Push* which contains $\Delta Stack$ was expanded.

The $\Xi$ symbol indicates an operation where the state does not change. The declaration part of the $\Xi$ convention is the same as the declaration part of the $\Delta$ convention, but the predicate has, in addition to the predicates of the $\Delta$ convention, more items to ensure that the state is unchanged.

Schema calculus is used to build complex schema from simple ones. The expansion for schema calculus expresses the schema box forms definition of a schema, which is defined by the schema calculus. The schema calculus expansion has two approaches, partial expansion, which does not expand included schema and full expansion, which expands the entire included schema. There are several main schema operators: schema conjunction, schema disjunction, schema composition, schema piping and schema negation. Our Web environment supports all of them. The expanding requirements for each operator are defined as follows:


Web browsing formal specification - Microsoft In...
File Edit View Favorites Tools Help
Address http://nt-appn.comp.nus.edu.sg/fm/zml/stack.xml
Go
Done
Internet

*StackInit*

*Stack*

$items = \langle\rangle$

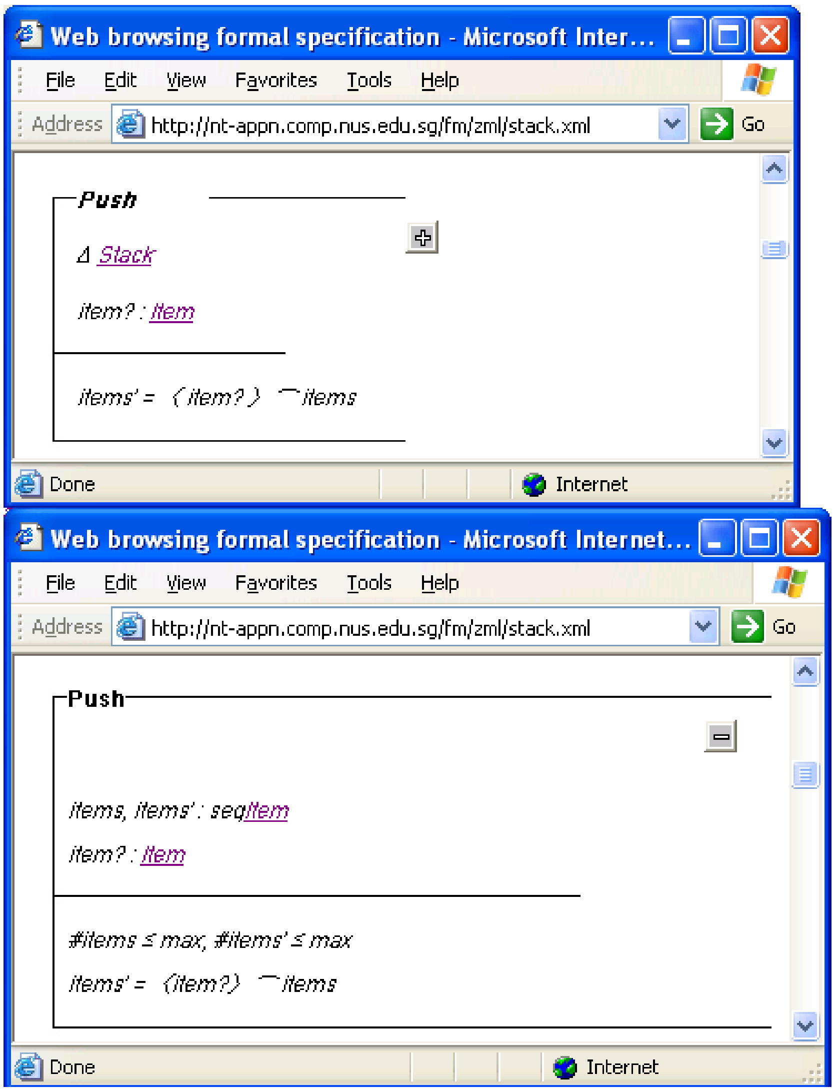
StackInit

$items : \mathrm{seq}\, Item$

$\# items \leq max$

$items = \langle\rangle$

Figure 3.2: Schema inclusion expansion


Web browsing formal specification - Microsoft Inter...
File Edit View Favorites Tools Help
Address http://nt-appn.comp.nus.edu.sg/fm/zml/stack.xml
Go
Done
Internet
Web browsing formal specification - Microsoft Internet...
File Edit View Favorites Tools Help
Address http://nt-appn.comp.nus.edu.sg/fm/zml/stack.xml
Go
Done
Internet

**Push**

$\Delta$ Stack

$item? : Item$

$items' = \langle item? \rangle \frown items$

**Push**

$items, items' : seq Item$

$item? : Item$

$\#items \leq max, \#items' \leq max$

$items' = \langle item? \rangle \frown items$

Figure 3.3: $\Delta$ convention expansion

**Schema conjunction** The declaration part of a schema defined by schema conjunction is obtained by merging the declarations of the schemas on the right-hand side, as explained for schema inclusion. The predicate part is the conjunction of the predicate parts of the schemas on the right, including the implicit predicates.

**Schema disjunction** The declaration part of a schema defined by schema disjunction is obtained by merging the declarations of the schemas on the right-hand side. The predicate part is the disjunction of the predicate parts of the schemas on the right.

**Schema composition** The property of schema composition is constructed as follows. The property of the first schema is included, but all the dashed names are redecorated with a decorator not used in either schema. The property of the second schema is included, but all the undashed names are decorated with the same decorator as was used in the first schema. The newly decorated names are hidden with an existential quantifier.

**Schema piping** The schema defined by piping two schemas is constructed as follows. The output of the first schema is matched with inputs of the second schema. For each matching output and its corresponding input, a name not in scope is chosen, and both are renamed to it. The predicates from two schema combined by conjunction, and the new name are hidden.

**Schema negation** To negate a schema, we expand all the included schemas, and negate the predicate of the expanded schema.

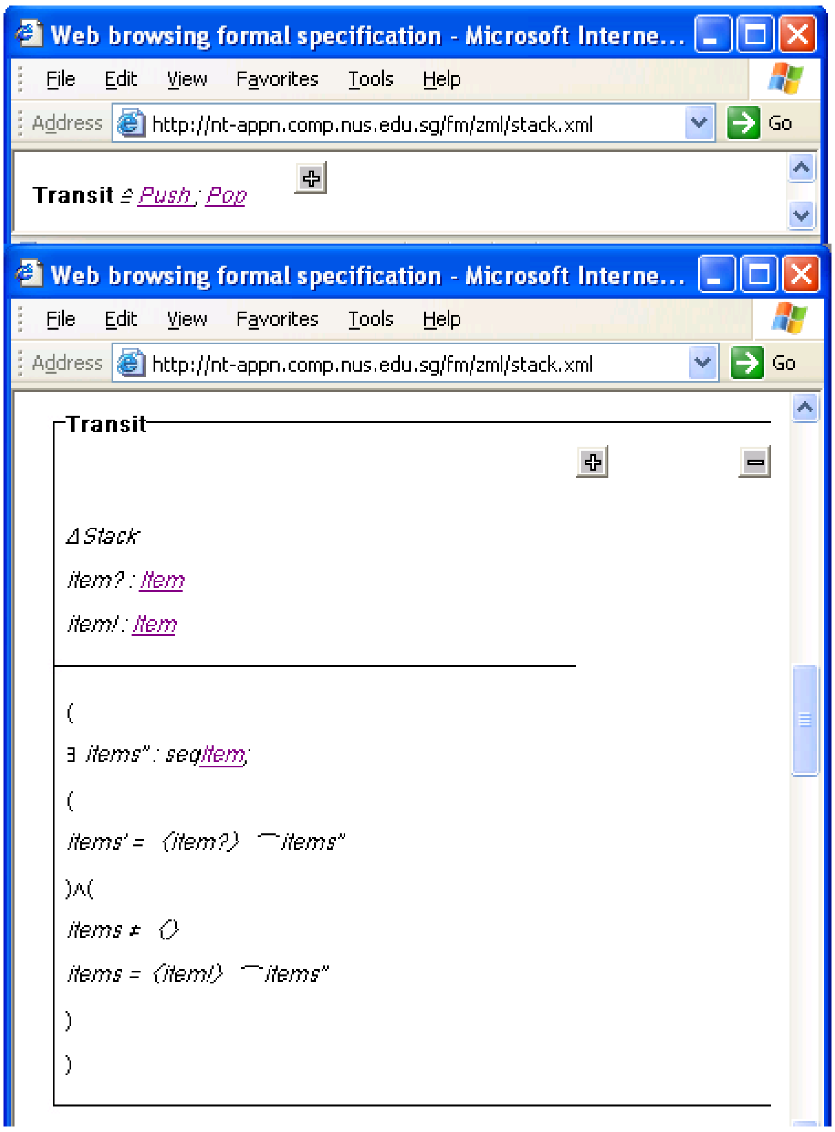
Figure 3.4 and 3.5 show both partial and full expansion of the schema *Transit*, which was defined by the schema composition as:

$Transit \mathrel{\widehat{=}} Push \mathbin{;} Pop$

### 3.3.4 Server side transformation

As mentioned in Section 3.3.2 the current ZML Web environment is based on client (browser) side transformation. It is not compatible with browsers that do not support XSL technology presently such as Netscape. To make the ZML environment available to all kinds of browsers, we can perform the transform on the server side and send back pure HTML to the browsers. XSL transformation on the server is bound to be a major part of the Internet Information Server (IIS) work tasks in the future, as we will see a growth in the specialized browser market (for example the use of Braille, Speaking Web, Web Printers, Handheld PCs, Mobile Phones ... [94]). The following Active Server Pages (ASP) code for transforming the XML file to HTML on the server side can achieve this.

```
<%
```


Web browsing formal specification - Microsoft Interne...
File Edit View Favorites Tools Help
Address http://nt-appn.comp.nus.edu.sg/fm/zml/stack.xml
Go

Transit ≙ Push ; Pop

Web browsing formal specification - Microsoft Interne...
File Edit View Favorites Tools Help
Address http://nt-appn.comp.nus.edu.sg/fm/zml/stack.xml
Go

Transit

$\Delta Stack$

$item? : Item$

$item! : Item$

(

$\exists\ items'' : seq Item;$

(

$items' = \langle item? \rangle \frown items''$

$) \wedge ($

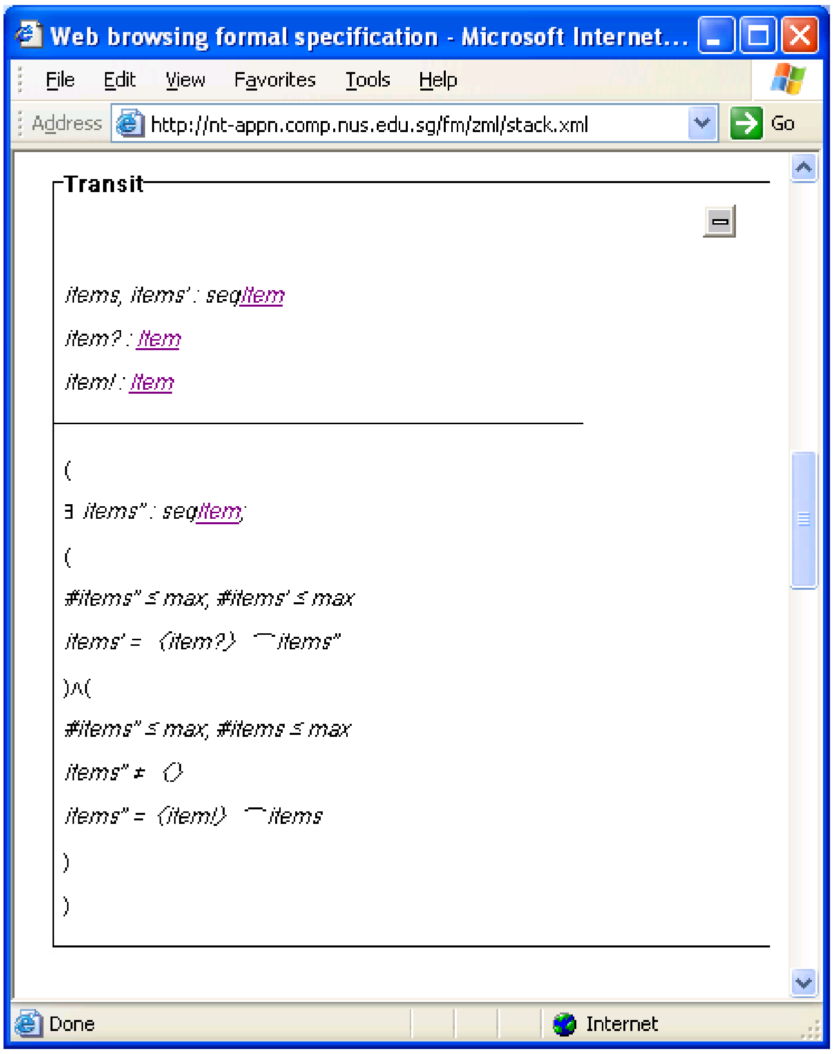
$items \neq \langle\rangle$

$items = \langle item! \rangle \frown items''$

)

)

Figure 3.4: Schema calculus partial expansion


Web browsing formal specification - Microsoft Internet...
File Edit View Favorites Tools Help
Address http://nt-appn.comp.nus.edu.sg/fm/zml/stack.xml
Go
Done
Internet

Transit

*items, items' : seqitem*

*item? : item*

*item! : item*

(

∃ *items'' : seqitem;*

(

*#items'' ≤ max, #items' ≤ max*

*items' = ⟨item?⟩ ⁀ items''*

)∧(

*#items'' ≤ max, #items ≤ max*

*items'' ≠ ⟨⟩*

*items'' = ⟨item!⟩ ⁀ items*

)

)

Figure 3.5: Schema calculus full expansion

```
    'Load the XML
    set xml = Server.CreateObject("Microsoft.XMLDOM")
    xml.async = false
    xml.validateOnParse = true
    xml.load(Server.MapPath("stack.xml"))
    'Load the XSL
    set xsl = Server.CreateObject("Microsoft.XMLDOM")
    xsl.async = false
    xsl.load(Server.MapPath("objectzednewnt.xsl"))
    'Transform the file
    Response.Write(xml.transformNode(xsl))
%>
```

The first block of code creates an instance of the Microsoft XML parser, and validates and loads the XML file into memory. The Microsoft XML parser is a COM component that implements the W3C XML Document Object Model (DOM). As a W3C specification, the objective for the XML DOM has been to provide a standard programming interface to a wide variety of applications for accessing and manipulating XML documents. The second block of code creates another instance of the parser and loads the XSL document into memory. The last line of code transforms the ZML document via the XSL style sheet, and then returns the resultant HTML to the browser.

## 3.4 Conclusions

The main contribution of this chapter is the demonstration of the XML/XSL approach to the development of a Web browsing environment for Z family languages. The ZML Web environment includes the auto type referencing and browsing facilities such as the Z schema calculus and Object-Z/TCOZ inheritance expansions.

Our ideas for putting Z family on the Web can be easily adopted by other formal specification notations, such as VDM and VDM++. In fact, since TCOZ includes most Timed CSP constructs, its Web environment can be used for process algebra (CSP/Timed-CSP) specifications. Perhaps this may create a new culture for constructing formal specifications on the Web in XML rather than in LaTeX. We hope it can be the starting point for developing a standard XML environment for all formal notations (including integrated formal notations, i.e., RAISE [65], SOFL [50] and so on): a formal specification Markup Language (FML). This may also make an impact on formal methods education through the Web.

Since we have constructed a Web XSL environment as close as possible to the LaTeX style files for Z/Object-Z (fuzz.sty and oz.sty), one immediate work is to develop a translation tool to map existing Z/Object-Z specifications in LaTeX to the ZML format [67]. Perhaps a reverse tool is also necessary as long as LaTeX is not totally replaced by XML technology.

# Chapter 4

# Semantic Web for Extending and Linking Formalisms

This chapter presents a flexible Semantic Web environment for formal specification languages.

## 4.1 Introduction

Various formal notations are often extended and combined for modelling large and complex systems. Unlike UML [72], an industrial effort for standardizing diagrammatic notations, a single dominating integrated formal method may not exist in the near future. The reason may be partially due to the fact that there are many different well established individual schools, e.g., VDM forum, Z/B users, CSP group, CCS/$\pi$-calculus family etc. Another reason may be due to the open nature of the research community, i.e. FME (www.fmeurope.org), which is different from the industrial 'globalization' community, i.e. OMG (www.omg.org).

Regardless of whether there will be or there should be an ultimate integrated formal method (like UML), *diversity* seems to be the current reality for formal methods and their integrations. Such diversity may have an advantage, that is, different formal methods and their combinations may be effective for developing various kinds of complex systems. The best way to support and popularize formal methods and their effective combinations is to build a widely accessible, extensible and integrated environment.

In Chapter 3 a new environment for browsing the Z families formal models on the Web was developed. However under this environment, the formalisms are difficult to extend and integrate. The reason is that XML focuses on the syntax of the document – how the document was presented. XML defines the number of elements of certain types, their attributes and ordering, and the sorts of text that can appear in datatypes. When extending or integrating formalisms, we focus more on the semantic of the language. For example, to integrate Object-Z with CSP, we care about whether a resource is a class or operation, not in what kind of syntax the resource was encoded.

The SW related techniques, like RDF, are about things in the world - people who have names, create documents, have friends. RDF is about real things in the world not the documents that describe them. This makes SW a good candidate to provide a widely accessible, extensible and integrated environment for formalisms.

The main contribution of the Semantic Web environments for formalisms is that they provide formal specifications on the Web together with additional semantic information. Furthermore, they facilitate collaborative formal design and some static semantics checking. For instance, given two CSP processes $P_1$ and $P_2$, the following incorrect CSP expression

$$P_1 \rightarrow P_2$$

will be detected by the CSP Semantic Web environment via an RDF validator. This thesis will focus on how these environments can be easily extended and integrated to form new environments for the extension and combination of formalisms.

In this chapter we first use Z [98] and CSP [38] as examples to demonstrate how a Semantic Web environment for formal specification languages can be developed.

After that we show these environments can be further extended and integrated. Furthermore we illustrate how specification comprehension can be supported by RDF queries.

## 4.2 Semantic Web for formal specifications

### 4.2.1 Semantic Web environment — DAML+OIL for Z

Firstly, a DAML+OIL definition for the Z language is developed according to its syntax and static semantics. This definition (a DAML+OIL ontology itself) provides information about the interpretation of the statements given in a Z-RDF instant data model. Part of the DAML+OIL definitions (for constructing a Z schema) is as follows:

```
<rdf:RDF
  xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd = "http://www.w3.org/2000/10/XMLSchema#"
  xmlns:daml = "http://www.daml.org/2001/03/daml+oil#"
  xmlns:z = "http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#">

<!-- some definition omitted -->
  <rdfs:Class rdf:ID="Schemadef">
    <rdfs:label>Schemadef</rdfs:label>
  </rdfs:Class>
  <rdfs:Class rdf:ID="Schemabox">
    <rdfs:label>Schemabox</rdfs:label>
    <rdfs:subClassOf rdf:resource="#Schemadef"/>
    <rdfs:subClassOf>
      <daml:Restriction daml:cardinalityQ="1">
```

```
        <daml:onProperty rdf:resource="#name"/>
      </daml:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <daml:Restriction daml:minCardinality="0">
        <daml:onProperty rdf:resource="#del"/>
        <daml:toClass rdf:resource="#Schemadef"/>
      </daml:Restriction>
    </rdfs:subClassOf>

    <!-- some definition omitted -->
    <rdfs:subClassOf>
      <daml:Restriction daml:minCardinality="0">
        <daml:onProperty rdf:resource="#decl"/>
      </daml:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <daml:Restriction daml:minCardinality="0">
        <daml:onProperty rdf:resource="#predicate"/>
      </daml:Restriction>
    </rdfs:subClassOf>
  </rdfs:Class>
```

(note that `xmlns` stands for XML name space)

The DAML+OIL class `Schemadef` represents the Z schemas. The class `Schemabox`, a subclass of `Schemadef`, represents the Z schemas defined in schema box form. The class `Schemabox` models a type whose instance may consist of a `name`, a number of declarations `decl` and some `predicate` definitions. In addition, a `Schemabox` instance may also have zero or more properties `del` whose value must be another `Schemadef` instance (for capturing the Z $\Delta$-convention). As the thesis focuses on demonstrating the approach, other Semantic Web environments for Z constructs are left out but can be found at:

```
http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z.daml
```

Under the Semantic Web environment for the Z language, Z specifications as RDF instance files can be edited (by any XML editing tool).

The Z notation contains a rich set of mathematical symbols. The unicode DTD defined in Chapter 3 is reused here. We have developed an XSLT program (http://nt-appn.comp.nus.edu.sg/fm/zdaml/rdf2zml.xsl) to transform the RDF environment into ZML, the XML environment for display/browsing Z on the Web directly (using the IE Web browser).

The following is a simple *Buffer* schema and a *Join* operation.

$[MSG]$

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{Buffer}} \\ \mathit{max} : \mathbb{Z} \\ \mathit{items} : \operatorname{seq} \mathit{MSG} \\ \hline \#\mathit{items} \leq \mathit{max} \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{Join}} \\ \Delta \mathit{Buffer} \\ i? : \mathit{MSG} \\ \hline \#\mathit{items} < \mathit{max} \land \\ \mathit{items}' = \langle i? \rangle ^\frown \mathit{items} \land \\ \mathit{max}' = \mathit{max} \\ \hline \end{array}$$

The partial of corresponding RDF definition is as Figure 4.1 which is a graphical representation of the following RDF document.

```
<z:Type rdf:ID="msg">
  <z:type>MSG</z:type>
```

```
</z:Type>
<z:Schemabox rdf:ID="buffer">
  <z:name>Buffer</z:name>
    <z:decl> <z:Decl z:name="max" z:dtype="&integer;"/> </z:decl>
    <z:decl> <z:Decl z:name="items" z:dtype="&seq; MSG"/> </z:decl>
    <z:predicate> #items &leq; max </z:predicate>
</z:Schemabox>
<z:Schemabox rdf:ID="join">
  <z:name>Join</z:name>
  <z:del rdf:resource="#buffer"/>
  <z:decl> <z:Decl z:name="i?" z:dtype="MSG"/> </z:decl>
  <z:predicate>#items &lt; max  &land;
    items'=  &lseq; i? &rseq; &cat; items &land; max' = max
  </z:predicate>
</z:Schemabox>
```

Note that the RDF file is in XML format which can be edited by XML editing tools, i.e. XMLSpy. Alternatively, this RDF specification can be treated as an interchange format which can be generated from ZML or from LaTeXvia our tools.

