Citation for published version:
Hopton, L 2009, *Modelling Institutions using Answer Set Programming: Enhancing the Institution Action Language*. Department of Computer Science Technical Report Series, no. CSBU-2009-06, Department of Computer Science, University of Bath, Bath. U. K.

Publication date:
2009

Link to publication

©The Author May 2009

University of Bath

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 07. Dec. 2019

Department of Computer Science

Technical Report

Undergraduate Dissertation: Modelling Institutions using Answer Set Programming: Enhancing the Institution Action Language

Luke Hopton

Technical Report 2009-06 **May 2009**

ISSN 1740-9497
Editor: Dr Marina De Vos

Copyright ©May 2009 by the author(s).

Contact Address:
Technical Report Editor
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497
Editor: Dr Marina De Vos

Modelling Institutions using Answer Set Programming: Enhancing the Institution Action Language

Luke Hopton

Bachelor of Science in Computer Science with Honours
The University of Bath
April 2009

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:

Abstract

Institutions are normative multi-agent systems. Cliffe et al. (2005) have developed a model of multiple interacting institutions in answer set programming (ASP). ASP is a declarative programming paradigm which provides non-monotonic reasoning. As a part of this work, they have developed InstAL, an action language which is used for the specification of institutions. These specifications are then translated into ASP. In this project we enhance the support for InstAL in two areas. We present an overview of the software engineering process used to implement InstEdit, an editor for InstAL. InstEdit supports the institution designer in creating and reasoning about InstAL specifications. We present also InstQL, the institution query language. This language is designed for the specification of queries for InstAL institutions. As with InstAL, the semantics of InstQL are given by translation into ASP. The development of this language is informed by motivating example queries. These are queries specified in ASP for an existing institution. We demonstrate how these can be expressed in InstQL and translate them back into ASP to verify the correctness of the semantics. We discuss how InstQL allows us to perform prediction, postdiction and planning and identify a restricted form of linear temporal logic that can be expressed in InstQL. Finally, we present the development of a tool to automate the translation of InstQL into ASP. We conclude that these additions aid the institution designer when working with InstAL and provide the foundation for further enhancements.

Acknowledgments

Thanks to Owen Cliffe, Marina de Vos and Julian Padget for not only being the much-referenced "Cliffe et al", but also for their help and advice. Special thanks to Marina, my supervisor, for her guidance throughout.

Thanks also to my family and friends (especially Ell) for love and support. Thanks to my parents for their financial support also: I literally couldn't have done it without you.

Modelling Institutions using Answer Set Programming: Enhancing the Institution Action Language

Submitted by: Luke Hopton

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).
This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Contents

1 Introduction **1**

1.1 Problem Description . . . 1

1.2 Report Structure . . . 2

1.3 Notational Conventions . . . 3

2 Literature Review **4**

2.1 Agents and Multi-Agent Systems . . . 4

2.1.1 What is an Agent? . . . 4

2.1.2 Multi-Agent Systems . . . 8

2.1.3 Institutions . . . 9

2.2 Formal Languages for MAS Reasoning . . . 11

2.2.1 Modal Logic . . . 12

2.2.2 Event Calculus . . . 14

2.2.3 Action Languages . . . 16

2.2.4 ASP . . . 20

2.3 InstAL . . . 23

2.3.1 InstAL Language Overview . . . 23

2.3.2 State of Development . . . 24

2.4 Summary . . . 26

3 The Institution Editor **27**

3.1 Requirements . . . 28

3.1.1 Requirements Elicitation . . . 28

3.1.2 Requirements Specification . . . 30
3.2 High-Level Design . . . 32
3.3 Low-Level Design & Implementation . . . 34
3.3.1 Implementing a Text Editor . . . 34
3.3.2 Integration with InstAL . . . 36
3.3.3 Integrating the Answer Set Solver . . . 38
3.3.4 Adding Syntax Highlighting . . . 39
3.3.5 Integrating Queries and Visualisation . . . 41
3.4 Testing . . . 42
3.5 Summary . . . 43

4 A Query Language for InstAL 44
4.1 Example Queries . . . 46
4.1.1 Selected Queries . . . 46
4.1.2 Analysis of the Examples . . . 47
4.2 The Institution Query Language . . . 48
4.2.1 InstQL_α – Basic Queries . . . 48
4.2.2 InstQL_β – Concurrent Events . . . 52
4.2.3 InstQL_γ – Simple Ordering . . . 54
4.2.4 InstQL_δ – Precise Ordering . . . 55
4.2.5 A Note on Negation . . . 57
4.3 Example Queries Revisited . . . 59
4.4 Summary . . . 61

5 Using the Institution Query Language 62
5.1 Reasoning with InstQL . . . 62
5.1.1 General Reasoning with InstQL . . . 62
5.1.2 Agent Reasoning . . . 65
5.2 Modelling Linear Temporal Logic in InstQL . . . 65
5.2.1 Institutional LTL . . . 66
5.2.2 Expressing InstLTL in InstQL . . . 67

5.3 Implementing the Institution Query Language . . . 69
5.3.1 Requirements & Design . . . 69
5.3.2 Implementation & Testing . . . 71
5.4 Summary . . . 72

6 Conclusion **73**
6.1 Project Summary . . . 73
6.2 Evaluation . . . 74
6.3 Future Work . . . 76

A InstAL **82**
A.1 Institution Model . . . 82
A.1.1 Model Definition/Syntax . . . 82
A.1.2 Model Semantics . . . 85
A.2 InstAL . . . 88

B Testing Plans **92**
B.1 Release Testing . . . 92
B.2 Component Testing . . . 95
B.3 Testing the InstQL Translator . . . 96

C Source Code Listings **98**
C.1 *InstEdit* . . . 99
C.1.1 SyntaxHighlighter.java . . . 99
C.1.2 InstEditPanel.java . . . 102
C.1.3 InstalBuilder.java . . . 103
C.1.4 Translator.java . . . 105
C.1.5 InstalTranslator.java . . . 108
C.1.6 InstqlTranslator.java . . . 110
C.1.7 AnswerSetSolver.java . . . 111
C.1.8 Visualiser.java . . . 115
C.2 InstQL Translator . . . 117

C.2.1 Parser.pm 117
C.2.2 Util.pm 119

D A*QL*: A Query Language for Action Domains 121

List of Figures

3.1 Use cases for *InstEdit* 29

3.2 Sequence diagram without InstEdit 30

3.3 InstEdit architecture 33

3.4 Sequence diagram with InstEdit 33

3.5 Design of InstEdit main screen 35

3.6 InstEdit main screen 36

3.7 InstEdit settings screen 37

3.8 Syntax highlighting architecture. 41

4.1 Current InstAL process. 45

4.2 Proposed InstAL process. 46

5.1 State transition for prediction. 64

5.2 Architecture for the InstQL translator 70

A.1 Institutional event types. 83

List of Tables

2.1 Abbreviations in $\mathcal{C}$ 18

2.2 Abbreviations in $\mathcal{C}+$ 20

4.1 InstQL Syntax 56

B.1 Release testing. 93

B.2 Release testing (continued). 94

Chapter 1

Introduction

1.1 Problem Description

Intelligent *agents* are computer systems which are capable of autonomous action in order to achieve their goals (Wooldridge, 2002). A key feature of agents is their ability for communication and social interaction with other agents – within the context of a *multi-agent system* (MAS). Traditionally, MAS have been developed as closed systems where one organisation is responsible for all participants. In areas such as E-Commerce, there is a movement towards open systems where agents from different organisations are free to join and leave the system (Tadjouddine et al., 2008).

Such open systems propose new challenges for their developers. When a system is designed, it is not known which agents will join and who they will represent. We cannot assume that participating agents will follow the rules (Viganò and Colombetti, 2007). Enforcing rules could be achieved by making any action which is not permitted impossible. However, such a system would be overly restrictive and limiting. For this reason, the agents community has introduced *institutions* (Noriega, 1997). These normative multi-agent systems use concepts such as power, permission and obligation to restrict the behaviour of participating agents. The aim of institutions is to impose restrictions to protect agents (and the system itself) from other agents' actions, without limiting the autonomy of agents (Garcia-Camino et al., 2005).

Various techniques have been employed for modelling institutions. Among these is the user of *answer set programming* (ASP). ASP is attractive for reasoning issues of the kind we see in the multi-agents system domain for two reasons.

1. It is *non-monotonic* – if from a we conclude b, then it need be the case that from $a \wedge c$ we conclude b. This makes ASP suited to domains such as MAS where we must reason with partial information. An agent which gathers more information can alter its conclusions.

2. ASP provides two forms of negation: *negation as failure* and *classical negation*. The classical negation "$\neg x$" means x is not true. However, the negation as failure "**not** x" means we cannot show x to be true. Some declarative systems, such as Prolog, associate classical negation with negation as failure. The consequence of this is that if we cannot prove x, then we conclude that we know x to be false. This means we must know everything that is true – the *closed world assumption*. Since ASP does not associate these two forms of negation, it does not suffer from the closed world assumption.

Cliffe et al. have developed a model of multiple institutions in ASP (Cliffe et al., 2006). The approach includes a domain specific action language called *InstAL* (the institution action language). This language is used to specify institutions. InstAL specifications are then translated into ASP to allow reasoning about the institution. Layering an action language on top of ASP is attractive since it provides a higher-level way to specify institutions. Similar work has been done by Lifschitz and Turner (1999) who translate the action language $\mathcal{C}$ into ASP.

Tools have been developed to automate the translation of an institution specified in InstAL into ASP. An answer set solver can be used to solve the ASP model of the institution. The model is set up such that the answer sets correspond to traces of the institution. Further tools allow the traces to be visualised in various formats. This reasoning process is complex: it requires a number of stages, each with different tools, which are not connected. It is up to the institution designer to mange this process. While the tools for the reasoning process exist, there are no tools to support the creation of an institution in InstAL.

This project aims to ease the use of InstAL by developing a toolkit to support this process. The toolkit will provide support for creating institutions and control the reasoning process. This will unite the separate stages of reasoning about institutions with InstAL so that the institution designer can control this process from one system.

InstAL allows us to specify institutions at a high-level and in a domain specific language. When we come to reason about and query an institution model, this must be done in ASP. The power of ASP allows a large variety of queries about institutions to be created. However, creating these queries can be complex. Doing so requires knowledge of how the institution is modelled in ASP. Since the institution will have been specified in InstAL and than translated into ASP, we cannot assume that designers have this knowledge.

To solve this problem, the second aim of this project is to design a domain specific query language. Like InstAL, this will be translated into ASP. This will make reasoning about institutions easier.

1.2 Report Structure

In Chapter 2 we present an overview of relevant literature. This considers agents, multi-agent systems and institutions. We then consider some of the alternative formalisms used

to model MAS and institutions, including answer set programming. The work of Cliffe et al. on modelling institutions in ASP and specifying institutions in InstAL is reviewed. Chapter 2 concludes with a consideration of the current state of the development of InstAL as a tool for reasoning about institutions.

Chapter 3 describes the software development process of implementing *InstEdit* – the institution editor. This tool supports the creating of InstAL specifications and assists control of the process of reasoning about them. It assumes much of the burden of managing the tools involved in this process, making modeling institutions easier for the user.

A new query language for institutions specified in InstAL has been created. Chapter 4 discusses the development of this language, *InstQL* (the institution query language). We present examples of queries previously specified in ASP and demonstrate how these may be expressed in InstQL.

The use of InstQL is considered in Chapter 5. We descibe its use in general reasoning about institutions and relate it to a restricted form of linear temporal logic. Chapter 5 concludes by presenting the implementation of a tool to automate the translation of InstQL into ASP.

1.3 Notational Conventions

Throughout this work the following conventions are adopted:

1. The symbols $\top$ and $\bot$ are used to denote *true* and *false* (respectively).
2. The powerset of some set S is denoted $\mathbb{P}(S)$.
3. Classical negation is denoted by $\neg$ while negation as failure is denoted **not**.
4. Arrows of the form $\Rightarrow, \Leftarrow, \Leftrightarrow$ indicate logical implication while arrows of the form $\rightarrow, \leftarrow, \leftrightarrow$ represent functions etc.

Chapter 2

Literature Review

This chapter presents an overview of the theoretic background which provides the motivation for this project. It considers existing work in the domain to give an indication of the approaches taken to the modelling of institutions. Thus it provides an idea as to why it is worthwhile to enhance the InstAL language as a tool to aid in institutional modelling.

In Section 2.1 the concepts of *agents*, *multi-agent systems* and *institutions* are presented. Section 2.2 considers some of the approaches that have been taken to provide a formal basis to allow reasoning about institutions and multi-agent systems, including the use of answer set programming. Answer set programming has been used as a method for reasoning about institutions by Cliffe et al, who have developed the InstAL language as part of this work. Section 2.3 summarises action language InstAL and considers (p24) the current state of InstAL.

2.1 Agents and Multi-Agent Systems

This section introduces the concepts of *agents*, *multi-agent systems* and *institutions* which are fundamental to the problem domain of InstAL. This begins by examining the notion of an agent and considering possible definitions of an agent. Then this is extended to look at multi-agent systems (p8) where a number of agents interact and a specialised kind of multi-agent system, the institution is considered.

2.1.1 What is an Agent?

The concept of an *agent* is a well established one within the field of Computer Science. Despite this, there is no universally accepted definition of an agent (Wooldridge, 2002). There is, however, a generally agreed upon consensus on the kind of properties and behaviours an agent has and exhibits. The first (and perhaps most fundamental) aspect of this consensus is *autonomy* – indeed agents are often referred to as *autonomous agents* in the literature.

Maybe those active in the field cannot agree on exactly what an agent is, but it is certainly true that an agent is **not** a system which works under human control.

Another concept intrinsically tied to agents is *action*; agents (autonomously) act in order to produce some effect(s) (Wooldridge and Jennings, 1995). Action is itself a concept which is difficult to formally define. Consider the following example from Wooldridge (1992):

> "A classic example, due to the philosopher Searle, is that of Gavrilo Princip in 1914: did he pull a trigger, fire a gun, kill Archduke Ferdinand, or start World War I? Each of these seem to be equally valid descriptions of "the same event", and yet trying to isolate that event is notoriously difficult."

Wooldridge (Wooldridge, 1992; Wooldridge and Jennings, 1995) identifies that, as well as being autonomous, agent action is often described as *rational*. This is yet another concept which is difficult to specify, but intuitively this can be interpreted to mean that agents act for a reason, in order to achieve some goal or maximise some utility. The implication is that agents will not perform actions that are detrimental to their own agenda. Of course, this only raises further question and necessitates further concepts to be defined: what are *goals* and what is *utility*? These concepts will be further considered later in this section.

Another aspect of agents is that we consider them within the context of an *environment*. Agents do not exist as systems in isolation – any agent will almost certainly be situated within an environment. Russel and Norvig present a classification of environments as follows (cited in Wooldridge, 2002):

Accessible/Inaccessible. An environment is accessible if and only if agents can obtain complete (up to date) information on it at any given time.

Deterministic/Non-deterministic. An environment is deterministic if and only if any single action has a guaranteed effect. If there is uncertainty about the outcome of any action then the environment is non-deterministic.

Static/Dynamic. An environment is static (relative to a given agent) if it does not change except for the actions of that agent.

Discrete/Continuous. An environment is discrete if there are only a fixed and finite number of actions and percepts possible within it.

It is fairly apparent that any "interesting" problem will feature an environment in which several of the following are true: agents must reason with only partial information, effects of actions may not be guaranteed, a single agent will not be the only thing changing the environment and the number of possible actions and/or percepts will not necessarily be fixed or finite. That is to say, the *trivial case* in which we have an accessible, deterministic, static and discrete environment is unlikely and probably unreasonable in any system which approximates the real world.

Some additional characteristics that we may commonly expect of agents are (adapted from Wooldridge and Jennings, 1995; Wooldridge, 2002):

- agents are *reactive* – able to respond to changes in their environment
- agents are *proactive* – they are not entirely driven by external factors and are able to initiate action in order to achieve their goals
- agents are *interactive* – capable of social activity to work with other agents (or possibly humans) in order to satisfy their goals

These considerations of what constitutes an agent lead to the concept of a *goal*. This corresponds to the high-level notion of goal we may intuitively have; the goal of an agent is simply something it wants to achieve or bring about. More formally, if an environment, E, is characterised by a set of discrete[1] states $E = \{e_1, \ldots, e_n\}$ then two kinds of tasks for an agent become apparent:

1. *Achievement tasks* where the goal is to get the environment state into some desirable state $e_i \in G \subseteq E$
2. *Maintenance tasks* where the goal is to maintain the environment state and avoid some "bad" states $B \subseteq E$

A related concept is that of *utility* where the utility of a state describes how desirable it is for an agent. That is, there is some *utility function* for the agent:

$$u : E \to \mathbb{R} \tag{2.1}$$

This allows an agent to reason about how well it is doing and allows agents to perform tasks in order to maximise their utility.

Intentional Stance

Perhaps the most common view of agents is the *intentional stance*, based on the work of the philosopher Daniel Dennet. Dennet used the phrase *intentional system* to describe entities "whose rational behaviour can be predicted by the method of attributing belief, desires and rational acumen" (Dennet, 1978 (cited in Wooldridge, 2002)). Under the intentional stance, we interpret the actions of agents in the context of *beliefs*, *desires* and *intentions* (BDI).

[1]This does not imply a discrete environment; a continuous environment can be classified as a set of discrete states. Consider the example where a thermostat agent seeks to maintain the temperature of a room. The environment is described by a continuous attribute (temperature) but we can impose a classification such that temperature is a value from $\{\textit{too cold}, \textit{acceptable}, \textit{too hot}\}$ by simply choosing some boundary temperatures for these categories.

Beliefs are a representation of an agent's knowledge of its environment, *desires* are goals that the agent wants to achieve and *intentions* are plans to achieve those desires (Georgeff et al., 1999; Jo et al., 2004). The BDI approach is based on a model of human practical reasoning developed by Michael Bratman (Georgeff et al., 1999). Wooldridge (2002) presents a view of the BDI model which can be summarised as follows: let Bel be the set of possible beliefs an agent may have, Des be the set of possible desires, Int the set of intentions and Per a set of percepts (information an agent has sensed about its environment). At any time, an agent has a set of current beliefs (from the powerset of all possible beliefs – $\mathbb{P}(Bel)$) and similarly a current set of desires and intentions. Then the agent updates its beliefs, desires and intentions with the following functions:

- Updating *options* (i.e. current desires):

$$options : \mathbb{P}(Bel) \times \mathbb{P}(Int) \to \mathbb{P}(Des)$$

- *Filtering* possible options to commit to:

$$filter : \mathbb{P}(Bel) \times \mathbb{P}(Des) \times \mathbb{P}(Int) \to \mathbb{P}(Int)$$

- *Belief revision*:

$$brf : \mathbb{P}(Bel) \times Per \to \mathbb{P}(Bel)$$

The lack of any concrete definition of an agent and the difficulties incurred in trying to provide a generally applicable definition can be seen as symptomatic of the nature and use of agents. Agents are a very high-level abstraction which can be used to model social behaviour; the model is largely human-oriented – an application of Bratman's model of human action characterised by beliefs, desires and intentions has become one of the most accepted ways to reason about agents (Wooldridge, 1992). The very fact that this model was originally intended for human behaviour should give an indication as to the complexity of agents. Agents are built to autonomously act in order to achieve some high-level goal; this goal-oriented behaviour is different to the task-oriented behaviour more typical of software systems.

Task-oriented behaviour can be viewed as following an algorithm with given input to produce some output – very much the traditional territory of programs. Agents, by contrast, are given goals and must achieve these, using artificial intelligence and reasoning mechanisms to decide how. The process by which this can be done depends on their environment and the actions of other agents (see p8). This is much closer to the way humans work and explains why the BDI model of human action is so readily applied to software agents. Agents act in such a complex way it is easier for us to reason about them in the way we ourselves think and reason than as traditional algorithms.

While agents are certainly autonomous, there is little else we can say about them with any guarantee. It was identified above that we can expect agents to be reactive, proactive and interactive but this may not always be the case. Whilst usually an agent will interact

with others, there is nothing to prevent design of systems consisting of a single agent; in such a case there is no need for the agent to interact with others. Equally, we can imagine an agent that aims to achieve its goals only in the way it responds to changes in its environment. Such an agent is not really being proactive in generating change in the environment. Perhaps a little more contrived is the case of an agent which acts regardless of the environment, but we see that these characteristics are not requirements of an agent. As Nwana (1996) puts it:

> "... even within the software fraternity, the word "agent" is really an umbrella term for a heterogeneous body of research and development... When we really have to, we define an agent as referring to a component of software and/or hardware which is capable of acting exactingly..."

Agents may be applied to a diverse range of problems – it is the problem domain that will determine what a suitable agent is for that scenario. This explains why a formal definition of agents has not been agreed upon. Indeed, such a definition may be of only limited use. To encompass all the complexity and diversity of many heterogeneous agents, any global definition would have to be itself complex and broad-reaching. It is perhaps more useful to define what it means to be an agent for each problem domain in which we intend to deploy agents.

2.1.2 Multi-Agent Systems

Assuming we have decided upon some definition of an agent, then denoting each agent as A_i, a naïve definition of a *multi-agent system* (MAS) would be as follows:

$$M = \langle \{A_1, \ldots, A_n\}, E \rangle \tag{2.2}$$

That is, a multi-agent system is a set of agents $\{A_1, \ldots, A_n\}$ in some environment E. While this captures the nature of a MAS at the most basic level, as with the agents themselves, there are many ways in which we can extend this definition. For example, since each A_i acts in the environment, A_i defines a "sphere of influence" in E – i.e. those parts of E that A_i can change (or at least influence the change of) (Wooldridge, 2002). For two agents A_i and A_j, their spheres of influence may be totally separate or may overlap or may even coincide. What the definition in (2.2) omits is any consideration of relationships between agents. For example, the MAS may feature "power" relationships where one agents is the "boss" of another (Wooldridge, 2002).

Sichman et al. (1994) present a classification of relationships between agents (with respect to their goals) as follows:

Independence. An agent is independent of another if it can achieve all of its goals without any actions of the other agent.

Unilateral dependence. An agent A_i unilaterally depends on A_j if A_i requires A_j to perform some actions in order to achieve its goals but A_j can achieve all of its goals independently of the actions of A_i.

Mutual dependence. Two agents are mutually dependent if they require actions of the other in order to achieve the *same* goal.

Reciprocal dependence. Two agents are reciprocally dependent on each other if each requires the other to perform some actions to achieve *different* goals.

In addition, a MAS allows for different specifications of goals. Goals can belong to agents as previously discussed, but in addition collective goals may be assigned to some subset of the agents in the system (Gaudou et al., 2008). Such cooperation in order to achieve collective tasks can necessitate to the formation of *coalitions*; Shehory and Kraus (1998) explain that:

> "The allocation of tasks to groups of agents in necessary when tasks cannot be performed by single agents or when single agents perform them inefficiently."

As well as cooperating to achieve collective tasks, agents within a MAS may cooperate to achieve the goals of individual agents – a "you scratch my back, I'll scratch yours" type of situation. Here, individual agents cooperate in order to increase their own utility (Shehory and Kraus, 1998).

Clearly, the simple definition of a MAS in (2.2) needs to be extended in order to fully capture what is meant by "multi-agent system". However, just as with the concept of an *agent*, MAS is not a well defined and exact notion. A whole variety of ideas are covered by this term and what exactly is meant by MAS depends on the context in which it is used. The interactions that will occur within a MAS depend on the origin and type of agents that will participate. In the simplest case all agents will have been developed by the same programmer and so their interactions are no different from those between different components of each agent – they can be considered all subsystems where the MAS itself is the whole system. Such a situation is certainly not the norm (Shoham and Tennenholtz, 1995) but will likely involve a very different formalism of the concept of MAS than one in which "agents are programmed individually in an unconstrained fashion" (Shoham and Tennenholtz, 1995).

2.1.3 Institutions

One particular type of MAS is the *institution* (or *electronic institution*). Institutions are *normative* mutli-agent systems in that they impose some form of laws/rules on participating agents which encourage adherence to "normal" behaviour (as defined by the system designer(s)). The following example is taken from Shoham and Tennenholtz (1995) to outline the motivation for norms:

> "...using the domain of mobile robots for illustration, when two robots note they are on a collision course with one another, they may either appeal to some

> central traffic controller for coordination advice, or alternatively they might engage in a negotiation resulting (say) in each robot moving slightly to its right... Why not adopt a convention, or as we'd like to think of it, a social law, according to which each robot keeps to the right of the path? *If each robot obeys the convention*, we will have avoided all head on collisions without any need for either a central arbiter or negotiation."

The italics above have been added to emphasise the fact that if the designer has decided that all robots will travel on the right to prevent collisions, this will only resolve the situation if the robots adhere to this norm. If the system does nothing to enforce this norm, then there is nothing to stop, for example, a robot designer programming his robot to travel down the centre of a corridor so that it can move past "traffic" on the right. There are reasons that an agent may not adhere to norms, other than the kind of intentional deviance given in the previous case. What if an agent breaks down on the right and blocks the way? Other agents must move from the right, potentially into the path of agents traveling the other way, in order to pass. As well as introducing some motivation for normative MAS, this example has illustrated possible reasons why agents may choose to disobey norms (either it may benefit them in the case of the "speedy" robot or that circumstances prevent compliance).

Institutions are used extensively in so-called *open systems* or *open interaction systems* to define expected behaviour in such systems (e.g. Fornara and Colombetti, 2008; Garcia-Camino et al., 2005). Garcia-Camino et al. (2005) define the key characteristics of an open system as:

Heterogeneity – agents within open systems are likely to have been developed by different organisations, for different purposes, in potentially different languages.

Reliability – open systems must continue to function while individual components are repaired or replaced.

Accountability/legitimacy – the system must prevent agents from performing "deviant" actions which may threaten the functioning of the system.

Societal change – societies represented by open systems are not static and the system must be able to change to reflect this.

In such a system, "The normative component is fundamental because it can be used to specify the expected behavior of the interacting agents" (Fornara and Colombetti, 2008) and it provides a mechanism to constrain agent behaviour without placing too much restriction on their autonomy (Garcia-Camino et al., 2005). This is because, rather than expressly forbidding certain actions, an institution is "a set of conventions on how participants are supposed to act" (Noriega, 1997).

These conventions are expressed in an institution by means of identifying whether actions are *permitted* or not. Whether an action is permitted is determined by some conditions within the institution and/or relating to the agent that performed the action. Equally,

institutions may impose *obligations* – i.e. agents must perform certain actions before some time frame expires. When an agent performs an action which is not permitted or fails to perform an action it was obliged to, then that agent has *violated* the rules of the institution. The institution will then impose some *sanction* or *violation event* (Cliffe et al., 2006) to punish the agent for the breach of rules (Aldewereld et al., 2006; Fornara and Colombetti, 2008; Garcia-Camino et al., 2005).

In addition to placing constraints on what actions agents should perform, institutions also provide "new possibilities of actions" (Gaudou et al., 2008). Consider the following example from (Cliffe et al., 2006):

> "...a marriage ceremony will only bring about the married state, if the person performing the ceremony is empowered so to do."