### 4.2.2 Semantic Web environment — DAML+OIL for CSP

Similarly a Semantic Web environment for CSP can be constructed based on its definition. Part of the DAML+OIL definitions (for constructing a CSP process) is as follows:

```
<!-- some definition omitted -->
<rdfs:Class rdf:ID="Event">
  <rdfs:label>Event</rdfs:label> </rdfs:Class>
<rdfs:Class rdf:ID="Process">
  <rdfs:label>Process</rdfs:label> </rdfs:Class>
```

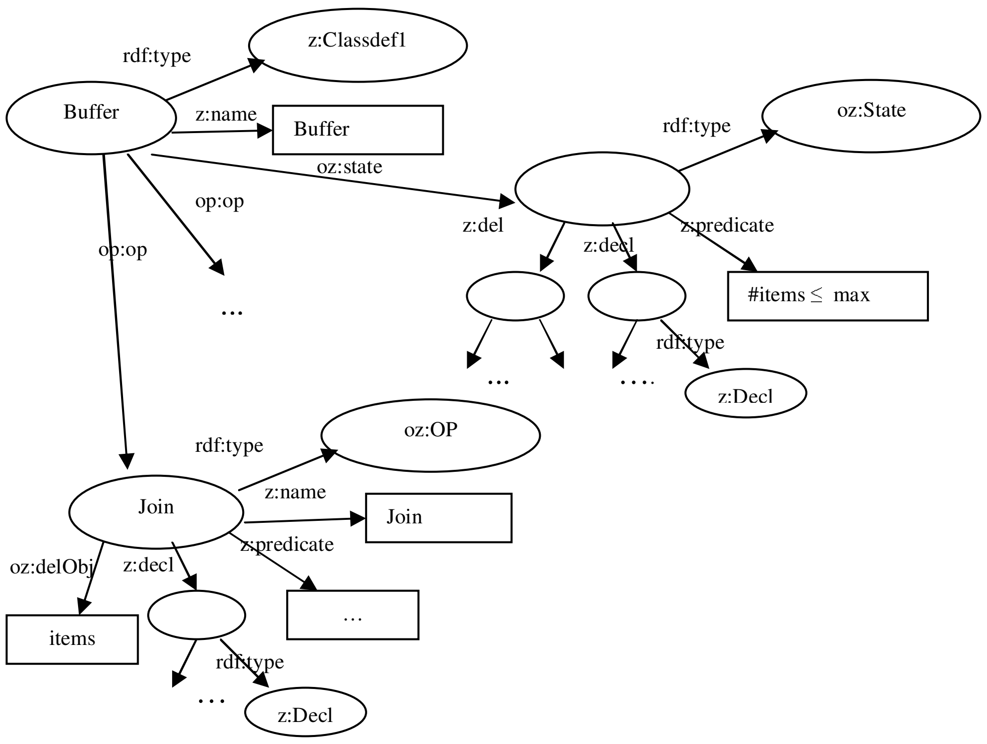


Figure 4.1: Z in Semantic Web environment

```
<rdfs:Class rdf:ID="Simevent">
  <rdfs:label>SimpleEvent</rdfs:label>
  <!-- some definition omitted -->
</rdfs:Class>

<rdfs:Class rdf:ID="Communication">
  <rdfs:label>Communication</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Event"/>
  <!-- some definition omitted -->
</rdfs:Class>

<!--STOP process-->
<rdfs:Class rdf:ID="Stop">
  <rdfs:label>STOP</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Process"/>
</rdfs:Class>
<!--prefix process-->
<rdfs:Class rdf:ID="PrefixPro">
  <rdfs:label>prefixPro</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Process"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#prefix"/>
      <daml:toClass rdf:resource="#Event"/>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#toProc"/>
      <daml:toClass rdf:resource="#Process"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</rdfs:Class>
```

It states that there are two major kinds of constructs in CSP, events and processes. Events can be classified into simple ones and communications containing channels and messages. Processes can be classified into various forms including a special event `STOP`, prefix, sequential etc. For example the parallel processes of process $P1$ and $P2$ will be represented in DAML+OIL as following:

```
<csp:ParallelPro>
  <csp:subprocess rdf:resource="P1"/>
  <csp:subprocess rdf:resource="P2"/>
</csp:ParallelPro>
```

As mentioned in before, these Semantic Web environments provide formal specifications on the Web together with additional semantic information, which make these environments easily extended and integrated to form new environments for the extension and combination of formalisms.

### 4.2.3 Extending Z to Object-Z

Object-Z [24, 80] is an object-oriented extension to Z. A Z specification defines a number of state and operation schemas. In contrast, Object-Z associates individual operations with one state schema. The collective definition of a state schema with its associated operations constitutes the definition of a class. Each class has one state schema, at most one initial schema and number of operation schema. The state schema can be viewed as a nameless Z schema. The initial schema can be viewed as a Z schema which only contains some predicate properties. The following demonstrates parts of the Semantic Web environment for Object-Z. It extends the Z SW environment.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:oz="http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#"
  xmlns:z="http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#"
  xmlns="http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#">
 <daml:Ontology rdf:about="">
   <daml:imports rdf:resource=
     "http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z"/>
 </daml:Ontology>
```

```
<rdfs:Class rdf:ID="State">
  <rdfs:label>State</rdfs:label>
  <rdfs:subClassOf rdf:resource="z:Schemabox"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="z:name"/>
        <daml:hasValue>
          <xsd:string rdf:value=""/>
        </daml:hasValue>
     </daml:Restriction>
  </rdfs:subClassOf>
</rdfs:Class>
<rdfs:Class rdf:ID="Init">
  <rdfs:label>INIT</rdfs:label>
  <!-- some definition omitted -->
</rdfs:Class>
<rdfs:Class rdf:ID="OP">
  <rdfs:label>OP</rdfs:label>
  <!-- some definition omitted -->
</rdfs:Class>

<rdfs:Class rdf:ID="Message">
  <rdfs:label>Message</rdfs:label>
  <rdfs:subClassOf rdf:resource="#OP"/>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="oz:receiver"/>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#method"/>
      <daml:toClass rdf:resource="#OP"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</rdfs:Class>

  <!-- some definition omitted -->
<rdfs:Class rdf:ID="Classdef"/>
```

```
  <rdfs:Class rdf:ID="Classdef1">
    <rdfs:label>Classdef1</rdfs:label>
    <rdfs:subClassOf rdf:resource="#Classdef"/>
    <rdfs:subClassOf>
      <daml:Restriction daml:cardinality="1">
        <daml:onProperty rdf:resource="z:name"/>
      </daml:Restriction>
    </rdfs:subClassOf>
    <!-- some definition omitted -->
    <rdfs:subClassOf>
      <daml:Restriction>
        <daml:maxCardinality>1</daml:maxCardinality>
        <daml:onProperty rdf:resource="#state"/>
        <daml:toClass rdf:resource="#State"/>
      </daml:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <daml:Restriction>
        <daml:onProperty rdf:resource="#op"/>
        <daml:toClass rdf:resource="#OP"/>
      </daml:Restriction>
    </rdfs:subClassOf>
  </rdfs:Class>
  <daml:DatatypeProperty rdf:ID="delObj">
    <rdfs:range rdf:resource=
      "http://www.w3.org/2000/10/XMLSchema#string"/>
  </daml:DatatypeProperty>
</rdf:RDF>
```

This Object-Z Semantic Web environment imports the definition of Z. Note that `Message` class is used to define message passing. It consists of a `receiver` property (object reference) and a `method` property (the operation of the declared class of the receiver).

A `Classdef1` class (an Object-Z class defined by a class box) was defined to have the following properties.

- a `name` property,
- a `state` property whose value must be a `State` class object,
- some `op` properties whose values must be `OP` class objects etc.

The `State` class is a subclass of `Schemabox` (class for a Z schema defined in schema box form). That is a `State` object is a special `Schemadef` object satisfying the restriction that the `name` property has no value. The `OP` class is the same as class `Schemadef` (for Z schema) except a new property `delObj` was added to it. This is due to the difference between the semantic requirements of $\Delta$ list in Z and Object-Z. In Z the entity following $\Delta$ is the name of state schema name, and in Object-Z the entity following the $\Delta$ are variables defined in the class state schema.

Consider the buffer example in Object-Z:

*Buffer*

$$\begin{array}{|l}
max : \mathbb{N} \\
items : \operatorname{seq} MSG \\
\hline
\# items \leq max
\end{array}
\qquad
\begin{array}{|l}
\textsc{Init} \\
\hline
items = \langle \, \rangle
\end{array}$$

*Join*

$\Delta(items)$
$i? : MSG$

$\#items < max$
$items' = \langle i? \rangle ^\frown items$

*Leave*

$\Delta(items)$
$i! : MSG$

$\#items \neq 0$
$items = items' ^\frown \langle i! \rangle$

Under the Semantic Web environment this Buffer class can be edited as the following RDF file.

```
<oz:Classdef1 rdf:ID="buffer">
  <z:name>Buffer</z:name>
    <oz:state>
      <oz:State>
        <z:decl> <z:Decl z:name="max" z:dtype="&integer;"/> </z:decl>
        <z:decl> <z:Decl z:name="items" z:dtype="&seq; MSG"/> </z:decl>
        <z:predicate>#items &leq; max</z:predicate>
      </oz:State>
   </oz:state>
    <!-- some definition omitted -->
   <oz:op>
    <oz:OP rdf:ID="join">
      <z:name>Join</z:name>
      <oz:delObj> items</oz:delObj>
      <z:decl> <z:Decl z:name="i?" z:dtype="MSG"/> </z:decl>
      <z:predicate>#items &lt; max  &land;
                   items'=  {i?} &cat; items
      </z:predicate>
    </oz:OP>
   </oz:op>
</oz:Classdef1>
```

### 4.2.4 Extending CSP to TCSP

The extension from CSP to TCSP can be achieved in a similar way. The following is part of the Semantic Web environment for TCSP.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:tcsp="http://nt-appn.comp.nus.edu.sg/fm/zdaml/TCSP#"
  xmlns:csp="http://nt-appn.comp.nus.edu.sg/fm/zdaml/CSP#"
  xmlns="http://nt-appn.comp.nus.edu.sg/fm/zdaml/TCSP#">
   <daml:Ontology rdf:about="">
     <daml:imports rdf:resource=
        "http://nt-appn.comp.nus.edu.sg/fm/zdaml/CSP"/>
   </daml:Ontology>
   <!--timed event-->
   <rdfs:Class rdf:about="csp:Event">
     <rdfs:subClassOf>
       <daml:Restriction daml:minCardinality="0">
         <daml:onProperty rdf:resource="#etime"/>
       </daml:Restriction> </rdfs:subClassOf>
   </rdfs:Class>
   <daml:DatatypeProperty rdf:ID="etime">
       <rdfs:range rdf:resource=
       "http://www.w3.org/2000/10/XMLSchema#string"/>
   </daml:DatatypeProperty>
   <!--Wait process-->
   <rdfs:Class rdf:ID="Wait">
     <rdfs:label>WAIT</rdfs:label>
     <rdfs:subClassOf rdf:resource="#process"/>
       <daml:Restriction daml:minCardinality="0">
         <daml:onProperty rdf:resource="#etime"/>
       </daml:Restriction> </rdfs:subClassOf>
   </rdfs:Class>
   <!-- some definition omitted -->
```

This TCSP environment is derived by first importing the definition of CSP, and then defining a new property `etime` for the events in CSP. The property `etime` shows the time of occurrence of events. Several new types of process are also defined. For example, the WAIT process is just a subclass of a general process.

One interesting point is that the physical size or number of 'subclass' clauses in the DAML+OIL file (above) may provide an indication of the degree of extension (how much modification and extension has been developed in the new language). Such a concrete number or ratio may give us some quantified comparison, perhaps indicating how new (or faithful) is Object-Z relative to Z, TCSP to CSP or VDM++ to VDM.

In the next section, we will focus on one of the essential parts of this chapter – the use of the Semantic Web for linking formalisms.

## 4.3 Semantic Web for linking formalisms

Various modelling methods can be used in an effective combination for designing complex systems if the semantic links between those methods can be clearly established and defined. Given two sets of formalisms, say state-based ones and event-based ones, it is not too surprising to see that different possible integrations are more than the cross-product of the two sets. This is simply because the different semantic links between the two formalisms lead to different integrations. Furthermore, the semantic links can be uni-directional and bi-directional.

Let's consider the case of linking Object-Z and CSP. Smith and Derrick's approach [81] is to identify Object-Z operations with CSP channel/events and Object-Z classes with CSP processes. The CSP-OZ approach taken by Fischer and Wehrheim [29] is similar to Smith and Derrick's approach except that it divides each Object-Z operation into two separate operations (enable and effect events). The TCOZ approach [55] identifies Object-Z operations with CSP processes.

Despite the differences, all those integrations are useful for modelling different kinds of complex systems. For example, Smith and Derrick's approach is good at modelling a system with a group of simple passive components and complex concurrent interactions (at a system level) between those components. On the other hand, TCOZ is good at modelling a system with complex components which may have their own thread of control and support multi-layer compositions and concurrency.

In this section, we will demonstrate how the Object-Z and (T)CSP Semantic Web environments can be linked to support both the Smith/Derrick and TCOZ approaches.

### 4.3.1 $class \Longrightarrow process$

In Smith/Derrick's approach [81], Object-Z classes are modelled as CSP processes and the Object-Z operations are modelled as CSP events. The event corresponding to an operation is a communication event with the operation name as the channel and the mapping from its parameters to their values as the value passed on that channel. In this approach any two operations with the same name and parameters will be modelled by identical events when their parameters have the same values and hence will be able to synchronize. There are two main phases in specifying a concurrent system.

- The first phase is to decompose the complex system into components and specify each of these components using Object-Z.
- The second phase involves the specification of the system using CSP operators.

Considering the specification of two communicating buffers, the following model demonstrates this approach:

$$\begin{aligned}
&Buffer_1 \mathrel{\widehat=} Buffer[Transfer/Leave]\\
&Buffer_2 \mathrel{\widehat=} Buffer[Transfer/Join]\\
&System \mathrel{\widehat=} Buffer_1 \mathbin{|[}\, Transfer \,\mathbin{]|} Buffer_2
\end{aligned}$$

where the two buffers ($Buffer_1$ and $Buffer_2$) communicate through channel $Transfer$.

The semantic environment for this approach can be achieved in the following way:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:oz="http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#"
  xmlns:csp="http://nt-appn.comp.nus.edu.sg/fm/zdaml/CSP#"
  xmlns:app1="http://nt-appn.comp.nus.edu.sg/fm/zdaml/APP1#">
 <daml:Ontology rdf:about="">
   <daml:imports rdf:resource=
     "http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ"/>
   <daml:imports rdf:resource=
     "http://nt-appn.comp.nus.edu.sg/fm/zdaml/CSP"/>
 </daml:Ontology>
 <rdfs:Class rdf:about="oz:Classdef">
   <rdfs:subClassOf rdf:resource="csp:Pro"/>
 </rdfs:Class>
 <rdfs:Class rdf:about="oz:OP">
   <rdfs:subClassOf rdf:resource="csp:Event"/>
 </rdfs:Class>
  <!--operation is one kind of process-->
</rdf:RDF>
```

It firstly imports the definition of CSP and Object-Z. The Object-Z class is declared as a subclass of the CSP process and the Object-Z operation (extended from Z operation schema) is declared as a subclass of the CSP event.

The above two buffers example can be encoded in the Semantic Web environment in the following way:

```
 <oz:Classdef2 rdf:ID="buffer1">
   <z:name>Buffer1</z:name>
   <oz:rename> Transfer/Leave</oz:rename>
   <oz:eqclass rdf:resource="#buffer"/> </oz:Classdef2>
 <oz:Classdef2 rdf:ID="buffer2">
   <z:name>Buffer2</z:name>
   <oz:rename> Transfer/Join</oz:rename>
   <oz:eqclass rdf:resource="#buffer"/> </oz:Classdef2>
 <oz:Classdef2 rdf:ID="system">
   <z:name>System</z:name>
```

```
    <oz:eqclass>
        <csp:ParallelPro>
            <csp:subprocess rdf:resource="buffer1"/>
            <csp:subprocess rdf:resource="buffer2"/>
            <csp:ParaSync>Transfer</csp:ParaSync>
        </csp:ParallelPro> </oz:eqclass>
  </oz:Classdef2>
```

### 4.3.2 *operation* $\Longleftrightarrow$ *process*

The TCOZ approach is to identify Object-Z operations as CSP processes and all the communication must go through the explicitly declared channels. The behavior of an active object is explicitly captured by a CSP process. To achieve this approach several new elements are introduced. They are:

**Chan** A channel is declared in an object's state.

**Main** This process defines the dynamic control behavior of an active object.

The environment for this approach can be achieved in the following way:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:tcoz="http://nt-appn.comp.nus.edu.sg/fm/zdaml/TCOZ#"
  xmlns:oz="http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#"
  xmlns:csp="http://nt-appn.comp.nus.edu.sg/fm/zdaml/CSP#"
```

```
 xmlns="http://nt-appn.comp.nus.edu.sg/fm/zdaml/TCOZ#">
<daml:Ontology rdf:about="">
  daml:imports rdf:resource=
    "http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ"/>
  <daml:imports rdf:resource=
    "http://nt-appn.comp.nus.edu.sg/fm/zdaml/CSP"/>
</daml:Ontology>
<rdfs:Class rdf:about="oz:State">
  <rdfs:subClassOf>
    <daml:Restriction daml:minCardinality="0">
      <daml:onProperty rdf:resource="csp:chan"/>
          </daml:Restriction>
  </rdfs:subClassOf>
  <!-- the channel can be declared in sate schema-->
</rdfs:Class>
<daml:ObjectProperty rdf:ID="MAIN">
  <rdfs:range rdf:resource="csp:Process"/>
  <rdfs:domain rdf:resource="#Classdef"/>
</daml:ObjectProperty>
<rdfs:Class rdf:about="oz:OP">
  <rdfs:subClassOf rdf:resource="csp:Process"/>
</rdfs:Class>
<rdfs:Class rdf:about="csp:Process">
  <rdfs:subClassOf rdf:resource="oz:OP"/>
</rdfs:Class>
<!--operation is one kind of process-->
</rdf:RDF>
```

Note that the DAML+OIL allows the subclass-relation between classes to be cyclic, since a cycle of subclass relationships provides a useful way to assert equality between classes. In TCOZ, the two communicating buffer system (with timing constraints on input and output operations) can be modelled as:

*TBuffer*

*Buffer*

$left, right : \mathbf{chan}$ [input and output channels]

$t_j, t_l : \mathbb{T}$ [time durations for Join and Leave operations]

$$\textsc{Main} \mathrel{\widehat=} \mu\, Q \bullet ([i : MSG] \bullet left?i \rightarrow Join \bullet \textsc{Deadline}\, t_j \,\Box\, [size \neq 0] \bullet right!last(items) \rightarrow Leave \bullet \textsc{Deadline}\, t_l);\ Q$$

*TSystem*

$l : TBuffer[middle/right]$

$r : TBuffer[middle/left]$

$$\textsc{Main} \mathrel{\widehat=} l \,|[\, middle \,]|\, r$$

In the Semantic Web environment, the class *TSystem* can be encoded as follows:

```
<oz:Classdef1 rdf:ID="tsystem">
  <z:name>TSystem</z:name>
  <oz:state> <oz:State>
     <z:decl>
       <z:Decl z:name="l" z:dtype="TBuffer[middle/right]"/></z:decl>
     <z:decl>
       <z:Decl z:name="r" z:dtype="TBuffer[middle/left]"/></z:decl>
  </oz:State> </oz:state>
  <oz:MAIN>
     <csp:parallelPro>
       <csp:subprocess> <oz:Message oz:receiver="l"
         oz:method="#TBMAIN"></oz:Message> </csp:subprocess>
       <csp:subprocess> <oz:Message oz:receiver="r"
         oz:method="#TBMAIN"></oz:Message> </csp:subprocess>
       <csp:ParaSync>middle</csp:ParaSync>
     </csp:parallelPro> </oz:MAIN>
  </oz:Classdef1>
```

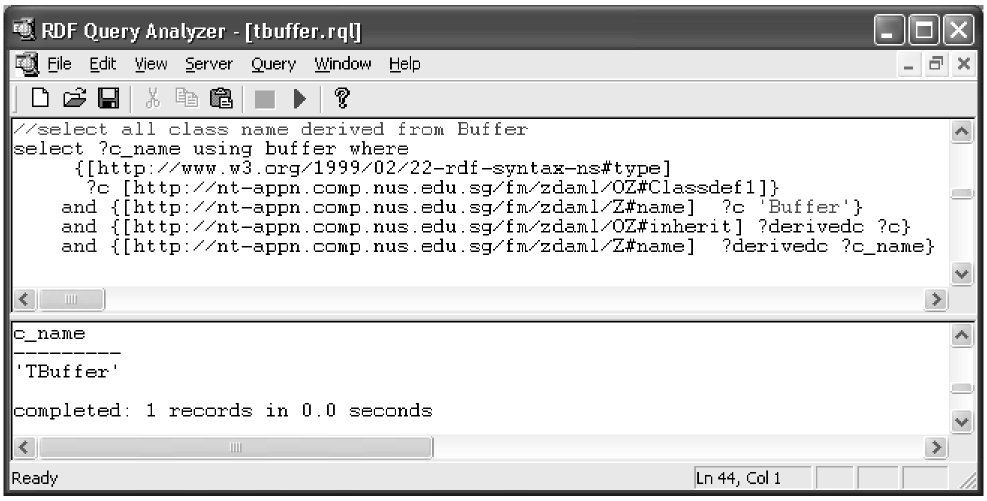
Clearly, unlike Smith and Derrick's approach, TCOZ is not a simple integration of Object-Z and TCSP, like CSP-OZ, TCOZ extends the two base notations with some new language constructs. Another distinct difference is that the semantic link between operation vs. process in TCOZ is bi-directional ($\Longleftrightarrow$), while in Smith and Derrick's approach, the semantic link between class and process has a single direction ($\Longrightarrow$). By building the Semantic Web environments for the two approaches, one can improve the understanding of the difference. Such a Semantic Web environment is applicable for many other integrated formalisms.

## 4.4 Specification comprehension

One of the major contributions of the RDF model, introduced by the Semantic Web community, is that it allows us to do more accurate and more meaningful searching. This strength of RDF can be applied in the specification context leading to the notion of *specification comprehension.* Useful RDF queries can be formulated for comprehending specification models particularly when models are large and complex.

There are many RDF query systems available or under development. In this thesis the RDFQL [42], an RDF query language developed by Intellidimension, is used to demonstrate some queries which can be achieved in the environment.

Based on our simple *Buffer* and *TBuffer* examples, the following demonstrates various queries expressed in RDFQL.


RDF Query Analyzer - [tbuffer.rql]
File Edit View Server Query Window Help

```
//select all class name derived from Buffer
select ?c_name using buffer where
     {[http://www.w3.org/1999/02/22-rdf-syntax-ns#type]
      ?c [http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#Classdef1]}
    and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name]  ?c 'Buffer'}
    and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#inherit] ?derivedc ?c}
    and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name]  ?derivedc ?c_name}
```

```
c_name
----------
'TBuffer'

completed: 1 records in 0.0 seconds
```

Figure 4.2: Find all the sub-classes

### 4.4.1 Inter-class queries

Two typical queries can be formulated for searching and understanding class relationships, such as inheritance hierarchy and composition structure.

(Inheritance) Find all the sub-classes derived from the class *Buffer* (Figure 4.2)

```
Query:
 select ?c_name using buffer where
     {[http://www.w3.org/1999/02/22-rdf-syntax-ns#type]
        ?c [http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#Classdef1]}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name]  ?c 'Buffer'}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#inherit] ?derivedc ?c}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name]  ?derivedc ?c_name}
Result: TBuffer
```

Here we only present the usage of the underlying query engine. To make this tool more useful, a cleaner interface is needed. A GUI is Currently being implemented by the group[1].

(Composition:) Find all classes containing *Buffer* instances (as attributes)

```
Query:
 select ?c_name using buffer where
     {[http://www.w3.org/1999/02/22-rdf-syntax-ns#type]
       ?c [http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#Classdef1]}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name]  ?c ?c_name}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#state] ?c ?s}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#decl] ?s ?d}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#dtype] ?d ?dt}
 and (INSTR(?dt, 'Buffer') = 1)
Result: TSystem
```

### 4.4.2 Intra-class queries

A number of queries can be built for search/understanding class content (this is useful particularly when a class is large and has many operations).