The issue here is the concept of *institutional power* – anyone can perform a marriage ceremony but it will not have the effect of marrying the participants (i.e. bringing about the "married state") unless the agent performing the ceremony has the institutional power to do so. Institutional power is a property of the institution itself; a suitable agent (perhaps a priest, registrar etc) will perform actions in a marriage ceremony that any agent can perform. The institution must empower an agent to perform the ceremony meaningfully in order to bring about the married state. Thus when this agent performs the ceremony, the institution recognises its effect.

Cliffe et al. (2006) introduce the concept of multi-institutions. This allows us to consider interactions between different institutions. Rather than modelling large and complex social structures as a monolithic whole, this allows us to separate out different parts as individual institutions. This allows:

> "...the possibility of using institutions as a means for abstraction (capturing increasing levels of specificity at lower levels) and also as a means for delegation (whereby one institution relies on the behaviour of another to augment its function)." (Cliffe et al., 2006)

As with many of the concepts in the multi-agent systems domain, there is no generally accepted formal definition for an institution. One formal model, that used by the action language InstAL (p23), is presented in Appendix A.1.

2.2 Formal Languages for MAS Reasoning

There have been many approaches to the formal/logical modelling of agents and MAS. A formal model is needed to give some semantics to these concepts (which are intuitive, high-level and somewhat ambiguous notions as detailed in Section 2.1). Giving such a system formal semantics allows us to reason about the properties and behaviour of MAS and

institutions. Just as the lack of consensus as to a general definition of agents was discussed above (p4), there are many different approaches to logical modelling and reasoning.

A fundamental method used for modelling agents, epistemic logics with modal operators, is presented in Section 2.2.1. However, there are problems with such logics and so approaches such as the Event Calculus (Section 2.2.2) and action languages (Section 2.2.3) are considered. Section 2.2.4 presents an introduction to Answer Set Programming, a declarative paradigm with well-defined formal semantics. Some more recent work has used ASP in order to model MAS (De Vos and Vermeir, 2004; Nieuwenborgh et al., 2007) and Cliffe et al. (2005; 2006; 2007; 2008).

2.2.1 Modal Logic

Modal logic is "the logic of necessity" (Hodges, 2001) – it extends first order logic by considering *necessary* and *contingent* truths. Intuitively, if some proposition is necessarily true then it **must** be true. If, however, the proposition is a contingent truth then it **might** be true. Much of this section is based on Moore (1985) and Wooldridge and Jennings (1995).

Modal logic is typically used with the *possible-worlds* semantics as an epistemic logic for agent reasoning, an approach based on Hintikka (1962) and Kripke (1963) – modal logics are an established technique for reasoning about intention. Under the possible-worlds semantics, an agent believes in a number of different worlds; as the agent gains more knowledge it is able to eliminate some worlds as not being possible (i.e. those that are inconsistent with its beliefs). An advantage of such an approach is that is says nothing about the structure of the agent's reasoning.

This approach is formulated in *normal modal logics* which extend the syntax of classical propositional logic with two new operators ($\Box$ and $\Diamond$) as follows (Wooldridge and Jennings, 1995):

- Let $P = \{p, q, r, \ldots\}$ be a countable set of atomic propositions, then $p \in P$ is a formula
- For formulae ϕ and ψ then the following are all formulae:
  - $\top$ (true)
  - $\neg\psi$
  - $\phi \vee \psi$
  - $\Box\psi$
  - $\Diamond\psi$

Then a model, M, is:

$$M = \langle W, R, \pi \rangle \tag{2.3}$$

Where W is a set of possible worlds, $R \subseteq W \times W$ is a binary relation on W that indicates which worlds are possible relative to each other and π is a valuation function that indicates which propositions are true in a given world ($\pi : W \to \mathbb{P}(P)$). The semantics of such logics are determined by the satisfaction relation $\models$ which holds between a pair of a model M and reference world w and a formula as follows:

$$\begin{array}{lll}
\langle M,w\rangle \models \top & & \\
\langle M,w\rangle \models p \in P & \iff & p \in \pi(w) \\
\langle M,w\rangle \models \neg\psi & \iff & \langle M,w\rangle \not\models \psi \\
\langle M,w\rangle \models (\psi \vee \phi) & \iff & \langle M,w\rangle \models \psi \vee \langle M,w\rangle \models \phi \\
\langle M,w\rangle \models \Box\psi & \iff & \forall w' \in W \cdot (w,w') \in R \Rightarrow \langle M,w'\rangle \models \psi \\
\langle M,w\rangle \models \Diamond\psi & \iff & \exists w' \in W \cdot (w,w') \in R \Rightarrow \langle M,w'\rangle \models \psi
\end{array}$$

A formula which is satisfied by some model and world pair is called *satisfiable*; a formula which cannot be satisfied by any model/world pair is *unsatisfiable.* If $M = \langle W, R, \pi\rangle$ and $\forall w \in W \cdot \langle M,w\rangle \models f$ then f is *true* in M. If a formula is true in each of a class of models, it is *valid* in that class. If it is valid under the class of all models then it is *valid simpliciter.* If f is valid simpliciter, we denote this $\models f$.

There are two fundamental axioms of such a modal logic – the first is *axiom K*:

$$\models \Box(\psi \Rightarrow \phi) \Longrightarrow (\Box\psi \Rightarrow \Box\psi) \tag{2.4}$$

The second is the *necessitation rule*:

$$(\models \psi) \Rightarrow (\models \Box\psi) \tag{2.5}$$

Such a modal logic is translated into an epistemic logic for a MAS with agents $A_1, \ldots, A_n$ by replacing the operator $\Box$ with K_i (for each agent). The formula $K_i\psi$ means "A_i knows ψ". A model becomes $M = \langle W, (R_1, \ldots, R_n), \pi\rangle$ where R_i defines worlds between which A_i cannot distinguish. This allows us to reason about what agents know (and what they know they know as in $K_iK_i\psi$ etc).

Problems of modal methods

In such a system, the fundamental axioms of (2.4) and (2.5) pose a serious problem. From (2.4) it follows that an agent's knowledge is closed under implication (logical consequence). From (2.5) it follows that any agent knows all valid formulae - amongst other things this includes infinitely many tautologies. These are both counter-intuitive to a definition of knowledge; combined they form the problem of *logical omniscience.*

Such epistemic logics are amongst the most fundamental and long-standing approaches to applying a logical formalisms to agent systems. Various extensions have been made to address the problem of logical omniscience such as using a meta-language and an object-language as done by Moore (1985). Wooldridge and Jennings (1995) present a summary of several such approaches.

These methods, however, are all *mentalistic* approaches in that they focus on "agents' internal mental states, and attitudes" (Cliffe, 2007). However, there are problems reasoning in this way, summarised by Cliffe (2007) as:

- mentalistic approaches assume a *common rationality* for all agents, i.e. all agents will interpret the same act in the same way
- in general, it is not (practically) verifiable that agents actually held the preconditions for an action in their mental state before performing it (that they "respect the semantics" of the communication framework (Wooldridge, 1998))

To address these, we need to used models based on *social semantics* of a system which limit semantics to those properties that are visible to an external observer. In addition, these methods are more tailored to the application of modelling institutions as a whole (rather than individual agents).

In addition, classical first-order logic is *monotonic* – that is to say that for formulae ψ and ϕ, the following is true (Mueller, 2006):

$$\psi \models \phi \Longrightarrow \forall \psi' (\psi \wedge \psi' \models \phi) \tag{2.6}$$

This means that, in the event of adding more 'knowledge' we can draw more conclusions, but we cannot revise those we have already drawn. Clearly this is not desirable for reasoning about institutions – if an agent reaches some conclusion we want it to be able to change its mind about that conclusion if it gains more information. What we need, then, is some kind of representation which allows for *non-monotonic* reasoning.

2.2.2 Event Calculus

The *Event Calculus* (EC) is a *many-sorted first-order logic* designed for commonsense reasoning, that addresses time by the notion of events. This section is based on Kowalski and Sergot (1986) who provide an overview of the EC and Mueller (2006) who provides a complete definition of EC.

Unlike the *situation calculus*, which uses global states ("situations"), EC deals with time periods defined by local events. While the situation calculus uses *branching time*, time in EC is *linear*. The situation calculus suffers from the *frame problem* – it is necessary to reason that a relationship in a situation (and is not affected by some event) will continue to hold in the next situation. The need to deduce inertia in this fashion makes the situation calculus "so computationally inefficient as to be intolerable" (Kowalski and Sergot, 1986). McCarthy and Hayes (1969) provide a fuller description of the frame problem.

EC is a many-sorted logic: there is a set of sorts and for each sort in this set a (possibly empty) set of subsorts. Each constant, variable, function symbol and the arguments to predicates and function symbols have a specified sort. EC uses three sorts:

1. An *event* sort.

2. A *fluent* sort (Boolean fluents).
3. A *timepoint* sort. The subsort of timepoints is the real number sort.

We present here the main predicates of EC as described by Mueller (2006). Variables of the event sort are denotes $e, e_1, e_2, \ldots$; similarly the fluent sort is denoted f, f_i and the timepoint sort t, t_i.

$Happens(e, t)$ Event e happens at timepoint t.

$HoldsAt(f, t)$ Fluent f is true at timepoint t. If f is false at t we write $\neg HoldsAt(f, t)$.

$ReleasedAt(f, t)$ Fluent f is released from the *commonsense law of inertia*[2] at timepoint t. If f is bound by the commonsense law of inertia (at t) then we denote this $\neg ReleasedAt(f, t)$.

$Initiates(e, f, t)$ Event e occuring at t will cause f to be true and not released (from the commonsense law of inertia) after t.

$Terminates(e, f, t)$ Event e occuring at t will cause f to be false and not released after t.

$Releases(e, f, t)$ If e occurs at t then f will be released after t.

$Trajectory(f_1, t_1, f_2, t_2)$ If f_1 is initiated by an event occuring at t_1 and $t_2 > 0$ then f_2 will be true at $t_1 + t_2$.

$AntiTrajectory(f_1, t_1, f_2, t_2)$ If f_1 is terminated by an event at t_1 and $t_2 > 0$ then f_2 will be true at $t_1 + t_2$.

EC is non-monotonic due to the use of *circumscription*[3]. Circumscription is a second-order logic technique, used in EC to minimise the extension of predicates; it also provides non-monotonic reasoning as shown below. Consider the following examples (from Mueller (2006)):

$$Initiates(SwitchOn,\ LightOn, t)$$
$$Terminates(SwitchOff,\ LightOn, t)$$
$$F = Happens(SwitchOn,\ 3)$$

The circumscription of $Happens$ in formula F in this example gives us that:

$$(e = SwtichOn \wedge t = 3) \Leftrightarrow Happens(e, t) \tag{2.7}$$

This enables us to conclude (as we would expect) that $HoldsAt(LightOn, 7)$. But if we substitute F for F' where:

$$F' = Happens(SwitchOn, 3) \wedge Happens(SwitchOff, 6)$$

[2]The commonsense law of inertia says that unless a fluent is affected by an event, its value persists unchanged.

[3]For brevity, no formal definition of circumscription is given here, see Mueller (2006) for such a definition.

Then the circumscription of $Happens$ in (2.7) becomes:

$$(e = SwtichOn \wedge t = 3) \vee (e = SwitchOff \wedge t = 6) \Leftrightarrow Happens(e, t) \tag{2.8}$$

We no longer conclude $HoldsAt(LightOn, 7)$; we now conclude $\neg HoldsAt(LightOn, 7)$ as expected.

The Event Calculus is based on first-order and second-order logic, making it powerful and expressive. However, this comes at a computational cost (Cliffe, 2007). In the next section we consider *action languages*, reasoning mechanisms that rely instead on propositional logic.

2.2.3 Action Languages

Action languages allow us to examine the social semantics of an institution. This is achieved by focusing on the concrete actions performed by agents, rather than their knowledge and internal mental state (as in epistemic logics). A summary of action languages is given by Gelfond and Lifschitz (1998).

The following definitions are key in Gelfond and Lifschitz's descriptions of action languages.

Definition 2.1 *A* action signature *is a triple* $\langle \mathbf{V}, \mathbf{F}, \mathbf{A} \rangle$ *where* $\mathbf{V}$, $\mathbf{F}$ *and* $\mathbf{A}$ *are non-empty sets.* $\mathbf{V}$ *is a set of value names,* $\mathbf{F}$ *a set of fluent names and* $\mathbf{A}$ *of action names.*

The exact meanings of these sets are left abstract, but intuitively fluents represented by symbols in $\mathbf{F}$ are properties which take values from $\mathbf{V}$. Symbols of $\mathbf{A}$ represent acts that can be carried out by participants within the system.

Definition 2.2 *A* transition system *of an action signature* $\langle \mathbf{V}, \mathbf{F}, \mathbf{A} \rangle$ *is a set of states,* S*, a valuation function* $V : \mathbf{F} \times S \to \mathbf{V}$ *and a transition relation* $R \subseteq S \times \mathbf{A} \times S$*.*

States in S provide some abstract representation the condition of the world at a particular time. A triple $(s, A, s') \in R$ means that executing action A in state s might result in state s'. The possible results of executing A in s are given by $\{s' | (s, A, s') \in R\}$. Denoting this set R_A, we say that A is *executable* if and only if $R_A \neq \emptyset$ and that A is *deterministic* if and only if $|R_A| = 1$. The *value* of some fluent P in state s is $V(P, s)$.

Concurrent execution of actions is achieved by defining a set $\mathbf{E}$ of "elementary action names" and making the action names in $\mathbf{A}$ the result of a truth valued function over $\mathbf{E}$:

$$\mathbf{A} = \{\top, \bot\}^{\mathbf{E}} \tag{2.9}$$

Then we use a function $A(E) \in \{\top, \bot\}$ for some elementary action, E. To execute A we then execute all (elementary) actions such that $A(E) = \top$.

An action description is *propositional* if $\mathbf{V} = \{\top, \bot\}$. A *propositional interpretation* of some set of symbols X is a valuation for every symbol of X by a function $I : X \rightarrow \{\top, \bot\}$. We denote the value given to P under some interpretation, i, as $i(P)$.

The following sections on the action languages $\mathcal{A}$ and $\mathcal{C}$ are based on the summaries of these languages by Gelfond and Lifschitz (1998).

The $\mathcal{A}$ Action Language

$\mathcal{A}$ uses propositional action signatures of the form $\langle\{\top, \bot\}, \mathbf{F}, \mathbf{A}\rangle$.

Definition 2.3 *A* proposition *of $\mathcal{A}$ is a statement of the form:*

$$A \textit{ causes } L \textit{ if } F \tag{2.10}$$

Where A is an action, L a single literal (called the head*) and F a (possibly empty) conjunction of literals.*

When F is the empty conjunction we denote this $\top$ and we can abbreviate the proposition to:

$$A \text{ causes } L \tag{2.11}$$

An *action description* in $\mathcal{A}$ is a set of propositions. The transition system for an action description D is $\langle S, V, R\rangle$ where:

- S is the set of all interpretations of $\mathbf{F}$
- $V(s, P) = s(P)$
- Let $E(A, s)$ be the set of literals, L, such that the proposition

  $$A \text{ causes } L \text{ if } F$$

  is in D and the interpretation s satisfies F. Then R is:

  $$R = \big\{(s, A, s')|E(A, s) \subseteq s' \subseteq E(A, s) \cup s\big\} \tag{2.12}$$

There will be at most one s' for any A and s that satisfies (2.12) – i.e. $\mathcal{A}$ is deterministic. Inertia is also encoded in the language since (2.12) requires that fluent values in s' were either caused by the execution of A in s, or were in s already.

The $\mathcal{C}$ Action Language

$\mathcal{C}$ extends $\mathcal{A}$ in several ways: inertia in $\mathcal{C}$ is not a part of the language (i.e. we can introduce it if we wish), $\mathcal{C}$ allows for non-deterministic and/or concurrent execution of actions and

Proposition	**Abbreviation For**
U causes F if G	caused F if $\top$ after $G \wedge U$
inertial F	caused F if F after F
inertial $F_1, \ldots, F_n$	n separate inertial propositions
always F	caused F if $\bot$ after $\neg F$
nonexecutable U if F	caused $\bot$ after $F \wedge U$
default F if G	caused F if $F \wedge G$
U may cause F if G	caused F if F after $G \wedge U$

Table 2.1: Abbreviations in $\mathcal{C}$

$\mathcal{C}$ has two classes of proposition: *static* and *dynamic laws*. Action descriptions in $\mathcal{C}$ are of the form $\langle\{\top, \bot\}, \mathbf{F}, \{\top, \bot\}^{\mathbf{E}}\rangle$. Note that we also require $\mathbf{F} \cap \mathbf{E} = \emptyset$.

In $\mathcal{C}$ a *state formula* is a propositional combination of fluent names and a *formula* is a propositional combination of fluent and elementary action names. A *static law* is of the form:

$$\text{caused } F \text{ if } G \tag{2.13}$$

A *dynamic law* is of the form:

$$\text{caused } F \text{ if } G \text{ after } U \tag{2.14}$$

Where F and G are state formulae and U is a formula. As in $\mathcal{A}$, if $G = \top$ we can abbreviate these to get "caused F" and "caused F after U" respectively. We call F the *head* of the law. An action description in $\mathcal{C}$ is a set of static and dynamic laws.

Let D be an action description in $\mathcal{C}$, then the transition system described by D is $\langle S, V, R\rangle$ where:

- S is the set of all interpretations, s, of $\mathbf{F}$ such that for each static law "caused F if G", then if s satisfies G, s satisfies F
- $V(s, P) = s(P)$
- R is the set of triples (s, A, s') where:
  - if s' satisfies G and the static law "caused F if G" is in D then s' satisfies F
  - if s' satisfies G, $s \cup A$ satisfies U and the dynamic law "caused F if G after U" is in D then s' satisfies F

$\mathcal{C}$ features a number of abbreviations to ease use which are given in Table 2.1. In these laws, F and G are state formulae and U is a propositional combination of elementary action names (i.e. a formula with no fluent names).

The $\mathcal{C}+$ Action Language

Giunchiglia et al. (2001) propose $\mathcal{C}+$ an extension to $\mathcal{C}$ in order to allow for *multi-valued* fluents – that is, in the action signature $\langle \mathbf{V}, \mathbf{F}, \mathbf{A} \rangle$ we allow $\mathbf{V}$ to be any non-empty set. A detailed description of $\mathcal{C}+$ (and its extension to $(\mathcal{C}+)^{++}$) is provided by Sergot (2004) while a summary of $\mathcal{C}+$ is given by Mueller (2006). To do this, the concept of a *multi-valued propositional signature* is introduced:

Definition 2.4 *A* multi-valued propositional signature *is a set of symbols called* constants *and a non-empty set, $Dom(c)$, of symbols assigned to each constant c (the domain of the constant).*

An *atom* of a signature, σ, is an expression of the form $c = v$ where $c \in \sigma$ and $v \in Dom(c)$. An *interpretation* of a signature is a function which maps every constant of the signature to its domain. We say interpretation I *satisfies* the atom $c = v$ (denoted $I \models c = v$) if $I(c) = v$. Formulae are propositional combinations of atoms (and satisfaction is extended to formulae in the normal way for propositional logic). Interpretation I is a *model* for a set X of formulae if I satisfies all formulae in X. If all models for X satisfy some formula F then X *entails* F (denoted $X \models F$).

A $\mathcal{C}+$ description has a multi-valued signature $\sigma = \sigma^{fl} \cup \sigma^{act}$ where (σ^{fl} is a set of fluent symbols and σ^{act} a set of action symbols). A *state formula* is a formula of σ^{fl} and an *action* is an interpretation of σ^{act}.

Static and dynamic laws in $\mathcal{C}+$ look the same as in $\mathcal{C}$:

$$\text{caused } F \quad \text{if} \quad G \tag{2.15}$$

$$\text{caused } F \quad \text{if} \quad G \text{ after } H \tag{2.16}$$

Where F and G are state formulae and H is any formula of σ. As in $\mathcal{C}$ we call F the *head* of the formulae (in (2.15) and (2.16)) and an action description is a set of propositions. Again, a number of abbreviations are introduced for ease of use (see Table 2.2). In these abbreviations, F is a state formula, H a formula and α is a Boolean action symbol.

The semantics of the transition system defined by an action description in $\mathcal{C}+$ are different from those in $\mathcal{C}$ in that:

- S is the set of interpretations of σ^{fl} which satisfy $F \subset G$ for every static law "caused F if G" in D
- A formula F is *caused* in the transition (s, A, s') if either:
  - the static law "caused F if G" is in D and $s' \models G$
  - the dynamic law "caused F if G after H" is in D and $s' \models G \wedge s \cup A \models H$

Proposition	Abbreviation For
α causes F if H	caused F if $\top$ after $\alpha = \top \wedge H$
$\alpha_1, \ldots, \alpha_k$ causes F if H	caused F if $\top$ after $\alpha_1 = \top \wedge \cdots \wedge \alpha_k = \top \wedge H$
nonexecutable $\alpha_1, \ldots, \alpha_k$ if H	$\alpha_1, \ldots, \alpha_k$ causes $\bot$ if H
inertial F	caused F if F after F
never F	caused $\bot$ if F

Table 2.2: Abbreviations in $\mathcal{C}+$

The $(\mathcal{C}+)^{++}$ Action Language

$(\mathcal{C}+)^{++}$ is an extension to $\mathcal{C}+$ that is "designed for representing norms of behaviour and institutional aspects of (human and computer) societies" (Sergot, 2004). This is achieved through two main additions:

1. The ability to express "counts as" relationships between actions (cf. "conventional generation" of events, Goldman 1976 (cited in Cliffe, 2007)).
2. The ability to express permitted states of a transition system and permitted transitions.

The counts as relation (intuitively) tell us that the occurrence of one event (i.e. an action being performed) is also the occurrence of another. This can be used for the generation of institutional events from 'physical' events. More formally, we define this over types of transitions (s, A, s'). We add expressions of the form (where α and β are action formulae i.e. have signature σ^{act}):

$$\alpha \text{ counts_as } \beta \tag{2.17}$$

These expressions behave as Boolean fluents so they may be used in the both the head and body of other propositions. The set of action types is the powerset of action names, $\mathbb{P}(\mathbf{A})$. Transition (s, a, s') is of type $X \subseteq \mathbf{A}$ if and only if $a \in X$.

Permission is handled by *state permission laws* of the form:

$$\text{not-permitted } F \tag{2.18}$$

Where F is a fluent formula (i.e. has signature σ^{fl}). An *action permission law* is of the form :

$$\text{not-permitted } \alpha \text{ if } \psi \tag{2.19}$$

Where α is an action formula and ψ a formula.

For a detailed presentation of $(\mathcal{C}+)^{++}$ see (Sergot, 2004).

2.2.4 ASP

Answer Set Programming (ASP) is a declarative logic programming paradigm that allows for nonmonotonic reasoning. One might ask why ASP should be used when we already

have a well-established, declarative, nonmonotonic logic programming language in Prolog? The answer is that Prolog associates *negation as failure* with *classical negation*. These are two fundamentally different forms of negation; classical negation (e.g. $\neg x$) corresponds to "I know x is not true" while the negation as failure, **not** x, corresponds to "I do not know that x is true".

Since Prolog associates negation as failure with classical negation, it makes the *closed world assumption*. This is the assumption that a program specification is complete and so if x being true cannot be derived from the program, then x must be false. ASP features both classical negation and negation as failure, and so is not subject to the closed world assumption (Baral, 2003; Cliffe et al., 2005). This is useful in that it allows us to reason with incomplete information; in the case of agents within an institution, for example, this is obviously beneficial.

We present here an overview of ASP as formalised in the language AnsProlog* (Programming in Logic with Answer Sets) (Baral, 2003). In AnsProlog*, a *term* is defined as:

- a variable is a term
- a constant is a term
- for a function symbol f with arity n and terms $t_1, \ldots, t_n$ then $f(t_1, \ldots, t_n)$ is a term

If p is a predicate symbol of arity n then $p(t_1, \ldots, t_n)$ (where $t_1, \ldots, t_n$ are terms) is an *atom*. A term or an atom is called *gound* if it contains no variables. There are three types of *literals* in AnsProlog*:

1. If a is an atom then a and $\neg a$ are *literals*.
2. If a is an atom then a and **not** a are *naf-literals*.
3. If L is a literal then L and **not** L are *gen-literals* (called *extended* literals by Cliffe et al. (2005)).

A *rule*[4] has the form:

$$L_0 \leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n. \tag{2.20}$$

Where each L_i is a literal. In the above rule, $\{L_0\}$ is called the *head* of the rule. If the head of the rule is empty we may write this using $\perp$ as the head (or simply omit the head altogether as in "$\leftarrow a$."). Such a rule is called a *constraint*. The *body* of (2.20) is $L_1, \ldots, \textbf{not } L_n$. The body should be read as a conjunction. A rule with an empty body is called a *fact*:

$$L \leftarrow . \tag{2.21}$$

[4]The simplified version of a rule from Cliffe et al. (2005) where the head may contain at most one literal is used, rather than as in Baral (2003) where the head of a rule may contain an arbitrary number of literals.

This is abbreviated "$L.$". A *program* is a finite set of rules.

Definition 2.5 *For a program P, the set of all ground terms which may appear in P is called the* Herbrand Universe *of P. We denote this $\mathcal{U}_P$.*

For example, given the program $P = \{a., f(X).\}$ where a is a constant, f is a function symbol and X is a variable then $\mathcal{U}_P = \{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$.

Definition 2.6 *For a program P, the set of all ground atoms which may appear in P is called the* Herbrand Base *of P. We denote this $\mathcal{B}_P$.*

A literal is ground if the atom in the literal is ground. A rule is ground if all literals in the rule are ground. A program is ground is all rules in the program are ground. The ground version of a program P (denoted $ground(P)$) can be obtained by the process of *grounding* – the values variables can take are determined by the ground terms in the program; rules are grounded by replacing variables with (all possible combinations) of the ground values they make take.

Often we use instead AnsDatalog*, a subset of AnsProlog* in which function symbols are not permitted. Then the Herbrand Universe of an AnsDatalog* program is exactly the finite set of constants of the program.

An *answer set* of a program P is defined in terms of the *stable model semantics* (Gelfond and Lifschitz, 1988). For a program P let $\Pi = ground(P)$. An *Herbrand interpretation* of Π is a subset of $\mathcal{B}_\Pi$. The rule (2.20) is *satisfied* by interpretation I of Π under the following conditions:

- if $L_0 \neq \bot$ then $\{L_1, \ldots, L_m\} \subseteq I \wedge \{L_{m+1}, \ldots, L_n\} \cap I = \emptyset \Longrightarrow L_0 \in I$
- if $L_0 = \bot$ then $\{L_1, \ldots, L_m\} \nsubseteq I \vee \{L_{m+1}, \ldots, L_n\} \cap I \neq \emptyset$

An Herbrand interpretation of Π which satisfies every rule of Π is called an *Herbrand model* of Π. A model, M, is *minimal* if $\nexists M'(M' \subset M)$ where M' is also an Herbrand model of Π. AnsDatalog$^{-\mathbf{not}}$ is a subset of AnsDatalog* in which negation as failure is not allowed.