Find all the operations which may change the attribute *items*:

[1]I am very glad to see when I submit the final version of this thesis, the GUI has been built successfully

```
Query:
 select ?op_name using buffer where
     {[http://www.w3.org/1999/02/22-rdf-syntax-ns#type]
         ?c [http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#Classdef1]}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name] ?c 'Buffer'}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#op]  ?c ?op}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#delObj]  ?op 'items'}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name]  ?op ?op_name}
Result: Join, Leave
```

Find all the constant attributes in a class:

```
Query:
 select ?att using buffer where
     {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#state] ?c ?sta}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#decl] ?sta ?decl}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name]  ?decl ?att}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#delObj]  ?op ?att1}
 and (?att <> ?att1)
Result: max
```

Find all the operations which have the same interface (with common base names for output and input):

```
Query:
 select ?op_name1 ?op_name2 using buffer where
     {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#op]  ?c1 ?op1}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/OZ#op]  ?c2 ?op2}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name] ?op1 ?op_name1}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name] ?op2 ?op_name2}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#decl]  ?op1 ?d1}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name]   ?d1 ?n1}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#decl]  ?op2 ?d2}
 and {[http://nt-appn.comp.nus.edu.sg/fm/zdaml/Z#name]   ?d2 ?n2}
 and (?op1 <> ?op2) and (STRCMP(regexp(?n1,'*!'), regexp(?n2,'*?'))= 0)
Result: 'Join'    'Leave'
```

## 4.5 Chapter summary

This chapter focuses on building a Semantic Web (RDF/DAML+OIL) environment for supporting, extending and integrating many different formalisms. Such a *meta integrator* may bring together the strengths of various formal methods communities in a flexible and widely accessible fashion. The Semantic Web environment for formal specifications may lead to many benefits. One novel application which has been demonstrated in this chapter is the notion of specification comprehension based RDF query techniques. The review process of a large specification can be facilitated by various RDF queries.

# Chapter 5

# Checking and Reasoning About Semantic Web Through Alloy

In this chapter, we present how existing formal tools can be used to reason about SW ontologies.

## 5.1 Introduction

In the development of the Semantic Web there is a pivotal role for ontology, since it provides a representation of a shared conceptualization of a particular domain that can be communicated between people and applications. Reasoning can be useful at many stages during the design, maintenance and deployment of ontology. Because autonomous software agents may perform their reasoning and come to conclusions without human supervision, it is essential that the shared ontology is consistent. However, since the Semantic Web technology is still in the early stage, the reasoning and consistency checking tools are primitive.

The software modelling language Alloy [44] is suitable for specifying structural properties of software. SW is a well suited application domain for Alloy because relationships between Web resources are the focus points in SW and Alloy is a first order declarative language based on relations. Furthermore, Alloy specifications can be analyzed automatically using the Alloy Analyzer (AA) [45]. Given a finite scope for a specification, AA translates it into a propositional formula and uses SAT solving technology to generate instances that satisfy the properties expressed in the specification. We believe that if the semantics of the SW languages can be encoded into Alloy, then Alloy can be used to provide automatic reasoning and consistency checking services for SW. Various reasoning tasks can be supported effectively by AA.

The remainder of the chapter is organized as follows. In section 5.2 semantic domain and functions for the DAML+OIL constructs are defined in Alloy. Section 5.3 presents the translation from DAML+OIL document to an Alloy program. In section 5.4 different reasoning tasks are demonstrated. Section 5.5 concludes the chapter.

## 5.2 DAML+OIL semantic encoding

DAML+OIL has a well-defined semantics which has been described in a set of axioms [27]. In this section based on the semantics of DAML+OIL, we define the semantic functions for some important DAML+OIL primitives in Alloy. The complete DAML+OIL semantic encoding can be found in Appendix D.

### 5.2.1 Basic concepts

The semantic model for DAML+OIL is encoded in the module DAMLOIL. Users only need to import this module to reason about DAML+OIL ontology in Alloy.

```
module DAMLOIL
```

All the things described in the Semantic Web context are called resources. A basic type Resource is defined as:

```
sig Resource {}
```

All other concepts defined later are extended from the Resource. Property, which is a kind of Resource itself, relates Resource to Resource.

```
disj sig Property extends Resource
    {sub_val: Resource -> Resource}
```

Each Property has a relation sub_val from set <Property, Resource, Resource> with type <Resource, Resource, Resource> (since in Alloy subsignature does not introduce a new type). This relation can be regarded as an RDF statement, i.e., a triple of the form
<property(or predicate), subject, value(or object)>.

The class corresponds to the generic concept of type or category of resource. Each Class maps a set of resources via the relation instances, which contains all the instance resources. The keyword disj is used to indicate the Class and Property are disjoint.

```
disj sig Class extends Resource {instances: set Resource}
```

The DAML+OIL also allows the use of XML Schema datatypes to describe (or define) part of the datatype domain. However there are no predefined types in Alloy, so we treat Datatype as a special Class, which contains all the possible datatype values in the instances relation.

```
disj sig Datatype extends Class {}
```

### 5.2.2 Class elements

The subClassOf is a relation between classes. The instances in a subclass are also in the superclasses. A parameterized formula (a function in Alloy) is used to represent this concept.

```
fun subClassOf(csup, csub: Class)
  {csub.instances in csup.instances}
```

The disjointWith is a relation between classes. It asserts that there are no instances common with each other.

```
fun disjointWith (c1, c2: Class) {no c1.instances & c2.instances}
```

### 5.2.3 Property restrictions

A toClass function states that all instances of the class c1 have the values of property *P* all belonging to the class c2.

```
fun toClass (p: Property, c1: Class, c2: Class)
 {all r1, r2: Resource | r1 in c1.instances <=>
                         r2 in r1.(p.sub_val)=>r2 in c2.instances}
```

A hasValue function states that all instances of the class c1 have the values of property P as resource r. The r could be an individual object or a datatype value.

```
fun hasValue (p: Property, c1: Class, r: Resource)
    {all r1: Resource | r1 in c1.instances => r1.(p.sub_val) = r}
```

A cardinality function states that all instances of the class c1 have exactly N distinct values for the property P. The new version of Alloy supports some integer operations.

```
fun cardinality (p: Property, c1: Class, N: Int)
 {all r1: Resource| r1 in c1.instances <=> # r1.(p.sub_val) = int N}
```

### 5.2.4 Boolean combination of class expressions

The intersectionOf function defines a relation between a class c1 and a list of classes clist. The List is defined in the Alloy library. The class c1 consists of exactly all the objects that are common to all class expressions from the list clist.

```
fun intersectionOf (clist: List, c1: Class)
     {all r: Resource| r in c1.instances <=>
         all ca: clist.*next.val | r in ca.instances}
```

The unionOf function defines a relation between a class c1 and a list of classes clist. The class c1 consists of exactly all the objects that belong to at least one of the class expressions from the list clist. It is analogous to logical disjunction;

```
fun unionOf (clist: List, c1: Class)
    {all r: Resource| r in c1.instances <=>
      some ca: clist.*next.val| r in ca.instances}
```

### 5.2.5 Property elements

The subPropertyOf function states that psub is a subproperty of the property psup. This means that every pair (subject,value) that is in psup is in the psub.

```
fun subPropertyOf (psup, psub: Property)
    {psub.sub_val in psup.sub_val}
```

The domain function asserts that the property P only applies to instances of the class c.

```
fun domain (p: Property, c: Class)
    {(p.sub_val).Resource in c.instances}
```

The inverseOf function shows two properties are inverse.

```
fun inverseOf (p1, p2: Property) {p1.sub_val = ~(p2.sub_val)}
```

## 5.3 DAML+OIL to Alloy translation

In the previous section we defined the semantic model for the DAML+OIL constructs, so that analyzing DAML+OIL ontology in Alloy can be easily and effectively achieved. We also constructed an XSLT [93] stylesheet for the automatic translation from DAML+OIL file to into an Alloy program. [1]

[1]The details of the XSLT program and other information on this thesis can be found at: http://nt-appn.comp.nus.edu.sg/fm/alloy/

A set of translation rules translating from DAML+OIL ontology to an Alloy program are developed in the following presentation.

### 5.3.1 DAML+OIL class translation

$$\frac{C \in DAML\_class}{static\ disj\ sig\ C\ extends\ Class\{\}}$$

A DAML_class C will be transferred into a scalar C, constrained to be an element of the signature Class.

### 5.3.2 DAML+OIL property translation

$$\frac{P \in DAML\_property}{static\ disj\ sig\ P\ extends\ Property\{\}}$$

A DAML_property p will be translated into a scalar P, constrained to be an element of the signature Property.

### 5.3.3 Instance translation

$$\frac{x \in instancesof[Y]}{\begin{array}{l} static\ disj\ sig\ x\ extends\ Resource\{\} \\ fact\{\ x\ in\ Y.instances\} \end{array}}$$

A DAML instance x of class Y will be translated into a scalar x, constrained to be an element of the signature Resource. x is a subset of Y.instances.

### 5.3.4 Other translations

Other DAML+OIL constructs can be easily translated into the Alloy function we defined in the previous section. For example the following rule shows how to translate the DAML+OIL subclass relation into Alloy code.

$$\frac{subclass[X, Y], X \in DAML\_class, Y \in daml\_class}{fact\{subClassOf(X, Y)\}}$$

### 5.3.5 Case study

A classical DAML+OIL ontology, "animal relation" is used to illustrate how the translation and analysis could be achieved. The following DAML+OIL ontology defines two classes animal and plant which are disjoint. The eats and eaten_by are two properties, which are inverse to each other. The domain of eats is animal. The carnivore is a subclass of animal which can only eat animals.

```
<daml:Class rdf:ID="animal">
  <rdfs:label>animal</rdfs:label> </daml:Class>
<daml:Class rdf:ID="plant">
  <rdfs:label>plant</rdfs:label>
  <daml:disjointWith rdf:resource="#animal"/></daml:Class>
<daml:ObjectProperty rdf:about="eaten_by">
  <rdfs:label>eaten_by</rdfs:label>
</daml:ObjectProperty>
<daml:ObjectProperty rdf:about="eats">
  <rdfs:label>eats</rdfs:label>
  <daml:inverseOf rdf:resource="#eaten_by"/>
  <rdfs:domain><daml:Class rdf:about="#animal"/>
  </rdfs:domain></daml:ObjectProperty>
```

```
<daml:Class rdf:ID="carnivore">
  <rdfs:label>carnivore</rdfs:label>
  <rdfs:subClassOf rdf:resource="#animal"/>
  <rdfs:subClassOf>
    <daml:Restriction> <daml:onProperty rdf:resource="#eats"/>
      <daml:toClass rdf:resource="#animal"/>
    </daml:Restriction>
  </rdfs:subClassOf></daml:Class>
```

This DAML+OIL ontology will be translated into Alloy as follow:

```
module animal
/*import the library module we defined*/
open DAMLOIL
/* plant and animal are translated to two class instances. The key
 word static is used to a signature containing exactly one element.*/
static disj sig plant, animal extends Class {}

/* The disjoin element was translated into fact in Alloy */
fact {disjointWith(plant, animal)}

/* eats, eaten_by are translated to two property instances */
static disj sig eats, eaten_by extends Property {}
fact {inverseOf(eats, eaten_by)}
fact {domain(eats, animal)}

static disj sig carnivore extends Class{}
fact{subClass(animal, carnivore)}
fact{toClass(eats, carnivore, animal)}
```

We can check the consistency of the DAML+OIL ontology and do some reasoning readily.

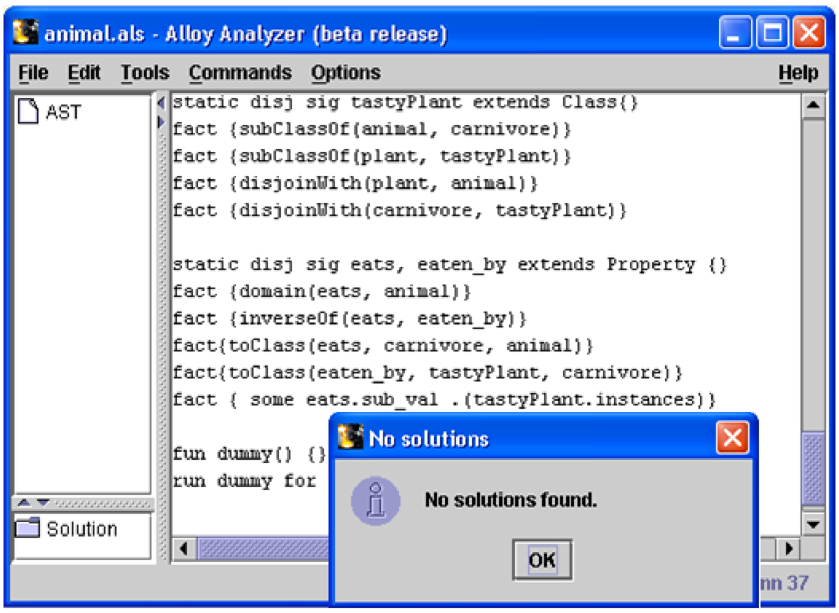
## 5.4 Analyzing DAML+OIL ontology

Reasoning is one of the key tasks for the Semantic Web. It can be useful at many stages during the design, maintenance and deployment of ontology.

There are two different levels of checking and reasoning, the conceptual level and the instance level. At the conceptual level, we can reason about class properties and subclass relationships. At the instance level, we can do the membership checking (instantiation) and instance property reasoning. The DAML+OIL reasoning tool, i.e. FaCT [41], can only provide conceptual level reasoning, while AA can perform both. The FaCT system was designed to be a terminological classifer (TBox) concerned only about the concepts, roles and attributes, not the instances. The Semantic Web reasoner based on the FaCT, like OILED, does not support instance level reasoning well.

### 5.4.1 Class property checking

It is essential that the ontology shared between autonomous software agents is conceptually consistent. Reasoning with inconsistent ontologies may lead to erroneous conclusions. In this section we give some examples of inconsistent ontology that can arise in ontology development, and demonstrate how these inconsistencies can be detected by the Alloy Analyzer. For example, we define another class tastyPlant which



```
static disj sig tastyPlant extends Class{}
fact {subClassOf(animal, carnivore)}
fact {subClassOf(plant, tastyPlant)}
fact {disjoinWith(plant, animal)}
fact {disjoinWith(carnivore, tastyPlant)}

static disj sig eats, eaten_by extends Property {}
fact {domain(eats, animal)}
fact {inverseOf(eats, eaten_by)}
fact{toClass(eats, carnivore, animal)}
fact{toClass(eaten_by, tastyPlant, carnivore)}
fact { some eats.sub_val .(tastyPlant.instances)}

fun dummy() {}
run dummy for
```

No solutions found.

Figure 5.1: Inconsistence example

is a subclass of plant and eaten by the carnivore. There is an inconsistency since by the ontology definition carnivores can only eat animals. Animals and plants are disjoint.

```
<daml:Class rdf:ID="tastyPlant">
 <rdfs:label>tastyPlant</rdfs:label>
 <rdfs:subClassOf rdf:resource="#plant"/>
 <rdfs:subClassOf>
   <daml:Restriction>
     <daml:onProperty rdf:resource="#eat_by"/>
     <daml:toClass rdf:resource="#carnivore"/>
   </daml:Restriction></rdfs:subClassOf>
 </daml:Class>
```

We translate the ontology into an Alloy program, add some facts to remove the trivial models (like every type is empty set) and load the program into the Alloy Analyzer.

The Alloy Analyzer will automatically check the consistency. We conclude that there is an inconsistency in the animal ontology since Alloy can not find any solutions satisfying all facts within the scope (Figure 5.1). Note that when Alloy can not find a solution, it may be due to the scope being too small. By picking a large enough scope, "no solution found' is very likely to mean that an inconsistency has occurred.

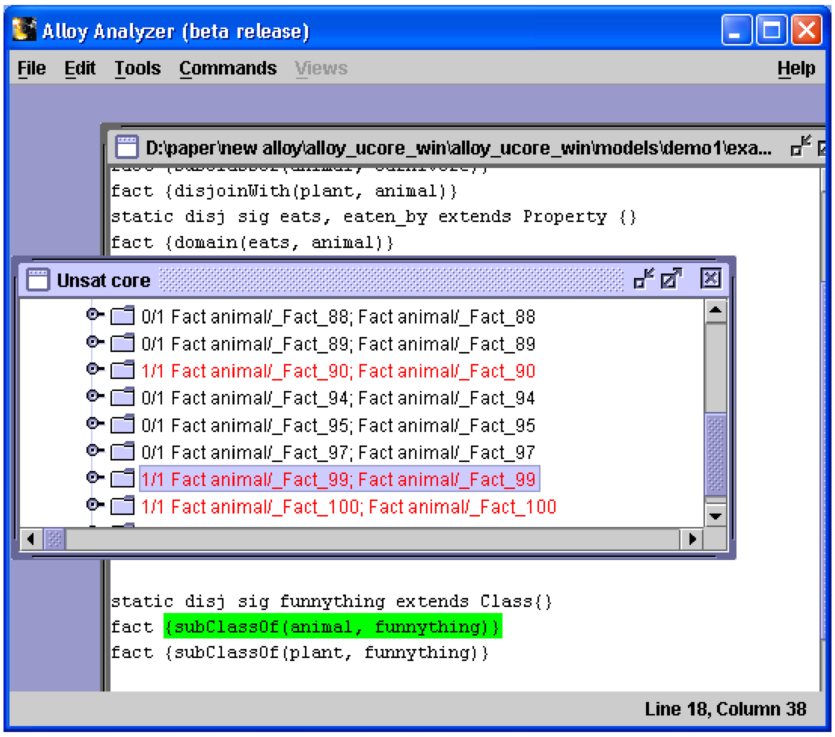
Let us take another example. Suppose we define that the polyphagic_animal eats at least two kind of things i.e polyphagic_animal objects have at least two distinct values for the property eats. There is also one kind of animal called picky_animal which only eats one other kind of animal. The ontology will be defined as follows:

```
<daml:Class rdf:ID="polyphagic_animal">
 <rdfs:label>polyphagic_animal</rdfs:label>
 <rdfs:subClassOf rdf:resource="#animal"/>
 <rdfs:subClassOf>
  <daml:Restriction>
    <daml:onProperty rdf:resource="#eats"/>
    <daml:minCardinality> 2 </daml:minCardinality>
  </daml:Restriction></rdfs:subClassOf></daml:Class>
<daml:Class rdf:ID="#picky_animal">
 <rdfs:label>picky_animal</rdfs:label>
 <rdfs:subClassOf rdf:resource="#animal"/>
 <rdfs:subClassOf>
   <daml:Restriction>
     <daml:onProperty rdf:resource="#eats"/>
     <daml:Cardinality> 1 </daml:Cardinality>
   </daml:Restriction></rdfs:subClassOf></daml:Class>
```

From the above ontology we can infer that the picky_animal is not a kind of polyphagic_animal, otherwise it would be an inconsistency that AA can easily pick up.

Besides discovering the existence of an inconsistency in ontology, tracing where the inconsistency arises from is also crucial for a reasoning tool to be practical. Without any tool support, identifying the conflicting knowledge could be frustrating. One possible systematic technique for finding the causes of inconsistent ontology is to manually remove individual knowledge information until the culprit is identified. This task can be lengthy and dangerous. In the latest version of Alloy [78], the "unsatisfied core" functionality of recent SAT solvers was utilized and it supports core extraction, a new analysis technique that helps to discover over-constraint in declarative models. This functionality can provide some assistance for the user to trace the inconsistency.

Extracting the unsatisfiable core of a CNF formula, that is a subset of the clause set sufficient to cause a contradiction, has been developed recently by satisfiability solvers [78, 32]. In the latest version of Alloy, the declarative model analysis has been cast as satisfiability instances and the unsatisfiable core has been mapped back onto the model. In other words, a user can identify the parts of model responsible for producing the unsatisfiable CNF core. Those parts, by themselves, suffice to produce an over-constraint, and their identification can help the user find the over-constraint. Using this functionality, the portions of the ontology which contradict each other can be traced readily. In the animal example, suppose a new class named funnything was defined to be a subclass of both animal and plant classes. It is easy to see that there is an inconsistency since the class animal and plant are disjoint. Alloy


Alloy Analyzer (beta release)
File Edit Tools Commands Views Help
D:\paper\new alloy\alloy_ucore_win\alloy_ucore_win\models\demo1\exa...
Unsat core
Line 18, Column 38

```
fact {disjoinWith(plant, animal)}
static disj sig eats, eaten_by extends Property {}
fact {domain(eats, animal)}
```

```
0/1 Fact animal/_Fact_88; Fact animal/_Fact_88
0/1 Fact animal/_Fact_89; Fact animal/_Fact_89
1/1 Fact animal/_Fact_90; Fact animal/_Fact_90
0/1 Fact animal/_Fact_94; Fact animal/_Fact_94
0/1 Fact animal/_Fact_95; Fact animal/_Fact_95
0/1 Fact animal/_Fact_97; Fact animal/_Fact_97
1/1 Fact animal/_Fact_99; Fact animal/_Fact_99
1/1 Fact animal/_Fact_100; Fact animal/_Fact_100
```

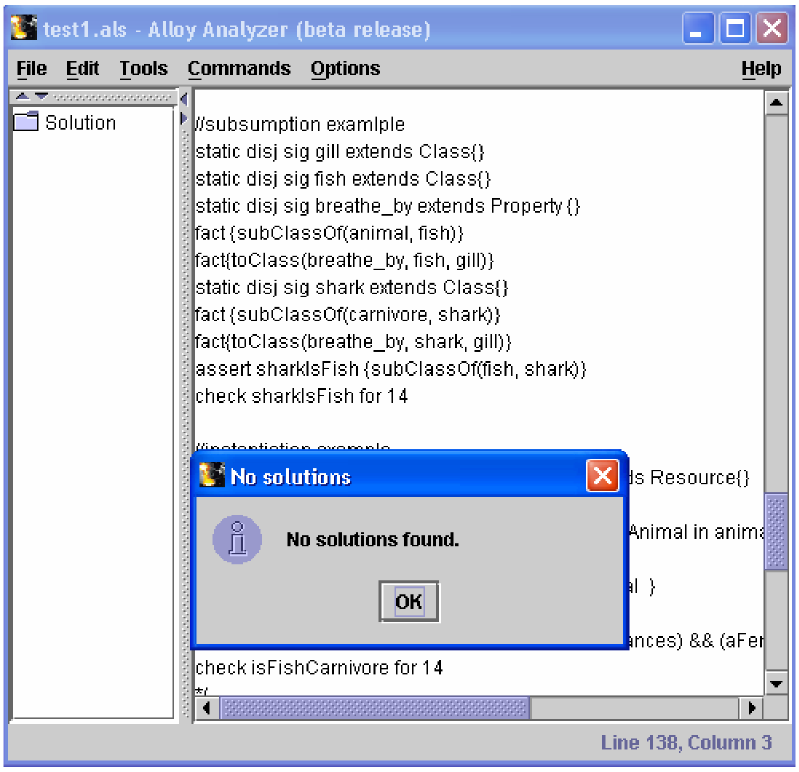
```
static disj sig funnything extends Class{}
fact {subClassOf(animal, funnything)}
fact {subClassOf(plant, funnything)}
```

Figure 5.2: Tracing the inconsistency

can automatically identify a set of knowledge which makes the ontology unsatisfiable (Figure 5.2). The unsatisfiability maybe due to the fact that funnything is a subclass of animal, funnything is a subclass of plant or animal and plant are disjoint classes, and so on.

### 5.4.2 Subsumption reasoning

The task of subsumption reasoning is to infer a DAML+OIL class is the subclass of another DAML+OIL class. We use the relationship between the fish, shark and dolphin as an example to demonstrate this kind of reasoning task. In the animal


test1.als - Alloy Analyzer (beta release)
File Edit Tools Commands Options Help
Solution
No solutions
OK
Line 138, Column 3

```
//subsumption examlple
static disj sig gill extends Class{}
static disj sig fish extends Class{}
static disj sig breathe_by extends Property{}
fact{subClassOf(animal, fish)}
fact{toClass(breathe_by, fish, gill)}
static disj sig shark extends Class{}
fact{subClassOf(carnivore, shark)}
fact{toClass(breathe_by, shark, gill)}
assert sharkIsFish{subClassOf(fish, shark)}
check sharkIsFish for 14

//instantiation example
                                          ds Resource{}

                                          Animal in anima

                                          al }

                                          ances) && (aFe
check isFishCarnivore for 14
*/
```

No solutions found.

Figure 5.3: Subsumption example

ontology a property breathe_by is defined. The fish is a subclass of the animal which breathe_by the gill.

```
<daml:ObjectProperty rdf:ID="breathe_by"/>
<daml:Class rdf:ID="gill">
  <rdfs:label>gill</rdfs:label></daml:Class>
<daml:Class rdf:ID="fish">
  <rdfs:label>fish</rdfs:label>
  <rdfs:subClassOf rdf:resource="#animal"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#breathe_by"/>
      <daml:toClass rdf:resource="#gill"/>
    </daml:Restriction></rdfs:subClassOf>
```

```
</daml:Class>
```

Since the purpose of this thesis is to demonstrate ideas, we keep the ontology simple. In reality there are some animals such as frogs and toads, which can respire by use of gills when they are young and by lungs when they reach adult stage. Also we do not consider the animals which respire by use of the pharyngeal lining or skin, like newborn Julia Creek dunnarts. We also define a class shark, a subclass of carnivore which breathe by the gill.

```
<daml:Class rdf:ID="shark">
  <rdfs:label>shark</rdfs:label>
  <rdfs:subClassOf rdf:resource="#carnivore"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#breathe_by"/>
      <daml:toClass rdf:resource="#gill"/>
    </daml:Restriction>
  </rdfs:subClassOf></daml:Class>
```

Several of the classes were upgraded to being defined when their definitions constituted both necessary and sufficient conditions for class membership, e.g., an animal is a fish if and only if it breathes by the gill. Additional subclass relationships can be inferred, i.e., the shark is also a subclass of fish. We transfer this ontology into an Alloy program and make an assertion that the shark is a subclass of fish. The Alloy analyzer will check the correctness of this assertion automatically (Figure 5.3). The Alloy Analyzer checks whether an assertion holds by trying to find a counterexample.

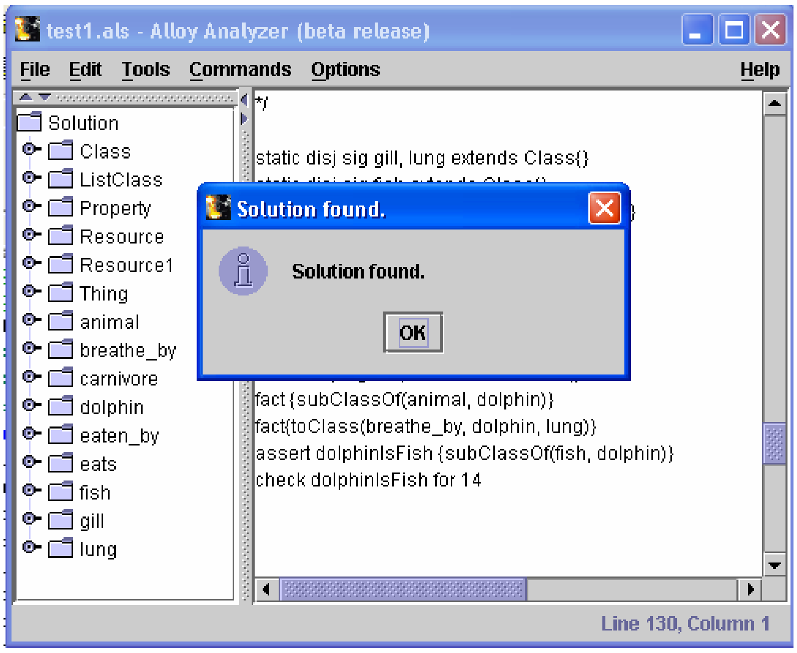
Note that "no solution" means no counterexample found, in this case, it strongly suggests that the assertion is sound. To make it more interesting, we define classes dolphin and lung. Dolphins are a kind of animal which breathe by lungs. The classes gill and lung are disjoint.

```
<daml:Class rdf:ID="lung">
  <rdfs:label>lung</rdfs:label>
  <daml:disjointWith rdf:resource="#gill"/></daml:Class>
<daml:Class rdf:ID="dolphin">
  <rdfs:label>dolphin</rdfs:label>
  <rdfs:subClassOf rdf:resource="#animal"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#breathe_by"/>
      <daml:toClass rdf:resource="#lung"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

Suppose we make an assertion that the dolphin is a kind of fish, the Alloy Analyzer will refute it since some counterexample was found (Figure 5.4). If we add that dolphin is a fish as a fact in the module, the AA will conclude that an inconsistency has arisen.

### 5.4.3 Debugging uncompleted ontology

Information in DAML+OIL is gathered into ontologies, which can then be from different parties and stored as documents in the World Wide Web. Some knowledge



```
*/

static disj sig gill, lung extends Class{}
static disj sig fish extends Class{}
fact {subClassOf(animal, dolphin)}
fact{toClass(breathe_by, dolphin, lung)}
assert dolphinIsFish {subClassOf(fish, dolphin)}
check dolphinIsFish for 14
```

Solution found.

Figure 5.4: Dolphin is not a fish

may be missed in the ontology. Reasoning about uncompleted ontologies may lead to some unexpected results. AA checks the assertion by generating counterexamples – structures or behaviors for which an expected property does not hold; from a counterexample, it is usually not too hard to figure out what's wrong. Looking at the counterexamples may provide some hints to the user on why the expected result does not hold and what knowledge is missing. For example, we want to show the DAML+OIL class dolphin and shark are disjoint. Intuitively, this is a correct statement since dolphin breathes by the gill while shark breathes by the lung. Gill and lung are disjoint. When the following assertion is added to Alloy, surprisingly AA concludes it is wrong.

```
assert disjointDS
   {disjointWith(shark, dolphin)}
```

By looking at the counterexamples graph, it has been noticed that all the counterexamples (an animal which is both a shark and a dolphin) generated by AA have empty values for the property breath by. In fact this unexpected result comes from the semantic of toClass construct in DAML+OIL. A DAML+OIL semantic can not deduce from a toClass restriction alone that there actually is at least one value for the property. A toClass restriction for a property is trivially satisfied for an instance that has no value for that property at all. The toClass restriction demands that all values of the property belong to a class, and if no such values exist, the restriction is trivially true. That is the reason why AA finds out the common instance, which does not breathe at all, for the class dolphin and class shark. To remove this expected result, extra knowledge needs to be added, e.g., an animal must breathe by something.

### 5.4.4 Instantiation

Instance level reasoning is one of the main contributions for reasoning over DAML+OIL ontology using Alloy. Currently some successful DAML+OIL reasoners like FaCT are designed for description logic (DL) T-box reasoning, which lacks support for instances.

In Alloy every expression denotes relations. The scalars will be represented by singleton unary relations - that is, relations with one column and one row. The instance level reasoning can be supported readily in Alloy.

Instantiation is a reasoning task which tries to check if an individual is an instance of a class. For example, we define two resources aFeralAnimal and aMeekAminal as the instances of class animal. aGill is an instance of class gill. aFeralAnimal eats aMeekAnimal and breathes by aGill. People may want to check if aFeralAnimal is a carnivore and a fish.

```
<animal rdf:ID="aMeekAnimal">
  <rdfs:label>aMeekAnimal</rdfs:label>
</animal>
<gill rdf:ID="aGill">
  <rdfs:label>aGill</rdfs:label>
</gill>
<animal rdf:ID="aFeralAnimal">
  <rdfs:label>aFeralAnimal</rdfs:label>
  <breathe_by rdf:resource="aGill"/>
  <eats rdf:resource="aMeekAnimal"/>
</animal>
```

We translate the ontology into an Alloy program and make an assertion as following:

```
static disj sig aFeralAnimal, aMeekAnimal extends Resource{}
static disj sig aGill extends Resource{}
fact {aFeralAnimal in animal.instances &&
      aMeekAnimal in animal.instances}
fact {aGill in gill.instances}
fact {(aFeralAnimal->aMeekAnimal) in eats.sub_val}
fact {(aFeralAnimal->aGill) in breathe_by.sub_val}
assert isFishCarnivore
```

```
  {(aFeralAnimal in fish.instances)
   && (aFeralAnimal in carnivore.instances)}
check isFishCarnivore for 15
```

AA concludes that this assertion is correct.

### 5.4.5 Instance property reasoning

Instance property reasoning (often regarded as knowledge querying) is important in Semantic Web applications. Since one of the promising strengths of Semantic Web technology is that it gives the agents the capability to do more accurate and more meaningful searches. The agent can answer some questions for which the answer is not explicitly stored in the knowledge base.

For example, the emerge_early and emerge_later are two properties, which are inverse to each other. Animal A emerged earlier than B if the species of A emerges earlier than the species of B on the earth. emerge_early is transitive. Three animal instances firstDinosaur, firstApe and firstHuman are defined. firstDinosaur emerge_early than firstApe and firstApe emerge_early than firstHuman. One possible question people may ask is whether firstHuman is emerge_later than firstDinosaur. With the assistance of Alloy reasoner, such questions can be answered.

```
fact{TransitiveProperty(emerge_early)}
static disj sig firstDinosaur, firstApe, firstHuman extends Resource{}
fact { firstDinosaur in animal.instances
```

```
    && firstApe in animal.instances
    && firstHuman in animal.instances}
fact {(firstDinosaur->firstApe) in emerge_early.sub_val}
fact {(firstApe->firstHuman) in emerge_early.sub_val}
assert hum {(firstHuman->firstDinosaur) in emerge_later.sub_val}
check hum for 14
```

AA concludes that this assertion is correct.

The correctness of the translation has been verified by many different test cases. A same problem has been sent to existing SW tools, theorem provers and Alloy; the same conclusions are drawn. Furthermore, the DAML+OIL has well defined semantics in first order logic and Alloy is also based on the first-order logic. The soundness of the translation can also be proved easily.

## 5.5 Chapter summary

The main contribution of this chapter is that it develops the semantic models for DAML+OIL language constructs in Alloy and the systematic translation rules and (XSLT) program which can translate DAML+OIL ontology to Alloy automatically. With the assistance of Alloy Analyzer (AA), we also demonstrated that the consistency of the SW ontology can be checked automatically and different kinds of reasoning tasks can be supported.

Alloy was chosen over other modelling techniques because

- Alloy is based on relations, and relations between Web resources are the focus issues in SW.
- Alloy has an impressive automatic tool support.
- Alloy provides automatical debugging assistance (counter example and UNSAT core).
- Alloy has a relatively simple syntax and semantics which allows us to quickly justify our ideas – reasoning SW using Formal methods tools.

However, the automation of Alloy sacrifices the scalability. The approach we present here can only deal with the ontologies with relatively small size. Based on the same idea, we also attempt to use the theorem prover, i.e. Z/EVES, to reason the SW ontology [17]. The theorem prover can handle large sized ontologies, but it requires the user's interaction. Here we do not claim that Alloy is the only and best formal tool to reason over SW ontologies, but we do claim that it is an effective attempt with certain novel and irreplaceable advantages like full automation and promising debugging assistance. In fact, it is unlikely in the near future that both expressive and automatic tool will be developed. Currently, it is desirable if we can combine the strength from different ontology reasoning tools. In [16], we present the methodology of checking DAML+OIL ontologies using tools RACER, Z/EVES and AA in conjunction.

We believe SW is a new novel application domain for Alloy. Recently, the technique/tool developed in this chapter was successfully applied to a military case study [23]. Alloy was used to check and reason about a *plan ontology* [48] developed by a research team at DSO National Laboratories in Singapore.

Recently, some researchers have begun to explore the potential of combining Web technologies and SE technologies together, e.g. [60]. However there has not been much work done on the application of formal techniques for Semantic Web. In the next chapter we try to extract Web ontology from TCOZ requirement models, which is a very different approach from the techniques demonstrated in this chapter – checking and reasoning about Semantic Web ontology by encoding the semantics of DAML+OIL into the Alloy system.

# Chapter 6

# TCOZ Approach to Semantic Web Service Design

In the previous chapter, we demonstrated that the formal tool, i.e. Alloy, could be used to reasoning the existing SW ontology; that is the ontology has already been built. It could have been revised from a existing ontology or extracted from natural language documents. One natural and interesting question will be, if users want to build a new Semantic Web system, how can formal techniques help on the development process. This question will be answered in this chapter. In this chapter, we present in several ways TCOZ as a specification technique can contribute to the Semantic Web based system development.

## 6.1 Introduction

As the next generation of Web, the Semantic Web (SW) [3] provides computer-interpretable markup of the Web's content and capability, thus enabling automation of many tasks currently performed by humans. Among the most important Web resources are those that provide service. The Web services, as the key application of SW, are Web-accessible programs and devices that will proliferate the Web. Some SW services have been developed recently, e.g. ITTALKS [13].

SW is highly distributed, and different parties may have different understandings for the same concept. One important concept in SW service is ontology. Ontology is the basis for constructing common understanding through explicitly defined relations. The most typical kind of ontology for the Web has taxonomy and a set of constraints. RDFS and DAML+OIL languages can be used to define the ontology. Another important concept in SW service is the semantic markup of service. Semantic markup of the content and capability of Web services – what a service does, how to use it, what its effect will be – will enable easy automation of a variety of reasoning tasks, currently performed manually by human beings, or through arduous hand-coding that enables subsequent automation. DAML-S [12] is such a semantic markup language for Web service.

SW services may have intricate data state, complex process behavior and concurrent interactions. The design of such SW service systems requires precise and powerful modelling techniques to capture not only the ontology domain properties but also the services' process behavior and functionalities. It is desired to have a powerful formal notation to precisely design the Web system.

Timed Communicating Object Z (TCOZ) [55] is a formal specification language which builds on the strengths of Object-Z in modelling complex data and state with strength of Timed CSP in modelling real-time concurrency.

We believe that TCOZ as a high level design technique can contribute to the semantic-web-based system development in many ways. In support of this claim, we conduct a SW service case study, i.e., the online talk discovery system, and apply TCOZ to the design stage to demonstrate how TCOZ can be used as high level design language to specify SW services. The following characteristics of many Web services make TCOZ a good candidate to design such a system.

- A complex Web service system often has both intricate data state and process control aspects. An integrated formal modelling language, like TCOZ, has the strength to model such systems.
- A Web service agent often provides several kinds of different services concurrently. TCOZ has the multithreaded capabilities to capture that.
- A complex Web service system is often composed from sub-services. The sub-

services may be provided by other agents, which have their own thread of control. It can be modelled by the active objects feature in TCOZ.

- A Web service includes highly distributed components with various synchronous and asynchronous communications. It can be specified with various TCOZ communication interfaces – channels, sensors and actuators.
- A Web service like an online hospital or online bank may have critical timing requirements. TCOZ can capture the real-time requirement well.

Furthermore, the chapter presents the development of the systematic translation rules and tools to automatically extract the Web ontology and semantic markup for the SW services from the formal TCOZ design model. This online talk discovery system is a simplified version of the ITTALKS system [13] which is a real life SW service case study.

The remainder of the chapter is organized as follows. Section 6.2 formally specifies the functionalities of the Semantic Web service case study (talk discovery system). Section 6.3 presents the tool which extracts the ontology used by the SW service from the TCOZ design model automatically. Section 6.4 presents the tool which extracts the semantic markup for SW service from the TCOZ design model automatically. Section 6.5 concludes the chapter.

## 6.2 The talk discovery system

In this section, an online talk discovery system is used as a case study to demonstrate how TCOZ notation can be applied to the Semantic Web service development.

### 6.2.1 System scenario

The talk discovery system is a Web portal offering access to information about talks and seminars. This Web portal can provide not only the talk's information corresponding to the user's profile in terms of his interest and location constraints, but also can further filter the IT related talks based on information about the user's personal schedule, etc.

In the course of operation, the talk discovery system discovers that there is an upcoming talk that may interest a registered user based on information in the user's preferences, which have been obtained from his online, DAML-encoded profile. Upon receiving this information, the user's User Agent needs to know more; it consults with its Calendar agent to determine the user's availability, and with the MapQuest agent to find the distance from the user's office to the talk's venue. We assume that a user only wants to attend the talks located within a few miles from his office. Finally, after evaluating the information and making the decision, the User Agent will send a notification back to the talk discovery agent indicating that the user will/will not plan to attend. The completed functionality of the ITTALKS system can be found at `http://www.ittalks.org/jsp/Controller.jsp`.

### 6.2.2 Formal model of the talk discovery system

The system involves four different intelligent agents which communicate interactively. They are the user's Calendar agent, MapQuest agent, user's personal agent and the talk discovery agent.

#### Calendar agent

Firstly, the *DATE* and *TIME* set are defined by the Z given type definitions. As this thesis focuses only on demonstrating the approach, we try to make the model simple. Z given type is chosen to define *TIME*, *DATE* and some other concepts. These concepts can be subdivided into detailed components, e.g., the *TIME* comprises hour, minute, and second. The more detailed the model is, the more detailed ontology will be derived automatically from our tool. This tool will be further discussed in the later section.

The *DateTime* is defined as a schema with two attributes date and time.

$[\mathit{TIME}, \mathit{DATE}]$

```
┌─ DateTime ─────────────────────
│ date : DATE;   time : TIME
└────────────────────────────────
```

The Calendar agent maintains a schedule for each eligible user and supplies some related services. Each eligible user must have a personal ID [*PID*] registered. This *id* is used to validate the identity of users when the system receives requests. The Calendar agent has an *ID manager* which provides functions for identity certifying. It may use Web security techniques like digital signatures to ensure the service is only available to the valid users.

The following specifies the ID manager:

*PIDManager*

$$\begin{array}{|l} ids : \mathbb{P}\, PID \\ add, remove : \mathbf{chan} \\ check : \mathbf{chan} \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{\textsc{Init}} \\ ids = \varnothing \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{AddPID} \\ \Delta(ids) \\ id? : PID \\ \hline ids' = ids \cup \{id?\} \\ \hline \end{array} \qquad \begin{array}{|l} \multicolumn{1}{l}{RemovePID} \\ \Delta(ids) \\ id? : PID \\ \hline ids' = ids - \{id?\} \\ \hline \end{array}$$

$$\begin{array}{l}
New \mathrel{\widehat=} [id : PID \mid id \notin ids] \bullet \\
\qquad add?id \rightarrow AddPID \\
Delete \mathrel{\widehat=} [id : PID \mid id \in ids] \bullet \\
\qquad remove?id \rightarrow RemovePID \\
Validate \mathrel{\widehat=} [id : PID] \bullet check?id \rightarrow \\
\qquad ([id \in ids] \bullet check!\,\mathrm{true} \rightarrow Skip \\
\qquad \Box\ [id \notin ids] \bullet check!\,\mathrm{false} \rightarrow Skip) \\
\textsc{Main} \mathrel{\widehat=} \mu\, N \bullet (New \Box Delete \Box Validate);\ N
\end{array}$$

The *Status* defined by the Z free type definition indicates if a person is free or busy.

$$Status ::= FREE \mid BUSY$$

*Calendar*

$$
\begin{array}{|l}
\hline
timetable : (PID \times DateTime) \rightarrow Status \\
upd, checktm : \textbf{chan} \\
check : \textbf{chan} \\
\hline
\end{array}
$$

*Upd*

$$
\begin{array}{|l}
\Delta(timetable) \\
id? : PID;\ t? : DateTime;\ \ s? : Status \\
\hline
timetable' = timetable \oplus \{(id?, t?, s?)\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
Update \mathrel{\widehat=} [id : PID;\ t : DateTime;\ s : Status] \\
\quad \bullet\ upd?(id, t, s) \rightarrow check!id \rightarrow \\
\qquad (check?\,\mathrm{false} \rightarrow \textsc{Skip} \,\square\, check?\,\mathrm{true} \rightarrow Upd) \\
Check\_Status \mathrel{\widehat=} [id : PID;\ t : DateTime] \\
\quad \bullet\ checktm?(id, t) \rightarrow check!id \rightarrow \\
\qquad (check?\,\mathrm{false} \rightarrow \textsc{Skip} \,\square \\
\qquad check?\,\mathrm{true} \rightarrow checktm!timetable(id, t) \rightarrow Skip) \\
\textsc{Main} \mathrel{\widehat=} \mu\, N \bullet (Update \,\square\, Check\_Status);\ N
\end{array}
$$

*Update* is used to update the timetable. The operation *Check_Status* is used to check whether a person is available or not for a particular time slot.

**MapQuest agent**

MapQuest agent is a third party agent supplying the service for calculating the distance between two places.

Firstly, the *PLACE* is defined as a Z given type. The MapQuest agent contains a set of places in its domain and a database storing the distance between any two places.

$[PLACE]$

*MapQuest*

$$\begin{array}{l}
places : \mathbb{P}\, PLACE \\
distance : places \times places \rightarrow \mathbb{R}^{+} \\
dist : \mathbf{chan}
\end{array}$$

$$\begin{array}{l}
Get\_dis \mathrel{\widehat=} [p_1, p_2 : places] \\
\quad \bullet\ dist?(p_1, p_2) \rightarrow dist!distance(p_1, p_2) \rightarrow Skip \\
\textsc{Main} \mathrel{\widehat=} \mu\, N \bullet Get\_dis;\ N
\end{array}$$

**Personal agent**

The personal agent keeps the user's profile including user's name, office location, interests, etc.

$[NAME, SUBJECT]$

*Person*

$$\begin{array}{l}
id : PID \\
name : NAME \\
office : PLACE \\
interests : \mathbb{P}\, SUBJECT \\
upd, talkch, dist, checktm : \mathbf{chan}
\end{array}$$

$$\begin{array}{l}
Check \mathrel{\widehat=} [tk : Talk] \bullet talkch?(id, tk) \rightarrow \\
\qquad ((checktm!(id, tk.dt) \rightarrow [tresult : Status] \\
\qquad\quad \bullet\ checktm?tresult \rightarrow Skip) \parallel \\
\qquad (dist!(office, tk.place) \rightarrow [dresult : \mathbb{R}^{+}] \\
\qquad\quad \bullet\ dist?dresult \rightarrow Skip)); \\
\qquad [tresult = FREE \wedge dresult < 5] \\
\qquad\quad \bullet\ talkch!(id, GO) \rightarrow \\
\qquad\qquad upd!(id, tk.dt, BUSY) \rightarrow Skip \\
\qquad \Box\ [tresult = BUSY \vee dresult \geqslant 5] \bullet \\
\qquad\quad talkch!(id, NO) \rightarrow Skip \\
\textsc{Main} \mathrel{\widehat=} \mu\, N \bullet Check;\ N
\end{array}$$

After receiving an interested talk information from the talk discovery agent (defined later), the personal agent uses operation *Check* to communicate with his calendar agent to check whether the user is free or not and with the MapQuest agent to ensure the talk will be held nearby. In our system we assume that a user only wants to attend the talks located within five miles from his office. If the user could attend the talk, the personal agent will inform the discovery agent and connect the calendar agent to update the user's timetable.

**Talk discovery agent**

Schema *Talk* is defined for a general talk type. The *interested_subjects* records the interested subjects for the users.

*Talk*

$$\begin{array}{l} place : PLACE \\ dt : DateTime \\ subject : \mathbb{P}\, SUBJECT \end{array}$$

$$notify ::= GO \mid NO$$

$$interested\_subjects : Person \leftrightarrow SUBJECT$$

*Discovery*

$$\begin{array}{l} users : \mathbb{P}_1\, Person \\ talkch : \mathbf{chan} \\ monitor : Talk\ \mathbf{sensor} \end{array}$$

$$
\begin{array}{|l}
\textsc{Main} \mathrel{\widehat=} \mu\, M \bullet [t : \mathit{Talk}] \bullet \mathit{monitor}?t \\
\qquad\quad \rightarrow \ ||| \ [u : \mathit{users}] \bullet \\
\qquad\qquad ([\mathit{interested\_subjects}(\!|\ \{u\}\ |\!) \cap t.\mathit{subject} \neq \varnothing] \bullet \\
\qquad\qquad\quad \mathit{talkch}!(u, t) \rightarrow [\mathit{response} : \mathit{notify}] \bullet \\
\qquad\qquad\qquad \mathit{talkch}?(u, \mathit{response}) \rightarrow \mathit{Skip} \\
\qquad\qquad \Box\ [\mathit{interested\_subjects}(\!|\ \{u\}\ |\!) \cap t.\mathit{subject} = \varnothing] \bullet \mathit{Skip});\ M \\
\hline
\end{array}
$$

The talk discovery system senses market updates, finding new talks information. Once a new talk is found, it sends a notification to all the users who may be interested.

A number of instances can be created also.

$$
\begin{array}{|l}
\mathit{National\_University\_Singapore} : \mathit{Place} \\
\mathit{atalk} : \mathit{Talk} \\
\hline
\mathit{atalk.place} = \mathit{National\_University\_Singapore} \\
\ldots
\end{array}
$$

## 6.3 Extracting Web ontology from the TCOZ model

It is important to have a thoroughly designed ontology since it will be shared by different agents and it forms the foundation of all agents' service. However designing a clear and consistent ontology is not a trivial job. It is useful to have some tool support in designing the ontology.

In this section, we will demonstrate the development of an XSL [93] program to automatically extract the ontology related domain properties from the static aspects of TCOZ formal models (encoded in ZML format [85]). The ontology for the system

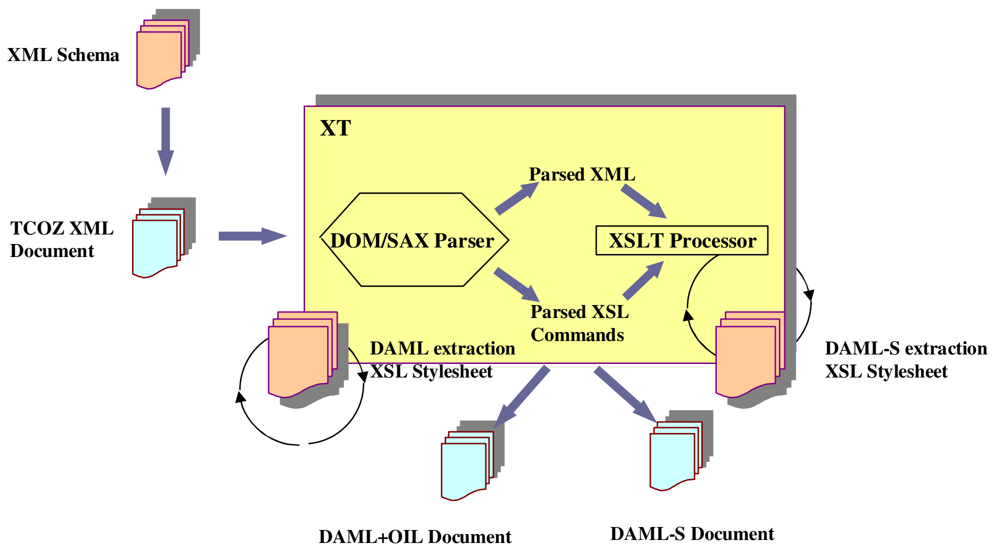


Figure 6.1: TCOZ DAML+OIL/DAML-S projection

can be resolved readily from the static parts of TCOZ design documents. In the next section, we will demonstrate tools to automatically extract the semantic markup for service from dynamic aspects of TCOZ formal models.

ZML (details have been presented in Chapter 3) is an XML environment for Z family notations (Z/Object-Z/TCOZ). It encodes the Z family documents in XML format so that the formal model can be easily browsed by the Web browser (e.g. Internet Explorer). The eXtensible Stylesheet Language (XSL) [93] is a stylesheet language to describe rules for matching and translating XML documents. In our case we translate the ZML to DAML+OIL and DAML-S. The main process and techniques for the translation are depicted by Figure 6.1.

A set of translation rules translating from TCOZ model (in ZML) to DAML+OIL ontology is developed in the following presentation.

### 6.3.1 Given type translation

The given types in the TCOZ model are directly translated into DAML+OIL classes. This rule is applicable to the given types defined in both inside and outside of a class definition. The translation can be expressed as the following rule:

$$\frac{[T]}{T \in daml\_class}$$

For example, the given type *TIME* can be translated into a class in DAML+OIL with *time* as ID.