Definition 2.7 *An* answer set *of an AnsDatalog$^{-\mathbf{not}}$ program is a minimal Herbrand model of that program.*

To get from an AnsDatalog* program to an AnsDatalog$^{-\mathbf{not}}$ program, we use the *Gelfond-Lifschitz reduct* (Gelfond and Lifschitz, 1988). Given an AnsDatalog* program Π and a set of ground atoms M then we can obtain an AnsDatalog$^{-\mathbf{not}}$ program Π_M as follows:

- Remove from Π any rule containing **not** p in its body where $p \in M$
- Remove from the body of all remaining rules any literal **not** s where $s \notin M$

We are now able to define the answer sets of a (ground) AnsDatalog* program Π.

Definition 2.8 *The* answer sets *of a ground AnsDatalog* program* Π *are all sets of ground literals M such that M is an answer set of* Π_M.

ASP is "powerful and intuitive... for modelling reasoning and verification tasks" (Cliffe et al., 2005). In addition, there are a number of established answer set solvers (i.e. programs which will take an ASP program and compute answer sets). Two of the most commonly used are Smodels (Niemelä and Simons, 1997) and DLV (Eiter et al., 1998). Cliffe et al use action languages in combination with ASP to model and reason about institutions; an action language is used to model the actions and events then this is mapped to ASP, allowing us to query and verify the models (Cliffe et al., 2006).

2.3 InstAL

Cliffe et al. (2005; 2006; 2007; 2008) have worked on modelling institutions using ASP. As part of this work, the language InstAL (*institution action language*) has been developed in order to aid the modelling of institutions. InstAL was proposed by Cliffe (2007), this represents the most detailed source on the language. In addition see (Cliffe et al., 2005) for their model of institutions, (Cliffe et al., 2006) for an overview of InstAL and (Cliffe et al., 2008) for an example of the use of InstAL.

Institutions specified in InstAL conform to the formal model of institutions described in Appendix A.1 (p82). An InstAL specification is translated into an ASP implementation of the model. The details of this translation and how the model is implemented in ASP are given by Cliffe (2007).

This section presents a summary of their work in the context of this project. Section 2.3.1 gives a summary of the InstAL language and Section 2.3.2 gives an overview of the current situation regarding the development of the language.

2.3.1 InstAL Language Overview

This section presents InstAL, the institution action language, by a series of examples. These examples are taken from Cliffe et al. (2008) in order to illustrate InstAL. They describe the use of InstAL to model an institution which regulated a single round of bidding in a Dutch auction. A full presentation of the language is given in Appendix A.2.

An institution specification begins by naming the institution being described:

```
institution dutch;
```

InstAL allows the definition of types simply by naming them. The domains of specified

types must then be given in separate *domain definition* files. Types are declared as follows:

```
type Bidder;
type Auct;
```

Events are declared giving the type, name and parameters (if any):

```
create event createdar;
exogenous event annbid(Bidder, Auct);
inst event bid(Bidder, Auct);
```

Fluents (Boolean properties of the system) are declared similarly:

```
fluent havebid;
fluent onlybidder(Bidder);
```

The effects of events on fluents are given using *initiation* and *termination* rules and events can *generate* other events:

```
annbid(B, A) generates bid(B, A);
bid(B, A) initiates havebid, onlybidder(B) if not havebid;
bid(B, A) terminates pow(bid(B, A)), perm(bid(B, A)), perm(annbid(B, A));
```

In the above description, we see two of the three built in fluents:

- `fluent perm(Event);` – the specified event is permitted
- `fluent pow(Event);` – the specified event is empowered (i.e. effective)
- `fluent obl(Event,Event,Event);` – an obligation to perform the first event before the second occurs; if this is not fulfilled the third event is triggered as a sanction

2.3.2 State of Development

An InstAL problem consists of (from Cliffe, 2007):

- One or more institution descriptions (as defined in Appendix A.2, p88). Each description describes a single institution or multi-institution system.
- A domain description which defines the domains of types and static properties.
- A trace program providing definitions of the sets of traces to be generated.
- A query program to specify properties of the institution which we are interested in. This restricts the traces generated to those matching the conditions of the query.

Currently, the institution and domain descriptions are written using the InstAL syntax defined above (in any ASCII text editor). The trace and query programs are written as ASP programs using the syntax of Smodels (Niemelä and Simons, 1997). Currently there is a tool which takes the institution and domain descriptions and produces an answer set program (using Smodels syntax). This, together with the query and trace programs, are grounded using LParse (part of the Smodels toolset). The grounded program from LParse is then solved by Smodels (that is, the answer set(s) of the program are computed). There are tools to visualise the answer sets either as a list of (observable) events in the trace represented by the answer sets or as a graph showing state transitions. An output graph is described in the GraphViz language – GraphViz tools are be used to produce a graph from this.

This set of InstAL and external tools allows successful modelling of and reasoning about institutions. However, the tool set is in a fairly early stage of development and the following areas for extension have been identified:

- The need for a more integrated toolset to unite all of these stages involving a number of separate programs (several of which are “third-party” programs).
- The InstAL language provides a layer of abstraction from ASP that eases the specification of institutions and multi-institution systems. No such abstraction is provided for the query and trace programs which must be specified in ASP directly. The ability to specify typical queries in a syntax similar to InstAL would be beneficial.
- New output formats could be provided (e.g. output to LaTeX)
- Development of an enhanced editor for InstAL with the kind of features we find typically in IDEs.
- There is no well defined methodology for specifying institutions in InstAL. This can cause a problem of ‘where to start?’ This could be addressed by either formalising in the language description the order of an institution specification or just developing and documenting a ‘best practice’ methodology for the use of InstAL.

With these areas in mind, this project will focus largely on two issues: provided a more comprehensive toolset for institution development with InstAL and developing a simple query language to ease complex reasoning about institutions. Since InstAL generates ordered traces, queries on institutional models developed in InstAL typically involve determining properties of these traces, whether any traces exist that allow the system to reach certain states etc. That is, queries are largely *temporal* in nature. This means that developing a query language as part of/to work alongside InstAL can build on *temporal logics* such as CTL* (and its various sub-languages) (Emerson and Halpern, 1986) as well as the action and event based languages already considered. Any query language developed would only be in an embryonic stage given the scope of this work and so must be designed so that it could be extended to become fully functional query language in the future.

2.4 Summary

An *agent* is used in the Computer Science community to mean a range of related concepts. Section 2.1 began by considering the question “What is an agent?”. This showed that, while we may have an intuitive notion of an agent as some autonomous system that acts rationally to achieve its goals (whatever they may be), there is no generally agreed (formal) definition of an agent. Typically, we mean by “agent” a very high-level software system which performs intelligent and complex reasoning, exhibiting behaviour that is often compared to the way we think and reason. This gives rise to interpretations such as BDI where agents are considered using models originally intended for reasoning about human thought.

A key part of agent behaviour (usually) is the ability to socially interact with other agents – hence we use *multi-agent systems*. In order to impose some constraint and norms on agents within multi-agent systems, *institutions* were introduced. These use the concepts such as *permission*, *obligation*, *violations* and *institutional power* to allow for constraints to be imposed on MAS without imposing too great a restriction on agent autonomy.

In order to provide a mechanism for reasoning about agents and MAS, work has been done on developing formal models. Section 2.2 described a number of approaches to this. One such approach was the more recent development of *answer set programming*. This forms the basis for work on institutional modelling by Cliffe et al, who proposed the language InstAL as a tool to ease this process. Section 2.3 provided an introduction to InstAL and described the current state of development (and which areas remain open).

Chapter 3

The Institution Editor

One of the main aims of this project was to develop additional tool support for the development of institutions using InstAL. The current state of support of the development process is given by Cliffe (2007):

- one or more single or multi-institution InstAL descriptions are specified (in ASCII text)
- a domain definition is specified to ground types and static properties (in ASCII text)
- the InstAL specifications and domain definition are given as input to a command-line tool which translates these into a set of answer set programs
- a query program is specified in ASP (but see Chapter 4 (p44) for changes made to this stage)
- a trace program is generated by a command-line tool
- the ASP programs generated from the specifications are combined with the trace and query and given as input to LParse (Syrjänen, n.d.) which grounds the programs
- the grounded programs are solved by Smodels (Niemelä and Simons, 1997)
- the answer sets computed by Smodels can be visualised:
  - using a tool called *InstViz*
  - using a command-line tool which produces a graph described in the language of GraphViz[1]

This process is relatively complex and requires a number of separate tools:

1. A text editor to create InstAL specifications.

[1] `http://www.graphviz.org`

2. The translator that generates ASP from InstAL (`genasp`).
3. A utility to generate the time instants for traces (`gentime`).
4. An answer set grounder (LParse).
5. An answer set solver (Smodels).
6. A tool to visualize output which may be:
   (a) *InstViz*.
   (b) The utility to generate a GraphViz graph (`gengraph`).

Note also that the development of a query language documented in Chapter 4 adds an additional tool to translate the new query language into ASP. This tool is referred to in this chapter as the *InstQL translator*. A prototype implementation of this tool is detailed in Section 5.3 (p69).

The highest level goal of developing tool support for InstAL is to create a system that will allow the user to develop and reason about institution specifications. That is, in addition to offering functionality to write InstAL specifications, the tool will control the above reasoning process. This chapter details the development of such a system: *InstEdit*, the institution editor.

3.1 Requirements

Requirements for *InstEdit* were ascertained by consultation with the InstAL research community. These are the people who have developed InstAL as a tool to aid in modelling of institutions. As such, they represent the stakeholders for *InstEdit*. This section describes the requirements for *InstEdit* derived from discussion with this community. The fact that the intended user base is a small number of academics in the field of computer science makes the requirements elicitation process relatively straightforward.

3.1.1 Requirements Elicitation

This section gives (informally) the findings of discussion with the system stakeholders. Since the number of stakeholders is small, requirements were ascertained by informal interviews. This is an expensive technique for requirements elicitation, but the low number of stakeholders made it possible.

Three use cases for the system are identified (Figure 3.1):

- *creating institutions* – writing InstAL specifications (including domain definitions)

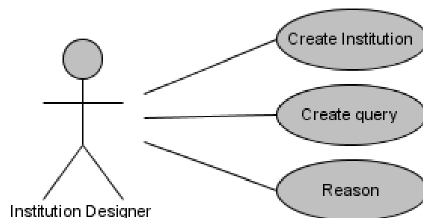

Figure 3.1: Use cases for *InstEdit*

- *creating queries* for institutions – writing queries in InstQL (a new query language for InstAL, see Chapter 4)
- *reasoning* about institutions – carrying out the process outlined above

The system must support the user in these three use cases. Since both InstAL and InstQL are written as ASCII text files, this means providing some text editor capabilities. Enhancing these basic facilities by features such as syntax highlighting, found in many integrated development environments (IDEs). To support the user in reasoning about institution specifications, the system must control and co-ordinate the reasoning process outlined at the start of this chapter.

As previously mentioned, this process involves a number of stages and separate tools. Figure 3.2 presents an analysis of the process as a sequence diagram. This highlights not only the number of stages and tools involved but also how dependent on the user the process is. The tools are all largely disjointed and it is up to the user to co-ordinate the process and launch each stage. Such a situation is undesirable as it requires a lot of effort from the user. To make reasoning with InstAL easier, the system should hide much of the complexity of the process from the user.

Current work with InstAL is carried out on the GNU/Linux operating system. The InstAL tools are written in Perl which is provided by many Linux distributions and LParse and Smodels also use Linux as their 'home environment'. However, Perl distributions exist for other platforms and LParse/Smodels have successfully been ported to other operating systems (Syrjänen, n.d.). At some future point, a different answer set grounder/solver may be used. It is desirable to make the system portable so that future development and use of the InstAL toolkit is not limited to a specific platform.

Work on the system should continue beyond the scope of this project. This project aims to deliver an initial functional version which can be extended and modified as required by the InstAL community. They will be responsible for maintaining the system after this project. The system should be adaptable to cope with possible future changes. For example, at some future point the answer set solver and grounder used may change as new grounders such as GrinGo (Gebser et al., 2007) and solvers such as Platypus (Gressmann et al., 2005) offer faster computation.

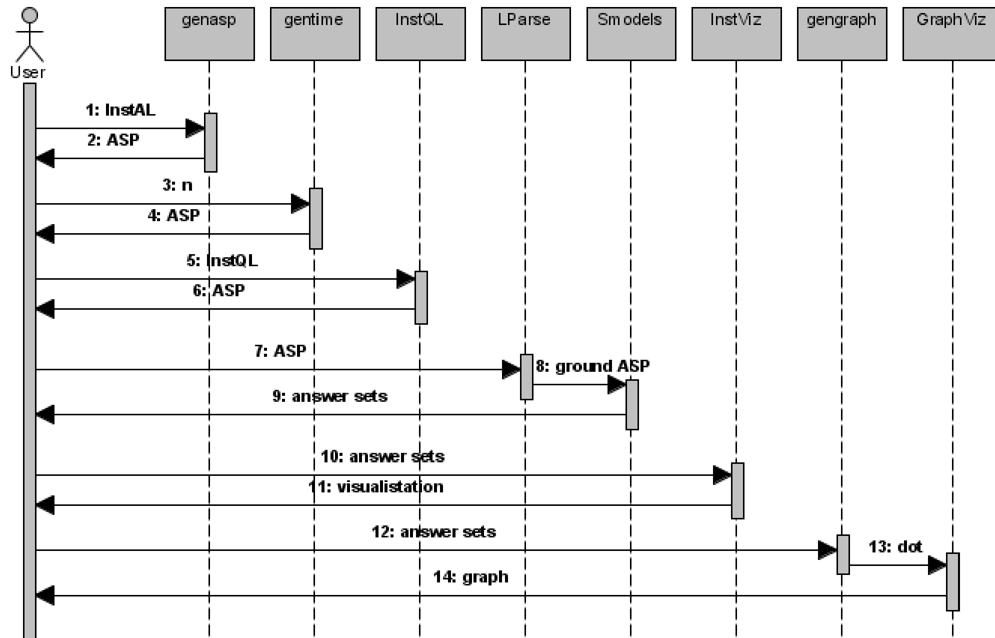

Figure 3.2: Sequence diagram of the current InstAL reasoning process.

3.1.2 Requirements Specification

In this section we formalise the properties, features and constraints described in the previous section into a requirements specification for *InstEdit*. This is split into two parts: a set of functional requirements describing features of the system and non-functional requirements describing more abstract constraints.

Functional Requirements

This section presents the functional requirements for *InstEdit*. Requirements beginning "The system **must**..." indicate high priority requirements that must be part of a successful system. Failure to deliver any such requirement indicates a failure of the project. Requirements beginning "The system **should**..." indicate lower priority requirements. An implementation failing to deliver some or all of these requirements is considered a (partial) success.

F1 The system **must** offer functionality of a simple text editor.

- **F1.1** The system **must** be able to save ASCII text files.
- **F1.2** The system **must** be able to open saved ASCII text files.

- **F1.3** The system **must** be able to edit ASCII text files.

F2 The system **should** provide syntax highlighting to help users detect errors in input.

F3 The system **must** automate the InstAL reasoning process.

- **F3.1** The system **must** provide a link to `genasp`.
- **F3.2** The system **must** provide a link to `gentime`.
- **F3.3** The system **must** provide a link to LParse.
- **F3.4** The system **must** provide a link to Smodels.
- **F3.5** The system **should** support output as a graph.
  - **F3.5.1** The system **should** provide a link to `gengraph`.
  - **F3.5.2** The system **should** provide a link to GraphViz.
- **F3.6** The system **should** provide a link to *InstViz*.
- **F3.7** The system **should** provide a link to the InstQL translator.

In requirements **F3.1** to **F3.7**, by "provide a link to" we mean that the user must be able to launch the specified external tool from *InstEdit*. This tool should operate on the relevant type of file for the institution the user is currently working on. For example, in the case of LParse (the ASP grounder used) the input should be the ASP representation of the institution that the user is currently working on. The output and results of doing so must be made available to the user within *InstEdit* (where appropriate).

Requirements **F1** and **F2** support the use cases of creating institutions and queries. Requirement **F3** supports the reasoning about institutions use case.

Non-Functional Requirements

Non-functional requirements are constraints on a system rather than specifications of the functionality that it must provide (Sommerville, 2004). The non-functional requirements for *InstEdit* arise from the fact that it is intended to be a system that will continue to evolve and be maintained by the InstAL research community beyond the scope of this project.

N1 The system **should** provide a graphical user interface to allow control of the various tools.

N2 The system **must** be maintainable.

- **N2.1** Maintenance documentation (such as APIs) **must** be produced during development to allow others to continue/revise development after the project is complete.
- **N2.2** The system **must** be adaptable. The architecture must permit changes of components/external tools with minimum impact.

N3 The system **should** be portable and run on as many platforms as possible.

N3.1 The system **must** run under the GNU/Linux operating system.

With the requirements specified, a testing plan was devised. The testing strategy used is described in Section 3.4 and the test plan is given in Appendix B.1.

3.2 High-Level Design

The software process used for development was an agile process based on incremental delivery. The functional requirements for the system (p30) are specified at a high level. There is scope for flexibility in how a system could satisfy these requirements. This makes an incremental process beneficial as it provides the ability to explore the requirements throughout the development process. This section describes the high-level architectural design of *InstEdit*; the following section presents the low-level designs for each increment as development progressed.

It was decided to implement *InstEdit* using Java. Java offers GUI support through the `Swing` and `AWT` libraries. Java is a very portable programming language due to its virtual machine architecture. The Javadoc tool enables generation of APIs from source code comments. Therefore, Java provides support for requirements **N1**, **N2.1** and **N3** making it an appropriate language for this project.

In order to satisfy requirement **N2.2**, it was necessary to devise a software architecture that allows for change. The most likely aspect of the system to change is the external tools. New versions of the InstAL and InstQL tools may be developed or a different answer set solver and/or grounder may be used. Even a more fundamental change such as using an alternative formalism to ASP would be reflected by a change in tools use. The process of writing InstAL specifications as text files and reasoning about them is much less likely to change. To this end, it was decided to handle all interaction with external tools through a sub-system. This allows external tools to change, so long as the interface between the sub-system and the rest of the system remains constant.

Figure 3.3 shows the high-level architecture of *InstEdit*. Three main sub-systems are identified:

- *GUI* – a sub-system to provide a text editor with a graphical interface with which the user can interact
- *translators* – a sub-system to control the external tools (since most of the tools perform some kind of translation on data e.g. InstAL to ASP, ASP to answer sets etc)
- *application* – a sub-system to provide utilities for *InstEdit* e.g. saving/loading settings such as locations of external tools

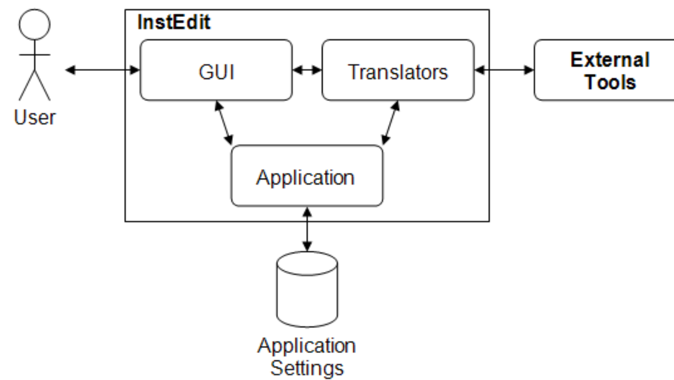

Figure 3.3: *InstEdit* architecture

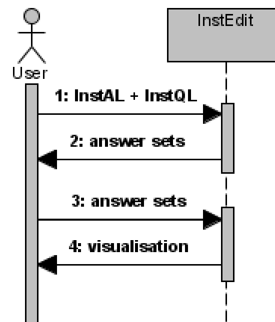

Figure 3.4: Sequence diagram of the InstAL reasoning process using *InstEdit*

To control the InstAL reasoning process from *InstEdit*, it was decided that the process should be split into two major stages. The first stage is translate and solve institution specification(s) (with optional domain definition and query) to get back answer sets corresponding to traces of the system. The second is to visualise the answer sets, either using *InstViz* or GraphViz. The split is made here since one, both or neither visualisation methods may be used an the answer sets. Splitting the process at this point allows the answer sets to be saved and then subsequently visualised without needing to translate and solve the specifications again.

The user will be able to launch these two stages from *InstEdit*. While the underlying reasoning process remains unchanged from that shown in Figure 3.2, *InstEdit* hides this from the user. From the user's perspective, the reasoning process becomes a much simplified version as illustrated in Figure 3.4.

Implementation of this architecture was organised as an incremental process over a number of feature-boxed versions. These versions cover all the functional requirements.

Version 0.1 Develop a text editor. Satisfies requirement **F1** (and all sub-requirements).

Version 0.2 Integrate with `genasp` to allow single institutions to be created and converted to ASP. Satisfies requirement **F3.1**.

Version 0.3 Integrate with `gentime`, LParse and Smodels. This version allows one or more single institutions, a multi-institution definition and domain definition to be translated into an ASP program and the answer sets of the program to be calculated. Satisfies requirements **F3.2**, **F3.3** and **F3.4**.

Version 0.4 Add syntax highlighting capabilities to the text editor. Satisfies requirement **F2**

Version 0.5 Integrate with visualisation tools. Integrate with the InstQL translator. This version will allow the full reasoning process to be carried out: the user specifies InstAL institution(s), domain definitions and an InstQL query. These are translated into an ASP programming which is solved and the answer sets can be visualised using *InstViz* or as a state transition graph created by GraphViz. Satisfies requirements **F3.5** (and sub-requirements), **F3.6** and **F3.7**.

Throughout the development process, the following principles were applied:

- All classes were commented with Javadoc style comments. This allows Javadoc to be run on the code and generate an API for the program. This aids maintenance and satisfies requirement **N2.1**.
- The system should be platform independent. For example, all file paths referenced in code should use a system-dependent file separator (available through the standard Java library class `System`) rather than hard-coding any specific file separator. This practice, combined with the wide availability of the Java runtime environment, satisfies requirement **N3**.

3.3 Low-Level Design & Implementation

This section presents the iterative design and implementation process used to deliver the five increments of *InstEdit* identified in the previous section.

3.3.1 Implementing a Text Editor

InstEdit 0.1 implemented the core of the GUI sub-system of the overall application. It provides the main application window which provides the functionality to edit text files. This satisfies requirements **F1**, **F1.1**, **F1.2**, **F1.3** and **N1**. This section describes the design and implementation of this increment.

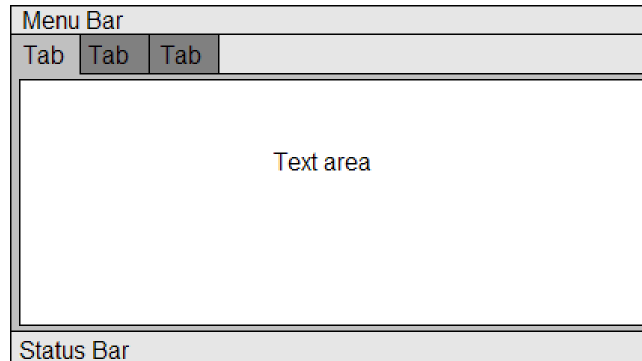

Figure 3.5: User interface design for the *InstEdit* main form.

The first stage of implementing *InstEdit* was to design the user interface. The interface designed is simple, more akin to a text editor than a rich IDE. This provides an uncluttered interface which should not distract the user. The selected design is shown in Figure 3.5. The features of the design are as follows:

- a *menu bar* to provide access to commands (such as save or open a file).
- a *tabs bar*, with tabs titled by the names of files open. *InstEdit* provides a tabbed interface which allows the user to work on multiple documents at once within the same window.
- an editable *text area* which will display the contents of the open file. Switching to a different tab will switch to a different text area showing the file open within that tab.
- a *status bar* to provide information to the user such as position of the caret within the file.

This design is intended to be simple and familiar. It is influenced by a number of common text editors and similar applications. To increase usability, the menus in the application have names which are commonly used such as "File", "Edit" and "View". These will contain options which are typically located on these menus (e.g. "Save" in the "File" menu), allowing the user to carry over experience in other applications to *InstEdit*. Figure 3.6 illustrates how this design was realised in the final application.

To facilitate future additions to *InstEdit*, an abstract class called `InstEditPanel` was implemented. This provides the superclass for anything to be displayed within a tab. This provides a flexible structure since the application can show anything in a tab which inherits from `InstEditPanel`. Future modifications can extend this class to provide new functionality within the tabbed interface. The API for this class specifies simple features that all components will require such as management of the tab title.

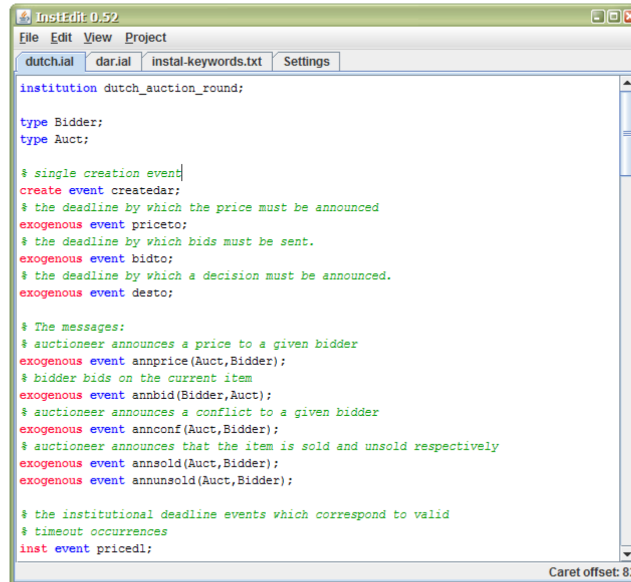

Figure 3.6: Screen shot of the final version of the *InstEdit* main form.

The text area used was an instance of class `JTextArea`. This is a simple text component in which all text must have a uniform format. The implementation of *InstEdit 0.4* (p39) delivers a tab with a different type of text area that supports rich formatting. This makes use of `InstEditPanel` in that both text areas used are provided within a subclass of `InstEditPanel`. The class which displays the text area is called `TextAreaPanel`. The simpler text area was used first (rather than using partial features of the more complex version from the outset) in order to enable a rapid prototype satisfying the requirements for this increment.

In order to provide user feedback, two listeners were added to the text area. One listens for key events which represent changes to the file being edited. This allows a label in the status bar to indicate whether all changes have been saved or not. The application also maintains this information internally so that it can prompt a user if the application is closed when there are unsaved changes. These measures prevent a user from accidentally losing work by forgetting to save. The second listener responds to movement of the caret by updating the status bar to display its current position within the file. This can help the user find errors since the external tools may give the line number of an error in a file.