```
<daml:Class rdf:ID="time">
  <rdfs:label>TIME</rdfs:label>
</daml:Class>
```

### 6.3.2 Axiomatic (Function and Relation) definition translation

The translation from functions and relations in TCOZ to DAML+OIL ontology requires several cases.

$$\frac{\begin{array}{|l} R : B \leftrightarrow (\rightarrow, \nrightarrow) C \\ \hline \ldots \end{array} \qquad B, C \in daml\_class}{R \in daml\_objectproperty[B \leftrightarrow (\rightarrow, \nrightarrow) C]}$$

The relation $R$ will be translated into a DAML+OIL property with $B$ as the domain class and $C$ as the range class. For total functions we restrict the $daml : cardinality$ property to be one and for partial functions we restrict the $daml : maxCardinality$ property to be one.

In our talk discovery case study, the relation *interested_subjects* can be translated into DAML+OIL as:

```
<daml:ObjectProperty rdf:ID="interested_subjects">
  <rdfs:domain rdf:resource="#person"/>
  <rdfs:range rdf:resource="#subject"/>
</daml:ObjectProperty>
```

### 6.3.3 Z Axiomatic (Subset and Constant) definition translation

**Subset:** In this situation, if $N$ corresponds to a DAML+OIL class, then $M$ will be translated into a DAML+OIL subclass of $N$. If $N$ corresponds to a DAML+OIL property, then $M$ will be translated into a DAML+OIL subproperty of $N$. The translation rules for the subset are:

$$\frac{\begin{array}{|l} M : \mathbb{P}\, N \\ \hline ... \end{array} \qquad N \in daml\_class}{M \in daml\_subclass[N]} \qquad \frac{\begin{array}{|l} M : \mathbb{P}\, N \\ \hline ... \end{array} \qquad N \in daml\_objectproperty}{M \in daml\_subproperty[N]}$$

**Constant:** In this situation, $X$ will be translated into an instance of $Y$. The following is the translation rule:

$$\frac{\begin{array}{|l} x : Y \\ \hline ... \end{array} \qquad Y \in daml\_class}{x \in instantceof[Y]}$$

For example, the *National_University_Singapore* and *atalk* defined in a previous section can be translated to

```
<place rdf:ID="National_University_Singapore"/>
<talk rdf:ID="atalk">
   <rdfs:label>atalk</rdfs:label>
   <talk_place  rdf:resource="#National_University_Singapore"/>
    ...
</talk>
```

### 6.3.4 Z state schema translation

A Z state schema can be translated into a DAML+OIL class. Its attributes are translated into DAML+OIL properties with the schema name as domain DAML+OIL class and the Z type declaration as range DAML+OIL class. In order to resolve the name conflict between same attribute names used in different schemas, we use the schema name appended with attribute name as the ID for the DAML+OIL property.

$$\begin{array}{|l} \multicolumn{1}{l}{S} \\ \hline x : T_1; \quad y : \mathbb{P}\, T_2 \\ \hline \ldots \\ \hline \end{array} \qquad T_1, T_2 \in \mathit{daml\_class}$$

---

$$\begin{array}{l} S\_y \in \mathit{daml\_objectproperty}[S \leftrightarrow T_2] \\ S \in \mathit{daml\_class},\ S\_x \in \mathit{daml\_objectproperty}[S \rightarrow T_1], \end{array}$$

For example the *Talk* schema defined in a previous section can be translated to DAML+OIL as:

```
<daml:Class rdf:ID="talk">
  <rdfs:label>Talk</rdfs:label>
</daml:Class>
<daml:ObjectProperty rdf:ID="talk_place">
  <rdf:type rdf:resource="
    http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdf:domain rdf:resource="#talk"/>
  <rdf:range rdf:resource="#place"/>
</daml:ObjectProperty>  ...
<daml:ObjectProperty rdf:ID="talk_subject">
  <rdf:domain rdf:resource="#talk"/>
  <rdf:range rdf:resource="#subject"/>
</daml:ObjectProperty>
```

### 6.3.5 Class translation

An Object-Z class can be translated into a DAML+OIL class. Its attributes defined in state schema are translated into DAML+OIL properties with the class name as domain DAML+OIL class and the type declaration as range DAML+OIL class. Other translation details are similar to the Z state schema translation defined above.

$C$ $\qquad\qquad$ $T_1, T_2 \in daml\_class$

$x : T_1;\ \ y : \mathbb{P}\, T_2$

...

...

---

$C\_y \in daml\_objectproperty[C \leftrightarrow T_2]$
$C \in daml\_class,\ \ C\_x \in daml\_objectproperty[C \rightarrow T_1],$

For example the *Person* class defined in a previous section can be translated to DAML+OIL as:

```
<daml:Class rdf:ID="person">
  <rdfs:label>Person</rdfs:label>
</daml:Class>
<daml:ObjectProperty rdf:ID="person_id">
  <rdf:type rdf:resource="
     http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdf:domain rdf:resource="#person"/>
  <rdf:range rdf:resource="#PID"/>
</daml:ObjectProperty>
 ...
```

Other translation rules are omitted as the aim of this thesis is to demonstrate the approach rather than providing the complete XSL program design.

## 6.4 Extracting DAML-S ontology from the TCOZ model

In the previous two sections we demonstrated how TCOZ can be used to capture the requirement of Semantic Web applications and how to project TCOZ models to DAML+OIL ontology automatically. DAML+OIL ontology is used to define the common understanding for certain concepts. The dynamic aspects of Semantic Web service, which define what is the service done and how it behaves is also crucial. Recently, DAML-S [12] emerges to define such information for SW services. Extracting the semantic markup information (i.e. DAML-S) for a Semantic Web service from the formal requirement model is another important research work. In this section, we will demonstrate the development of another XSL program to automatically extract DAML-S information from TCOZ formal models. The semantic markup for the system can be resolved from the TCOZ design documents also.

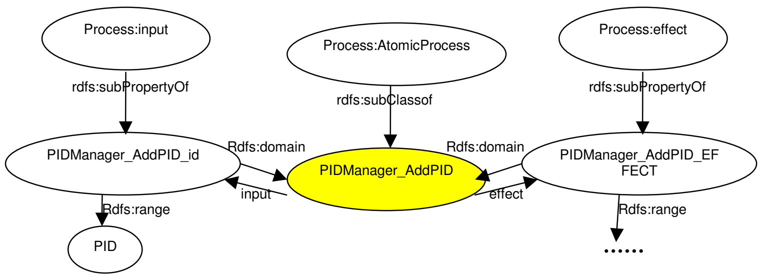


Figure 6.2: The DAML-S process ontology for *AddID* service

### 6.4.1 Translation rules

A set of translation rules translating from TCOZ model to DAML-S semantic markup for Semantic Web services are developed in the following:

**Basic rule 1 (R1):**

Each operation in TCOZ is modelled as a process (AtomicProcess or CompositeProcess) in DAML-S. In TCOZ, operations are discrete processes which specify the computation behavior and interaction behaviors. From a dynamic view, the state of an object is subject to change from time to time according to its interaction behavior, which is defined by operation definitions. At the same time the service process allows one to effect some action or change in the world. The connection between operations in TCOZ and service process in Semantic Web services is obvious. In order to resolve the name conflict between the same operation names used in different classes we use the class name appended with operation name as the ID for the process.

**Basic rule 2 (R2):**

In the case that an operation that invokes no other operations, the operation is translated as an AtomicProcess.

```
┌ C ─────────────────────────
│ ┌ O ───────────────────────
│ │ ...
│ ├──────
│ │ pre(O)
│ │ post(O)
│ └──────────────────────────
└────────────────────────────
```

[pre(O) is a precondition of the operation O. post(O) is a postcondition of the operation O.]

$C\_O \in damls\_AtomicProcess \land C\_O\_pre(O) \in damls\_precondition$

$\land\ C\_O\_post(O) \in damls\_effect$

A precondition appearing in a TCOZ operation schema definition is modelled as *precondition* in the respective service process. A postcondition appearing in a TCOZ operation schema definition is modelled as *effect* in the respective service process.

**Basic rule 3 (R3):**

```
┌ C ─────────────────────────
│ ┌ O ───────────────────────
│ │ i? : T
│ │ o! : T
│ │ ...
│ └──────────────────────────
└────────────────────────────
```

$C\_O\_i \in damls\_input \land C\_O\_o \in damls\_output$

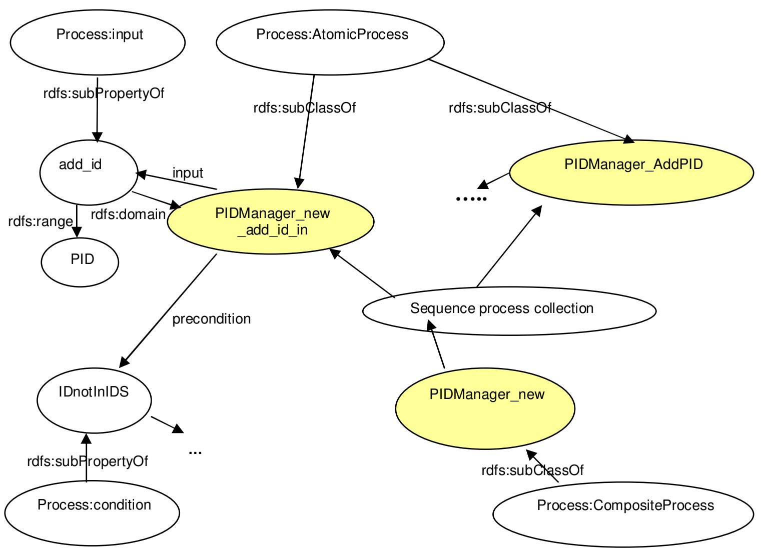


Figure 6.3: The DAML-S process ontology for *New* service

An input appearing in a TCOZ operation schema definition is modelled as *input* in the respective service process. An output appearing in a TCOZ operation schema definition is modelled as *output* in the respective service process.

**Basic rule 4 (R4):**

$C$

$\mathbf{O} \mathrel{\widehat=} \ldots$

---

$C\_O \in damls\_CompositeProcess$

In the case that an operation calls other operations, the operation is translated as a composite process.

**Basic rule 5 (R5):**

Communication in TCOZ is modelled as an atomic process with input or output.

$$\begin{array}{|l} \hline C \\ O \mathrel{\widehat{=}} ...\mathbf{Ch?i} \rightarrow ... \\ \hline \end{array}$$

---

$$C\_O\_Ch \in \mathit{damls\_AtomicProcess} \wedge C\_O\_Ch\_i \in \mathit{damls\_input}$$

$$\begin{array}{|l} \hline C \\ O \mathrel{\widehat{=}} ...\mathbf{Ch!o} \rightarrow ... \\ \hline \end{array}$$

---

$$C\_O\_Ch \in \mathit{damls\_AtomicProcess} \wedge C\_O\_Ch\_o \in \mathit{damls\_output}$$

In DAML-S, atomic processes, in addition to specifying the basic actions from which larger processes are composed, can also be thought of as the communication primitives of an (abstract) process specification.

**Basic rule 6 (R6):**

Each TCOZ process primitive will be translated into the proper DAML-S composite process. For example, the following two rules show how the translation is done for the sequential and parallel processes in TCOZ.

$$\begin{array}{|l} \hline C \\ O \mathrel{\widehat{=}} P_1;\ P_2 \\ \hline \end{array} \qquad [P_1 \text{ and } P_2 \text{ are process components}]$$

---

$$C\_O \in \mathit{damls\_Sequence}[P_1, P_2]$$

$C$

$O \mathrel{\widehat=} P_1 \parallel P_2$ [$P_1$ and $P_2$ are process components]

---

$C\_O \in \mathit{damls\_Split}[P_1, P_2]$

Other translation rules for process primitive are omitted due to the limited space.

**Basic rule 7 (R7):**

$C$

$O \mathrel{\widehat=} [G]..$

---

$C\_O\_G \in \mathit{damls\_precondition}$

The guards in TCOZ model are used to control the input of an operation. The guards are modelled as preconditions.

Other translation rules are omitted as the aim of this thesis is to demonstrate the approach rather than providing the complete XSL program design.

### 6.4.2 Case study

The *PIDManager* class defined for the Calendar agent will be used to demonstrate the translation. The *PIDManager* class has five operations, *AddPID*, *RemovePID*, *New*, *Delete* and *Validate*. Each of them will be translated into a *process*.

The operation *AddPID* is an operation invokes no other operations, so it will be translated as an AtomicProcess (R2). Some standard header information is generated first.

```
<!--Header Information-->
<?xml version='1.0' encoding='ISO-8859-1'?>
<rdf:RDF xmlns:rdf = "&rdf;#" ...>
<daml:Ontology rdf:about="">
  <daml:imports rdf:resource="&daml;"/>
  <daml:imports rdf:resource="&service;"/>
  <daml:imports rdf:resource="&process;"/>
  ....
</daml:Ontology>
<!--PIDmanager AddPId process-->
<daml:Class rdf:ID="PIDManager_AddPID">
 <rdfs:subClassOf
   rdf:resource ="&process;#AtomicProcess"/>
 <rdfs:subClassOf>
   <daml:Restriction daml:cardinality="1">
     <daml:onProperty rdf:resource="#AddPID_id"/>
   </daml:Restriction>
 </rdfs:subClassOf>
 ...
</daml:Class>
```

Figure 6.2 shows the semantic markup for service *AddID* in the graphical format. The DAML-S code in RDF format can be found in Appendix E.

The operation *AddPID* has one input *id*? declared to be type *PID*. It will be translated into *input* (*PIDManager_AddPID_id*) in DAML-S (R3).

The operation *AddPID* has one predicate $ids' = ids \cup \{id?\}$ which is a postcondition. It will be translated into *effect* (*PIDManager_AddPID_EFFECT*) in DAML-S (R2). The operation *RemovePID* can be translated similarly.

The operation *New* calls the other operation *AddPID*, so it is translated as a composite process (R4). It performs two subprocesses *PIDManager_AddPID_add_id_in* and *PIDManager_AddPID* in sequence. The *PIDManager_AddPID_add_id_in* process represents the communication on channel *add* (R5). The guard of the operation is translated as the precondition (*IDnotInIDS*)(R7). Figure 6.3 shows the semantic markup DAML-S for the operation *New*.

The operation *Delete* and *Valide* can be similarly translated.

## 6.5 Chapter summary

In this chapter, we demonstrate that TCOZ can be used as a high level design language for modelling the SW services ontology and functionalities. Another major contribution of this chapter is that it develops systematic transformation rules and tools which can automatically project TCOZ models to DAML+OIL ontology and DAML-S semantic markup.

# Chapter 7

# Conclusions and directions for further research

This chapter summarizes the main contributions of the thesis and discusses possible directions for further research.

## 7.1 Thesis main contributions and influence

The content of the thesis demonstrate the latest investigations on the links between Semantic Web and formal methods. It shows that these two communities can benefit from each other in different ways.

- This thesis developed a Web environment for the Z family languages based on XML/XSL transformation. The ZML Web environment provides a feasible means of constructing, displaying and resource sharing formal specification models on the web. It includes the auto type referencing, static syntax checking and browsing facilities such as the Z schema calculus and Object-Z/TCOZ inheritance expansions. As part of the Community Z Tools (CZT) initiative [58] project, this work is continuing with the definition of a standard markup language [90] for the ISO Z standard [1]. Hopefully it will become part of the ISO Z standard in the future. This will also make an impact on formal methods education through the internet.
- This thesis also developed a Semantic Web (RDF/DAML) environment for supporting, extending and integrating many different formalisms. Such a *meta integrator* may bring together the strengths of various formal methods communities in a flexible and widely accessible fashion. This Semantic Web environment for formal specifications also leads to many benefits. One novel application which

has been demonstrated in this thesis is the notion of specification comprehension based RDF query techniques.

- This thesis presented an Alloy semantic models for DAML+OIL and the systematic transformation rules and (XSLT) program which can translate DAML+OIL ontology to Alloy automatically. With the assistance of Alloy Analyzer (AA), we demonstrated that the consistency of the SW ontology can be checked automatically and different kinds of reasoning tasks can be supported. This work also forms the starting point for applying other formal methods and tools i.e. Z [17] and HOL/Isabelle [88].

In summary, as demonstrated above there is a clear synergy between SW languages and formal specifications. The investigation into links between those two paradigms will lead to great benefits for both areas.

## 7.2 Directions for further research

The following topics, arising out of this thesis, seem worthy of further research.

### 7.2.1 Enrich the Semantic Web environment for different formalisms

In Chapter 4 we use Z [98] and CSP [38] as examples to demonstrate how an extendable and flexible Semantic Web environment for formal specification languages can be developed. This environment can be further enriched by including some other formalisms like $\pi$-calculus, B, and VDM.

### 7.2.2 Analysis/Reasoning about Semantic Web ontology using other formal tools

In Chapter 5 we demonstrate how the Alloy Analyzer could be used to reason about Semantic Web ontology. Recently we investigated how other formal tools like theorem provers, e.g. HOL/Isabelle [66] or Z/EVES [73] can be used to reason about Semantic Web ontology.

There are some pros and cons between these different approaches. Being a model checker liked tool, reasoning in Alloy is fully automated and if there is an inconsistency, Alloy can give a counter example so that it is easier to trace the origin of the inconsistency. On the other hand, Alloy is not very scalable. Since it performs exhaustive search, it can only handle ontologies with no more than twenty entities. Moreover, Alloy does not support concrete domains such as integer, etc.

These characteristics make Alloy more automated, but less powerful and expressive than Z/EVES. Compared to Semantic Web-specific reasoning tools and Alloy, the apparent disadvantage of theorem prover approach, i.e. Z/EVES or HOL/Isabelle, is that it has a lower degree of automation and can only perform reasoning tasks interactively. However, the high degree of expressiveness of Z language or HOL implies that it can capture properties beyond ontology languages and applying theorem prover to checking ontologies gives us more confidence in the correctness of ontology related properties.

The new ontology language OWL Full is designed to be very expressive and reasoning will be generally undecidable [15]. As a result, proof process will be inevitably interactive. Therefore, theorem prover is a natural choice for reasoning over OWL language. Extending the support to OWL will be one of the future work directions.

### 7.2.3 Analysis/Reasoning about DAML-S using formal tools

Following the same motivation of reasoning about DAML+OIL by Alloy in chpater 5, we believe that if the DAML-S languages can be transferred into some formal language like CSP, TCOZ, SPIN [30] or PORMELA [70], then the formal tools can be used to provide automatic reasoning and consistency checking services for DAML-S. Strictly speaking, DAML-S is one kind of process algebra language, just like CSP. Reasoning about DAML-S through CSP verification tools like FDR [51] seems to be a natural attempt. FDR (Failures-Divergence Refinement) is a model-checking tool for CSP. Its method of establishing whether a property holds is to test for the refinement of a transition system capturing the property by the candidate machine. There is also the ability to check determinism of a state machine. Reasoning about DAML-S will be a novel application domain for FDR.

### 7.2.4 Time extension for DAML-S

When we develop the rule and tools extracting the semantic markup (DAML-S) for a Web Service from the TCOZ model in Chapter 6, one observation is that some time related features of TCOZ can not be expressed in DAML-S. The reason is that the time issue has not been included in the current version of DAML-S. However people do realize that temporal concept is crucial for the Web Service semantic markup [12] and some incipient research on temporal ontology are undergoing. In this section we propose a timed extension for DAML-S process ontology. Two more process flow control constructors *Timeout* and $Timed - interrupt$ were defined.

**Timeout:** The semantics is that the process tries to invoke $P$ first but, if unable to make it within *outtime* time, process evokes $Q$.

**Timed-interrupt:** The semantics is that the process invokes $P$ first but, if unable to finish it within *outtime* time, process evokes $Q$.

The definition of the *Timeout* construct is defined as:

```
 <daml:Class rdf:ID="Timeout">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#components"/>
      <daml:toClass rdf:resource="#ProcessComponentBag"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>

<daml:Property rdf:ID="outtime">
  <rdfs:domain rdf:resource="#Timeout"/>
</daml:Property>

<daml:Property rdf:ID="P">
  <rdfs:domain rdf:resource="#Timeout"/>
  <rdfs:range rdf:resource="#ProcessComponent"/>
</daml:Property>

<daml:Property rdf:ID="q">
  <rdfs:domain rdf:resource="#Timeout"/>
  <rdfs:range rdf:resource="#ProcessComponent"/>
</daml:Property>
```

### 7.2.5 Soundness proof of the translation between TCOZ and DAML+OIL, DAML-S

In Chapter 6 we presented systematic translation rules and tools which can project TCOZ models to DAML+OIL ontology and DAML-S automatically. The soundness of these translation rules can be formally proved. As an ongoing work we give a semantic foundation for these tools showing that they are defining morphisms between the logical systems underlying the three specification languages. To do that the institution is introduced for formalizing the logic underlying the specification langauge TCOZ, the Web ontology language DAML+OIL and DAML-S.

# Bibliography

[1] ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics. International Standard.

[2] K. Araki, A. Galloway, and K. Taguchi, editors. *IFM'99: Integrated Formal Methods, York, UK*. Springer-Verlag, June 1999.

[3] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. Scientific American, May 2001.

[4] J.C. Bicarregui and B. M. Matthews. Integrating EXPRESS and SGML for Document Modelling in Control Systems Design. In *EUG'95, 5th Annual EXPRESS User Group International Conference*, 1995.

[5] J. P. Bowen and D. Chippington. Z on the Web using Java. In Bowen et al. [6], pages 66–80.

[6] J. P. Bowen, A. Fett, and M. G. Hinchey, editors. *ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users, Berlin, Germany,*

*24–26 September 1998*, volume 1493 of *Lect. Notes in Comput. Sci.* Springer-Verlag, 1998.

[7] D. Brickley and R. V. Guha (editors). Resource description framework (rdf) schema specification 1.0. `http://www.w3.org/TR/2000/CR-rdf-schema-20000327/`, March, 2000.

[8] J. Broekstra, M. Klein, S. Decker, D. Fensel, and I. Horrocks. Adding Formal Semantics To The Web: Building On Top Of RDF Schema. In *ECDL Workshop on the Semantic Web: Models, Architectures and Management*, 2000.

[9] M. Butler. csp2B: A Practical Approach To Combining CSP and B. In Wing et al. [96].

[10] M. Butler, L. Petre, and K. Sere, editors. *IFM'02: Integrated Formal Methods,*, Lect. Notes in Comput. Sci. Springer-Verlag, October 2002.

[11] P. Ciancarini, C. Mascolo, and F. Vitali. Visualizing Z notation in HTML documents. In Bowen et al. [6], pages 81–95.

[12] DAML Service Coalition. Daml-s – version 0.9. `http://www.daml.org/services/daml-s/`, 2003.

[13] R. Cost, T. Finin, A. Joshi, and etc. Ittalks: A case study in the semantic web and daml. In *proceedings of the International Semantic Web Working Symposium*, July 2002.

[14] R. S. M. de Barros. Deriving relational database programs from formal specifications. In *Proceedings of FME'94: Industrial Benefit of Formal Methods*, pages 703–723, Barcelona, October 1994. Springer-Verlag.

[15] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. S. (editors). Owl web ontology language 1.0 reference. `http://www.w3.org/TR/owl-ref/`.

[16] J. S. Dong, C. H .Lee, Y. F. Li, and H. Wang. A combined approach to checking web ontologies. In *The 13th ACM International World Wide Web Conference (WWW'04)*. ACM Press, May 2004.

[17] J. S. Dong, C. H .Lee, Y. F. Li, and H. Wang. Verifying daml+oil and beyond in z/eves. In *The 26th International Conference on Software Engineering (ICSE'04)*. IEEE Press, May 2004.

[18] J. S. Dong, Y. F. Li, J. Sun, J. Sun, and H. Wang. XML-based Static Type Checking and Dynamic Visualization for TCOZ. In *The 4th International Conference on Formal Engineering Methods*. Springer-Verlag, October 2002.

[19] J. S. Dong and B. Mahony. Active Objects in TCOZ. In J. Staples, M. Hinchey, and S. Liu, editors, *the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 16–25. IEEE Press, December 1998.

[20] J. S. Dong, J. Sun, and H. Wang. Semantic Web for Extending and Linking Formalisms. In *Formal Methods Europe (FME'02 - FLoC)*, pages 587–606. Springer-Verlag, July 2002.

[21] J. S. Dong, J. Sun, and H. Wang. Z Approach to Semantic Web. In *The 4th International Conference on Formal Engineering Methods*, pages 156–167. Springer-Verlag, October 2002.

[22] J. S. Dong, J. Sun, and H. Wang. Checking and Reasoning about Semantic Web through Alloy. In *12th Internation Symposium on Formal Methods Europe (FM'03)*. Springer-Verlag, September 2003.

[23] J. S. Dong, J. Sun, H. Wang, C. H. Lee, and H. B. Lee. Analysing Web Ontology in Alloy: A Military Case Study. In *The 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, pages 542–547, San Francisco, USA, July 2003.

[24] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan, March 2000.

[25] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[26] M. Ghallab et. al. PDDL-The Planning Domain Definition Language V. 2. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[27] R. Fikes and D. L. McGuinness. An axiomatic semantics for rdf, rdf schema, and daml+oil. Technical Report KSL-01-01, Knowledge Systems Laboratory, 2001.

[28] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, MD, USA, 1994. ACM Press.

[29] C. Fischer and H. Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In Araki et al. [2].

[30] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[31] A. J. Galloway and W. J. Stoddart. An operational semantics for ZCCS. In Hinchey and Liu [37], pages 272–282.

[32] E. Goldberg and Y. Novikov. Verification of Proofs of Unsatisfiability for CNF Formulas. In *Proc. Design, Automation and Test in Europe: DATE'03*, Munich, Germany, March 2003. IEEE Press.

[33] W. Grieskamp, T. Santen, and B. Stoddart, editors. *IFM'00: Integrated Formal Methods,*, Lect. Notes in Comput. Sci. Springer-Verlag, October 2000.

[34] I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, 2nd edition, 1993.

[35] I. J. Hayes, editor. *Specification Case Studies.* Series in Computer Science. Prentice Hall, 1987.

[36] I. J. Hayes and B. P. Mahony. Using units of measurement in formal specifications. *Formal Aspects of Computing*, 7(3), 1995.

[37] M. Hinchey and S. Liu, editors. *the IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, Hiroshima, Japan, November 1997. IEEE Press.

[38] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985.

[39] I. Horrocks. DAML+OIL: a description logic for the semantic web. *Bull. of the IEEE Computer Society Technical Committee on Data Engineering*, 25(1):4–9, March 2002.

[40] I. Horrocks, D. McGuinness, and C. Welty. Digital libraries and web-based information systems. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 427–449. Cambridge University Press, 2003.

[41] I. Horrocks. The FaCT System. *Tableaux'98, Lecture Notes in Computer Science*, 1397:307–312, 1998.

[42] Intellidimension Inc. Rdfql reference manual. `http://www.intellidimension.com/RDFGateway/Docs/rdfqlmanual.asp`, 2001.

[43] International Organization for Standardization, Geneva. *Units of Measurement: Handbook on International Standards for Units of Measurement*, 1979.

[44] D. Jackson. Micromodels of software: Lightweight modelling and analysis with alloy. Available: http://sdg.lcs.mit.edu/alloy/book.pdf, 2002.

[45] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering: ICSE'2000*, pages 730–733, Limerick, Ireland, June 2000. ACM Press.

[46] G. Kaiser, S. Dossick, W. Jiang, and J. Yang. An Architecture for WWW-based Hypercode Environments. In R. Adrion, A. Fuggetta, and R. Taylor, editors,

*The 19th International Conference on Software Engineering (ICSE'97)*, pages 3–13, Boston, USA, 1997. IEEE Press.

[47] O. Lassila and R. R. Swick (editors). Resource description framework (rdf) model and syntax specification. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`, Feb, 1999.

[48] C. H. Lee. Phase I Report for Plan Ontology. DSO National Labs, Singapore, 2002.

[49] H. J. Levesque, R. Reiter, Y. Lesperance, F. Z. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[50] S. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering*, 24(1), January 1998.

[51] Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR2 User Manual. `http://www.fsel.com/documentation/fdr2/html/index.html`, May 2000.

[52] B. Mahony and J. S. Dong. Network Topology and a Case Study in TCOZ. In J. Bowen, A. Fett, and M. Hinchey, editors, *The 11th International Conference*

*of Z Users*, volume 1493 of *Lecture Notes in Computer Science*, pages 308–327, Berlin, Germany, September 1998. Springer-Verlag.

[53] B. Mahony and J. S. Dong. Overview of the semantics of TCOZ. In Araki et al. [2], pages 66–85.

[54] B. Mahony and J. S. Dong. Sensors and Actuators in TCOZ. In Wing et al. [96], pages 1166–1185.

[55] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.

[56] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Kyoto, Japan, April 1998. IEEE Press.

[57] A. Martin and A. Simpson. Generalising the Z Schema Calculus: Database Schema and Beyond. In *The 10th Asia-Pacific Software Engineering Conference (APSEC'03)*, pages 28–48. IEEE Press, 2003.

[58] A. P. Martin. Community Z Tools Initiative, 2001. http://web.comlab.ox.ac.uk/oucl/work/andrew.martin/CZT/.

[59] D. Martin, A. Cheyer, and D. Moran. The Open Agent Architecture: a framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1/2):91–128, 1999.

[60] C. Mascolo, W. Emmerich, and A. Finkelstein. XML technologies and software engineering. In *International Conference on Software Engineering*, pages 775–776, 2001.

[61] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[62] R. Milner. *Communicating with Mobile Agents: The pi-Calculus.* Cambridge University Press, 1999.

[63] M. Minsky. A framework to represent knowledge. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, 1975.

[64] S. Narayanan. Reasoning about actions in narrative understanding. In *International Joint Conference on Artificial Intelligence (IJCAI'1999)*, pages 350–357. Morgan Kaufmann Press, 1999.

[65] M. Nielsen, K. Havelund, R. Wagner, and C. George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1:85–114, 1989.

[66] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[67] Formal Specification Research (NUS). Z notation in LaTeX to/from XML translator, April 2003. `http://www-appn.comp.nus.edu.sg/~rpfm/LTFZ/`.

[68] R. Paige. Formal method integration via heterogeneous notations. PhD Dissertation, University of Toronto, 1997.

[69] R. Paige. A meta-method for formal method integration. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313, pages 473–494. Springer-Verlag, 1997.

[70] P. Pettersson and K. G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[71] M. R. Quillian. Semantic memory. In Marvin, editor, *Semantic Information Processing*, pages 227–270. The MIT, 1968.

[72] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Languauge Reference Manual*. Addison-Wesley, 1999.

[73] M. Saaltink. The Z/EVES system. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer-Verlag, 1997.

[74] C. Schlenoff and etc. The Process Specification Language (PSL): Overview and Version 1.0 Specification. Technical Report NISTIR 6459, National Institute of Standards and Technology, Gaithersburg, MD, 2000.

[75] M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26, 1991.

[76] S. Schneider and J. Davies. A brief history of Timed CSP. *Theoretical Computer Science*, 138, 1995.

[77] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe. Timed CSP: Theory and Practice. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lect. Notes in Comput. Sci.*, pages 640–675. Springer-Verlag, 1992.

[78] I. Shlyakhter, R. Seater, M. Sridharan, D. Jackson, and M. Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Proc. 18th IEEE International Conference on Automated Software Engineering: ASE'03*. IEEE Press, 2003.

[79] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[80] G. Smith. *The Object-Z Specification Language.* Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[81] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in System Design*, 18:249–284, 2001.

[82] J. M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, 1989.

[83] J. Sun. *Tools And Verification Techniques For Integrated Formal Methods.* PhD thesis, National University of Singapore, 2003.

[84] J. Sun, J. S. Dong, J. Liu, and H. Wang. A XML/XSL Approach to Visualize and Animate TCOZ. In J. He, Y. Li, and G. Lowe, editors, *The 8th Asia-Pacific Software Engineering Conference (APSEC'01)*, pages 453–460. IEEE Press, 2001.

[85] J. Sun, J. S. Dong, J. Liu, and H. Wang. Object-Z Web Environment and Projections to UML. In *WWW-10: 10th International World Wide Web Conference*, pages 725–734. ACM Press, May 2001.

[86] J. Sun, J. S. Dong, J. Liu, and H. Wang. A Formal Object Approach to the Design of ZML. *Annals of Software Engineering*, 13(1-4):329–356, June 2002.

[87] K. Taguchi and K. Araki. The State-Based CCS Semantics for Concurrent Z Specification. In Hinchey and Liu [37], pages 283–292.

[88] Y. Tang. Reasoning about semantic web in isabelle/hol. Master Thesis, Nov, 2003.

[89] H. Treharne and S. Schneider. Using a Process Algebra to Control B OPERATIONS. In Araki et al. [2].

[90] M. Utting, I. Toyn, J. Sun, A. Martin, J. S. Dong, N. Daley, and D. Currie. Zml: Xml support for standard z. In *3nd International Conference of Z and B Users (ZB'03)*, LNCS. Springer, June 2003.

[91] F. van Harmelen, P. F. Patel-Schneider, and I. Horrocks (editors). Reference description of the daml+oil ontology markup language. Contributors: T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, P. Hayes, J. Heflin, J. Hendler, O. Lassila, D. McGuinness, L. A. Stein, ..., March, 2001.

[92] World Wide Web Consortium (W3C). Extensible markup language (xml). `http://www.w3.org/XML`.

[93] World Wide Web Consortium (W3C). Extensible stylesheet language (xsl). `http://www.w3.org/Style/XSL`.

[94] w3schools.com. Xsl - on the server. `http://www.w3schools.com/xsl/xsl\_server.asp`, 2000.

[95] A. Walshe. Formal methods for database language design and constraint handling. *Software Eng. Journal*, 4(1):15–24, January 1989.

[96] J. Wing, J. Woodcock, and J. Davies, editors. *FM'99: World Congress on Formal Methods*, Lect. Notes in Comput. Sci., Toulouse, France, September 1999. Springer-Verlag.

[97] J. Woodcock and A. Cavalcanti. The Steam Boiler in A Unified Theory of Z And CSP. In J. He, Y. Li, and G. Lowe, editors, *The 8th Asia-Pacific Software Engineering Conference (APSEC'01)*, pages 291–298. IEEE Press, 2001.

[98] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall International, 1996.

[99] P. Zave and M. Jackson. Where do operations come from?: A multiparadigm specification technique. *IEEE Transactions on Software Engineering*, 22(7):508–528, July 1996.

[100] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Software Engineering and Methodology*, 6(1):1–30, January 1997.

# Appendix A

# Glossary of Z Notation

This appendix presents a glossary of the Z notation used in this thesis. The glossary is based on the glossary of Z notation presented in Hayes[35] with modifications to reflect more closely the more recent Z notation of Spivey[82].

## Mathematical Notation

### Definitions and declarations

Let $x, x_k$ be identifiers and let $T$, $T_k$ be non-empty, set-valued expressions.

| | |
|---|---|
| $LHS == RHS$ | Definition of $LHS$ as syntactically equivalent to $RHS$. |
| $LHS[X_1, X_2, \ldots, X_n] == RHS$ | Generic definition of $LHS$, where $X_1, X_2, \ldots, X_n$ are variables denoting formal parameter sets. |

| | |
|---|---|
| $x : T$ | A declaration, $x : T$, introduces a new variable $x$ of type $T$. |
| $x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n$ | List of declarations. |
| $x_1, x_2, \ldots, x_n : T$ | $== x_1 : T;\ x_2 : T;\ \ldots;\ x_n : T$ |
| $[X_1, X_2, \ldots, X_n]$ | Introduction of free types named $X_1, X_2, \ldots, X_n$. |

## Logic

Let $P$, $Q$ be predicates and let $D$ be a declaration or a list of declarations.

| | |
|---|---|
| $\mathit{true}, \mathit{false}$ | Logical constants. |
| $\neg\ P$ | Negation: "not $P$". |
| $P \land Q$ | Conjunction: "$P$ and $Q$". |
| $P \lor Q$ | Disjunction: "$P$ or $Q$ or both". |
| $P \Rightarrow Q$ | $== (\neg\ P) \lor Q$<br>Implication: "$P$ implies $Q$" or "if $P$ then $Q$". |
| $P \Leftrightarrow Q$ | $== (P \Rightarrow Q) \land (Q \Rightarrow P)$<br>Equivalence: "$P$ is logically equivalent to $Q$". |

| | |
|---|---|
| $\forall\, x : T \bullet P$ | Universal quantification: "for all $x$ of type $T$, $P$ holds". |
| $\exists\, x : T \bullet P$ | Existential quantification: "there exists an $x$ of type $T$ such that $P$ holds". |
| $\exists_1\, x : T \bullet P$ | Unique existence: "there exists a unique $x$ of type $T$ such that $P$ holds". |
| $\forall\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$ | "For all $x_1$ of type $T_1$, $x_2$ of type $T_2$, …, and $x_n$ of type $T_n$, $P$ holds." |
| $\exists\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$ | Similar to $\forall$. |
| $\exists_1\, x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \bullet P$ | Similar to $\forall$. |
| $\forall\, D \mid P \bullet Q$ | $\Leftrightarrow \forall\, D \bullet P \Rightarrow Q$ |
| $\exists\, D \mid P \bullet Q$ | $\Leftrightarrow \exists\, D \bullet P \land Q$ |
| $t_1 = t_2$ | Equality between terms. |
| $t_1 \neq t_2$ | $\Leftrightarrow \neg\,(t_1 = t_2)$ |

## Sets

Let $X$ be a set; $S$ and $T$ be subsets of $X$; $t, t_k$ terms; $P$ a predicate; and $D$ declarations.

| | |
|---|---|
| $t \in S$ | Set membership: "$t$ is a member of $S$". |
| $t \notin S$ | $\Leftrightarrow \neg\ (t \in S)$ |
| $S \subseteq T$ | $\Leftrightarrow (\forall x : S \bullet x \in T)$<br>Set inclusion. |
| $S \subset T$ | $\Leftrightarrow S \subseteq T \wedge S \neq T$<br>Strict set inclusion. |
| $\varnothing$ | The empty set. |
| $\{t_1, t_2, \ldots, t_n\}$ | The set containing the values of terms $t_1, t_2, \ldots, t_n$. |
| $\{x : T \mid P\}$ | The set containing exactly those $x$ of type $T$ for which $P$ holds. |
| $(t_1, t_2, \ldots, t_n)$ | Ordered n-tuple of $t_1, t_2, \ldots, t_n$. |
| $T_1 \times T_2 \times \ldots \times T_n$ | Cartesian product: the set of all n-tuples such that the $k$th component is of type $T_k$. |

| | |
|---|---|
| $\mathit{first}(t_1, t_2, \ldots, t_n)$ | |
| | $== t_1$ |
| | Similarly, $\mathit{second}(t_1, t_2, \ldots, t_n) == t_2$, etc. |
| $\{x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n \mid P\}$ | |
| | The set of all n-tuples $(x_1, x_2, \ldots, x_n)$ with each $x_k$ of type $T_k$ such that $P$ holds. |
| $\{D \mid P \bullet t\}$ | The set of values of the term $t$ for the variables declared in $D$ ranging over all values for which $P$ holds. |
| $\{D \bullet t\}$ | $== \{D \mid \mathit{true} \bullet t\}$ |
| $\mathbb{P}\, S$ | Powerset: the set of all subsets of $S$. |
| $\mathbb{P}_1\, S$ | $== \mathbb{P}\, S \setminus \{\varnothing\}$ |
| | The set of all non-empty subsets of $S$. |
| $\mathbb{F}\, S$ | $== \{T : \mathbb{P}\, S \mid T \text{ is finite }\}$ |
| | Set of finite subsets of $S$. |
| $\mathbb{F}_1\, S$ | $== \mathbb{F}\, S \setminus \{\varnothing\}$ |
| | Set of finite non-empty subsets of $S$. |
| $S \cap T$ | $== \{x : X \mid x \in S \land x \in T\}$ |
| | Set intersection. |

| | |
|---|---|
| $S \cup T$ | $== \{x : X \mid x \in S \lor x \in T\}$<br>Set union. |
| $S \setminus T$ | $== \{x : X \mid x \in S \land x \notin T\}$<br>Set difference. |
| $\bigcap SS$ | $== \{x : X \mid (\forall S : SS \bullet x \in S)\}$<br>Intersection of a set of sets; $SS$ is a set containing as its members subsets of $X$, i.e. $SS : \mathbb{P}(\mathbb{P}\,X)$. |
| $\bigcup SS$ | $== \{x : X \mid (\exists S : SS \bullet x \in S)\}$<br>Union of a set of sets; $SS : \mathbb{P}(\mathbb{P}\,X)$. |
| $\#S$ | Size (number of distinct members) of a finite set. |

## Numbers

| | |
|---|---|
| $\mathbb{R}$ | The set of real numbers. |
| $\mathbb{Z}$ | The set of integers (positive, zero and negative). |
| $\mathbb{N}$ | $== \{n : \mathbb{Z} \mid n \geq 0\}$<br>The set of natural numbers (non-negative integers). |
| $\mathbb{N}_1$ | $== \mathbb{N} \setminus \{0\}$<br>The set of strictly positive natural numbers. |

| | |
|---|---|
| $m\,..\,n$ | $== \{k : \mathbb{Z} \mid m \leq k \land k \leq n\}$ The set of integers between $m$ and $n$ inclusive. |
| $min\,S$ | Minimum of a set; for $S : \mathbb{P}_1\,\mathbb{Z}$, $min\,S \in S \land (\forall\, x : S \bullet x \geq min\,S)$. |
| $max\,S$ | Maximum of a set; for $S : \mathbb{P}_1\,\mathbb{Z}$, $max\,S \in S \land (\forall\, x : S \bullet x \leq max\,S)$. |

## Relations

A binary relation is modelled by a set of ordered pairs hence operators defined for sets can be used on relations. Let $X$, $Y$, and $Z$ be sets; $x : X$; $y : Y$; $S$ be a subset of $X$; $T$ be a subset of $Y$; and $R$ a relation between $X$ and $Y$.

| | |
|---|---|
| $X \leftrightarrow Y$ | $== \mathbb{P}(X \times Y)$ The set of relations between $X$ and $Y$. |
| $x\ \underline{R}\ y$ | $== (x, y) \in R$ $x$ is related by $R$ to $y$. |
| $x \mapsto y$ | $== (x, y)$ |
| $\{x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n\}$ | $== \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ The relation relating $x_1$ to $y_1$, $x_2$ to $y_2$, …, and $x_n$ to $y_n$. |

| | |
|---|---|
| $\operatorname{dom} R$ | $== \{x : X \mid (\exists\, y : Y \bullet x\ \underline{R}\ y)\}$<br>The domain of a relation: the set of $x$ components that are related to some $y$. |
| $\operatorname{ran} R$ | $== \{y : Y \mid (\exists\, x : X \bullet x\ \underline{R}\ y)\}$<br>The range of a relation: the set of $y$ components that some $x$ is related to. |
| $R_1 \fatsemi R_2$ | $== \{x : X;\ z : Z \mid (\exists\, y : Y \bullet x\, R_1\, y \land y\, R_2\, z)\}$<br>Forward relational composition; $R_1 : X \leftrightarrow Y$; $R_2 : Y \leftrightarrow Z$. |
| $R_1 \circ R_2$ | $== R_2 \fatsemi R_1$<br>Relational composition. This form is primarily used when $R_1$ and $R_2$ are functions. |
| $R^{\sim}$ | $== \{y : Y;\ x : X \mid x\ \underline{R}\ y\}$<br>Transpose of a relation $R$. |
| $\operatorname{id} S$ | $== \{x : S \bullet x \mapsto x\}$<br>Identity function on the set $S$. |
| $R^k$ | The homogeneous relation $R$ composed with itself $k$ times: given $R : X \leftrightarrow X$,<br>$R^0 = \operatorname{id} X$ and $R^{k+1} = R^k \fatsemi R$. |

| | |
|---|---|
| $R^+$ | $== \bigcup\{n : \mathbb{N}_1 \bullet R^n\}$ |
| | $= \bigcap\{Q : X \leftrightarrow X \mid R \subseteq Q \land Q \mathbin{⨾} Q \subseteq Q\}$ |
| | Transitive closure. |
| $R^*$ | $== \bigcup\{n : \mathbb{N} \bullet R^n\}$ |
| | $= \bigcap\{Q : X \leftrightarrow X \mid \mathrm{id}\, X \subseteq Q \land R \subseteq Q \land Q \mathbin{⨾} Q \subseteq Q\}$ |
| | Reflexive transitive closure. |
| $R(\!\| S \|\!)$ | $== \{y : Y \mid (\exists\, x : S \bullet x \; \underline{R} \; y)\}$ |
| | Image of the set $S$ through the relation $R$. |
| $S \lhd R$ | $== \{x : X;\; y : Y \mid x \in S \land x \; \underline{R} \; y\}$ |
| | Domain restriction: the relation $R$ with its domain restricted to the set $S$. |
| $S \mathbin{⩤} R$ | $== (X \setminus S) \lhd R$ |
| | Domain subtraction: the relation $R$ with the elements of $S$ removed from its domain. |
| $R \rhd T$ | $== \{x : X;\; y : Y \mid x \; \underline{R} \; y \land y \in T\}$ |
| | Range restriction to $T$. |
| $R \mathbin{⩥} T$ | $== R \rhd (Y \setminus T)$ |
| | Range subtraction of $T$. |

| | |
|---|---|
| $R_1 \oplus R_2$ | $== (\mathrm{dom}\, R_2 \ndres R_1) \cup R_2$<br>Overriding; $R_1, R_2 : X \leftrightarrow Y$. |

## Functions

A function is a relation with the property that each member of its domain is associated with a unique member of its range. As functions are relations, all the operators defined above for relations also apply to functions. Let $X$ and $Y$ be sets, and $T$ be a subset of $X$ (i.e. $T : \mathbb{P}\, X$).

| | |
|---|---|
| $f\ t$ | The function $f$ applied to $t$. |
| $X \pfun Y$ | $== \{f : X \leftrightarrow Y \mid (\forall\, x : \mathrm{dom}\, f \bullet (\exists_1\, y : Y \bullet x\ \underline{f}\ y))\}$<br>The set of partial functions from $X$ to $Y$. |
| $X \rightarrow Y$ | $== \{f : X \pfun Y \mid \mathrm{dom}\, f = X\}$<br>The set of total functions from $X$ to $Y$. |
| $X \pinj Y$ | $== \{f : X \pfun Y \mid (\forall\, y : \mathrm{ran}\, f \bullet (\exists_1\, x : X \bullet x\ \underline{f}\ y))\}$<br>The set of partial one-to-one functions (partial injections) from $X$ to $Y$. |
| $X \rightarrowtail Y$ | $== \{f : X \pinj Y \mid \mathrm{dom}\, f = X\}$<br>The set of total one-to-one functions (total injections) from $X$ to $Y$. |

| | |
|---|---|
| $X ⤀ Y$ | $== \{f : X ⇸ Y \mid \operatorname{ran} f = Y\}$<br>The set of partial onto functions (partial surjections) from $X$ to $Y$. |
| $X \twoheadrightarrow Y$ | $== (X ⤀ Y) \cap (X \rightarrow Y)$<br>The set of total onto functions (total surjections) from $X$ to $Y$. |
| $X ⤖ Y$ | $== (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y)$<br>The set of total one-to-one onto functions (total bijections) from $X$ to $Y$. |
| $X ⇻ Y$ | $== \{f : X ⇸ Y \mid f \in \mathbb{F}(X \times Y)\}$<br>The set of finite partial functions from $X$ to $Y$. |
| $X ⤕ Y$ | $== \{f : X ⤔ Y \mid f \in \mathbb{F}(X \times Y)\}$<br>The set of finite partial one-to-one functions from $X$ to $Y$. |
| $(\lambda\, x : X \mid P \bullet t)$ | $== \{x : X \mid P \bullet x \mapsto t\}$<br>Lambda-abstraction: the function that, given an argument $x$ of type $X$ such that $P$ holds, gives a result which is the value of the term $t$. |

$(\lambda\, x_1 : T_1;\ \ldots;\ x_n : T_n \mid P \bullet t)$

$$== \{x_1 : T_1;\ \ldots;\ x_n : T_n \mid P \bullet (x_1, \ldots, x_n) \mapsto t\}$$

| | |
|---|---|
| disjoint$[I, X]$ | $== \{S : I \rightarrow\!\!\!\!\!\!+\ \mathbb{P}\,X \mid \forall\, i, j : \operatorname{dom} S \bullet i \neq j \Rightarrow S(i) \cap S(j) = \varnothing\}$ Pairwise disjoint; where $I$ is a set and $S$ an indexed family of subsets of $X$ (i.e. $S : I \rightarrow\!\!\!\!\!\!+\ \mathbb{P}\,X$). |
| $S$ $\underline{\text{partitions}}$ $T$ | $== S \in \text{disjoint} \wedge \bigcup \operatorname{ran} S = T$ |

## Sequences

Let $X$ be a set; $A$ and $B$ be sequences with elements taken from $X$; and $a_1, \ldots, a_n$ terms of type $X$.

| | |
|---|---|
| $\operatorname{seq} X$ | $== \{A : \mathbb{N}_1 \rightarrow\!\!\!\!\!\!+\ X \mid (\exists\, n : \mathbb{N} \bullet \operatorname{dom} A = 1..n)\}$ The set of finite sequences whose elements are drawn from $X$. |
| $\operatorname{seq}_\infty X$ | $== \{A : \mathbb{N}_1 \rightarrow\!\!\!\!\!\!+\ X \mid A \in \operatorname{seq} X \vee \operatorname{dom} A = \mathbb{N}_1\}$ The set of finite and infinite sequences whose elements are drawn from $X$. |
| $\#A$ | The length of a finite sequence $A$. (This is just '#' on the set representing the sequence.) |
| $\langle\rangle$ | $== \{\}$ The empty sequence. |
| $\operatorname{seq}_1 X$ | $== \{s : \operatorname{seq} X \mid s \neq \langle\rangle\}$ The set of non-empty finite sequences. |

| | |
|---|---|
| $\langle a_1, \ldots, a_n \rangle$ | $= \{1 \mapsto a_1, \ldots, n \mapsto a_n\}$ |
| $\langle a_1, \ldots, a_n \rangle \frown \langle b_1, \ldots, b_m \rangle$ | $= \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$<br>Concatenation.<br>$\langle\rangle \frown A = A \frown \langle\rangle = A.$ |
| $\mathit{head}\ A$ | The first element of a non-empty sequence:<br>$A \neq \langle\rangle \Rightarrow \mathit{head}\ A = A(1).$ |
| $\mathit{tail}\ A$ | All but the head of a non-empty sequence:<br>$\mathit{tail}\ (\langle x \rangle \frown A) = A.$ |
| $\mathit{last}\ A$ | The final element of a non-empty finite sequence:<br>$A \neq \langle\rangle \Rightarrow \mathit{last}\ A = A(\#A).$ |
| $\mathit{front}\ A$ | All but the last of a non-empty finite sequence:<br>$\mathit{front}\ (A \frown \langle x \rangle) = A.$ |
| $\mathit{rev}\ \langle a_1, a_2, \ldots, a_n \rangle$ | $= \langle a_n, \ldots, a_2, a_1 \rangle$<br>Reverse of a finite sequence; $\mathit{rev}\ \langle\rangle = \langle\rangle$. |
| $\frown / AA$ | $= AA(1) \frown \ldots \frown AA(\#AA)$<br>Distributed concatenation; where $AA : \mathrm{seq}(\mathrm{seq}(X))$. $\frown/\langle\rangle = \langle\rangle$. |

| | |
|---|---|
| $A \subseteq B$ | $\Leftrightarrow \exists\, C : \mathrm{seq}_{\infty} X \bullet A \frown C = B$ <br> $A$ is a prefix of $B$. (This is just '$\subseteq$' on the sets representing the sequences.) |
| $\mathrm{squash}\, f$ | Convert a finite function, $f : \mathbb{N} \mathrel{+\!\!\!\!\twoheadrightarrow} X$, into a sequence by squashing its domain. That is, $\mathrm{squash}\{\} = \langle\rangle$, and if $f \neq \{\}$ then $\mathrm{squash}\, f = \langle f(i)\rangle \frown \mathrm{squash}(\{i\} \lhd f)$, where $i = min(\mathrm{dom}\, f)$. For example, $\mathrm{squash}\{2 \mapsto A, 27 \mapsto C, 4 \mapsto B\} = \langle A, B, C\rangle$. |
| $A \upharpoonright T$ | $== \mathrm{squash}(A \rhd T)$ <br> Restrict the range of the sequence $A$ to the set $T$. |

## Bags

| | |
|---|---|
| $\mathrm{bag}\, X$ | $== X \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{N}_1$ <br> The set of bags whose elements are drawn from $X$. A bag is represented by a function that maps each element in the bag onto its frequency of occurrence in the bag. |
| $[\![\,]\!]$ | The empty bag $\varnothing$. |
| $[\![x_1, x_2, \ldots, x_n]\!]$ | The bag containing $x_1, x_2, \ldots, x_n$, each with the frequency that it occurs in the list. |

| | |
|---|---|
| $\mathit{items}\ s$ | $== \{x : \mathrm{ran}\, s \bullet x \mapsto \#\{i : \mathrm{dom}\, s \mid s(i) = x\}\}$ <br> The bag of items contained in the sequence $s$. |

## Axiomatic definitions

Let $D$ be a list of declarations and $P$ a predicate.

The following axiomatic definition introduces the variables in $D$ with the types as declared in $D$. These variables must satisfy the predicate $P$. The scope of the variables is the whole specification.

$$\begin{array}{|l} D \\ \hline P \end{array}$$

## Generic definitions

Let $D$ be a list of declarations, $P$ a predicate and $X_1, X_2, \ldots X_n$ variables.

The following generic definition is similar to an axiomatic definition, except that the variables introduced are generic over the sets $X_1, X_2, \ldots X_n$.

$$\begin{array}{|l} [X_1, X_2, \ldots X_n] \\ D \\ \hline P \\ \hline \end{array}$$

The declared variables must be uniquely defined by the predicate $P$.

# Schema Notation

## Schema definition

A schema groups together a set of declarations of variables and a predicate relating the variables. If the predicate is omitted it is taken to be true, i.e. the variables are not further restricted. There are two ways of writing schemas: vertically, for example,

$$\begin{array}{|l}
\hline
S \\
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
x \leq \# y \\
\hline
\end{array}$$

and horizontally, for the same example,

$$S == [x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid x \leq \# y]$$

Schemas can be used in signatures after $\forall$, $\lambda$, $\{...\}$, etc.:

$$(\forall S \bullet y \neq \langle\rangle) \Leftrightarrow (\forall x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid x \leq \# y \bullet y \neq \langle\rangle)$$

| | |
|---|---|
| $\{S\}$ | Stands for the set of objects described by schema $S$. In declarations $w : S$ is usually written as an abbreviation for $w : \{S\}$. |

## Schema operators

Let $S$ be defined as above and $w : S$.

| | |
|---|---|
| $w.x$ | $== (\lambda S \bullet x)(w)$ |
| | Projection functions: the component names of a schema may |

be used as projection (or selector) functions, e.g. $w.x$ is $w$'s $x$ component and $w.y$ is its $y$ component; of course, the predicate '$w.x \leq \# w.y$' holds.

$\theta S$ The (unordered) tuple formed from a schema's variables, e.g. $\theta S$ contains the named components $x$ and $y$.

**Compatibility** Two schemas are compatible if the declared sets of each variable common to the declaration parts of the two schemas are equal. In addition, any global variables referenced in predicate part of one of the schemas must not have the same name as a variable declared in the other schema; this restriction is to avoid global variables being *captured* by the declarations.

**Inclusion** A schema $S$ may be included within the declarations of a schema $T$, in which case the declarations of $S$ are merged with the other declarations of $T$ (variables declared in both $S$ and $T$ must have the same declared sets) and the predicates of $S$ and $T$ are conjoined. For example,