3.3.2 Integration with InstAL

The second increment, *InstEdit 0.2*, integrates the text editor delivered by the first increment with `genasp`. This gives us a tool which allows the user to create an InstAL specification for an institution and then translate it into ASP. This first version of inte-

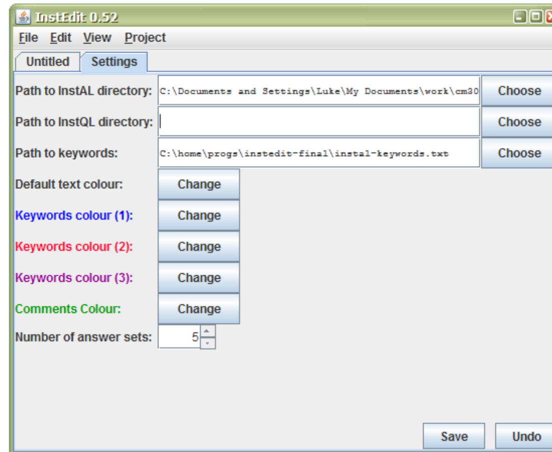

Figure 3.7: Screen shot of the settings tab of *InstEdit*.

gration is able to translate only a single file at a time, and so allows only translation of single institutions. Since multi-institutions require multiple single institution specifications and another file to specify how these institutions interact, translating these is not possible under *InstEdit 0.2*. This increment delivered requirement **F3.1**.

In order to achieve integration, this increment provided the basics of the application and translators sub-systems. These were combined with the base of the GUI sub-system from *InstEdit 0.1* to allow the user to control the translation process. The integration is achieved by calling `genasp` as an external process through the Java runtime environment. This technique allows a command (as one would normally type at a command-line) to be called from an application. This provides a simple approach to integrating `genasp`, a Perl script, within *InstEdit* (a Java application). This maintains a loose coupling between the translator and editor for InstAL. This allows changes to either program without affecting the other. As outlined in the design 3.2 (see Figure 3.3, p33), the loose coupling is ensured within *InstEdit* by providing a class in the translators sub-system to provide the interface for calling `genasp`.

Calling `genasp` in this manner creates the requirement for *InstEdit* to know where `genasp` is located. The application sub-system provides this functionality. This sub-system provides the facility to store application settings for *InstEdit* to disk and to load these at runtime. One such setting stores the location of `genasp` on the user's system. In order to allow the user to manage these settings, a new kind of tab was developed in the GUI sub-system. This tab has a simple design allowing the user to enter values for settings in fields or launch dialogs to select values. The final version of this tab is illustrated in Figure 3.7. The development of `InstEditPanel` for the first increment made it simple to develop this new kind of tab and add it within the application.

Another new kind of tab was added to display output from processes to the user. The

external tools are all command-line applications which print information and diagnostic text to the user. When the user launches external tools, a new tab is opened within *InstEdit* and output information from the tools is displayed in an uneditable text area on the tab.

A criticism of the approach adopted by *InstEdit* in connecting to the external tools is that these are called synchronously. Translating and solving complex institution descriptions can take a significant amount of time. While the external processes are running, the user must wait for the processes to complete. This is unsatisfactory as there is little feedback to inform the user of which stage in the process the system is executing. A better solution is to run the external processes asynchronously in a separate thread – this is left for future development.

When translation is complete, another new tab is opened which displays the ASP program generated. This uses the same type of tab as for creating the institution, allowing the user to add to or edit the program if required.

3.3.3 Integrating the Answer Set Solver

Requirements **F3.2**, **F3.3** and **F3.4** are satisfied by *InstEdit 0.3*. This extends the previous increment to complete the reasoning process up to visualisation of the traces (answer sets). This required integrating `gentime`, LParse and Smodels with *InstEdit*.

While `genasp` and `gentime` are command-line programs (and so calling them as external processes is the most logical way), Smodels offers a C++ library. The Java Native Interface (JNI) allows Java applications to access C/C++ libraries. This meant Smodels could be accessed as an external process (in the same manner as for `genasp` in *InstEdit 0.2*) or through the JNI. Using the library through JNI would allow greater control over how Smodels is integrated with the program. However, this approach was not used because:

- calling LParse and Smodels as external processes provides a looser coupling between *InstEdit* and the answer set solver used. Due to requirement **N2.2** this is desirable. The cost of moving from Smodels to a different solver is minimised using this approach.
- calling LParse and Smodels as external processes means the link to external tools is standardised since the InstAL tools are called in this manner.

LParse, Smodels and `gentime` were all integrated with *InstEdit* in the same was as `genasp` (see p36).

A complete InstAL reasoning problem consists of several files. For each (single) institution we have an InstAL specification. We also have a multi-institution specification which defines the operation of the system as a whole and a domain definitions file. To support working with multiple files, support for *projects* was added to *InstEdit*. When creating a project, the user gives the project a name and specifies where output for that project should be

saved. The user may then identify a multi-institution specification, a domain definition, an InstQL query (added for *InstEdit 0.5*) and many institution specifications. These are translated and combined with the time instants program created by `gentime` and some base institution programs provided with the InstAL tools (these specify properties of the model common to all institutions such as inertia of fluents etc). These programs are all grounded and solved together. The translators sub-system is designed to work with projects. A new tab type was added, with a similar design to the settings tab, to allow the user to create and manage projects.

To provide flexibility for the user, three modes of translation are provided:

1. Translate a project from InstAL into ASP, generate a specified number of time instants and solve the programs to compute traces. (This is the complete reasoning process up to visualisation.)
2. Translate a project from InstAL into ASP. This allows the user to inspect the generated ASP and if required make additions to the program (for example, query conditions written in ASP and not InstQL).
3. Solve an ASP program. This can be used following step 2 to complete the process.

During the translation process, intermediate representations are saved to the directory associated with the project. Translating and solving a project takes the institution designer from the InstAL specification(s) straight to the answer sets. However, the ASP programs created by `genasp` and `gentime` are saved, along with the output of Smodels (the answer sets). This makes maximum information available to the institution designer for later review if required.

3.3.4 Adding Syntax Highlighting

InstEdit 0.4 satisfies requirement **F2** by providing syntax highlighting. This helps users spot syntax errors and makes institution specifications more easily readable by highlighting keywords of InstAL and comments.

As mentioned during the implementation of *InstEdit 0.1* (p34), this required using a new kind of text area which supported multiple formats. The component used is `JTextPane` from the `Swing` library. Much of the functionality of `TextAreaPanel` (the subclass of `InstEditPanel` that contains the text area for *InstEdit 0.1*) was refactored into a new class `EditorPanel`. This extends `InstEditPanel` and provides the superclass for both `TextAreaPanel` and a new class `TextPanePanel`. `TextPanePanel` provides an editor that supports syntax highlighting for InstAL and InstQL documents.

Two types of highlighting are provided by *InstEdit*: keywords and comments. Three different levels of keyword highlighting are defined: language keywords, event types and predefined types and fluents. The definition of what the keywords are (and which level they belong to) is specified in an external file. This allows keywords to be added and removed

as InstAL develops. The application settings contain the path to this file and a class in the application subsystem provides the functionality to read this file and load lists of keywords (of each type). This class then provides a method to determine which keyword list (if any) a specified string belongs to. In addition, the application settings store the colours in which to highlight normal text, comments and each level of keywords. This allows the user to personalise the colour scheme used.

Actually implementing syntax highlighting involves two stages: recognising a change has occurred in a document and responding by highlighting the affected text. The architecture of `Swing` allows a document listener to monitor the text area used; the listener will be notified when any change occurs. Several alternative approaches to highlighting were considered:

1. A *listener* monitors the document for changes. As changes occur, the listener highlights the affected text.
2. A *listener* monitors the document for changes. When a change occurs, the listener stores the affected portion of the document. A *highlighter* asynchronously performs the stored highlights.
3. A *highlighter* asynchronously polls the document at short intervals to see if has changed since the last poll. If so the changed region is re-highlighted.

Approach 3 was dismissed for the amount of work this would create. The highlighter would need to maintain a record of the document state to detect changes. This would require maintaining a copy of the whole document contents in memory, which is computationally expensive. This method does not make use of the facilities provided by the `Swing` library.

The first option offers the advantage that changes will be guaranteed to be reflected in the document instantly. The problem with performing a synchronous highlight in this manner is that should a large change occur (e.g. pasting of a large volume of text) the user would have to wait for the highlighting to occur.

For these reasons, method 2 was selected. Each new `TextPanePanel` starts a syntax highlighter in a new thread. A listener for the text area in the panel records the location of changes and adds these to the end of a first-in-first-out queue. The highlighter monitors the head of this queue and when the queue is non-empty it removes the item at the head and highlights the appropriate region. This architecture is given in Figure 3.8.

In order to minimise the effort expended by the highlighter, the implementation seeks to minimise the area deemed affected by a change. This extends beyond the changed region because, for example, typing the last letter to complete a keyword affects the entire word or typing the comment symbol affects the rest of that line. Since InstAL has no multi-line constructs, only the line of the change is highlighted.

There are two cases when we need to highlight the entire file. The first is when a document is opened. This is handled by the syntax highlighter automatically since when text is loaded

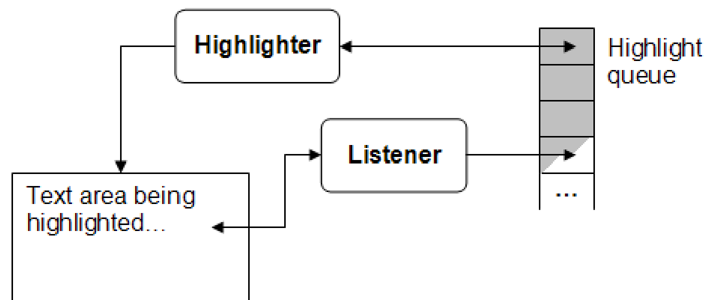

Figure 3.8: Syntax highlighting architecture.

into the text pane, this generates a change which spans the entire document. The second is when the user changes the application settings. Recall that the application settings define the colours that the syntax highlighter uses and the location of the file which defines the keywords to highlight. Changing the settings may change one or both of these properties. To deal with this case, a method is defined in `TextPanePanel` to add to the queue an item which indicates a change spanning the whole document.

3.3.5 Integrating Queries and Visualisation

The final increment of *InstEdit* completes the reasoning process. It integrates *InstEdit* with the InstQL translator, allowing queries to be specified and executed within *InstEdit*. In addition, *InstEdit* is linked with *InstViz*, `gengraph` and GraphViz to allow traces to be visualised. This fulfils requirements **F3.5** (and all its sub-requirements), **F3.6** and **F3.7**.

The file paths to tools and library programs for InstAL and InstQL must be known by *InstEdit*. Previous versions stored the path to each required file separately in the application settings. Since all the tools and programs required for translating and visualising InstAL are provided in the same directory by the InstAL tools, this was simplified by storing only the path to that directory. A new class within the application sub-system is then responsible for constructing file paths for the required files from this directory. The same is done for InstQL. This simplifies access to the external tools.

Syntax highlighting is extended to InstQL. InstQL supports comments in the same format as InstAL and has its own set of keywords. For simplicity, the InstQL keywords are stored in the same file as those for InstAL. The syntax highlighting component is then able to highlight InstAL specifications and InstQL queries since it knows of the keywords for both. The system at present does not distinguish between the two, since a text area does not identify the type of document within it. There is no way to signal a highlighting mode to the highlighter, this is left for a future extension.

As discussed in the high-level design (p32), the reasoning process is split into computing

and visualising traces. *InstEdit 0.3* (p38) provides the process up to computing traces (with the exception of queries). *InstEdit 0.5* allows the second stage to be carried out. The user is presented with two methods to visualise traces: *InstViz* or generating a graph.

When the user opts to generate a graph, `gengraph` and then GraphViz are called to produce the graph in PostScript format. Since *InstEdit* provides only tabs to view/edit text files, it is not currently possible to view this graph within *InstEdit*. However, the system design regarding tabs allows a new kind of tab to be developed and integrated easily that will allow PostScript files to be viewed within *InstEdit*. This is left for a future version.

3.4 Testing

In this section we present an overview of the testing process used for *InstEdit*. The main testing was black-box release testing to verify that each increment met its requirements. In addition, white-box component testing was performed for the syntax highlighting component of the GUI sub-system. This was the most complex portion of the application to implement and so additional testing was performed to ensure it functioned correctly.

In general, the functionality of *InstEdit* specified by the requirements (p30) is simple to verify. For example, for requirement **F1.1**, either the system is able to correctly save a file, or it is not. This makes black-box testing an appropriate strategy for *InstEdit*.

After each increment was completed, we performed release testing on that increment. We also performed regression testing for previous increments. This ensured integrating the new functionality had not caused problems with the existing functionality.

The exception to this strategy was requirement **F2**. Since this feature is more complex, white-box testing was performed to make sure the syntax highlighting component functioned correctly. The release testing for *InstEdit 0.4* was just regression testing to ensure implementing syntax highlighting had not caused problems elsewhere in the system.

This testing strategy was efficient in that it ensured that *InstEdit* meets its requirements and time intensive white-box testing was used only where appropriate.

Tests 8 and 9 for syntax highlighting revealed a flaw in the highlighting algorithm. The original implementation deemed the line on which the change occurred (determined by the location within the document of the start of the change) to be affected. This worked for normal typing but failed in two cases:

1. Pasting in text that spans several lines. The change starts at the point where the text was pasted but carries on up to the line where the pasted text ends.

2. Typing return to terminate a line when there is text on that line after the caret. The change is recorded as occuring on the line where return was pressed but may require the next line to be highlighted. For example, if the line is terminated part way through a keyword then this is split into two words which (in general) are not keywords. Since

> InstAL has single-line comments, terminating the line during a comment moves text down to the next line and out of the comment.

The algorithm was revised to start highlighting at the beginning of the line on which the change starts. The highlighting continues until the end of the line on which the change ends.

3.5 Summary

This chapter described the development process for *InstEdit*, the institution editor. This application allows the user to create InstAL specifications and reason about them within the same system. The InstAL reasoning process requires a number of separate tools. Prior to *InstEdit*, the user was responsible for managing this complex process. *InstEdit* provides control for much of the process, making development easier.

The requirements for *InstEdit* were ascertained by informal interviews with the InstAL development community. Section 3.1 (p28) describes the process of elicitation and motivation for the requirements. We then present the functional and non-functional requirements for *InstEdit* in Section 3.1.2.

Section 3.2 (p32) discusses the high-level design decisions taken. We decided to implement *InstEdit* in Java using an incremental delivery process. The design of *InstEdit* aims to deliver an application which is modifiable and maintainable to support a developing research area. In Section 3.3 (p34) we then describe selected issues arising during the incremental delivery of *InstEdit* in accordance with the high-level design.

InstEdit was tested according to a black-box release testing strategy (as discussed in Section 3.4, p42). This is augmented with white-box component testing for the syntax highlighting element of the system. Testing plans are given in Appendix B.

Chapter 4

A Query Language for InstAL

One of the major goals of the project is to design and implement a query language to operate on institution descriptions written in InstAL. As an action language specific to institutional modelling, InstAL provides a useful abstraction for the institution designer away from the underlying formal model specified in ASP. However, this level of abstraction is lost when it comes to specifying queries about an institution. This is a vital part of the modelling process but as yet there is no query language and so queries must be specified in ASP directly. This is undesirable for these reasons:

1. InstAL provides a level of abstraction above ASP making it easier to specify institutions. This is lost when it comes to query specification and the designer must use ASP which is more complex than required.
2. In order to write ASP queries for an InstAL specification, the designer is required to know how the InstAL will be translated into ASP.
3. A fully layered approach with both the specification of the institution(s) and queries in specific languages would allow the underlying logic to be altered without requiring any changes to institutions developed in InstAL.

The process of modelling an institution (or several institutions in a multi-institution specification) using InstAL and the currently available tools (up to the computation of answer sets) is shown in Figure 4.1 (adpated from Cliffe, 2007). The aim is to develop a query language in which the user can describe desired queries at a higher level and tools to translate this language into ASP. This will modify the early stages of Figure 4.1 to produce the development process shown in Figure 4.2.

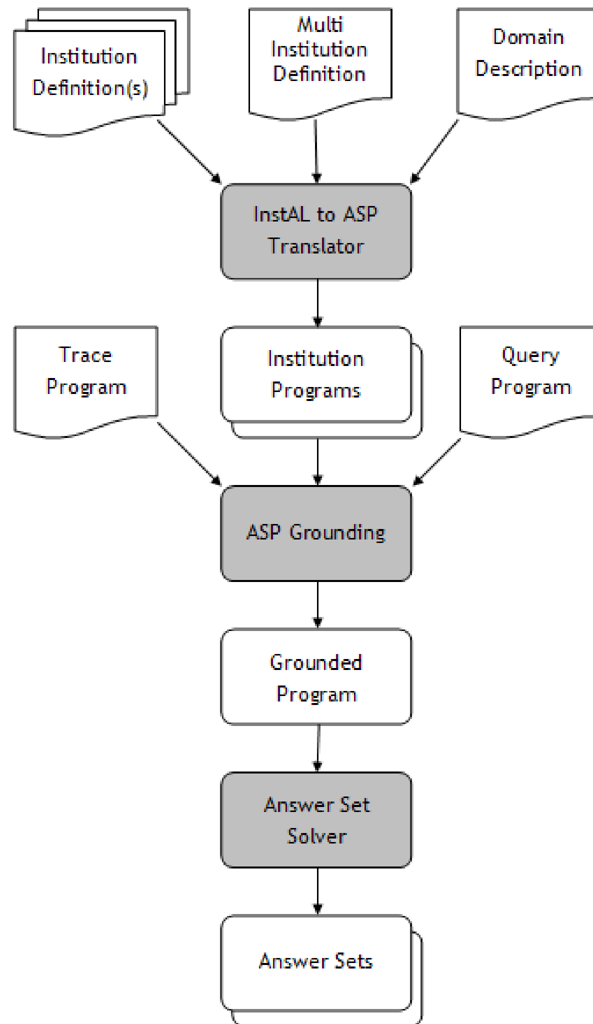

Figure 4.1: The current InstAL development process.

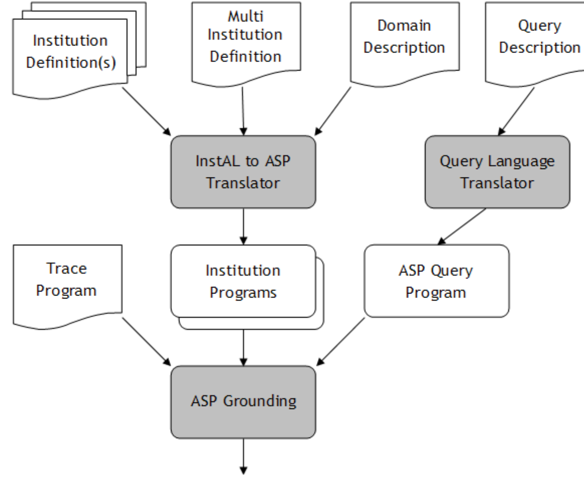

Figure 4.2: The proposed new InstAL development process. (Only initial stages shown.)

4.1 Example Queries

To guide the development of a query language for institutional models written in InstAL, five existing queries are considered. These are taken from an existing institutional model. The selected model is that of a single bidding round in a Dutch auction as described by Cliffe et al. (2007; 2008). This is the most complex institution to have been modeled in InstAL to date. These queries have been selected to cover conditions of differing complexity and involve reasoning over different parts of the model. It is believed these are representative of the typical kinds of queries that will be made of models.

4.1.1 Selected Queries

The first case is a simple constraint involving event occurrence. This query states that answer sets corresponding to traces in which the event `badgov` occurs at any point should be excluded. The key part of this condition is that an event occurs at **any** time.

$$\begin{aligned} \texttt{bad} &\leftarrow \texttt{occurred(badgov, I), instant(I).} \\ \perp &\leftarrow \texttt{bad.} \end{aligned} \tag{Q1}$$

Similarly, the second query involves the a fluent being true at **any** time during the execution. This time, only those answer sets corresponding to traces that satisfy the condition should be included.

$$\begin{aligned} \texttt{hadconflict} &\leftarrow \texttt{holdsat(conflict, I), instant(I).} \\ \perp &\leftarrow \texttt{not hadconflict.} \end{aligned} \tag{Q2}$$

In the third case, the query condition is for an event to occur **at the same time** as a fluent is true. Again, only answer sets in which the condition is satisfied should be included.

$$\begin{aligned} \texttt{restarted} &\leftarrow \texttt{occurred(desdl, I), holdsat(conflict, I),} \\ &\quad\ \texttt{instant(I).} \\ \bot &\leftarrow \texttt{not restarted.} \end{aligned} \tag{Q3}$$

The fourth case declares a parametrised condition. Whilst in the previous queries we considered conditions that are true/false of a whole model, this case declares a condition `startstate` that is true of a particular fluent. In addition, this query requires that fluent is true in the state **after** an event occurs.

$$\begin{aligned} \texttt{startstate(F)} &\leftarrow \texttt{holdsat(F, I1), occurred(createdar, I0),} \\ &\quad\ \texttt{next(I0, I1), ifluent(F).} \end{aligned} \tag{Q4}$$

The fifth case is a complex query involving many constraints. This query features the use of previously declared conditions in subsequent conditions. (Note that one of these, `startstate(F)`, is the condition specified in query (Q4).)

$$\begin{aligned} \texttt{startstate(F)} &\leftarrow \texttt{holdsat(F, I1), occurred(createdar, I0),} \\ &\quad\ \texttt{next(I0, I1), ifluent(F).} \\ \texttt{restartstate(F)} &\leftarrow \texttt{holdsat(F, I1), occurred(desdl, I0),} \\ &\quad\ \texttt{holdsat(conflict, I0),} \\ &\quad\ \texttt{next(I0, I1), ifluent(F).} \\ \texttt{missing(F)} &\leftarrow \texttt{startstate(F), not restartstate(F), ifluent(F).} \\ \texttt{added(F)} &\leftarrow \texttt{restartstate(F), not startstate(F), ifluent(F).} \\ \texttt{invalid} &\leftarrow \texttt{missing(F), ifluent(F).} \\ \texttt{invalid} &\leftarrow \texttt{added(F), ifluent(F).} \\ \bot &\leftarrow \texttt{not invalid.} \end{aligned} \tag{Q5}$$

4.1.2 Analysis of the Examples

The sample queries detailed above (p46) reveal a number of features which should be possible in an useful query language. All of the queries involve conditions relating to the occurrence of events and/or the value of fluents. These are the basics properties of a model in which we are interested in querying. These conditions need to be joined together using a logical **and** (indicated in ASP by comma-separated conditions). In addition, in (Q5) we see the need to join conditions using logical **or**. This is indicated in the ASP by two rules for `invalid`; collectively these read “the institution is invalid if any fluent is missing **or** any fluent is added”.

The query language must provide a way for defined conditions to restrict the answer sets of the model. That is, to make queries effective. In ASP this is done by specifying constraints (i.e. rules with an empty head). From queries (Q2), (Q3) and (Q5) we see that negation is required. In ASP, this is provided by negation as failure. Query (Q5) illustrates the use of

defined conditions as part of further complex conditions. While not strictly necessary, this is a convenient way to decompose complex conditions and enable reuse of sub-conditions.

Queries (Q1) and (Q2) illustrate conditions where we are interested in events occuring/fluents holding at any point in the model. Query (Q3) involves a simultaneous condition on events/fluents. Query (Q4) features the specification of an event occuring/a fluent holding after a condition on another event/fluent.

4.2 The Institution Query Language

In this section, we present the development of successive versions of an institution query language. This language is called InstQL (*institution query language*), to indicate its relation to InstAL. InstQL has a syntax influenced by that of InstAL in that it provides high-level declaration of concepts. Whereas in InstAL these are events, fluents and associated rules for institutions, in InstQL there are two concepts: *constraints* and *conditions*. A constraint is an assertion of a property which must be true of an InstAL model. A condition specifies a property of an InstAL model. Conditions can be declared in relation to other conditions and constraints can involve declared conditions. Like InstAL, the semantics of InstQL are defined in ASP. This makes the language directly executable through the use of an answer set solver such as DLV (Eiter et al., 1998) or Smodels (Niemelä and Simons, 1997).

InstQL was incrementally specified to simplify the development process. Each version of the language adds some new functionality to allow a specific type of reasoning. The initial version provides the base reasoning capabilities and features of the language. Each successive version then built this to provide new temporal relationships between conditions.

4.2.1 InstQL_α – Basic Queries

InstQL_α is the simplest variant of InstQL; this initial version provides the basic language structure. Reasoning over institution specifications concerns the occurrence of events and value of fluents within the institution(s). The motivation for the design of InstQL_α was to produce a language that allowed the specification of conditions involving events and fluents. From the example queries (p46) we see that the effect of a query should be to specify a constraint over the possible answer sets of the model. Therefore, the purpose of InstQL_α is to provide a way to express queries that will restrict the possible traces based on whether certain events do/do not occur and fluents are (not) true. This section first prevents the syntax of InstQL_α using Backus-Naur Form (BNF) and then the semantics of the translation of InstQL_α into ASP.

InstQL_α Syntax

As terminal symbols, InstQL_α provides a definition of various types of names which are built up as follows:

```
<variable> ::= [A-Z][a-zA-Z0-9_]*
<variable_list> ::= <variable> , <variable_list> | <variable>
<name> ::= [a-z][a-zA-Z0-9_]*
<param_list> ::= ( <variable_list> )
<identifier> ::= <name> <param_list> | <name>
```

The definition of a variable name given conforms with that of Lparse/Smodels (Niemelä and Simons, 1997), the underlying answer set grounder and solver used by InstAL. An *identifier* gives us an arbitrary name which may have variable parameters – note that InstAL allows the definition of parametrised events and fluents.

InstQL$_\alpha$ provides two *predicates* which form the basis of all InstQL queries. The first is `happens(Event)` – this means the specified event (defined in the corresponding InstAL specification) occurs at any point during the lifetime of the institution. The second is `holds(Fluent)`, which means that the specified fluent is true at any point during the lifetime of the institution. That is:

```
<predicate> ::= happens( <identifier> ) | holds( <identifier>)
```

Where the *identifier* corresponds to an event e (in the first case) or a fluent f (in the second case) that is defined in the InstAL specification. In terms of the model defined in Appendix A.1 (p82), we have that $e \in \mathcal{E}_i$ and $f \in \mathcal{F}_i$. Negation (as failure) is provided in InstQL$_\alpha$ by the unary operator `not`:

```
<literal> ::= not <predicate> | <predicate>
```

Named conditions may be defined and subsequently referenced by name. This allow complex criteria to be built up using sub-conditions. For example, suppose we have defined a condition called `my_cond` which specifies some desired property of the institution. We can then join this with other criteria e.g. "`my_cond and happens(e)`". Sub-conditions may be referenced within rules as *condition literals*:

```
<condition_literal> ::= not <identifier> | <identifier>
```

Note that this allows for parametrised conditions to be defined by the definition of an *identifier*. The building block of query conditions is the *term*:

```
<term> ::= <literal> | <condition_literal>
```

Terms may be grouped and connected by the connectives `and` and `or` which provide logical conjunction and disjunction.

```
<conjunction> ::= <term> and <conjunction> | <term>
<disjunction> ::= <conjunction> or <disjunction> | <conjunction>
```

This allows us to combine *predicates* and named conditions with arbitrary combinations of the logical operators `and`, `or`, `not`. For example:

```
happens(e1) or holds(f1) and not happens(e3)
     or not my_condition and holds(f2)
```

Note that this definition implicitly gives `or` higher precedence over `and`. That is, the statement "`X or Y and Z`" is read as "either `X` is true, or both `Y` and `Z` are true". No construction is provided to change this order to get the result "`(X or Y) and Z`", but we shall see later that this is can be overcome with the declaration of conditions. Conditions may be declared using the `condition` keyword.

```
<condition_decl> ::= condition <identifier> : <disjunction> ;
```

This construction defines a *condition* with the specified name to have a value equal to the specified *disjunction*. This allows the *condition name* to be used as a *condition literal*. *Constraints* specify properties of the institution which must be true.

```
<constraint> ::= constraint <disjunction> ;
```

For example, consider the following InstQL$_\alpha$ query:

```
constraint happens(e);
```

This constraint indicates that only answer sets of the model that correspond to traces in which `e` occurs at some point should be considered. That is, we are only interested in those traces in which `e` occurs.

To illustrate how this syntax is used to form queries, consider a simple light bulb institution. The fluent `on` is true when the bulb is on. The event `switch` turns the light on or off. We can require that at some point the light is on:

```
constraint holds(on);
```

We can require that the light is never on:

```
condition light_on: holds(on);
constraint not light_on;
```

There is some subtlety here in that `light_on` is true if at any instant `on` is true. Therefore, if `light_on` is not true, there cannot be an instant at which `on` was true. See p57 for further discussion of negation in InstQL. What about if the bulb is broken – the switch is pressed but the light never comes on? This can be expressed as:

```
constraint not light_on and happens(switch);
```

InstQL$_\alpha$ **Semantics**

The semantics of an InstQL$_\alpha$ query are defined by the translation function T which translates InstQL$_\alpha$ code into ASP. This function takes a symbol of InstQL$_\alpha$ (as defined above) and generates a set of (partial) ASP rules. Typically, this set is a singleton; only expressions involving disjunctions generate more than one rule. Set notation is dropped in the case where a singleton set is generated. The semantics of predicates are defined as follows:

$$T(\texttt{happens(e)}) = \texttt{occurred(e,I),instant(I),event(e)} \tag{4.1}$$

$$T(\texttt{holds(f)}) = \texttt{holdsat(f,I),instant(I),ifluent(f)} \tag{4.2}$$

For a literal of the form `not P` (where `P` is a predicate) the semantics are:

$$T(\texttt{not } P) = \texttt{not } T(\texttt{P}) \tag{4.3}$$

For a condition literal the semantics are as follows:

$$T(\texttt{conditionName}) = \texttt{conditionName} \tag{4.4}$$

$$T(\texttt{not conditionName}) = \texttt{not conditionName} \tag{4.5}$$

A conjunction of terms is handled as follows:

$$T(t_1 \texttt{ and } t_2 \texttt{ and} \cdots \texttt{ and } t_n) = T(t_1),\, T(t_2), \ldots, T(t_n) \tag{4.6}$$

A disjunction produces a set of rules. However, this is defined slightly differently for a condition declaration and a constraint.

$$\begin{aligned} T(\texttt{condition conditionName}:\; & C_1 \texttt{ or } C_2 \texttt{ or } \cdots \texttt{or } C_n;) = \\ & \{\texttt{conditionName} \leftarrow T(C_i). \mid 1 \le i \le n\} \end{aligned} \tag{4.7}$$

$$\begin{aligned} T(\texttt{constraint}\quad & C_1 \texttt{ or } C_2 \texttt{ or } \cdots \texttt{or } C_n;) = \\ & \{\texttt{newName} \leftarrow T(C_i). \mid 1 \le i \le n\} \cup \{\bot \leftarrow \texttt{not newName}.\} \end{aligned} \tag{4.8}$$

Note that the ASP term `newName` stands for any identifier which is unique within the ASP program taken by combining the translations of the InstAL description and InstQL$_\alpha$ query. In addition, each time instant `I` generated in the translation of a predicate represents a name for a time instant that is unique within the InstQL$_\alpha$ query. For example, the translation of "`happens(a) and happens(b)`" would be:

```
occurred(a, I), instant(I), event(a),
occurred(b, J), instant(J), event(b)
```

Such that $\texttt{I} \neq \texttt{J}$. Recall that a condition name may be parametrised. Since an InstQL variable matches a variable for Smodels, no extra work is required in the semantics to deal with this case. For example, the condition "`condition ever(E): happens(E);`" (which just defines an alias for `happens`) can be translated in accordance with the above

to get "`ever(E) ← occurred(E, I), instant(I), event(E).`". This is the required construction for a parametrised ASP rule.

Note that the scope of a variable within an InstQL query is bound within the conjunction within which it is used. This is because the translation of a disjunction will place different conjunctions within different ASP rules. For example:

```
constraint not happens(E) or happens(E) and holds(f);
```

Will be translated to:

```
c0 ← not occurred(E, I), event(E), instant(I).
c0 ← occurred(E, I), event(E), instant(I),
     holdsat(f, J), ifluent(f), instant(J).
⊥  ← not c0.
```

This means that the two instances of `E` in the query need not be grounded with the same event.

Above (p50), we mentioned that the precedence of `or` over `and` means that an expression which means "`(X or Y) and Z`" cannot be directly expressed. However, the following InstQL$_\alpha$ query has the required meaning:

```
condition disjunction: X or Y;
constraint disjunction and Z;
```

This becomes apparent when we apply the translation function to this query. The resulting ASP program is:

```
disjunction ← X.
disjunction ← Y.
newName     ← disjunction, Z.
⊥           ← not newName.
```

This tells us to compute only answer sets in which `newName` is true. This is the case only when both `disjunction` and `Z` are true. Then `disjunction` is true if either `X` or `Y` is true. This is the result we want.

4.2.2 InstQL$_\beta$ – Concurrent Events

While InstQL$_\alpha$ provides the basics for a query language over InstAL models, it has several limitations. We may commonly wish to specify queries of the form "X and Y happen at the same time". That is, we may wish to talk about events occurring at the same time as one or more fluents are true, simultaneous occurrence of events or combinations of

fluents being simultaneously true (and/or false). To allow this kind of reasoning, InstQL$_\beta$ introduces the keyword `while` to indicate that literals are true simultaneously. This allows *while expressions* which link multiple literals that must be true simultaneously.

Note that while expressions are only defined over literals constructed from predicates (i.e. `happens` and `holds`) and not condition literals involving condition names. This is because, for example, a condition may mean "event `e` never occurs". It does not make sense to define an expression that says something like "event `d` occurs at the same time as (`e` never occurs)".

The syntax of InstQL$_\beta$ preserves that of InstQL$_\alpha$ except that a new statement, the while expression, is defined and the definition for a term is modified. A while expression is defined as follows:

```
<while_expr> ::= <literal> while <while_expr> | <literal>
```

This means that whereas `happens(e)` still means "event `e` occurs at some point during the institution lifetime" when occurring on its own, the expression `happens(e) while happens(d)` means "at some point during the institution lifetime, events `d` and `e` happen simultaneously". The definition of a term is modified as follows:

```
<term> ::= <while_expr> | <condition_literal>
```

Since the rest of the syntax of InstQL$_\beta$ is identical to that of InstQL$_\alpha$ (p48) we see that `while` has lower precedence than `and` (and therefore, `or` also). That is, the expression "L_1 `or` L_2 `while` L_3 `and` L_4" corresponds to "either L_1 is true, or L_2 and L_3 are true at the same time and L_4 is true (at any time)".

Returning to the light bulb institution, InstQL$_\beta$ allows us to specify that we want only traces where the light was turned off at some point:

```
constraint happens(switch) while holds(on);
```

Or that at some point the light was turned left on:

```
constraint holds(on) while not happens(switch);
```

Again, the semantics of InstQL$_\beta$ are given in terms of the translation function T. The semantics of a predicate are redefined as follows:

$$T(\texttt{happens(e)}) = \texttt{occurred(e,I)} \tag{4.9}$$

$$T(\texttt{holds(f)}) = \texttt{holdsat(f,I)} \tag{4.10}$$

The semantics for a while expression are:

$$T(L_1 \texttt{ while } L_2 \texttt{ while } \cdots \texttt{ while } L_n) = T(L_1), T(L_2), \ldots, T(L_n), \texttt{instant(I)} \tag{4.11}$$

This is such that for each literal, the same instant label is used. The label (which in InstQL_α was unique to each predicate) is now unique to each while expression. That is we require that, for example:

$$\begin{aligned} T(\texttt{happens(e)} \ & \texttt{while holds(f) and happens(e}') \texttt{ while holds(f}')) = \\ & \texttt{occurred(e, I), holdsat(f, I), instant(I),} \\ & \texttt{occurred(e}', \texttt{J), holdsat(f}', \texttt{J), instant(J)} \end{aligned}$$

Where $\texttt{I} \neq \texttt{J}$. Note that in the special case of a while expression with only one literal we get:

$$T(L_1^w) = T(L_1), \ \texttt{instant(I)}$$

(Where L_1^w denotes L_1 interpreted as a while expression rather than as a literal.) That is, we get a time instant unique to that literal as in InstQL_α – this allows us to still talk about literals that are true at any time during the institution lifetime. This is because InstQL_β is defined so that it is a super-language of InstQL_α – any InstQL_α query is an InstQL_β query also.

4.2.3 InstQL_γ – Simple Ordering

InstQL_β allows us to express query conditions requiring certain literals to be true simultaneously. InstQL_γ adds the ability to express orderings over events. This is done with the `after` keyword. This allows statements of the form:

```
holds(f1) while not holds(f2) after happens(e1) after happens(e2)
```

The intended meaning of the above statement is as follows:

- at some time instant k the event `e2` occurs
- at some other time instant j the event `e1` occurs
- at some other time instant i the fluent `f1` is true but the fluent `f2` is not true
- these time instants are ordered such that $i > j > k$ (i.e. k is the earliest time instant)

More formally, the syntax of an *after expression* is defined as:

```
<after_expr> ::= <while_expr> after <after_expr> | <while_expr>
```

Then, a term is redefined as:

```
<term> ::= <after_expr> | <condition_literal>
```

As with a while expression, this means that an after expression cannot involve a condition literal. This is done for the same reason as with a while expression; there is no guarantee an arbitrary condition relates to some particular moment in time and so it does not make sense to discuss properties of a model after the condition.

Once again returning to the light bulb institution, InstQL$_\gamma$ allows us to specify a query which requires the light to be switched twice (or more):

```
constraint happens(switch) after happens(switch);
```

Or that once that light has is on, it cannot be switched off again:

```
condition switch_off: happens(switch) after holds(on);
constraint not switch_off;
```

The semantics of an after expression are defined as follows:

$$T(W_1 \quad \texttt{after} \cdots \texttt{after}\ W_n) = T(W_1), \ldots, T(W_n),\ \texttt{after}(\texttt{i}_1, \texttt{i}_2),\ \texttt{after}(\texttt{i}_2, \texttt{i}_3), \ldots, \texttt{after}(\texttt{i}_{\texttt{n}-1}, \texttt{i}_\texttt{n}) \tag{4.12}$$

Where $\texttt{i}_\texttt{k}$ is the time instant generated by translating the while expression W_k and `after` is a predicate defined in ASP as follows:

$$\texttt{after}(\texttt{I}, \texttt{J}) \leftarrow \texttt{next}(\texttt{J}, \texttt{I}). \tag{4.13}$$

$$\texttt{after}(\texttt{I}, \texttt{J}) \leftarrow \texttt{after}(\texttt{I}, \texttt{K}),\ \texttt{after}(\texttt{K}, \texttt{J}),\ \texttt{instant}(\texttt{I}),\ \texttt{instant}(\texttt{J}), \texttt{instant}(\texttt{K}). \tag{4.14}$$

The rule (4.13) gives us that if i is the next time instant after j, then i is after j. Rule (4.14) gives us that the `after` relation is closed under transitivity. That is, if there exists a time instant k that is after j, such that i is after k, then i is after j also.

The only addition to InstQL$_\beta$ made by InstQL$_\gamma$ is the after expression. This is defined so that its simplest case is just a single while expression and we have that any InstQL$_\beta$ query is also a query of InstQL$_\gamma$ – again this is a super-language.

4.2.4 InstQL$_\delta$ – Precise Ordering

InstQL$_\gamma$ allows us reason about orderings of events. For example, we can say "`happens(e) after happens(d)`" which means that if some event `d` occurs between the time instants t_i and t_{i+1} then the event `e` occurs between t_j and t_{j+1} such that $j \geq i + 1$. However, if we consider queries (Q4) and (Q4) we see the need to specify properties of the institution **in the very next state** after some other property held. That is, we need to say that not only does something literal hold after some other literal, but that this is precisely one time instant later.

Expression		Definition
`<variable>`	::=	`[A-Z][a-zA-Z0-9_]*`
`<variable_list>`	::=	`<variable> , <variable_list> | <variable>`
`<name>`	::=	`[a-z][a-zA-Z0-9_]*`
`<param_list>`	::=	`( <variable_list> )`
`<identifier>`	::=	`<name> <param_list> | <name>`
`<predicate>`	::=	`happens( <identifier> ) | holds( <identifier>)`
`<literal>`	::=	`not <predicate> | <predicate>`
`<while_expr>`	::=	`<literal> while <while_expr> | <literal>`
`<after>`	::=	`after( <integer> ) | after`
`<after_expr>`	::=	`<while_expr> <after> <after_expr>`
		`| <while_expr>`
`<condition_literal>`	::=	`not <identifier> | <identifier>`
`<term>`	::=	`<after_expr> | <condition_literal>`
`<conjunction>`	::=	`<term> and <conjunction> | <term>`
`<disjunction>`	::=	`<conjunction> or <disjunction> | <conjunction>`
`<condition_decl>`	::=	`condition <identifier> : <disjunction> ;`
`<constraint>`	::=	`constraint <disjunction> ;`

Table 4.1: InstQL Syntax

InstQL$_\delta$ extends InstQL$_\gamma$ to allow us to specify such precise orderings between literals. Rather than just providing the facility to specify a literal occurs/holds in the next time instant, this is generalised to say that a literal happens n time instants after another. That is, for a fluent that does (not) hold at time instant t_i or an event that occurs between t_i and t_{i+1}, we can talk about literals that hold at t_{i+n} or occur between t_{i+n} and t_{i+n+1}.

This is achieved by modifying the definition of an after expression as follows:

```
<after> ::= after | after( <integer> )
<after_expr> ::= <while_expr> <after> <after_expr> | <while_expr>
```

An *after expression* may contain only the `after` operator from InstQL$_\gamma$ and not use the new `after(n)` operator. This means that any InstQL$_\gamma$ query is a query of InstQL$_\delta$ also. We have the following inclusions: InstQL$_\alpha$ $\subset$ InstQL$_\beta$ $\subset$ InstQL$_\gamma$ $\subset$ InstQL$_\delta$ – each successive version of InstQL is a strict super-language of the previous.

This modified version of the after expression completes the definition of the syntax of InstQL. Table 4.1 summarises the syntax of the language.

The semantics are given for the binary operator `after(n)` – these can easily be generalised for after expressions built of sequences of `after(n)` operators mixed with `after` operators.

$$T(W_i \texttt{ after(n) } W_j) = \quad T(W_i),\ T(W_j),\ \texttt{after}(\texttt{t}_\texttt{i}, \texttt{t}_\texttt{j}, \texttt{n}) \tag{4.15}$$

Where $\texttt{t}_\texttt{i}$ and $\texttt{t}_\texttt{j}$ are the time instants generated by W_i and W_j respectively. This is defined

such that we require $\mathtt{n} > 0$. The three-argument form of `after` is defined in ASP as:

$$\mathtt{number}(1 \,..\, 1000). \tag{4.16}$$
$$\mathtt{after}(\mathtt{I},\ \mathtt{J},\ 1) \leftarrow \mathtt{next}(\mathtt{J},\ \mathtt{I}). \tag{4.17}$$
$$\mathtt{after}(\mathtt{I},\ \mathtt{J},\ \mathtt{N}) \leftarrow \mathtt{next}(\mathtt{K},\ \mathtt{I}),\ \mathtt{after}(\mathtt{K},\ \mathtt{J},\ \mathtt{N}-1),\ \mathtt{number}(\mathtt{N}). \tag{4.18}$$

The predicate `number` must be defined to restrict the variable `N`. This is done for the numbers $1 \leq \mathtt{n} \leq 1000$ since this seems a sufficiently large range to not be limiting[1]. We require that in operator `after(n)`, $\mathtt{n} > 0$ and this now imposes a de facto upper limit as well.

We now provide a concrete example of the translation of an *after expression* to illustrate this process:

$$\begin{aligned} T\ & (\mathtt{happens(e)\ while\ holds(f)\ after\ happens(d)\ after(3)\ holds(g)}) = \\ & \mathtt{occurred(e,\ t_i),\ event(e),\ holdsat(f,\ t_i),\ ifluent(f),} \\ & \mathtt{instant(t_i),\ occurred(d,\ t_j),\ event(d),\ instant(t_j),} \\ & \mathtt{holdsat(g,\ t_k),\ ifluent(g),\ instant(t_k),} \\ & \mathtt{after(t_i,\ t_j),\ after(t_j,\ t_k, 3).} \end{aligned}$$

4.2.5 A Note on Negation

It may be tempting to interpret the InstQL$_\alpha$ query "`constraint not happens(e);`" to mean "consider only traces where event `e` never happens". However, the definition of the semantics of a literal (predicate rather than condition) given in (4.3) mean that this is not the case. The translation into ASP of this query is as follows:

$$\begin{aligned} \mathtt{n1} & \leftarrow \mathtt{not\ occurred(e,\ I),\ event(e),\ instant(I).} \\ \perp & \leftarrow \mathtt{not\ n1.} \end{aligned}$$

This says we should include only answer sets for which `n1` is true. This is satisfied for any trace where for some time instant `I`, the predicate `occurred(e, I)` is not true. This means the query means "consider only traces where **there exists** an instant `I` such that `e` does not occur at `I`." The naïve interpretation outlined above corresponds to "consider only traces where **for all** instants `I`, `e` does not occur at `I`."

The 'for all' interpretation seems more intuitive in this simple case. The semantics are defined for the 'there exists' interpretation for two reasons. Firstly, this parallels the intuitive interpretation for positive predicates. Consider the query "`constraint happens(e);`". The intuitive (and correct) interpretation is that at some time instant, `e` happens – not that `e` happens at all time instants. Secondly, and more importantly, the existentially quantified interpretation is correct in the more general case. Consider instead the following query:

```
constraint happens(d) while not happens(e);
```

[1]Though this can be increased at the cost of grounding efficiency if required.

The interpretation for this (both the intuitive and the correct one) is that there exists a time instant such that `d` happens at this instant and that `e` does not happen at this time instant. For expressions involving `after` and `while` the interpretation of "`not happens(e)`" needs to be that there exists an instant at which `e` does not occur. In the special case where these operators are not used, this is not the most intuitive interpretation of the negation. Treating this case specially would make the semantics of `not` (applied to a predicate) context-sensitive which would complicate implementation and understanding of InstQL.

The intended (i.e. universally quantified) interpretation that `e` never happens can be achieved as follows:

```
condition ever(E) : happens(E);
constraint not ever(e);
```

This translates into ASP as:

$$\begin{aligned} \texttt{ever(E)} &\leftarrow \texttt{occurred(E, I), event(E), instant(I).} \\ \texttt{n1} &\leftarrow \texttt{not ever(e).} \\ \bot &\leftarrow \texttt{not n1.} \end{aligned}$$

This illustrates the fundamental difference between a negated condition and a negated predicate. The scope of negation in a predicate literal is limited to `occurred` or `holdsat` in the underlying ASP. In a condition literal, however, negation extends over the entire rule. Therefore, the program above says that `ever(E)` is true if event `E` occurs at some point during the institution lifetime. The condition with temporary name `n1`, (generated by translation) is true if it is not true that `e` ever occurs. That is, `e` never occurs; the entire rule is negated. The final rule constrains the answer sets to consider only those in which `n1` is true.

Furthermore, a similar misinterpretation may occur with queries such as:

```
constraint not happens(d) while happens(e);
```

A naïve interpretation of this query could be "at no time instant does `d` occur if `e` occurs". Such an interpretation exaggerates the scope of the negation. The negation applies only to "`happens(d)`" and so therefore the correct interpretation is "there exists a time instant where `d` does not occur but `e` does". The query obtained by swapping "`not happens(d)`" and "`happens(e)`" is equivalent (since the two conditions are in conjunction, which is commutative) but clarifies this issue. Phrased the other way round, it is not possible to attribute a scope for the negation beyond "`happens(d)`". (This phrasing is shown above in this section.)

A query to state that `d` and `e` must not happen simultaneously can be constructed as follows:

```
condition simultaneous : happens(d) while happens(e);
constraint not simultaneous;
```

4.3 Example Queries Revisited

Having defined (various versions of) the query language InstQL, we know return to the example queries for InstAL institutions specified on page 46. This section gives equivalent InstQL queries to the ASP ones previously used. We then show the translation of these queries into ASP to illustrate the correctness of InstQL semantics.

For (Q1) the following InstQL$_\alpha$ query is equivalent:

$$\begin{array}{l}\texttt{condition bad : happens(badgov);}\\ \texttt{constraint not bad;}\end{array} \tag{IQ1}$$

The translation of this into ASP (in accordance with the translation function T) is as follows:

$$\begin{aligned} T(\text{IQ1}) &= \texttt{bad} \leftarrow T(\texttt{happens(badgov)}).\\ &\quad\ \texttt{n1} \leftarrow T(\texttt{not bad}).\\ &\quad\ \bot \leftarrow \texttt{not n1}.\\ &= \texttt{bad} \leftarrow \texttt{occurred(badgov, I), event(badgov), instant(I)}.\\ &\quad\ \texttt{n1} \leftarrow \texttt{not bad}.\\ &\quad\ \bot \leftarrow \texttt{not n1}.\end{aligned}$$

This is equivalent to (Q1). The only difference is that the translation has produced an extra rule for `n1`. This says that `n1` is true if `bad` is not true. The constraint permits only answer sets in which `n1` is true (and therefore bad is not true). In (Q1), the constraint permits only answer sets in which `bad` is not true.

An InstQL$_\alpha$ query that is equivalent to (Q2) is:

$$\texttt{constraint holds(conflict);} \tag{IQ2}$$

The translation of this into ASP is as follows:

$$\begin{aligned} T(\text{IQ2}) &= \texttt{n1} \leftarrow T(\texttt{holds(conflict)}).\\ &\quad\ \bot \leftarrow \texttt{not n1}.\\ &= \texttt{n1} \leftarrow \texttt{holdsat(conflict, I), ifluent(conflict), instant(I)}.\\ &\quad\ \bot \leftarrow \texttt{not n1}.\end{aligned}$$

This is exactly equivalent to (Q2) with the literal `hadconflict` renamed to `n1`.

The following InstQL$_\beta$ query is equivalent to (Q3):

$$\texttt{constraint happens(desdl) while holds(conflict);} \tag{IQ3}$$

The translation of (IQ3) into ASP is as follows:

$$T(\text{IQ3}) = \texttt{n1} \leftarrow T(\texttt{happens(desdl) while holds(conflict)}).$$

$$
\begin{aligned}
&\bot \leftarrow \texttt{not n1}.\\
= \;& \texttt{n1} \leftarrow T(\texttt{happens(desdl)}),\ T(\texttt{holds(conflict)}),\ \texttt{instant(I)}.\\
&\bot \leftarrow \texttt{not n1}.\\
= \;& \texttt{n1} \leftarrow \texttt{occurred(desdl, I), event(desdl),}\\
&\texttt{holdsat(conflict, I), instant(I).}\\
&\bot \leftarrow \texttt{not n1}.
\end{aligned}
$$

This is exactly equivalent to (Q3) with the literal `restarted` renamed to `n1`.

For (Q4), the following InstQL$_\delta$ query is equivalent:

$$\texttt{condition startstate(F) : holds(F) after(1) happens(createdar);} \tag{IQ4}$$

The translation of this into ASP is as follows:

$$
\begin{aligned}
T(\text{IQ4}) \;=\;& \texttt{startstate(F)} \leftarrow T(\texttt{holds(F) after(1) happens(createdar)}).\\
=\;& \texttt{startstate(F)} \leftarrow T(\texttt{holds(F)}),\ T(\texttt{happens(createdar)}),\ \texttt{next(J, I)}.\\
=\;& \texttt{startstate(F)} \leftarrow \texttt{holdsat(F, I), ifluent(F), instant(I),}\\
& \texttt{occurred(createdar, J), event(createdar), instant(J), after(I, J, 1).}
\end{aligned}
$$

For (Q5) the following InstQL$_\delta$ query is equivalent:

```
condition   startstate(F) : holds(F) after(1) happens(createdar);
condition   restartstate(F) : holds(F) after(1) happens(desdl)
            while holds(conflict);
condition   missing(F) : startstate(F) and not restartstate(F);        (IQ5)
condition   added(F) : restartstate(F) and not startstate(F);
constraint  missing(F) or added(F);
```

The steps of translating (IQ5) into ASP are omitted for brevity but the translated query is as follows:

$$
\begin{aligned}
\texttt{startstate(F)} &\leftarrow \texttt{holdsat(F, I), ifluent(F), instant(I),}\\
&\quad \texttt{occurred(createdar, J), event(createdar), instant(J),}\\
&\quad \texttt{after(I, J, 1).}\\
\texttt{restartstate(F)} &\leftarrow \texttt{holdsat(F, I), ifluent(F), instant(I),}\\
&\quad \texttt{occurred(desdl, J), event(desdl), holdsat(conflict, J),}\\
&\quad \texttt{ifluent(conflict), instant(J), after(I, J, 1).}\\
\texttt{missing(F)} &\leftarrow \texttt{startstate(F), not restartstate(F).}\\
\texttt{added(F)} &\leftarrow \texttt{restartstate(F), not startstate(F).}\\
\texttt{n1} &\leftarrow \texttt{missing(F).}\\
\texttt{n1} &\leftarrow \texttt{added(F).}\\
\bot &\leftarrow \texttt{not n1.}
\end{aligned}
$$

This is equivalent to (Q5). In the rules for `startstate(F)` and `restartstate(F)`, the translation of (IQ5) specifies `instant(I)` for each time instant used in the rule. This is not necessary since the query also specifies `next(J, I)` which will be defined only for time instants. The extra predicates arise from the translation of a while expression where they are necessary if the expression is not part of an after expression. Apart from this, the only way the translation of the InstQL$_\delta$ query differs from the original ASP query is that `invalid` is renamed to `n1` (i.e. a name generated during translation).

4.4 Summary

This chapter presented *InstQL* – the institution query language. This new language is designed for the construction of queries to allow reasoning about institutions specified in InstAL. InstQL maintains the level of abstraction above the underlying logic (ASP) which InstAL provides for institution specification. Prior to the advent of InstQL, this abstraction was lost and the institution designer was forced to specify queries in ASP directly. This requires knowledge of how an InstAL specification is translated into ASP and ties the development process to the use of ASP. Having abstraction above ASP at all stages of development allows the underlying logic to be changed at some future stage if desired.