```
── T ──────────────────────────
│ S
│ z : ℕ
├──────────
│ z < x
───────────────────────────────
```

is equivalent to

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\
\hline
x, z : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
\hline
x \leq \# y \land z < x \\
\hline
\end{array}$$

The included schema ($S$) may not refer to global variables that have the same name as one of the declared variables of the including schema ($T$).

**Decoration** Decoration with subscript, superscript, prime, etc: systematic renaming of the variables declared in the schema. For example, $S'$ is

$[x' : \mathbb{N};\ y' : \operatorname{seq} \mathbb{N} \mid x' \leq \# y']$.

$\neg\, S$ The schema $S$ with its predicate part negated. For example, $\neg\, S$ is $[x : \mathbb{N};\ y : \operatorname{seq} \mathbb{N} \mid \neg\, (x \leq \# y)]$.

$S \land T$ The schema formed from schemas $S$ and $T$ by merging their declarations and conjoining (and-ing) their predicates. The two schemas must be compatible (see above).

Given $T == [x : \mathbb{N};\ z : \mathbb{P}\, \mathbb{N} \mid x \in z]$, $S \land T$ is

$$\begin{array}{|l}
\multicolumn{1}{l}{S \land T} \\
\hline
x : \mathbb{N} \\
y : \operatorname{seq} \mathbb{N} \\
z : \mathbb{P}\, \mathbb{N} \\
\hline
x \leq \# y \land x \in z \\
\hline
\end{array}$$

$S \vee T$

The schema formed from schemas $S$ and $T$ by merging their declarations and disjoining (or-ing) their predicates. The two schemas must be compatible (see above). For example, $S \vee T$ is

```
┌─ S ∨ T ──────────────────────────
│ x : ℕ
│ y : seq ℕ
│ z : ℙ ℕ
├──────────────
│ x ≤ #y ∨ x ∈ z
└──────────────────────────────────
```

$S \Rightarrow T$

The schema formed from schemas $S$ and $T$ by merging their declarations and taking 'pred $S \Rightarrow$ pred $T$' as the predicate. The two schemas must be compatible (see above). For example, $S \Rightarrow T$ is

```
┌─ S ⇒ T ──────────────────────────
│ x : ℕ
│ y : seq ℕ
│ z : ℙ ℕ
├──────────────
│ x ≤ #y ⇒ x ∈ z
└──────────────────────────────────
```

$S \Leftrightarrow T$

The schema formed from schemas $S$ and $T$ by merging their declarations and taking 'pred $S \Leftrightarrow$ pred $T$' as the predicate. The two schemas must be compatible (see above). For example, $S \Leftrightarrow T$ is

```
── $S \Leftrightarrow T$ ──────────────
│ $x : \mathbb{N}$
│ $y : \operatorname{seq} \mathbb{N}$
│ $z : \mathbb{P}\,\mathbb{N}$
├──────────
│ $x \leq \# y \Leftrightarrow x \in z$
└──────────────────────
```

$S \setminus (v_1, v_2, \ldots, v_n)$

Hiding: the schema $S$ with variables $v_1, v_2, \ldots, v_n$ hidden – the variables listed are removed from the declarations and are existentially quantified in the predicate. The parantheses may be omitted when only one variable is hidden.

$S \upharpoonright (v_1, v_2, \ldots, v_n)$

Projection: The schema $S$ with any variables that do not occur in the list $v_1, v_2, \ldots, v_n$ hidden – the variables are removed from the declarations and are existentially qualified in the predicate. For example, $(S \wedge T) \upharpoonright (x, y)$ is

```
── $(S \wedge T) \upharpoonright (x, y)$ ──────────────
│ $x : \mathbb{N}$
│ $y : \operatorname{seq} \mathbb{N}$
├──────────
│ $(\exists\, z : \mathbb{P}\,\mathbb{N} \bullet$
│       $x \leq \# y \wedge x \in z)$
└──────────────────────
```

The list of variables may be replaced by a schema; the variables declared in the schema are used for projection.

$\exists D \bullet S$

Existential quantification of a schema.

The variables declared in the schema $S$ that also appear in the declarations $D$ are removed from the declarations of $S$. The predicate of $S$ is existentially quantified over $D$. For example, $\exists x : \mathbb{N} \bullet S$ is the following schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{\exists x : \mathbb{N} \bullet S} \\ \hline
y : \operatorname{seq} \mathbb{N} \\ \hline
\exists x : \mathbb{N} \bullet \\
\qquad x \leq \# y \\ \hline
\end{array}$$

The declarations may include schemas. For example,

$$\begin{array}{|l}
\multicolumn{1}{l}{\exists S \bullet T} \\ \hline
z : \mathbb{N} \\ \hline
\exists S \bullet \\
\qquad x \leq \# y \land z < x \\ \hline
\end{array}$$

$\forall D \bullet S$

Universal quantification of a schema.

The variables declared in the schema $S$ that also appear in the declarations $D$ are removed from the declarations of $S$. The predicate of $S$ is universally quantified over $D$. For example, $\forall x : \mathbb{N} \bullet S$ is the following schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{\forall x : \mathbb{N} \bullet S} \\ \hline
y : \operatorname{seq} \mathbb{N} \\ \hline
\forall x : \mathbb{N} \bullet \\
\qquad x \leq \# y \\ \hline
\end{array}$$

The declarations may include schemas. For example,

$$\begin{array}{|l}
\multicolumn{1}{l}{\forall S \bullet T} \\
\hline
z : \mathbb{N} \\
\hline
\forall S \bullet \\
\qquad x \leq \# y \land z < x \\
\hline
\end{array}$$

## Operation schemas

The following conventions are used for variable names in those schemas which represent operations, that is, which are written as descriptions of operations on some state,

**undashed** state before the operation,

**dashed** state after the operation,

**ending in "?"** inputs to (arguments for) the operation, and

**ending in "!"** outputs from (results of) the operation.

The basename of a name is the name with all decorations removed.

| | |
|---|---|
| $\Delta S$ | $\widehat{=} S \land S'$ <br> Change of state schema: this is a default definition for $\Delta S$. In some specifications it is useful to have additional constraints on the change of state schema. In these cases $\Delta S$ can be explicitly defined. |

$\Xi S$ $\quad \widehat{=} [\Delta S \mid \theta S' = \theta S]$

No change of state schema.

## Operation schema operators

$\mathrm{pre}\, S$ Precondition: the after-state components (dashed) and the outputs (ending in "!") are hidden, e.g. given,

$$\begin{array}{|l}
\multicolumn{1}{l}{S} \\ \hline
x?, s, s', y! : \mathbb{N} \\ \hline
s' = s - x? \land y! = s' \\ \hline
\end{array}$$

$\mathrm{pre}\, S$ is,

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathrm{pre}\, S} \\ \hline
x?, s : \mathbb{N} \\ \hline
\exists\, s', y! : \mathbb{N} \bullet \\
\qquad s' = s - x? \land y! = s' \\ \hline
\end{array}$$

$S;\ T$ Schema composition: if we consider an intermediate state that is both the final state of the operation $S$ and the initial state of the operation $T$ then the composition of $S$ and $T$ is the operation which relates the initial state of $S$ to the final state of $T$ through the intermediate state. To form the composition of $S$ and $T$ we take the pairs of after-state components of $S$ and before-state components of $T$ that have the same basename, rename each pair to a new variable, take the conjunction of the resulting schemas, and hide the new variables. For example,

$S;\ T$ is,

$$\begin{array}{|l}
\hline
S;\ T \\
x?, s, s', y! : \mathbb{N} \\
\hline
(\exists\, ss : \mathbb{N} \bullet \\
\qquad ss = s - x? \land y! = ss \\
\qquad \land\, ss \leq x? \land s' = ss + x?) \\
\hline
\end{array}$$

# Appendix B

# Concrete Syntax of Object-Z

The following concrete syntax of Object-Z is an extension of the concrete syntax of Z presented by Spivey[82]. It is given in an extension to Backus-Naur Form (BNF) defined in [82]. Optional phrases are enclosed in slanted square brackets. NL denotes new line.