Section 4.1 introduced motivating examples for InstQL. It presented five queries specified in ASP for the Dutch auction round institution (Cliffe et al., 2008). These queries represent the typical reasoning done about institution specifications and so informed the language design. Then in Section 4.2 we presented a definition of four successive versions of InstQL. Each version is a strict super-language of the previous that adds new reasoning capabilities. This section gave the syntax of InstQL as summarised in Table 4.1 and its semantics, defined by translation into ASP. Finally, Section 4.3 returned to the queries of Section 4.1 and gave a specification of these same queries in InstQL. In addition, the translation of these queries back into ASP (in accordance with the definition given in Section 4.2) was given to illustrate that the InstQL queries are equivalent to their ASP counterparts.

The next chapter discusses practical aspects of reasoning with InstQL. It explains the use of InstQL for various types of reasoning and details the development of a system to automate translation of InstQL into ASP.

Chapter 5

Using the Institution Query Language

Having defined InstQL, the institution query language, in Chapter 4, we know discuss the practical aspects of reasoning using InstQL. This chapter considers how general reasoning may be carried out using InstQL. In addition, InstQL is demonstrated to be capable of expressing a limited version of linear temporal logic (Pnueli, 1977). We then detail the development of a system which automates the translation of InstQL into ASP that was outlined in the previous chapter.

5.1 Reasoning with InstQL

Section 4.3 (p59) gave some examples of the use of InstQL to specify queries for InstAL specifications. This section furthers this by detailing the use of InstQL to accomplish general reasoning tasks. We describe how to use InstQL to perform three "standard" (Sergot, 2004) types of computational reasoning: prediction, postdiction and planning. We then discuss possible uses of InstQL by agents.

5.1.1 General Reasoning with InstQL

Prediction is the problem of ascertaining the resulting state for a given (partial) sequence of actions and initial state. That is, suppose some transition system is in state S and a sequence $A = a_1, \ldots, a_n$ of actions occurs. Then the prediction problem (S, A) is to decide the set of states $\{S'\}$ which may result. Postdiction is the opposite problem – if a system is in state S' and we know that $A = a_1, \ldots, a_n$ have occurred, then the problem (A, S') is to decide the set $\{S\}$ of states that could have held before A. The planning problem (S, S') is to decide which sequence(s) of actions, $\{A\}$, will bring about state S' from state S.

Recall from definition A.3 that a state of an institution is described by those fluents that are

true in the state and those that are false in the state. That is, a state is $S = \{f_1, \dots, f_n\} \cup \neg\{g_1, \dots, g_k\}$ where f_i are the fluents true under S and g_i those false under S. This may then be encoded in InstQL as the while expression:

$$\begin{aligned} S \;=\; & \texttt{holds}(f_1)\ \texttt{while} \cdots \texttt{while}\ \texttt{holds}(f_n)\ \texttt{while} \\ & \texttt{not holds}(g_1)\ \texttt{while} \cdots \texttt{while not holds}(g_k) \end{aligned} \tag{5.1}$$

When reasoning about institutions, action sequences correspond to sequences of observable events. InstQL allows reasoning about sequences of any events in $\mathcal{E}_{\mathcal{M}}$, including those in $\mathcal{E}^i_{obs}$. A sequence of events $E = e_1, \dots, e_n$ may be encoded as an after expression. If we have complete information, then we know that e_1 occurred, then e_2 at the next time instant and so on up to e_n with no other events occurring in between. In this case, we can express E as follows:

$$E = \texttt{happens}(e_n)\ \texttt{after(1)} \cdots \texttt{after(1)}\ \texttt{happens}(e_1) \tag{5.2}$$

This can be generalised to the case where e_{i+1} occurs after e_i with some known number $k \geq 0$ of events happening in between:

$$\texttt{happens}(e_{i+1})\ \texttt{after}(k+1)\ \texttt{happens}(e_i)$$

Alternatively if we do not know k (i.e. we know that e_{i+1} happens later than e_i but zero or more events occur in between) we can express this as:

$$\texttt{happens}(e_{i+1})\ \texttt{after happens}(e_i)$$

We can combine these cases throughout the formulation of E to represent the amount of information available.

Given an initial state S and a sequence of events E, the prediction problem (S, E) can be expressed in InstQL as:

```
constraint E after(1) S;
```

This query limits traces to those in which at some point S holds and following S the events of E occur in sequence. Any traces satisfying this query will then contain the states $\{S'\}$. This permits no observable events to occur between S holding and the first event of E. As with relations between the events in E, we can use instead `after` or `after(n)` as dictated by the amount on information available.

Given a sequence of events E and a resulting state S', the postdiction problem (E, S') can be expressed as:

```
constraint S after(1) E;
```

This requires S to hold in the next instant following the final event of E. Again, a different form of after can be used if appropriate.

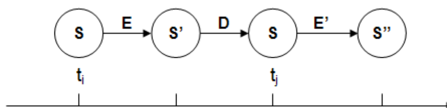

Figure 5.1: State transition for prediction.

Given a pair of states S and S' the planning problem (S, S') can be expressed in InstQL as:

```
constraint S' after S;
```

This allows any non-empty sequence of observable events to bring about the transition from S to S'. If we want to consider plans of length k (i.e. $E = e_1, \ldots, e_k$) then we express this:

```
constraint S' after(k) S;
```

These solutions require at least knowledge of the ordering of events that have occurred. For the sequence $e_1, \ldots, e_n$ we require that $e_1 < \cdots < e_n$ (i.e. e_1 occurs before e_2 etc). It may be the case that we know two successive events in the sequence are a specific number of instants apart, but if not we must know that they are one or more time instants apart. This may not always be the case. For example, in a prediction problem we may know the initial state and a set of events that have occurred but not the order in which those events occurred. The general case of this is that we have a set of event sequences (which may be of length one if we have only a set of events). That is, suppose we know that event sequence E occurred (with as before, the events of E happening in order) and event sequence E' also occurred, but we do not know how the events of E relate to those of E'. Naïvely, it seems we can the solve the modified planning problem $(S, \{E, E'\})$ with the query:

```
constraint E after S and E' after S;
```

This query allows only traces such that at some point S held and following this, the events of E occurred in order and following S the events of E' also occurred in order. The problem is is that there is not guarantee that S must hold at the same time in each case. The first term requires S holds at t_i and following t_i, E occurs. Then the second term requires that S holds at t_j and following t_j, E' occurs. We cannot be sure $t_i = t_j$. Figure 5.1 illustrates such a situation. A trace of this kind would be permitted by the above query. In general, when we know that the events of E and E' and possibly others have happened, this is an acceptable result. In the case where we know that the only events to have happened are those of E and E', we can prevent such a situation through the addition of the following constraint:

```
condition between: E after happens(Event) after E'
                   or E' after happens(Event) after E;
constraint not between;
```

The condition `between` is true if all the events of E occur, then (some time later) some other event and then (some time later) those of E'. Similarly, for E' occurring first. The constraint excludes traces for which `between` is true – i.e. only those traces in which no events occur in between E and E'. Should E and E' overlap, `between` will not be true and so such traces will be permitted.

5.1.2 Agent Reasoning

There are two distinct types of reasoning about institutions. The first is the verification and exploration of institution properties by institution designers. After specifying an institution, queries can be used to ensure desired properties of the model and elicit emergent properties that were perhaps not intended. This is the use of InstQL which has been discussed up to this point. This is the primary use of InstQL.

The second case for reasoning about an institution is for an agent within that institution as part of deciding on its actions. If we imagine a system in which an agent has perfect information about the institution it belongs to, this can be provided by the InstAL specification of the institution. If the agent also has a system to sense the current institution state, it could use InstQL queries over the institution specification as part of its reasoning process.

Two types of goals are common for agents: *achievement* tasks and *maintenance* tasks (Wooldridge, 2002). In an achievement task, the agent attempts to bring about some goal state G. If the agent is able to determine the current state C (or at least some approximation of the current state when the environment is not accessible) then the possible ways in which G can be brought about are the traces which are solutions to the planning problem (C, G). This can be determined using InstQL as described above. The agent can then use this information to aid selection of its next action.

For a maintenance task, there is some state B which the agent seeks to avoid. Again, solving the planning problem (C, B) will tell the agent ways in which B might be brought about. It can then attempt to avoid any action which is involved in those traces resulting in B. Agents can also use prediction as outlined above to assess the consequences of possible actions.

5.2 Modelling Linear Temporal Logic in InstQL

Linear Temporal Logic (LTL) (Pnueli, 1977) provides us with a formalism for reasoning about paths of state transition systems. In LTL, we have a set AP of *atomic propositions*. The syntax of LTL ($L(F, U)$ in Emerson and Halpern, 1986) is defined as follows:

- $p \in AP$ is a formula of LTL
- $\neg f$ is a formula if f is a formula

- $f \vee g$ is a formula if f and g are formulae
- $f \wedge g$ is a formula if f and g are formulae
- $\Diamond f$ is a formula if f is a formula ("eventually f")
- fUg is a formula if f and g are formulae ("f until g")

We abbreviate $\neg\Diamond\neg f$ by $\Box f$ ("always f").

Definition 5.1 *A* structure *is a triple* $M = (\mathbf{S}, \mathbf{X}, \mathbf{L})$ *where* $\mathbf{S}$ *is a non-empty set of* states, $\mathbf{X}$ *a non-empty set of* paths *and* $\mathbf{L} : \mathbf{S} \rightarrow \mathbb{P}(AP)$ *a* labelling function *which assigns to each state a set of propositions true in that state. A path is a non-empty sequence of states* $x = s_0 s_1 s_2 \ldots$. *We denote by* x^k *the suffix of path* x *starting with the* k^{th} *state. That is* $x^k = s_k s_{k+1} s_{k+2} \ldots$ *In addition, we use* $first(x)$ *to denote the first state in path* x*.*

The semantics of LTL are defined inductively by the relation $\models$ (based on those of Emerson and Halpern, 1986; Emerson, 1990; Sistla and Clarke, 1985; Heljanko and Niemel, 2003). Let $M = (\mathbf{S}, \mathbf{X}, \mathbf{L})$ be a structure and $x \in \mathbf{X}$, then:

$$\begin{array}{lcl}
M, x \models p \in AP & \iff & p \in \mathbf{L}(first(x)) \\
M, x \models \neg f & \iff & M, x \not\models f \\
M, x \models f \vee g & \iff & M, x \models f \text{ or } M, x \models g \\
M, x \models f \wedge g & \iff & M, x \models f \text{ and } M, x \models g \\
M, x \models \Diamond f & \iff & \exists i \cdot M, x^i \models f \\
M, x \models fUg & \iff & \exists i \cdot M, x^i \models g \wedge (\forall j < i \cdot M, x^j \models f)
\end{array}$$

5.2.1 Institutional LTL

To apply LTL to institutions, we take for our structure $\mathcal{M}$, a multi-institution system. For the atomic propositions we take $AP = \mathcal{E}_{\mathcal{M}} \cup \mathcal{F}_{\mathcal{M}}$. We let $\mathbf{S} = \Sigma_{\mathcal{M}}$, the set of all possible institutional states. We take for $\mathbf{X}$ the set of all possible traces of the system. Traces implicitly define a set of states since states are implied in the model by those fluents which are true at a given time.

We define a function $events : \mathbf{S} \rightarrow \mathbb{P}(\mathcal{E}_{\mathcal{M}})$ such that $events(s)$ denotes the set of events which occur between t_i and t_{i+1} (where s is the state of $\mathcal{M}$ at t_i). Since s is itself the set of fluents true at t_i we have that $\mathbf{L}(s) = s \cup events(s)$.

Let x^k denote the suffix of x starting at the kth time instant of trace x. We let $first(x)$ denote the state implied at the first time instant of trace x.

We can then define *InstLTL* (institutional LTL) as the following restricted form of LTL:

- $p \in AP$ is a *literal*
- $\neg p$ is a *literal* if $p \in AP$

- $\Diamond l$ is a formula if l is a literal ("eventually l")
- fUe is a formula if $f \in \mathcal{F}_{\mathcal{M}}$ and $e \in \mathcal{E}_{\mathcal{M}}$ ("f until e")
- $\neg f$ is a formula if f is a formula
- $f \vee g$ is a formula if f and g are formulae
- $f \wedge g$ is a formula if f and g are formulae

The restriction that $\Diamond$ and U can only be applied to literals rather than general formulae arises from the fact the InstLTL is designed so as to be expressible in InstQL. Again, we let $\Box p = \neg\Diamond\neg p$ ("always p").

The semantics of InstLTL are given as follows for some multi-institution system $\mathcal{M}$:

$$\begin{array}{lll}
\mathcal{M}, x \models p \in AP & \iff & p \in \mathbf{L}(first(x)) \\
\mathcal{M}, x \models \neg p & \iff & p \notin \mathbf{L}(first(x)) \\
\mathcal{M}, x \models \Diamond l & \iff & \exists i \cdot \mathcal{M}, x^i \models l \\
\mathcal{M}, x \models fUe & \iff & \exists i \cdot \mathcal{M}, x^i \models e \wedge (\forall j < i \cdot \mathcal{M}, x^j \models f) \\
\mathcal{M}, x \models f \vee g & \iff & \mathcal{M}, x \models f \text{ or } \mathcal{M}, x \models g \\
\mathcal{M}, x \models f \wedge g & \iff & \mathcal{M}, x \models f \text{ and } \mathcal{M}, x \models g \\
\mathcal{M}, x \models \neg f & \iff & \mathcal{M}, x \not\models f
\end{array}$$

5.2.2 Expressing InstLTL in InstQL

nstLTL (the restricted form of LTL defined above) may be expressed in InstQL. This section describes how various formulae of LTL may be expressed as conditions in InstQL. To make a formula f effective (i.e. only compute traces for which f is true) we simply add a constraint to the query that specifies the condition for f must hold. If we have defined f by the condition `f` then this is done by "`constraint f;`".

For formulae of the form "$\Diamond l$" we define the conditions:

$$\texttt{condition eventually(E)}: \quad \texttt{happens(E);} \tag{5.3}$$

$$\texttt{condition eventually(F)}: \quad \texttt{holds(F);} \tag{5.4}$$

$$\texttt{condition eventually_not(E)}: \quad \texttt{not happens(E);} \tag{5.5}$$

$$\texttt{condition eventually_not(F)}: \quad \texttt{not holds(F);} \tag{5.6}$$

This is the reason InstLTL requires that in "$\Diamond p$", p is a literal rather than general formula. These conditions only work for event/fluent literals. We also define the following abbreviation for "$\neg\Diamond p$":

$$\texttt{condition never(P)}: \texttt{ not eventually(P);} \tag{5.7}$$

Note the difference between `not eventually(P)` and `eventually_not(P)` as discussed earlier (p57). The following condition is defined for $\Box p$:

$$\texttt{condition always(P)}: \texttt{ not eventually_not(P);} \tag{5.8}$$

Defining until (pUq) is a more complex. Naïvely, we could attempt to define "f until e" as follows:

$$\begin{aligned}\texttt{condition false_before(F,E)}: &\quad \texttt{happens(E) after not holds(F);}\\ \texttt{condition until(F,E)}: &\quad \texttt{not false_before(F,E);}\end{aligned}$$

This gives us almost what we need. However, translating this into ASP we see that the condition is too strong:

$$\begin{aligned}\texttt{false_before(F,E)} &\leftarrow \texttt{occurred(E, I), event(E), instant(I),}\\ &\qquad \texttt{not holdsat(F, J), ifluent(F), instant(J), after(I, J).}\\ \texttt{until(F, E)} &\leftarrow \texttt{not false_before(F, E).}\end{aligned}$$

We can satisfy `false_before(f, e)` if we can find time instants t_i and t_j such that $t_j < t_i$, e happens at t_i and at t_j f is false. That is, f cannot be false before any occurrence of e. The correct semantics of until are that f cannot be false before the *first* occurrence of e (Heljanko and Niemel, 2003).

In order to achieve the correct semantics, we introduce a new fluent to the institution `happened(e)` to indicate that event $e \in \mathcal{E}_{\mathcal{M}}$ has happened at any time in the past during the current trace; this is defined as follows:

$$\texttt{ifluent(happened(E))} \leftarrow \texttt{event(E).} \tag{5.9}$$

$$\texttt{holdsat(happened(E), I)} \leftarrow \texttt{occurred(E, I), event(E), instant(I).} \tag{5.10}$$

$$\begin{aligned}\texttt{holdsat(happened(E), I)} \leftarrow\ &\texttt{occurred(E, J), after(I, J),}\\ &\texttt{event(E), instant(I), instant(J).}\end{aligned} \tag{5.11}$$

This allows us to then specify fUe for a fluent f and event e as follows:

$$\begin{aligned}\texttt{condition fb(F, E)}: &\quad \texttt{not holds(F) while not holds(happened(E));}\\ \texttt{condition until(F, E)}: &\quad \texttt{not fb(F, E) and eventually(E)}\\ &\quad \texttt{and eventually(F);}\end{aligned} \tag{5.12}$$

Recall that $\mathcal{M}, x \models fUe \Leftrightarrow \exists i \cdot \mathcal{M}, x^i \models e \wedge (\forall j < i \cdot \mathcal{M}, x^j \models f)$. The condition for `until` requires that at some time instant e happens ("`eventually(e)`"). (Note that we also specify "`eventually(F)`" otherwise the variable `F` will be unrestricted which prevents it from being grounded.) This satisfies the first half of the definition of the semantics of U. Suppose then that e does occur and that it first occurs at t_i. This means `happened(E)` will hold from t_i onwards. For fUe to be true we require that f must hold at all time instants before t_i. The condition `fb` is true if there exists a time instant t_j before t_i at which f does not hold. For `until(f, e)` to be true, `fb(f, e)` must be false. This means that for all $t_j < t_i$, f must be true as required.

Expressing compound formulae joined with the logical connectives $\vee, \wedge, \neg$ is simple given that we will have already specified conditions as defined above for sub-formulae. Then for

"$h = \neg f$", "$h = f \wedge g$" and "$h = f \vee g$" (respectively) we have:

$$\texttt{condition h}: \quad \texttt{not f;} \tag{5.13}$$
$$\texttt{condition h}: \quad \texttt{f and g;} \tag{5.14}$$
$$\texttt{condition h}: \quad \texttt{f or g;} \tag{5.15}$$

5.3 Implementing the Institution Query Language

Section 4.2 defines the translation of InstQL into ASP. In this section, we describe the implementation of a prototype for a tool to automate this translation. This tool is able to parse InstQL queries and generate the corresponding ASP. This provides the situation shown in Figure 4.2; the generated query program can be combined with the institution program (generated from the InstAL specification) to allow reasoning about the institution.

5.3.1 Requirements & Design

The intention for the prototype translator was to produce a system able to automate the translation process for InstQL (described in 4.2, p48). This enables practical use of InstQL. The prototype is intended as a 'proof of concept' for the automatic translation of InstQL and not fully functional tool.

With this aim in mind, a simple requirements specification for the translator is outlined below.

Requirements Specification

Q1 The translator must recognise InstQL$_\delta$ queries as specified in Table 4.1.

- **Q1.1** In addition to the syntax defined in Table 4.1, the translator should also support comments introduced by a "%" which should then extend to the end of the current line.

Q2 The translator must generate ASP from InstQL$_\delta$ queries.

- **Q2.1** The generated ASP must be in a format that will be accepted by Lparse/Smodels (Syrjänen, n.d.).
- **Q2.2** The generated ASP must conform to the definition of the translation of InstQL as given in Section 4.2 (p48).
  - **Q2.2.1** The translator must provide generation of time instants so that *literals* within the same *while expression* use the same time instant.
  - **Q2.2.2** The translator must ensure that *literals* within different *while expressions* use different time instants.

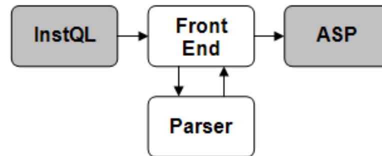

Figure 5.2: Architecture for the InstQL translator

Q3 The translator must provide a command line interface.

Q3.1 The interface must accept a file path to an InstQL query as an argument.

Q4 The translator must run on any system that supports InstAL.

Q5 The translator must be compatible with *InstEdit*.

Design

It was decided to implement the translator in two stages; the main stage is a parser that will parse InstQL queries and generate ASP from it. The other stage is the *front end* – this provides a command line utility that takes a filename as an argument and calls the parser to generate ASP from the contents of this file. This architecture is illustrated in Figure 5.2. The front end satisfies requirement **Q3** while the parser satisfies requirements **Q1** and **Q2**.

The parsing mechanism chosen was *recursive descent*. This top-down parsing algorithm allows relatively simple creation of parsers. The language selected for the parser implementation was Perl, a scripting language with good string-handling facilities. In addition, the InstAL tools are written in Perl and so this satisfies requirement **Q4**.

The Perl library module `Parse::RecDescent` supports the creation of recursive descent parsers. It was decided to use `Parse::RecDescent` since this offers a number of advantages. `Parse::RecDescent` generates recursive descent parsers from the grammars that they recognise. InstQL was defined by such a suitable grammar (Table 4.1). The grammar was designed to be right-recursive which allows recursive descent parsing. (Any recursive references by a rule in the grammar are not the first token mentioned in that rule.) In addition, `Parse::RecDescent` automatically splits its input into tokens. This means there is no need to write a separate lexical analyser.

Since the translation of InstQL into ASP is defined recursively, using a recursive descent parser means that as rules are recognised they can be translated into ASP 'on the fly' without need for first building a parse tree. Translating each kind of rule (as defined in Section 4.2, p48) can be done as a *semantic action* once the rule has been recognised by the parser. This approach greatly simplifies implementation of the parser but the drawback is that without first building a parse tree, error-handling is difficult.

The use of `Parse::RecDescent` means that the traditional approach of writing a lexical analyser to generate a token stream, a parser to organise the token stream into a parse tree and a back-end to operate on the parse tree can be simplified to a single component. This simplified approach is beneficial for this prototype of an InstQL translator.

5.3.2 Implementation & Testing

The InstQL translator was implemented in accordance with the design outlined above. The source code for the parser is provided in Appendix C.2. As planned in the design section, the parser generates ASP as it parses the query. This makes the parser implementation match closely the translation process outlined in Section 4.2 (p48). The parser was implemented as a Perl module using `Parse::RecDescent`.

The parser places no restrictions on condition names used. When a condition is defined, there is no requirement that the name is unique. This allows an alternate form of disjunction. For example:

```
condition event_fluent(X) : happens(X);
condition event_fluent(X) : holds(X);
```

Translates into ASP as:

```
event_fluent(X)  ←  occurred(X, I), event(X), instant(I).
event_fluent(X)  ←  holdsat(X, I), ifluent(X), instant(I).
```

This is equivalent to specifying the condition as:

```
condition event_fluent(X) : happens(X) or holds(X);
```

A module of utility functions for the parser was implemented. This provide management of time instant and condition names in order to ensure that when a name is generated it is unique. This is implemented through counters so that, for example, the time instants generated are called `I0`, `I1`, `I2` etc. In order to correctly generate ASP for *after expressions*, the utilities module also maintains lists of time instants generated and how far apart these should occur. At the end of an *after expression*, these lists are emptied and ASP generated from them. For example, suppose the instants list contains (`I0`, `I1`, `I2`, `I3`) and the list of how far apart these are is (2, 0, 4). The ASP generated in this case is:

```
after(I0, I1, 2), after(I1, I2), after(I2, I3, 4)
```

The front-end was implemented as a Perl script which takes a file path as a command-line argument. This takes the contents of the file (which are expected to be an InstQL query) and then strips any comments from the query. In accordance with requirement **Q1.1**, a comment is any text following a "%" up to the end of that line. This is consistent with the comment syntax used by InstAL. Handling this in the front-end makes use of Perl's

string handling capabilities and makes the parser more readable by separating comments from the actual language.

The front-end then calls the parser on the query (without comments). Each statement of the query (condition declaration or constraint, terminated by a semi-colon) produces an array of ASP rules. The parser returns an array of the results for all statements of the query. The front-end 'deciphers' this output by printing the contents of the arrays to standard output. This allows the translation of the query to be saved to file by redirecting the standard output of the process. Alternatively, an external program calling the process can use the output however it chooses.

Having implemented the translator, is was linked to *InstEdit* (see Section 3.3.5, p41). This satisfies requirement **Q5**. *InstEdit* was modified so as to be able to call the front-end for the translator as an external process. *InstEdit*saves the ASP output from the InstQL parser to a file. This program and the base InstQL programs (e.g. definition of `after`) are combined with the program generated from an institution. This ensures only traces satisfying the query are returned by the answer set solver.

In order to test the translator, the five sample queries from Section 4.3 (p59) were given as input to the translator. This provides a form of black-box testing to verify that the translator works as expected. The results of this testing are given in Appendix B.3.

5.4 Summary

In this chapter we discussed practical aspects of InstQL, the query language introduced in Chapter 4. These included how InstQL can be used for various kinds of reasoning. Section 5.1.1 (p62) demonstrated the use of InstQL for the common reasoning patterns of *prediction*, *postdiction* and *planning*.

Section 5.1.2 (p65) then discussed possible application of these reasoning methods by agents within an institution. With an agent's interpretation of the institution represented in InstAL, the agent can use InstQL to solve planning problems to determine how to achieve its goals. InstQL also could be used for the agent to predict the effects of its actions.

In Section 5.2 (p65) we introduced *InstLTL* a restricted form of linear temporal logic (LTL) that can be represented in InstQL. The details of expressing InstLTL formulae in InstQL were explained. A prototype of a translator to generate ASP from InstQL queries has been produced. The development of this translator was then described (Section 5.3, p69).

Chapter 6

Conclusion

6.1 Project Summary

This project set out to enhance InstAL, an action language designed to model institutions. We sought to do this in two ways: to provide a toolkit to assist institution designers using InstAL and to design a query language. These were realised as *InstEdit*, the institution editor, and InstQL, the institution query language.

Prior to this project, the process of using InstAL was as follows:

- Specify institution(s) in InstAL
- Translate institution(s) into an ASP institution program
- Write an ASP query program for the institution(s)
- Generate an ASP program to define the number of time instants over which to reason
- Combine the institution, query and time programs with some library base programs (specifying properties of the model) and ground
- Compute the answer sets of the ground program
- Visualise the output (which is a two stage process if an output graph is desired)

The work of this project has simplified this complex process. *InstEdit* was described in Chapter 3. This system provides the functionality of a text editor, augmented with syntax highlighting for InstAL and InstQL. This supports the institution designer at the stage of creating an institution in InstAL. *InstEdit* manages the process of reasoning about institutions for the user.

Multiple InstAL specifications (including domain definition files) can be written in *InstEdit*. An InstQL query (Chapter 4) for the system can be written, in the same editor. These

can be grouped together and managed as a *project*. From *InstEdit*, a command is available solve the project. This causes *InstEdit* to prompt for the number of time instants to use.

The time program is generated. The InstAL specifications and InstQL query are translated into ASP and combined with the time and base programs. The ASP is grounded and solved. This is all handled by the *InstEdit*. The output from these processes is displayed within *InstEdit* to inform the user of the results of solving and allow then to identify any errors in the institution(s).

The user can then launch *InstViz* from *InstEdit* to visualise the traces. Alternatively, another command in *InstEdit* is available to generate a graph description from the answer sets and call GraphViz to draw the graph.

Whereas previously queries for InstAL had to be specified in ASP, these can now be written in InstQL. This domain specific query languages allows the user to reason about institutions with greater ease. InstQL abstracts the user away from having to define time instants by allowing for events and fluents to be temporally related using the keywords `while` and `after`.

The translation of InstQL into ASP is defined (Section 4.2, p48). We considered how existing ASP queries for InstAL can be written in InstQL in Section 4.3 (p59) and demonstrated that following the translation we get equivalent ASP. The translation of InstQL into ASP was automated – the development of a tool to perform the translation is given in Section 5.3 (p69).

We introduced InstLTL (institutional linear temporal logic). This restricted form of linear temporal logic (Pnueli, 1977) can be expressed in InstQL. The details of expressing InstLTL in InstQL are provided in Section 5.2 (p65).

This project has enhanced InstAL as a tool for institutional modelling. There is now a dedicated toolkit, *InstEdit*, which simplifies the process of reasoning about institutions defined by InstAL. *InstEdit* supports this process from writing InstAL specifications up to visualising traces of the institution. We now have a query language that allows us to reason about properties of InstAL institutions. This language, InstQL, maintains the abstraction above ASP which InstAL provides for institution specification. It is higer-level than ASP and expressed in domain specific terms making writing queries easier.

6.2 Evaluation

The process of reasoning about institutions is now all controlled within one system. A comparison of Figures 3.2 (p30) and 3.4 (p33) illustrates the simplification of this process for the user that *InstEdit* provides. In addition, by providing project management and syntax highlighting, *InstEdit* makes the process of creating institutions (and multi-institution systems) easier for the institution designer.

InstEdit has been designed in such a way as to permit easy extension. It gives us a base

platform from which we can develop further functionality to aid development of institutions (see p 76). Should we wish to use a different answer set solver, this should be a simple modification thanks to the architecture of *InstEdit*. This makes *InstEdit* a useful tool, though there is scope for numerous extensions to its functionality.

InstQL provides specification of queries at a higher-level than those in ASP. Writing queries in ASP, we typically talk about when an event or fluent occurs while InstQL abstracts away from this. For example, in ASP we would write "`holdsat(f, i)`" (fluent `f` is true at time `i`) while in InstQL we write "`holds(f)`" (`f` is true at some point). InstQL makes it easy to express queries of the form "Does this combination of events and fluents ever happen?". For example, Cliffe et al. (2008) specify a query for the Dutch auction round institution which means "Does a decision deadline ever occur whilst we have more than one bid?". The following InstQL query specifies this condition:

```
constraint happens(desdl) while holds(conflict);
```

This permits only traces in which the deadline occurs whilst we have multiple bids (and are in a conflicted state). The semantics of InstQL are implicitly existential. The two base predicates (`happens` and `holds`) mean there exists some time instant at which a certain event occurs or a certain fluent holds. Universally quantified conditions are possible indirectly through negation (much as in ASP itself (Eiter et al., 2003)).

The existential semantics of InstQL allow easy formulation of queries where we require that a certain situation does (not) occur. In more complex situations, the inability to define time instant variables is a limiting factor in the expressive power of the language. This is what causes an unrestricted form of LTL to be expressible in InstQL. However, we demonstrated (p62) that the class of queries that can be expressed in InstQL is sufficient to solve *prediction*, *postdiction* and *planning* problems.

The experience gained in completing the project means that if it were to be repeated one major change would be made. The initial survey of the state of InstAL revealed a number of areas where there was scope for work to be done. This project addresses two such areas: the tool support and the lack of a query language. The project was successful in addressing both areas, though both have great scope for further improvement and development. If the project were to be repeated now, we would focus on only one of these areas. This would allow a more complete solution to be delivered in a specific area.

The motivation for working on both areas was to create solutions which left the areas in an 'acceptable state'. *InstEdit* delivers a text editor that is able to aid and control the InstAL reasoning process. This toolkit can serve as a base for future development to deliver a more feature rich IDE. InstQL provides a query language which allow a certain (common) class of query to be expressed easily. This can be extended to allow further queries to be specified. These were both deemed high-priority areas were development could have a large impact on the utility of InstAL. Following the work of this project, neither is an issue but neither is solved.

6.3 Future Work

Both *InstEdit* and InstQL provide a good platform for developing further the support for institutional modelling. *InstEdit* in particular is designed to be an extensible tool. Further features can be added to it to allow it to evolve into a richer development environment. For example, features such as code completion and syntax error detection could be introduced. We consider here two possible extensions of *InstEdit* which we believe would be of great value.

An institution debugger could be added to *InstEdit*. This would allow designers to better understand properties of institutions by stepping through traces. At each step, the debugger could provide the following information:

- the current institution state
- the observable event, e, that brought about the transition
- the set, E, of events generated by e
- the set of fluents initiated by E
- the set of fluents terminated by E

This would allow an institution designer to ascertain why certain conditions were occuring. Since this requires stepping through traces, it could be implemented as an alternative visualisation method.

InstAL supports output in graph format. This provides a state transition graph describing the institution. Nodes are labelled with fluents and edges with (observable) events. This provides a description of the institution. This process could be reversed to specify the institution by its state transition graph. A new tab type could be added to *InstEdit* to support visual editing of a graph. Tools could be created to generate an InstAL specification from this graph.

For InstQL, we discussed (p65) modelling a restricted form of LTL in InstQL. As mentioned above, the current language definition prevents unrestricted LTL being expressed in InstQL. The defnition could be expanded to provide a more powerful language allowing all of LTL to be expressed. This could be extended to other temporal logics such as CTL* (Emerson and Halpern, 1986).

The InstQL translator is a prototype only and provides poor error handling. Future development of this tool is required to produce a more robust translator. We could investigate the possibility of additional translators to map (subsets of) other related languages such as $(\mathcal{C}+)^{++}$ (Sergot, 2004) or the Event Calculus into InstQL. This would make using InstAL accessible to a greater part of the research community.

There is a considerable body of work in the literature on modelling institutions. However, as noted by Aldewereld et al. (2006), there is no definitive practical method established.

InstAL represents one such approach. Its treatment of multi-institution systems and basis in ASP make it unique. This project has begun the process of developing the support for InstAL required to make it an easy and practical tool for institutional modelling. Future development can realise this goal.

Bibliography

Aldewereld, H., Dignum, F., García-Camino, A., Noriega, P., Rodríguez-Aguilar, J. A. and Sierra, C. (2006), Operationalisation of norms for usage in electronic institutions, *in* 'AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems', ACM, New York, NY, USA, pp. 223–225.

Baral, C. (2003), *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press.

Cliffe, O. (2007), Specifying and Analysing Institutions in Multi-Agent Systems using Answer Set Programming, PhD thesis, Dept. of Computer Science, University of Bath.

Cliffe, O., De Vos, M. and Padget, J. (2005), Specifying and analysing agent-based social institutions using answer set programming, *in* 'Selected revised papers from the workshops on Agents, Norms and Institutions for Regulated Multi-Agent Systems (ANIREM) and Organizations and Organization-Oriented Programming (OOOP) at AAMAS'05', Springer-Verlag, Utrecht, The Netherlands.

Cliffe, O., De Vos, M. and Padget, J. (2006), Specifying and reasoning about multiple institutions, *in* 'Coordination, Organization, Institutions and Norms in Agent Systems (COIN'06)', Japan.

Cliffe, O., De Vos, M. and Padget, J. (2008), Embedding landmarks and scenes in a computational model of institutions, *in* 'Coordination, Organizations, Institutions, and Norms in Agent Systems III', Vol. 4870 of *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, pp. 41–57.

De Vos, M. and Vermeir, D. (2004), Extending answer sets for logic programming agents, *in* 'Annals of Mathematics and Artificial Intelligence', Vol. 42, Kluwer Academic Publishers, The Netherlands, pp. 103 – 139.

Eiter, T., Faber, W., Leone, N. and Pfeifer, G. (2003), 'Computing preferred answer sets by meta-interpretation in answer set programming', *Theory Pract. Log. Program.* **3**(4), 463–498.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G. and Scarcello, F. (1998), The KR system `dlv`: Progress report, comparisons and benchmarks, *in* A. G. Cohn, L. Schubert and S. C.

Shapiro, eds, 'Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)', Morgan Kaufmann Publishers, pp. 406–417.

Emerson, E. A. (1990), Temporal and modal logic, *in* J. van Leeuwen, ed., 'Handbook of Theoretical Computer Science', Elsevier, pp. 995–1072.

Emerson, E. A. and Halpern, J. Y. (1986), '"Sometimes" and "not never" revisited: on branching versus linear time temporal logic', *Journal of the ACM* **33**(1), 151–178.

Fornara, N. and Colombetti, M. (2008), Specifying and enforcing norms in artificial intelligence (short paper), *in* 'Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)', International Foundation for Autonomous Agents and Multiagent Systems, pp. 1481–1484.

Garcia-Camino, A., Noriega, P. and Rodriguez-Aguilar, J. A. (2005), Implementing norms in electronic institutions, *in* 'AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems', ACM, New York, NY, USA, pp. 667–673.

Gaudou, B., Longin, D., Lorini, E. and Tummolini, L. (2008), Anchoring institutions in agents' attitudes: towards a logical framework for autonomous multi-agent systems, *in* 'AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems', International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 728–735.

Gebser, M., Schaub, T. and Thiele, S. (2007), GrinGo: A new grounder for answer set programming, *in* 'Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning', Lecture Notes in Computer Science, Springer-Verlag, Tempe, AZ, USA, pp. 266–271.

Gelfond, M. and Lifschitz, V. (1988), The stable model semantics for logic programming, *in* 'Logic Programming: Proceedings of the Fifth International Conference and Symposium', MIT Press, pp. 1070–1080.

Gelfond, M. and Lifschitz, V. (1998), 'Action languages', *Electronic Transactions on AI* **2**, 193–210.

Georgeff, M., Pell, B., Pollack, M., Tambe, M. and Wooldridge, M. (1999), The belief-desire-intention model of agency, *in* J. Müller, M. P. Singh and A. S. Rao, eds, 'Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)', Vol. 1555, Springer-Verlag: Heidelberg, Germany, pp. 1–10.

Giunchiglia, E., Lee, J., Lifschitz, V. and Turner, H. (2001), Causal laws and multi-valued fluents, *in* 'Proceedings of Workshop on Nonmonotonic Reasoning, Action and Change (NRAC)'.

Gressmann, J., Janhunen, T., Mercer, R., Schaub, T., Thiele, S. and Tichy, R. (2005), Platypus: A platform for distributed answer set solving, *in* C. Baral, G. Greco, N. Leone and G. Terracina, eds, 'Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning', Lecture Notes in Computer Science, Springer-Verlag, Diamante, Italy, pp. 227–239.

Heljanko, K. and Niemel, I. (2003), Bounded LTL model checking with stable models, *in* 'Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning', Springer-Verlag, pp. 200–212.

Hodges, W. (2001), *Logic*, 2nd edn, Penguin Books.

Jo, C.-H., Chen, G. and Choi, J. (2004), A new approach to the BDI agent-based modeling, *in* 'SAC '04: Proceedings of the 2004 ACM symposium on Applied computing', ACM, New York, NY, USA, pp. 1541–1545.

Kowalski, R. and Sergot, M. (1986), A logic-based calculus of events, *in* 'New Generation Computing', Vol. 4, Ohmsha, Ltd., pp. 67–95.

Lifschitz, V. and Turner, H. (1999), Representing transition systems by logic programming, *in* 'Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning', pp. 92 – 106.

McCarthy, J. and Hayes, P. (1969), Some philosophical problems from the standpoint of artificial intelligence, *in* B. Meltzer and D. Michie, eds, 'Machine Intelligence 4', Edinburgh University Press, pp. 463–502.

Moore, R. C. (1985), A formal theory of knowledge and action, *in* J. R. Hobbs and R. C. Moore, eds, 'Formal Theories of the Commonsense World', Greenwood Publishing Group Inc., Westport, CT, USA.

Mueller, E. (2006), *Commonsense Reasoning*, Morgan Kaufmann, San Francisco, CA.

Niemelä, I. and Simons, P. (1997), Smodels - an implementation of the stable model and well-founded semantics for normal lp, *in* 'LPNMR '97: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning', Springer-Verlag, London, UK, pp. 421–430.

Nieuwenborgh, D. V., Vos, M. D., Heymans, S. and Vermeir, D. (2007), Hierarchical decision making in multi-agent systems using answer set programming, *in* 'Computational Logic in Multi-Agent Systems', Vol. 4372 of *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, pp. 20–40.

Noriega, P. (1997), Agent mediated auctions: The Fishmarket Metaphor, PhD thesis, Universitat Autónoma de Barcelona.

Nwana, H. S. (1996), Software agents: An overview, *in* 'Knowledge Engineering Review', Vol. 11, Cambridge University Press, pp. 1 –40.

Pnueli, A. (1977), The Temporal Logic of Programs, *in* '19th Annual Symp. on Foundations of Computer Science'.

Sergot, M. (2004), $(\mathcal{C}+)^{++}$: An action language for modelling norms and institutions, Technical Report 8, Department of Computing, Imperial College, London.

Shehory, O. and Kraus, S. (1998), 'Methods for task allocation via agent coalition formation', *Artif. Intell.* **101**(1-2), 165–200.

Shoham, Y. and Tennenholtz, M. (1995), 'On social laws for artificial agent societies: off-line design', *Artif. Intell.* **73**(1-2), 231–252.

Sichman, J. S., Conte, R., Demazeau, Y. and Castelfranchi, C. (1994), A social reasoning mechanism based on dependence networks, *in* 'Proc. 12th European Conference on Artificial Intelligence (ECAI'94)', Amsterdam, The Netherlands.

Sistla, A. P. and Clarke, E. M. (1985), 'The complexity of propistional linear temporal logics', *Journal of the ACM* **32**(3), 733–749.

Sommerville, I. (2004), *Software Engineering*, 7th edn, Addison-Wesley.

Syrjänen, T. (n.d.), 'Lparse 1.0 user's manual', Online [Accessed 14 April 2009]. Available from: `http://www.tcs.hut.fi/Software/smodels/`.

Tadjouddine, E., Guerin, F. and Vasconcelos, W. (2008), Abstractions for model-checking game-theoretic properties in auctions, *in* 'Proceedings of AAMAS 2008', Estoril, Portugal.

Viganò, F. and Colombetti, M. (2007), Symbolic model checking of institutions, *in* 'ICEC '07: Proceedings of the ninth international conference on Electronic commerce', ACM, New York, NY, USA, pp. 35–44.

Wooldridge, M. (1992), The Logical Modelling of Computational Multi-Agent Systems, PhD thesis, University of Manchester.

Wooldridge, M. (1998), Verifiable semantics for agent communication languages, *in* 'Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98)', IEEE Computer Society Press, pp. 349–356.

Wooldridge, M. (2002), *An Introduction to MultiAgent Systems*, Wiley, England.

Wooldridge, M. and Jennings, N. (1995), Agent theories, architectures, and languages: a survey, *in* 'ECAI-94: Proceedings of the workshop on agent theories, architectures, and languages on Intelligent agents', Springer-Verlag, pp. 1–39.

Appendix A

InstAL

This appendix presents work done by Cliffe et al.. Their work on institutions has produced the action language InstAL and a formal model of multi-institutions which provides the semantics for InstAL. This work is fundamental to this project and so we summarise it here. This appendix is based on Cliffe (2007) and Cliffe et al. (2005; 2006; 2008).

A.1 Institution Model

This section presents a summary of the institutional model developed by Cliffe et al. (2006) (see also Cliffe, 2007). This model is for not only of individual institutions, but also multi-institution systems where we have multiple, interacting institutions.

A.1.1 Model Definition/Syntax

Definition A.1 *A* multi-institution system *is a tuple $\mathcal{M} = \langle \mathcal{I}_1, \ldots, \mathcal{I}_n \rangle$ where each $\mathcal{I}_i$ is an institution.*

Definition A.2 *An* institution *is a tuple $\mathcal{I}_i = \langle \mathcal{E}_i, \mathcal{F}_i, \mathcal{C}_i, \mathcal{G}_i, \Delta_i \rangle$ where $\mathcal{E}_i$ is a set of events, $\mathcal{F}_i$ is a set of fluents, $\mathcal{C}_i$ a function defining causal rules (or consequences), $\mathcal{G}_i$ an event-generation function and Δ_i is the initial state of the institution.*

The set of events $\mathcal{E}_i$ is a set of symbols representing those events that can occur in the institution. It is split into two disjoint subsets, the *exogenous* or *observable* events $\mathcal{E}^i_{obs}$ and the *institutional* events $\mathcal{E}^i_{inst}$. Exogenous events are externally obeservable events such as communication acts between agents. Institutional events are those created by the semantics of the institution.

The institutional events are further broken down into two (disjoint) subsets, *institutional actions* $\mathcal{E}^i_{instact}$ and *violation events* $\mathcal{E}^i_{viol}$. The violation events are defined so that (at a

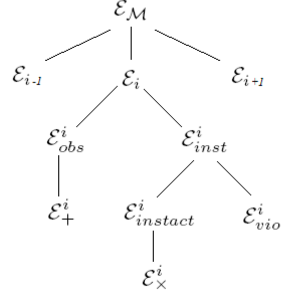

Figure A.1: Relationship between event types in the institutional model. If E' appears below E in the tree then $E' \subseteq E$.

minimum) there is a violation event for each institutional action and each exogenous event:

$$\forall e \in \mathcal{E}^i_{obs} \cup \mathcal{E}^i_{instact} \cdot viol(e) \in \mathcal{E}^i_{viol}$$

Note that additional violation events can be defined as the institution being modelled requires. The events of multi-institution system are given by:

$$\mathcal{E}_{\mathcal{M}} = \bigcup_{i=1}^{n} \mathcal{E}_i \tag{A.1}$$

The sets $\mathcal{E}^{\mathcal{M}}_{inst}$, $\mathcal{E}^{\mathcal{M}}_{obs}$, $\mathcal{E}^{\mathcal{M}}_{viol}$ and $\mathcal{E}^{\mathcal{M}}_{instact}$ are defined similarly. We define a subset of the exogenous events of the multi-institution system as the *envioronment states* – these are those events which have a concrete effect on the envioronment of $\mathcal{M}$. That is $\mathcal{E}^{\mathcal{M}}_{env} \subseteq \mathcal{E}^{\mathcal{M}}_{obs}$.

A set of *institutional creation* events are identified $\mathcal{E}^i_{+} \subseteq \mathcal{E}^i_{obs}$. A set of *dissolution* events are defined $\mathcal{E}^i_{\times} \subseteq \mathcal{E}^i_{instact}$. The relationship between the sets of events of an institution is shown in figure A.1.

The set of fluents is broken into *domain fluents* $\mathcal{D}_i$ (institution specific fluents) and *normative fluents*. The normative fluents are broken down into the following disjoint subsets:

- $\mathcal{W}_i$ a set of *institutional power* fluents of the form $pow(j,e) \cdot 1 \leq j \leq n, e \in \mathcal{E}_{instact}$
- $\mathcal{P}_i$ a set of *permissions* of the form $perm(e) \cdot e \in \mathcal{E}^i_{instact} \cup \mathcal{E}^i_{obs}$ denoting that event e is permitted. An event e' is forbidden if $perm(e') \notin P$
- $\mathcal{O}_i$ a set of obligations of the form $obl(e,d,v)$ where $e,d \in \mathcal{E}_i$ and $v \in \mathcal{E}^i_{inst}$ indicating an obligation to perform event e before the occurrence of event d or be sanctioned with violation v (note that v does not have to be a violation event of $\mathcal{E}^i_{viol}$ but any institutional event indicating a failure to satisfy the obligation).
- $\mathcal{S}_i$ a set of initiating powers of the form $inipow(j,f) \cdot 1 \leq j \leq n, f \in \mathcal{D}_i$ indicating that institution j is able to initiate fluent f in institution i

- $\mathcal{T}_i$ a set of terminating powers of the form $termpow(j, f) \cdot 1 \leq j \leq n, f \in \mathcal{D}_i$ indicating that institution j is able to terminate fluent f in institution i

In addition, for each institution the special fluent *live* is defined as a member of $\mathcal{F}_i$. This fluent is unique in that it is independent of institution semantics and indicates that the institution exists. Any creation event in $\mathcal{E}^i_+$ initiates *live* and any dissolution event in $\mathcal{E}^i_\times$ terminates it (see (A.3) for the definition of initiated and terminated fluents).

The set of fluents for a multi-institution system is defined as:

$$\mathcal{F}_\mathcal{M} = \bigcup_{i=1}^{n} \mathcal{F}_i \tag{A.2}$$

The state of an institution is defined by those fluents which are true, i.e. state $s \subseteq \mathcal{F}_i$. The state of a multi-institution system is a sequence of such states $(s_1, \ldots, s_n)$. Note that Sergot (2004) uses the state of a $(\mathcal{C}+)^{++}$ description to imply which fluents are true at that time; Cliffe et al. do the opposite and imply the state through the fluents.

State formulae are used to describe collections of states and are defined as follows:

Definition A.3 *The set of all* state formulae *for institution i, is denoted $\mathcal{X}_i$. $\mathcal{X}_i = \mathbb{P}(\mathcal{F}_i \cup \neg\mathcal{F}_i)$ where $\neg\mathcal{F}_i$ is this set of all the fluents of i negated.*

The set of all possible states for an institution is $\Sigma_i = \mathbb{P}(\mathcal{F}_i)$. For $\mathcal{M}$ the set of all possible states is given by $\Sigma_\mathcal{M} = \Sigma_1 \times \cdots \times \Sigma_n$.

The consequence function $\mathcal{C}_i$ is defined as follows:

$$\mathcal{C}_i : \mathcal{X}_i \times \mathcal{E}_i \to \mathbb{P}(\mathcal{F}_\mathcal{M}) \times \mathbb{P}(\mathcal{F}_\mathcal{M}) \tag{A.3}$$

The first set of fluents are those initiated by the causal rule and the second set those terminated by the rule. For state X and event e, the fluents initiated are denoted as $\mathcal{C}_i^\uparrow(X, e)$ and those terminated are denoted by $\mathcal{C}_i^\downarrow(X, e)$. Consequences are able to affect fluents in other institutions (provided the initiating institution has the power to initiate/terminate those fluents in the target institution).

The event-generation function, $\mathcal{G}_i$, is defined:

$$\mathcal{G}_i : \mathcal{X}_i \times \mathcal{E}_i \to \mathbb{P}(\mathcal{E}^\mathcal{M}_{inst}) \tag{A.4}$$

This generation of events corresponds to a “counts as” relation between events (e.g. as in $(\mathcal{C}+)^{++}$).

For a multi-institution, the set of state formulae is given by $\mathcal{X}_\mathcal{M} = \mathbb{P}(\mathcal{F}_\mathcal{M} \cup \neg\mathcal{F}_\mathcal{M})$. Event generation within a multi-institution relates events in different institutions. That is:

$$\mathcal{G}_\mathcal{M} : \mathcal{X}_\mathcal{M} \times (\mathcal{E}^\mathcal{M}_{inst} \cup \mathcal{E}^\mathcal{M}_{env}) \to \mathbb{P}(\mathcal{E}^\mathcal{M}_{obs}) \tag{A.5}$$

This differs from the single institution generation rule in that for an observable event to generate others, it must have been identified as an environment event. In addition, it is observable events that are generated within the multi-institution while institutional events are generated within a single institution. The model requires that only a single observable event may occur at any time. This is implicitly enforced by generation in single institutions. Within a multi-institution, it is necessary to enforce this property. This is done by defining the function $PR_{\mathcal{M}} : \mathbb{P}(\mathcal{E}_{\mathcal{M}}) \to \mathbb{P}(\mathcal{E}_{\mathcal{M}})$ that describes when an event may generate another. That is, $\{e'\} \in PR_{\mathcal{M}}(e)$ if and only if it is possible for e to generate e'.

Definition A.4 *For a multi-institution specification, $\mathcal{M}$, the function $PR_{\mathcal{M}}$ is defined as:*

$$\begin{aligned} PR_{\mathcal{M}}(E) = \{\ e' \in \mathcal{E}_{\mathcal{M}} \mid \quad & \exists \phi \in \mathcal{X}_{\mathcal{M}}\ \exists e \in E \cdot e' \in \mathcal{G}_{\mathcal{M}}(\phi, e) & \vee \\ & \exists \mathcal{I}_i \in \mathcal{M}\ \exists e \in E\ \exists \phi \in \mathcal{X}_i \cdot e' \in \mathcal{G}_i(\phi, e) & \vee \\ & \exists \mathcal{I}_i \in \mathcal{M}\ \exists e \in E\ \exists e'' \in \mathcal{E}_i \cdot obl(e'', e, e') \in \mathcal{F}_i & \} \end{aligned}$$

That is, the events that could be generated by e are those generated by e within the multi-institution, those generated within one of the component institutions or arise as a sanction through not executing e. Denoting by $(\{e\}, PR_{\mathcal{M}})^*$ the transitive closure of $\{e\}$ over $PR_{\mathcal{M}}$ we can then define a constraint on the generation rules of a multi-institution.

Definition A.5 $\mathcal{G}_{\mathcal{M}}$ *is* valid *with respect to a multi-institution specification $\mathcal{M}$ if and only if:*

$$\forall \mathcal{I}_i \in \mathcal{M}\ \forall \{e, e'\} \subseteq \mathcal{E}^i_{obs} \cdot e' \notin (\{e\}, PR_{\mathcal{M}})^*$$

A.1.2 Model Semantics

Given a state $S_k \in \Sigma_k$ and an expression $\phi \in \mathcal{X}_k$, the *satisfaction relation* for institution k is defined as follows:

$$S_k \models \phi \Leftarrow \begin{cases} \phi = \emptyset \\ \phi = \{p\} \wedge p \in S_i \\ \phi = \{\neg p\} \wedge p \notin S_i \\ \forall p \in \phi \cdot S_k \models \{p\} \end{cases} \tag{A.6}$$

Given a state $S_{\mathcal{M}} \in \Sigma_{\mathcal{M}}$ (and where the institution $\mathcal{I}_k$ has state S_k), the satisfaction relation between $S_{\mathcal{M}}$ and an expression $\phi \in \mathcal{X}_{\mathcal{M}}$ is defined as follows:

$$S_{\mathcal{M}} \models \phi \Leftarrow \begin{cases} \phi = \emptyset \\ \phi = \{p\} \wedge \exists \mathcal{I}_k \in \mathcal{M} \cdot p \in \mathcal{F}_i \wedge p \in S_i \\ \phi = \{\neg p\} \wedge \exists \mathcal{I}_k \in \mathcal{M} \cdot p \in \mathcal{F}_i \wedge p \notin S_i \\ \forall p \in \phi \cdot S_{\mathcal{M}} \models \{p\} \end{cases} \tag{A.7}$$

Event Generation

Event generation within the institution $\mathcal{I}_k$ is given by the function $GR : \Sigma_k \times \mathbb{P}(\mathcal{E}_k) \to \mathbb{P}(\mathcal{E}_k)$. This is defined as follows for state $S \in \Sigma_k$ and events $E \subseteq \mathcal{E}_k$:

$$\begin{array}{lll} GR(S,E) & = \{e \in \mathcal{E}_k \mid & \\ & e \in E & \vee \\ & \exists e' \in E, \phi \in \mathcal{X}_k, e \in \mathcal{G}_k(\phi, e') \cdot e \in \mathcal{E}^k_{instact} \wedge S \models pow(e) \wedge S \models \phi & \vee \\ & \exists e' \in E, \phi \in \mathcal{X}_k, e \in \mathcal{G}_k(\phi, e') \cdot e \in \mathcal{E}^k_{viol} \wedge S \models \phi & \vee \\ & \exists e' \in E \cdot e = viol(e') \wedge S \models \neg perm(e') \wedge e' \notin (\mathcal{E}^k_+ \cup \mathcal{E}^k_\times) & \vee \\ & \exists e' \in \mathcal{E}_k, d \in E \cdot S \models obl(e', d, e) & \} \end{array} \tag{A.8}$$

There are five cases for events generated by E given in the definition of $GR(S,E)$: events in E are preserved, empowered generation of institutional actions specified by $\mathcal{G}_k$ for the current state, generation of violations for the current state that are specified in $\mathcal{G}_k$ (note that it is not required these are empowered), violations arising from performing non-permitted events and sanctions from obligations whose deadline was in E.