```
SPECIFICATION  ::=  PARAGRAPH NL . . . NL PARAGRAPH

PARAGRAPH      ::=  [IDENT, . . . , IDENT]
                —   AXIOMATICBOX
                —   SCHEMABOX
                —   GENERICBOX
                —   CLASSBOX
                —   SCHEMANAME [GENFORMALS] ≙ SCHEMAEXP
                —   CLASSNAME [GENFORMALS] ≙ CLASSREF
                —   DEFLHS == EXPRESSION
                —   PREDICATE

                      │ DECLPART
AXIOMATICBOX ::=    [ ├──────────
                      │ AXIOMPART ]

                      ┌─SCHEMANAME [GENFORMALS]──
SCHEMABOX      ::=    │ DECLPART
                    [ ├──────────
                      │ AXIOMPART ]
                      └──────────────────────────
```

| | | |
|---|---|---|
| GENERICBOX | ::= | ═ [GENFORMALS] ═══<br>DECLPART<br>[ ───<br>AXIOMPART ] |
| CLASSBOX | ::= | ─ CLASSNAME [GENFORMALS] ─<br>[LOCALDEFS]<br>[STATE]<br>[INIT]<br>[OPN<br>⋮<br>OPN]<br>[ ───<br>HISTPRED ] |
| LOCALDEFS | ::= | LOCALDEF<br>⋮<br>LOCALDEF |
| LOCALDEF | ::= | INHERITEDCLASS |
| | — | [IDENT, . . . , IDENT] |
| | — | AXIOMATICBOX |
| | — | DEFLHS == EXPRESSION |
| STATE | ::= | [ DECLPART [ \| AXIOMPART] ] |
| | — | [ AXIOMPART ] |
| | — | STATEBOX0 |
| | — | STATEBOX1 |
| STATEBOX0 | ::= | ───<br>DECLPART<br>[ ───<br>AXIOMPART ] |
| STATEBOX1 | ::= | ───<br>AXIOMPART |
| INIT | ::= | INIT $\widehat{=}$ [ AXIOMPART ] |
| | — | INITBOX |

```
InitBox      ::=    ┌─ Init ──────────────────
                    │ AxiomPart
                    └─────────────────────────

Opn          ::=  OpnName ≙ OpnExp
               —  OpnBox0
               —  OpnBox1

OpnBox0      ::=    ┌─ OpnName ───────────────
                    │ OpnDeclPart
                  [ ├──────────────
                    │ AxiomPart ]
                    └─────────────────────────

OpnBox1      ::=    ┌─ OpnName ───────────────
                    │ AxiomPart
                    └─────────────────────────

DeclPart     ::=  BasicDecl Sep ... Sep BasicDecl

OpnDeclPart  ::=  Δ(DeltaList) [Sep DeclPart]
               —  DeclPart

AxiomPart    ::=  Predicate Sep ... Sep Predicate

Sep          ::=  ;     — NL

DefLhs       ::=  VarName [GenFormals]
               —  PreGen Ident
               —  Ident InGen Ident

SchemaExp    ::=  ∀ SchemaText • SchemaExp
               —  ∃ SchemaText • SchemaExp
               —  ∃₁ SchemaText • SchemaExp
               —  SchemaExp1
```

```
SchemaExp1    ::=  [ SchemaText ]
                —  SchemaRef
                —  ¬ SchemaExp1
                —  pre SchemaExp1
                —  SchemaExp1 ∧ SchemaExp1
                —  SchemaExp1 ∨ SchemaExp1
                —  SchemaExp1 ⇒ SchemaExp1
                —  SchemaExp1 ⇔ SchemaExp1
                —  SchemaExp1 ↾ SchemaExp1
                —  SchemaExp1 \ (DeclName, . . . , DeclName)
                —  SchemaExp1 ⨾ SchemaExp1
                —  (SchemaExp)

OpnExp        ::=  ∀ SchemaText • OpnExp
                —  ∃ SchemaText • OpnExp
                —  ∃₁ SchemaText • OpnExp
                —  OpnExp1

OpnExp1       ::=  [ OpnText ]
                —  OpnRef
                —  ¬ OpnExp1
                —  pre OpnExp1
                —  OpnExp1 ∧ OpnExp1
                —  OpnExp1 ∨ OpnExp1
                —  OpnExp1 ∥ OpnExp1
                —  OpnExp1 ⇒ OpnExp1
                —  OpnExp1 ⇔ OpnExp1
                —  OpnExp1 ↾ OpnExp1
                —  OpnExp1 \ (DeclName, . . . , DeclName)
                —  OpnExp1 • OpnExp1
                —  (OpnExp)

SchemaText    ::=  Declaration [ | Predicate]

OpnText       ::=  OpnDeclaration [ | Predicate]
                —  Predicate

SchemaRef     ::=  SchemaName Decoration [GenActuals]
```

| | | |
|---|---|---|
| InheritedClass | ::= | ClassRef [RenameList] [RedefList] |
| ClassRef | ::= | ClassName [GenActuals] |
| OpnRef | ::= | OpnName |
| | — | ObjRef.OpnName |
| Declaration | ::= | BasicDecl; . . . ; BasicDecl |
| OpnDeclaration | ::= | $\Delta$(DeltaList) [; Declaration] |
| | — | Declaration |
| BasicDecl | ::= | DeclName, . . . , DeclName : Type |
| | — | SchemaRef |
| | — | OpnRef |
| Type | ::= | Expression |
| | — | ClassRef |
| | — | $\downarrow$ClassRef |
| Predicate | ::= | $\forall$ SchemaText $\bullet$ Predicate |
| | — | $\exists$ SchemaText $\bullet$ Predicate |
| | — | $\exists_1$ SchemaText $\bullet$ Predicate |
| | — | Predicate1 |
| Predicate1 | ::= | Expression Rel Expression Rel . . . Rel Expression |
| | — | PreRel Expression |
| | — | SchemaRef |
| | — | OpnRef |
| | — | ObjRef.Init |
| | — | pre SchemaRef |
| | — | pre OpnRef |
| | — | *true* |
| | — | *false* |
| | — | $\neg$ Predicate1 |
| | — | Predicate1 $\wedge$ Predicate1 |
| | — | Predicate1 $\vee$ Predicate1 |
| | — | Predicate1 $\Rightarrow$ Predicate1 |
| | — | Predicate1 $\Leftrightarrow$ Predicate1 |
| | — | (Predicate) |

| | | |
|---|---|---|
| Rel | ::= | = —$\in$ — InRel |
| | | |
| Expression0 | ::= | $\lambda$ SchemaText $\bullet$ Expression |
| | — | Expression |
| | | |
| Expression | ::= | Expression InRel Expression |
| | — | Expression1 $\times$ Expression1 $\times \ldots \times$ Expression1 |
| | — | Expression1 |
| | | |
| Expression1 | ::= | Expression1 InFun Expression1 |
| | — | $\mathbb{P}$ Expression3 |
| | — | PreGen Expression3 |
| | — | $-$Expression3 |
| | — | Expression3 PostFun |
| | — | Expression3$^{\text{Expression}}$ |
| | — | Expression3$(\!\vert$ Expression0 $\vert\!)$ |
| | — | Expression2 |
| | | |
| Expression2 | ::= | Expression2 Expression3 |
| | — | Expression3 |
| | | |
| Expression3 | ::= | VarName [GenActuals] |
| | — | Number |
| | — | SchemaRef |
| | — | SetExp |
| | — | $\langle$ [Expression, . . . , Expression] $\rangle$ |
| | — | $[\![$ [Expression, . . . , Expression] $]\!]$ |
| | — | (Expression, . . . , Expression) |
| | — | $\theta$SchemaName Decoration |
| | — | Expression3.VarName |
| | — | (Expression0) |
| | | |
| HistPred | ::= | $\forall$ Declaration [ $\vert$ HistPred] $\bullet$ HistPred |
| | — | $\exists$ Declaration [ $\vert$ HistPred] $\bullet$ HistPred |
| | — | $\exists_1$ Declaration [ $\vert$ HistPred] $\bullet$ HistPred |
| | — | HistPred1 |

| | | |
|---|---|---|
| HISTPRED1 | ::= | PREDICATE |
| | — | $\overrightarrow{\text{OBJREF}}$ |
| | — | OPNREF **enabled** [ \| PREDICATE] |
| | — | OPNREF **occurs** [ \| PREDICATE] |
| | — | $\Box$HISTPRED1 |
| | — | $\Diamond$HISTPRED1 |
| | — | $\neg$ HISTPRED1 |
| | — | HISTPRED1 $\wedge$ HISTPRED1 |
| | — | HISTPRED1 $\vee$ HISTPRED1 |
| | — | HISTPRED1 $\Rightarrow$ HISTPRED1 |
| | — | HISTPRED1 $\Leftrightarrow$ HISTPRED1 |
| | — | (HISTPRED1) |
| RENAMELIST | ::= | [RENITEM, . . . , RENITEM] |
| RENITEM | ::= | FEATUREREN — PARAMREN |
| FEATUREREN | ::= | IDENT/IDENT |
| PARAMREN | ::= | OPNNAME[FEATUREREN, . . . , FEATUREREN] |
| REDEFLIST | ::= | [**redef** OPNNAME, . . . , OPNNAME] |
| DELTALIST | ::= | IDENT, . . . , IDENT |
| SETEXP | ::= | { [EXPRESSION, . . . , EXPRESSION] } |
| | — | {SCHEMATEXT [ • EXPRESSION] } |
| OBJREF | ::= | IDENT \| (IDENT, IDENT) |
| IDENT | ::= | WORD DECORATION |
| DECLNAME | ::= | IDENT — OPNAME |
| VARNAME | ::= | IDENT — (OPNAME) |

| | | |
|---|---|---|
| OpName | ::= | _InSym_ — PreSym_ — _PostSym — $\_\llparenthesis\_\rrparenthesis$ — $-$ |
| InSym | ::= | InFun — InGen — InRel |
| PreSym | ::= | PreGen — PreRel |
| PostSym | ::= | PostFun |
| Decoration | ::= | *[*Stroke ... Stroke*]* |
| GenFormals | ::= | [Ident, ..., Ident] |
| GenActuals | ::= | [Expression, ..., Expression] |

| | |
|---|---|
| Word | Undecorated name or special symbol |
| Stroke | Single decoration: $'$, ?, ! or a subscript digit |
| SchemaName | Same as Word, but used to name a schema |
| OpnName | Same as Word, but used to name an operation |
| ClassName | Same as Word, but used to name a class |
| InFun | Infix function symbol |
| | $\mapsto$ $..$ $+$ $-$ $\cup$ $\setminus$ $\frown$ $*$ DIV MOD $\cap$ $\upharpoonright$ $\fatsemi$ $\circ$ $\oplus$ $\lhd$ $\rhd$ $\ntriangleleft$ $\ntriangleright$ |
| InRel | Infix relation symbol |
| | $\neq$ $\notin$ $\subseteq$ $\subset$ $<$ $\leqslant$ $\geqslant$ $>$ partitions |
| InGen | Infix generic symbol |
| | $\leftrightarrow$ $\nrightarrow$ $\rightarrow$ $\rightarrowtail\!\!\!\!\!+$ $\rightarrowtail$ $\twoheadrightarrow\!\!\!\!\!+$ $\twoheadrightarrow$ $\rightarrowtail\!\!\!\!\twoheadrightarrow$ $\nrightarrow\!\!\!\!\!+$ $\rightarrowtail\!\!\!\!\!\nrightarrow$ |
| PreRel | Prefix relation symbol |
| | disjoint |
| PreGen | Prefix generic symbol |
| | $\mathbb{P}_1$ id $\mathbb{F}$ $\mathbb{F}_1$ seq $\text{seq}_1$ $\text{seq}_\infty$ bag |
| PostFun | Postfix function symbol |
| | $\_^{\sim}$ $\_^{*}$ $\_^{+}$ |
| Number | Unsigned decimal integer |

# Appendix C

# TCOZ glossary

| Notation | Explanation |
|---|---|
| **c** : **chan** | declare **c** to be a channel |
| **a** : **actuator** | declare **a** to be a actuator |
| **s** : **sensor** | declare **s** to be a sensor |
| $\perp$ | divergent process |
| STOP | deadlocked process |
| SKIP | terminate immediately |
| WAIT **t** | delay termination by **t** |
| **a** $\rightarrow$ **P** | communicate **a** then do **P** |

continued on next page

| Notation | Explanation |
|---|---|
| $\mathbf{a@t} \rightarrow \mathbf{P}$ | communicate **a** at time **t** then do **P** |
| $[\mathbf{t} : \mathbb{T}] \bullet \mathbf{a@t} \rightarrow \mathbf{P}$ | record time of **a** event in variable **t** |
| $\mathbf{c}.\mathbf{a}$ | communicate **a** on channel **c** |
| $\mathbf{c}?\mathbf{a}$ | input **a** on channel **c** |
| $\mathbf{c}!\mathbf{a}$ | output **a** from channel **c** |
| $[\mathbf{b}] \bullet \mathbf{P}$ | enable **P** only if **b** |
| $\mathbf{P};\ \mathbf{Q}$ | perform **P** until termination, then perform **Q** |
| $\mathbf{P} \Box \mathbf{Q}$ | perform the first enabled of **P** and **Q** |
| $[\mathbf{i} : \mathbf{I}] \bullet \mathbf{P}$ | perform **P** with first enabled value of **i** (indexed external choice) |
| $\mathbf{P} \sqcap \mathbf{Q}$ | perform either of **P** and **Q** |

continued on next page

| Notation | Explanation |
|---|---|
| $[\mathbf{i}! : \mathbf{I}];\ \mathbf{P}$ | perform **P** with any value of **i** (indexed internal choice) |
| $\mathbf{v} := \mathbf{e}$ | syntactic sugar for $[\Delta\mathbf{v} \mid \mathbf{v}' = \mathbf{e}]$ |
| $\mathbf{P} \setminus \mathbf{A}$ | hide the events **A** from the environment of **P** |
| $\mathbf{P} \mid[\, \mathbf{A} \,]\mid \mathbf{Q}$ | synchronise **P** and **Q** on events from **A** |
| $(\big\Vert\ \mathbf{p}_1, \ldots, \mathbf{p_n} \bullet \ldots;\ \mathbf{p_i} \xleftrightarrow{\mathbf{A}} \mathbf{p_j};\ \ldots)$ | network topology abstraction with parameters $\mathbf{p}_1, \ldots, \mathbf{p_n}$ and network connections including $\mathbf{p_i}$ communicating with $\mathbf{p_j}$ on private channels from **A** |
| $\mathbf{P} \mid\mid\mid \mathbf{Q}$ | **P** and **Q** running without sychronisations |

continued on next page

| Notation | Explanation |
|---|---|
| $\mathbf{P} \rhd\{\mathbf{t}\}\ \mathbf{Q}$ | if **P** does not begin by time **t**, perform **Q** instead |
| $\mathbf{P} \swarrow\!\!\!/\{\mathbf{t}\}\ \mathbf{Q}$ | perform **P** until time **t**, then transfer control to **Q** |
| $\mathbf{P} \triangledown \mathbf{e} \rightarrow \mathbf{Q}$ | perform **P** until exception **e**, then transfer control to **Q** |
| $\mathbf{P} \bullet \textsc{Deadline}\ \mathbf{t}$ | behaviours of **P** which terminate before time **t** |
| $\mathbf{P} \bullet \textsc{WaitUntil}\ \mathbf{t}$ | after **P** idle until time **t** |

# Appendix D

# Completed DAML+OIL semantic encoding

## D.1 Basic concepts

The semantic models for DAML+OIL are encoded in the module DAMLOIL.

```
module DAMLOIL
```

The semantic encoding for the basic concepts is summarized in Table D.1.

All the things described in Semantic web context are called resources. All other concepts defined later like Property and Class are extended from the Resource.

| **DAML+OIL primitive** | **Alloy semantic function** |
|---|---|
| Resource | sig Resource {} |
| DAML_Property | disj sig Property extends Resource {sub_val: Resource $\rightarrow$ Resource} |
| DAML_Class | disj sig Class extends Resource {instances: set Resource} |
| Datatype | disj sig Datatype extends Class {} |

Table D.1: DAML+OIL Semantic encoding (basic concepts)

| DAML+OIL primitive | Alloy semantic function |
|---|---|
| subClassOf | fun subClassOf(csup, csub: Class) {csub.instances in csup.instances} |
| disjointWith | fun disjointWith (c1, c2: Class) { no c1.instances & c2.instances} |
| disjointUnionOf | fun disjointUnionOf(clist: List, c1: Class) {c1.instances = clist.*next.val.instances all disj ca1, ca2: clist.*next.val \| no ca1.instances & ca2.instances } |
| sameClassAs | fun sameClassAs( c1, c2: Class) {c1.instances = c2.instances} |

Table D.2: DAML+OIL Semantic encoding (class elements)

## D.2 Class elements

The semantic encoding for the class elements is summarized in Table D.2. It includes constructs like subClassOf, disjointWith, disjointUnionOf and sameClassAs.

## D.3 Property restrictions

The semantic encoding for the property restrictions is summarized in Table D.3. A property restriction defines the class of all objects that satisfy the restriction. For example the toClass function states that all instances of the class c1 have the values of property P all belonging to the class c2. The other constructs include hasValue, hasClass, cardinality etc.

| **DAML+OIL primitive** | **Alloy semantic function** |
|---|---|
| toClass | fun toClass (p: Property, c1: Class, c2: Class) {all r1, r2: Resource \| r1 in c1.instances <=> r2 in r1.(**p.sub_val**) => r2 in c2.instances} |
| hasValue | fun hasValue (p: Property, c1: Class, r: Resource) {all r1: Resource \| r1 in c1.instances => r1.(**p.sub_val**)=r } |
| hasClass | fun hasClass (p: Property, c1: Class, c2: Class) {all r1: Resource \| r1 in c1.instances => some r1.(**p.sub_val**) & c2.instances} |
| cardinality | fun cardinality (p: Property, c1: Class, N: Int) {all r1: Resource \| r1 in c1.instances <=> # r1.(**p.sub_val**) = int N} |
| maxCardinality | fun maxCardinality (p: Property, c1: Class, N: Int) {all r1: Resource \| r1 in c1.instances <=> # r1.(**p.sub_val**) =< int N } |
| minCardinality | fun minCardinality (p: Property, c1: Class, N: Int) {all r1: Resource \| r1 in c1.instances <=> # r1.(**p.sub_val**) >= int N } |
| cardinalityQ | fun cardinalityQ (p: Property, c1: Class, N: Int, c2: Class) {all r1: Resource \| r1 in c1.instances <=> # r1.(**p.sub_val**) & c2.instances = int N } |
| maxCardinalityQ | fun maxCardinalityQ (p: Property, c1: Class, N: Int, c2: Class) {all r1: Resource \| r1 in c1.instances <=> # r1.(**p.sub_val**) & c2.instances =< int N } |
| minCardinalityQ | fun minCardinalityQ(p: Property, c1: Class, N: Int, c2: Class) {all r1: Resource \| r1 in c1.instances <=> # r1.(**p.sub_val**) & c2.instances >= int N} |

Table D.3: DAML+OIL Semantic encoding (Property restrictions)

| **DAML+OIL primitive** | **Alloy semantic function** |
|---|---|
| intersectionOf | fun intersectionOf (clist: List, c1: Class)<br>{all r: Resource\| r in c1.instances <=><br>all ca: clist.*next.val \| r in ca.instances} |
| unionOf | fun unionOf (clist: List, c1: Class)<br>{all r: Resource\| r in c1.instances <=><br>some ca: clist.*next.val\| r in ca.instances} |

Table D.4: DAML+OIL Semantic encoding (Boolean combination)

## D.4 Boolean combination of class expressions

The semantic encoding for the boolean combination of class expression is summarized in Table D.4.

## D.5 Property elements

The semantic encoding for the property elements is summarized in Table D.5. It includes subPropertyOf, samePropertyAs etc.

| **DAML+OIL primitive** | **Alloy semantic function** |
|---|---|
| subPropertyOf | fun subPropertyOf (psup, psub: Property)<br>{psub.sub_val in psup.sub_val } |
| domain | fun domain (p: Property, c: Class)<br>{(p.sub_val).Resource in c.instances } |
| range | fun range (p: Property, c: Class)<br>{Resource.(p.sub_val) in c.instances } |
| samePropertyAs | fun samePropertyAs(p1, p2: Property)<br>{p1.sub_val=p2.sub_val } |
| inverseOf | fun inverseOf (p1, p2: Property)<br>{p1.sub_val = ~(p2.sub_val)} |
| TransitiveProperty | fun TransitiveProperty(p: Property)<br>{all x, y, z: Resource \|<br>y in (p.sub_val).x && z in (p.sub_val).y<br>=><br>z in (p.sub_val).x } |
| UniqueProperty | fun UniqueProperty (p: Property)<br>{all x : Resource \| sole x.(p.sub_val) } |
| UnambigousProperty | fun UnambigousProperty(p: Property)<br>{all x : Resource \| sole (p.sub_val).x} |

Table D.5: DAML+OIL Semantic encoding (Property elements)

# Appendix E

# DAML-S process ontology for PIDManager (XML format)

he **PIDManager** class defined for the Calendar agent will be used to demonstrate the translation from TCOZ model to DAML-S document. The **PIDManager** class has five operations, **AddPID**, **RemovePID**, **New**, **Delete** and **Validate**. Each of them will be translated into a **process**.

The operation **AddPID** is an operation invokes no other operations, so it will be translated as an AtomicProcess (R2). Some standard header information is generated firstly.

```
<!--Header Information-->
<?xml version='1.0' encoding='ISO-8859-1'?>
...
```

```
<!--PIDmanager AddPId process-->
<daml:Class rdf:ID="PIDManager_AddPID">
 <rdfs:subClassOf rdf:resource
    ="&process;#AtomicProcess"/>
 <rdfs:subClassOf>
   <daml:Restriction daml:cardinality="1">
     <daml:onProperty
       rdf:resource="#AddPID_id"/>
   </daml:Restriction>
 </rdfs:subClassOf>
</daml:Class>
```

The operation **AddPID** has one input **id**? declared to be type **PID**. It will be translated into **input** in DAML-S (R4).

```
 <!--input-->
 <rdf:Property rdf:ID="PIDManager_AddPID_id">
  <rdfs:subPropertyOf rdf:resource="&process;#input"/>
  <rdfs:domain rdf:resource="#PIDManager_AddPID"/>
  <rdfs:range rdf:resource="#PID"/>
</rdf:Property>
```

The operation **AddPID** has one predicate $\mathbf{ids}' = \mathbf{ids} \cup \{\mathbf{id}?\}$ which involve post-states. It will be translated into **effect**

(**PIDManager_AddPID_EFFECT**) in DAML-S (R2).

```
<!--UnConditionalEffect parameter -->
 <daml:Property
     rdf:ID="PIDManager_AddPID_EFFECT">
  <rdfs:subPropertyOf
     rdf:resource="&process;#effect"/>
  <rdfs:domain rdf:resource="#PIDManager_AddPID"/>
  <rdfs:range><daml:Class>
     <rdfs:subClassOf rdf:resource
       ="&process;#UnConditionalEffect"/>
     <rdfs:subClassOf>
      <daml:Restriction>
       <daml:onProperty
```

```
          rdf:resource="&process;#ceEffect"/>
      <daml:toClass
          rdf:resource="#PIDManager_AddPIDEffect"/>
    </daml:Restriction>
   </rdfs:subClassOf>
   </daml:Class></rdfs:range>
 </daml:Property>
 <daml:Class rdf:ID="#PIDManager_AddPIDEffect">
  <rdfs:subClassOf rdf:resource="&daml;#Thing"/>
 </daml:Class>
```

The operation **RemovePID** can be translated similarly.

The operation **New** calls other operation **AddPID**, so it was translated as a composite process (R4). It perform two subprocess **PIDManager_AddPID_add_id_in** and **PIDManager_AddPID** in sequence. The **PIDManager_AddPID_add_id_in** process represents the communication on channel **add** (R5). The guard of the operation was translated as the precondition (**IDnotInIDS**)(R7).

```
 <-- "New" process-->
 <--Communication translated as atomic process R5-->
<daml:Class rdf:ID="PIDManager_New_add_id_in">
 <rdfs:subClassOf
    rdf:resource="&process;#AtomicProcess"/>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty
        rdf:resource="#PIDManager_New_add_id"/>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty
        rdf:resource="&process;#precondition"/>
      <daml:toClass rdf:resource ="#IDnotInIDS"/>
     </daml:Restriction>
   </rdfs:subClassOf>
```

```
</daml:Class>
<-- input from channel translated
    as input for process--!>
<rdf:Property rdf:ID="PIDManager_New_add_id">
 <rdfs:subPropertyOf
   rdf:resource="&process;#input"/>
 <rdfs:domain
   rdf:resource="#PIDManager_New_add_id_in"/>
 <rdfs:range rdf:resource="#PID"/>
</rdf:Property>
<--Guard --!>
<daml:Class rdf:ID="IDnotInIDS">
 <rdfs:subClassOf
    rdf:resource="&process;#Condition"/>
 <rdfs:subClassOf>
  <daml:Restriction>
    <daml:onProperty
       rdf:resource="&process;#Value"/>
    <daml:hasValue
       rdf:resource="&process;#True"/>
  </daml:Restriction>
</rdfs:subClassOf>
</daml:Class>
<-- "New" translated as compositeprocess--!>
<daml:Class rdf:ID="PIDManager_New">
 <rdfs:subClassOf
   rdf:resource="&process;#CompositeProcess"/>
 <rdfs:subClassOf>
   <daml:Restriction>
    <daml:onProperty rdf:resource=
      "&process;#composedOf"/>
     <daml:toClass>
      <daml:Class>
       <daml:intersectionOf
         rdf:parseType="daml:collection">
        <daml:Class rdf:about="&process;#Sequence"/>
         <daml:Restriction>
          <daml:onProperty
            rdf:resource="&process;#components"/>
           <daml:toClass><daml:Class>
```

```
              <process:listOfInstancesOf
                rdf:parseType="daml:collection">
               <daml:Class
                 rdf:about="#PIDManager_New_add_id_in"/>
                <daml:Class
                 rdf:about="#PIDManager_AddPID"/>
               </process:listOfInstancesOf>
              </daml:Class></daml:toClass>
            </daml:Restriction>
          </daml:intersectionOf>
        </daml:Class>
      </daml:toClass></daml:Restriction>
  </rdfs:subClassOf>
<!--some atomic derived IOEP was omitted here--> </daml:Class>
```

The operation **Delete** and **Valide** can be similarly translated.