Since $GR(S,E)$ preserves events in E, it is monotonic when applied to itself. At each step of the model, a single observable event is performed which generates some set of events. The set of all events generated by an observable event $e_{obs} \in \mathcal{E}^k_{obs}$ is given by $GR^\omega(S, \{e_{obs}\})$ – a fixed-point of $GR(S, \{e_{obs}\})$. Such a fixed-point can be reached since GR is monotonic when applied to itself. Let $GR^n(S,E)$ be the function defined recursively as:

$$GR^n(S,E) = \left\{ \begin{array}{lll} GR(S,E) & \Leftrightarrow & n = 1 \\ GR^{n-1}(S,E) & \Leftrightarrow & n \geq 2 \end{array} \right. \tag{A.9}$$

This allows us to give an explicit definition of $GR^\omega(S,E)$.

Definition A.6 *$GR^\omega(S,E) = E'$ such that $GR(S,E') = E'$ and there exists $n \geq 1$ such that $GR^n(S,E) = E'$.*

The function GR provides the means to obtain the set of events generated by another such set (in a given state). Fixed-points of the form GR^ω allow the complete set of events generated by a given set of events in a given state to be determined (specifically, this can be used to give the set of all events generated from a particular observable event). However, these two constructs alone are not sufficient to specify the generation of events within an institution. They allow (in the current state) for events to be generated before a creation event occurs, i.e. before the institution actually exists. The *event occurrence* function $OC : \Sigma_k \times \mathcal{E}^k_{obs} \to \mathbb{P}(\mathcal{E}_k)$ is used to obtain the complete set of events generated by an observable event as follows:

$$OC(S, e_{obs}) = \left\{ \begin{array}{lll} \emptyset & \Leftarrow & S \models \neg live \wedge e_{obs} \notin \mathcal{E}^k_+ \\ GR^\omega(S, \{e_{obs}\}) & \Leftarrow & S \models \neg live \wedge e_{obs} \in \mathcal{E}^k_+ \\ GR^\omega(S, \{e_{obs}\}) & \Leftarrow & S \models live \end{array} \right. \tag{A.10}$$

In a multi-institution, event generation is given by $GM : \Sigma_{\mathcal{M}} \times \mathbb{P}(\mathcal{E}_{\mathcal{M}}) \to \mathbb{P}(\mathcal{E}_{\mathcal{M}})$:

$$GM(S,E) = \{e \in \mathcal{E}_{\mathcal{M}} \mid \begin{array}{ll} e \in E & \vee \\ \exists e' \in E, \phi \in \mathcal{X}_{\mathcal{M}} \cdot e \in \mathcal{G}_{\mathcal{M}}(\phi, e') \wedge S \models \phi & \vee \\ \exists \mathcal{I}_k \in \mathcal{M}, S_k \in S, e' \in \mathcal{E}^k_{obs} \cdot e \in OC(S_k, e') & \} \end{array} \tag{A.11}$$

As with GR, since GM preserves events in E we can obtain a fixed-point $GM^{\omega}(S,E)$. Event occurrence in a mutli-institution is given by $OM : \Sigma_{\mathcal{M}} \times \mathcal{E}^{\mathcal{M}}_{env} \to \mathbb{P}(\mathcal{E}_{\mathcal{M}})$:

$$OM(S,e) = GM^{\omega}(S, \{e\}) \tag{A.12}$$

Event Effects

Initiation of fluents is provided by the function $INITI : \Sigma_k \times \mathcal{E}^k_{obs} \to \mathbb{P}(\mathcal{F}_k)$.

$$INITI(S,e) = \{p \in \mathcal{F}_k \mid \exists e' \in OC(S,e), \phi \in \mathcal{X}_k \cdot p \in \mathcal{C}^{\uparrow}_k(\phi, e') \wedge S \models \phi\} \tag{A.13}$$

Just as with the function GR, the case in which an institution is inactive (i.e. has not been created or has been dissolved) needs to be considered. This is achieved by the function $INIT : \Sigma_k \times \mathcal{E}^k_{obs} \to \mathbb{P}(\mathcal{F}_k)$:

$$\begin{array}{lll} INIT(S,e) & = \{p \in \mathcal{F}_k \mid & \\ & p \in INITI(S,e) \wedge S \models live \wedge \forall e' \in OC(S,e) \cdot e' \notin \mathcal{E}^k_{\times} & \vee \\ & (p \in INITI(S,e) \cup \Delta_k \cup \{live\} \wedge e \in \mathcal{E}^k_{+} \wedge S \not\models live \wedge & \\ & \forall e' \in OC(S,e) \cdot e' \notin \mathcal{E}^k_{\times}) & \} \end{array} \tag{A.14}$$

The first case of $INIT$ gives us that for an active institution fluents are initiated in accordance with the consequence function (provided no event generated by e causes dissolution). The second case provides initiation of the following fluents:

1. Those specified by the consequence function for e (i.e. those in $INITI(S,e)$).
2. Those true in the initial state, Δ_k.
3. The special *live* fluent.

This case occurs when the institution is inactive, e is a creation event and no event generated by e will cause dissolution. Termination of fluents is provided by $TERM : \Sigma_k \times \mathcal{E}^k_{obs} \to \mathcal{F}_k$:

$$TERM(S,e) = \{p \in S \mid \begin{array}{ll} \exists e' \in OC(S,e), \phi \in \mathcal{X}_k \cdot p \in \mathcal{C}^{\downarrow}_k(\phi, e') \wedge S \models \phi & \vee \\ \exists e', d \in \mathcal{E}_k \cdot p = obl(e', d, v) \wedge e' \in OC(S,e) & \vee \\ \exists e', d \in \mathcal{E}_k \cdot p = obl(e', d, v) \wedge d \in OC(S,e) & \vee \\ \exists e' \in OC(S,e) \cdot e' \in \mathcal{E}^k_{\times} & \} \end{array} \tag{A.15}$$

The first case deals with termination of fluents as defined by the consequence function $\mathcal{C}_k$. The second case terminates obligations that have been fulfilled. The third case deals with

obligations that have been violated (i.e. the deadline has expired). The final case terminates all fluents in the event of dissolution (note that this includes *live* if the institution is active). For termination, it is not necessary to address the case of the institution being inactive explicitly. If the institution is inactive, then by the definition of $INIT$, no fluents may be initiated. Therefore S is empty and so there is nothing to terminate. Note, however, that the definition of $TERM$ copes with this case since the fluents terminated are those of S which meet some conditions. When S is empty, nothing can be terminated.

A.2 InstAL

This description of the InstAL language is an overview of that given by Cliffe (2007) who introduced the language and (Cliffe et al., 2006) which presents a summary. Together these sources represent the most complete specification of InstAL.

An InstAL specification begins with the declaration of the name of the institution being modelled:

```
institution inst_name;
```

InstAL's type system is fairy simple – it allows for types to be declared (named) but the domains of the types do not form part of the specification. Types are declared as follows:

```
type Agent;
```

We can then declare in external files the domains of these types such as:

```
Agent: a1 a2 a3...
```

In addition to any user defined types, InstAL defines three standard types:

`Fluent:` The set of all grounded fluent literals that appear in the specification.

`Event:` The set of all event literals that appear in the specification.

`Inst:` The set of all unique institution names in a multi-institution specification. In the case where a single institution is being modelled, then this is a singleton set.

Domain fluents ($\mathcal{D}_i$ in the model) are declared with the types of any parameters as follows:

```
fluent fname1(Type1, Type2, ...);
fluent fname2;
```

In addition, institutional power, permission and obligation are modelled by the following implicitly defined fluents:

```
pow(Event);
perm(Event);
obl(Event, Event, Event);
```

InstAL allows the definition of *static properties* which are (semantically) fluents declared when an institution is specified that cannot be changed by the rules of the institution. For example, to define a static property corresponding to a relationship between agents (assuming definition of the agent type):

```
static father(Agent, Agent);
father(agent1, agent2).
```

Static properties are used to affect the way the specification is grounded. They cannot be used in the head of causal rules (see below).

Events are declared as an event type from the following list followed by the `event` keyword and then a unique event name (a lowercase identifier) and the types of parameters (if any) as with fluents. The event types (and how they correspond to the formal model) are:

`create`**:** Creation events in $\mathcal{E}_+$

`exogenous`**:** Observable/exogenous events in $\mathcal{E}_{obs}$

`inst`**:** Institutional events in $\mathcal{E}_{inst}$

`violation`**:** Violation events in $\mathcal{E}_{viol}$

`dest`**:** Dissolution events in $\mathcal{E}_\times$

A *fluent literal* or an *event literal* is a fluent or an event with all arguments replaced by literals from the domains of the argument types. For example if we have the type `Agent` then for the following fluent and event:

```
fluent buyer(Agent);
exogenous event sell(Agent, Agent);
```

Then (for literals `a1` and `a2` of type `Agent`) the following are a fluent and an event literal respectively:

```
buyer(a1);
sellto(a2, a1);
```

Fluent expressions may be used to qualify the applicability of rules. These are a comma-separated list of fluent literals (representing a conjunction) such as:

```
f1, not f2(a), f3(b,c)
```

The set of all possible fluent expressions in a specification corresponds to the state of states of the model ($\mathcal{X}_i$). *Static expressions* do not reference fluent literals.

Variables in InstAL are denoted by capitalised strings or "_" the special unbound variable. The static expressions `A=B` and `A!=B` can be used to require equality or inequality of variables (respectively) in a rule.

The initial state of an institution (Δ_i) is determined by *initial rules* of the form:

```
iniitially f1, f2(A), f3(A,B), ... if A!=B, ... ;
```

An initial rules is defined as the keyword `initially` followed by one ore more fluents (in a comma-separated list) followed (optionally) by the `if` keyword and a condition of one or more static expressions.

Causal rules correspond to the consequence function $\mathcal{C}_i$ of the formal model. These are a single event (the *trigger event*) followed by an operator (`initiates` or `terminates`) followed by one or more fluents that are initiated/terminated by the rule followed by (optionally) the `if` keyword then one or more fluent/static expressions that identify a *condition* on the rule being triggered. For example:

```
sell(A, O) terminates owns(A, O) if pow(sell(A, O));
buy(B, O) initiates owns(B, O);
```

A *generation rule* consists of a trigger event, the `generates` keyword, one or more events and optionally a condition of one or more fluent/static expressions. For example:

```
sellto(A, B) generates transaction(A, B) if A != B;
```

Multi-Institution Specifications

The specification of a multi-institution system begins with the name of the system:

```
multi mname;
```

Then *envioronment events* are defined (the set $\mathcal{E}^{\mathcal{M}}_{env}$ in the model). These are as with other event definitions but using the `env` event type. The types used as parameters of environment events are taken from the individual institutions in the system.

The last part of the system specification is generation of events which occur accross institutions – i.e. for when events in one institution trigger events in one or more others. The

syntax of this is exactly the same as for generation rules in a single institution, the events involved are simply drawn from all of the institutions in the system.

A single InstAL specification describes either one institution or one multi-institution system. As previously mentioned, we also have a *domain definition* identifying constants of each type in the institution(s).

Appendix B

Testing Plans

B.1 Release Testing

In this section, we present the results of realease testing. This was a black-box testing process. After each increment was implemented, the tests for all earlier increments were repeated as regression tests. These are not shown in the testing plan unless there was a change to the testing procedure. No black-box testing was performed for *InstEdit 0.4* (see Section B.2 for white-box testing performed instead). We need not test every error condition since we are not testing the external tools here, only that *InstEdit* can call them correctly.

Release	Test	Req.	Description	Expected Result	Actual Result	Pass?
0.1	1	**F1.1**, **F1.3**	Edit a file in *InstEdit*, save it and open in a text editor	Changes visible	Changes visible	Yes
0.1	2	**F1.2**	Open a file	Contents displayed	Contents displayed	Yes
0.2	3	**F3.1**	Translate valid institution	ASP shown	ASP shown	Yes
0.2	4	**F3.1**	Translate invalid institution	Error message from `genasp` shown	Error shown	Yes
0.2	5	**F3.1**	Request translate for invalid file path	Error message shown	Error shown	Yes
0.3	6	**F3.2**	Generate “abc” time instants	Error message shown	Nothing happens	No
0.3	7	**F3.2**	Generate 5 time instants	5 instants generated	5 instants generated	Yes
0.3	8	**F3.1**, **F3.2**, **F3.3**, **F3.4**	Solve a project with no institutions	Error message	Error message	Yes
0.3	9	**F3.1**, **F3.2**, **F3.3**, **F3.4**	Solve a project with 1 institution but without output location specified	Error message	Output written to working directory	No
0.3	10	**F3.1**, **F3.2**, **F3.3**, **F3.4**	Solve a project with one institution	Translation successful	Translation successful	Yes
0.3	11	**F3.1**, **F3.2**, **F3.3**, **F3.4**	Solve a project with 2 institutions	Translation successful	Translation successful	Yes
0.3	12	**F3.1**, **F3.2**, **F3.3**, **F3.4**	Solve a project with 2 institutions and a domain definition	Translation successful	Translation successful	Yes
0.3	13	**F3.1**, **F3.2**, **F3.3**, **F3.4**	Solve a project with 2 institutions, domain definition and multi-institution	Translation successful	Translation successful	Yes

Table B.1: Release testing.

Release	Test	Req.	Description	Expected Result	Actual Result	Pass?
0.5	14	**F3.7**	Solve a project with a query	Translation successful and traces reflect query	Translation successful and traces reflect query	Yes
0.5	15	**F3.6**	Launch *InstViz* on answer sets	*InstViz* opens	*InstViz*opens	Yes
0.5	16	**F3.6**	Launch *InstViz* before computing answer sets	Error message	Not possible: must select answer sets to visualise	Yes
0.5	17	**F3.5**	Generate graph from answer sets	PostScript graph produced	PostScript graph produced	Yes
0.5	18	**F3.5**	Generate graph before computing answer sets	Error message	Not possible: must select answer sets to visualise	Yes

Table B.2: Release testing (continued).

B.2 Component Testing

Due to the complexity of syntax highlighting, it was decided to perform white-box component testing on this part of the system. Examining the code revealed the following cases:

1. Keyword at line start. A line begins with the keyword "institution"; there is text after keyword.
    - **Expected result:** Keyword highlighted.
    - **Actual result:** Keyword highlighted.
    - **Pass/fail:** Pass.
2. Keyword in middle of line. A line contains the keyword "event"; there is text either side of the keyword.
    - **Expected result:** Keyword highlighted.
    - **Actual result:** Keyword highlighted.
    - **Pass/fail:** Pass.
3. Keyword at line end. A line ends with the keyword "institution"; there is text before keyword.
    - **Expected result:** Keyword highlighted.
    - **Actual result:** Keyword highlighted.
    - **Pass/fail:** Pass.
4. Comment at line start. A line starts with "%".
    - **Expected result:** Whole line highlighted as comment.
    - **Actual result:** Whole line highlighted as comment.
    - **Pass/fail:** Pass.
5. Comment in middle of line. Comment symbol typed in middle of a line of text.
    - **Expected result:** Comment symbol and all following text on that one line (only) highlighted as comment.
    - **Actual result:** Comment symbol and all following text on that one line (only) highlighted as comment.
    - **Pass/fail:** Pass.
6. Comment at line end. A lines ends with "%".
    - **Expected result:** Comment symbol only highlighted as keyword.
    - **Actual result:** Comment symbol only highlighted as keyword.

- **Pass/fail:** Pass.

7. Keyword in comment. Type a keyword after comment symbol.
    - **Expected result:** Keyword highlighted as comment.
    - **Actual result:** Keyword highlighted as comment.
    - **Pass/fail:** Pass.
8. Break a keyword onto multiple lines by pressing return part-way through keyword.
    - **Expected result:** Both parts of keyword are unhighlighted.
    - **Actual result:** Part of keyword before return is unhighlighted. Part following return (i.e. moved on to next line) remains highlighted.
    - **Pass/fail:** Fail.
9. Paste multiple-lines of text into document: "`inst event e; \n % comment`".
    - **Expected result:** Keywords "inst" and "event" highlighted. Comment highlighted.
    - **Actual result:** Keywords "inst" and "event" highlighted. Comment not highlighted.
    - **Pass/fail:** Fail.
10. Change application settings. Alter the colour of comment highlighting. (There is no need to check changing keywords since whenever settings change, same code is called regardless of what changed.)
    - **Expected result:** Comments in open documents rehighlighted to new colour.
    - **Actual result:** Comments in open documents rehighlighted to new colour.
    - **Pass/fail:** Pass.
11. Remove path to keywords file from application settings.
    - **Expected result:** Keywords in all open documents lose their highlighting. (The application is able to cope without having keywords defined.)
    - **Actual result:** Keywords in all open documents lose their highlighting.
    - **Pass/fail:** Pass.

B.3 Testing the InstQL Translator

In order to test the InstQL translator, it was used to test the sample queries from section 4.3 (p59). This section gives the output from the translator for these queries. It can be verified that this output matches the correct translation of these queries into ASP described in section 4.3 (up to the names generated during the translation).

(Note that due to a mistake in the InstAL tools, "`occurred`" is spelt "`occured`". For compatibility with InstAL, this is replicated in the InstQL translator.)

For query (IQ1) the output was:

```
bad :- occured(badgov, I7), event(badgov), instant(I7).
c0  :- not bad.
    :- not c0.
```

For query (IQ2) the output was:

```
c1 :- holdsat(conflict, I15), ifluent(conflict), instant(I15).
   :- not c1.
```

For query (IQ3) the output was:

```
c2 :- occured(desdl, I23), event(desdl), holdsat(conflict, I23),
      ifluent(conflict), instant(I23).
   :- not c2.
```

For query (IQ4) the output was:

```
startstate(F) :- holdsat(F, I33), ifluent(F), instant(I33),
                 occured(createdar, I35), event(createdar),
                 instant(I35), after(I33, I35, 1).
```

For query (IQ5) the output was:

```
startstate(F)   :- holdsat(F, I45), ifluent(F), instant(I45),
                   occured(createdar, I47), event(createdar),
                   instant(I47), after(I45, I47, 1).
restartstate(F) :- holdsat(F, I57), ifluent(F), instant(I57),
                   occured(desdl, I59), event(desdl),
                   holdsat(conflict, I59), ifluent(conflict),
                   instant(I59), after(I57, I59, 1).
missing(F)      :- startstate(F), not restartstate(F).
added(F)        :- restartstate(F), not startstate(F).
c3              :- missing(F).
c3              :- added(F).
                :- not c3.
```

Appendix C

Source Code Listings

Full code source code is provided by electronic submission and on CD. We present here only a subset of the code written.

For *InstEdit*, we include the syntax highlighter, `InstEditPanel` (the superclass for all components that are displayed in tabs) and the classes to translate data. `InstalBuilder` provides the interface between the GUI and translators sub-systems and the remaining classes form the translators sub-system.

For InstQL, we give the back end of the InstQL translator.

C.1 *InstEdit*

C.1.1 SyntaxHighlighter.java

```java
package lch21.instal.gui.syntax;


import lch21.instal.gui.TextPanePanel;
import lch21.instal.language.Keywords;
import lch21.instal.language.LanguageConstants;

import java.util.ArrayList;
import java.util.Queue;
import javax.swing.text.*;


/**
 * A new SyntaxHighlighter is started for each
      TextPanePanel to monitor for
 * changes in the underlying Document (by way of
      PendingHighlights) and
 * respond by performing syntax highlighting.
 * @author Luke Hopton
 */
public class SyntaxHighlighter extends Thread {
    private TextPanePanel parent;
    private HighlightStyles styles;

    /**
     * Creates a new SyntaxHighlighter to highlight the
          document displayed
     * in the specified panel.
     * @param parent Panel that this highlighter works
          on.
     */
    public SyntaxHighlighter(TextPanePanel parent) {
        this.parent = parent;
        styles = new HighlightStyles();
    }

    /**
     * Starts the syntax highlighting process for the
          parent panel.
     * The highlighter keeps highlighting until it is
          interrupted,
     * at which point it exits.
     */
    public void run() {
        Queue<PendingHighlight> jobs =
            parent.getHighlightList();

        PendingHighlight ph;

        // keep going until we are interrupted
        while (true) {
            // do all jobs in queue
            while ((ph = jobs.poll()) != null) {
                highlight(ph); // do highlighting
            }

            try {
                // wait for more jobs
                synchronized (parent) {
                    parent.wait();
                }
            } catch (InterruptedException ie) {
                // when the thread is interrupted, it
                    should cease execution
                return;
            }
        }
    }

    /**
     * Performs the syntax highlighting for a specific
          job.
     * @param job Information on the highlighting task
          to be performed.
     */
    private void highlight(PendingHighlight job) {
        StyledDocument doc = (StyledDocument)
            job.getDocument();

        int offset = job.getOffset();
        int jobStart = getLineStart(doc, offset);
        int jobLength = job.getLength();
        int jobEnd = getLineEnd(doc, offset + jobLength);
```

```
    int i = jobStart;

    // highlight one line at a time
    while (i < jobEnd && i >= 0) {
        highlightLine(doc, i);
        i = getLineEnd(doc, i);
        i = getLineStart(doc, ++i);
    }
}

/**
 * Highlights a single line of a document.
 * @oaram doc Document to highlight in.
 * @param start Start index of line within doc.
 */
private void highlightLine(StyledDocument doc, int
    start) {
    int end = getLineEnd(doc, start);
    int length = end - start;

    if (length <= 0) {
        return;
    }
    // initially, highlight whole line as normal
        text to fix some
    // issues with inserts picking up existing styles
    SimpleAttributeSet sas = styles.getStyle(
            Keywords.NONE);

    doc.setCharacterAttributes(start, length, sas,
        true);

    String s;

    try {
        s = doc.getText(start, length);
    } catch (BadLocationException ble) {
        System.err.println(
                "Bad location: start=" + start
                + ", lengtht=" + length);
        return;
    }
    // check for comments
    int cmt = s.indexOf(LanguageConstants.COMMENT);

    if (cmt != -1) {
        int commentStart = start + cmt;
        int commentEnd = getLineEnd(doc,
            commentStart);
        int commentLength = commentEnd -
            commentStart;
        SimpleAttributeSet cmtAttr =
            styles.getCommentStyle();

        doc.setCharacterAttributes(commentStart,
                commentLength, cmtAttr, true);
    }

    // highlight keywords up to comment (if any)
    StringBuilder buffer = new StringBuilder();

    for (int i = 0; i < s.length(); i++) {
        if (cmt != -1 && i >= cmt) {
            break;
        } // stop highlighting at comment

        String c = s.substring(i, i + 1);
        boolean delim = c.matches(
                LanguageConstants.REGEX_DELIMITER);

        if (!delim) {
            buffer.append(c);
        }

        if (delim && buffer.length() > 0) {
            int wordStart = start + i -
                buffer.length();

            highlightKeyword(doc, buffer.toString(),
                    wordStart);
            buffer.delete(0, buffer.length());
        }
        if (i == s.length() - 1 && buffer.length() >
            0) {
            int wordStart = start + i -
                buffer.length()
                    + 1;

            highlightKeyword(doc, buffer.toString(),
                    wordStart);
```

```
            buffer.delete(0, buffer.length());
        }
    }
}

/**
 * Highlights a keyword in a styled document.
 * @param doc Document to highlight in.
 * @param s Keyword to highlight.
 * @param end Start index of s within doc.
 */
private void highlightKeyword(StyledDocument doc,
    String s, int start) {
    int type = Keywords.typeOf(s);
    SimpleAttributeSet keyAttr =
        styles.getStyle(type);

    doc.setCharacterAttributes(start, s.length(),
            keyAttr, true);
}

/**
 * Returns the index within the document of the end
     of the line
 * (determined by a single line feed character
    <code>\n</code>)
 * containing <code>offset</code> or the end of the
     document
 * (whichever is encountered first).
 * @param doc Document to search.
 * @param offset Offset within document to begin
     search.
 * @return Index of end of line containing
    <code>offset</code>,
 * or <code>-1</code> if something went wrong.
 */
private int getLineEnd(Document doc, int offset) {
    int i = offset;

    if (i > doc.getLength()) {
        return -1;
    }

    String s = "";

    while (i <= doc.getLength() && !s.equals("\n")) {
        try {
            s = doc.getText(i++, 1);
        } catch (BadLocationException ble) {
            // TODO handle this nicely?
            System.err.println(
                    ble.getMessage() + " "
                    + ble.offsetRequested());
            return -1;
        }
    }

    return i - 1;
}

/**
 * Gets the index within the document of the start
     of the line containing
 * offset. Line start is determined by a single new
     line character.
 * @param doc Document to search.
 * @param offset Starting index.
 * @return Index of line start or 0 if document
     start found before line
 * start or <code>-1</code> if something went wrong.
 */
private int getLineStart(Document doc, int offset) {
    if (offset == 0) {
        return 0;
    }
    int i = offset + 1;
    String s = "";

    while (i > 0 && !s.equals("\n")) {
        try {
            i--;
            if (i == 0) {
                return 0;
            }
            s = doc.getText(i - 1, 1);
        } catch (BadLocationException ble) {
            System.err.println(
                    ble.getMessage() + " "
                    + ble.offsetRequested());
            return -1;
```

```
            }
        }

        return i;
    }
}
```

C.1.2 InstEditPanel.java

```
package lch21.instal.gui.components;


import javax.swing.JPanel;
import javax.swing.JOptionPane;


/**
 * An abstract implementation of a panel which should be
      the superclass
 * for any component displayed at "top level" within a
      tab on the main
 * form.
 * @see lch21.instal.gui.MainForm
 * @author Luke Hopton
 */
public abstract class InstEditPanel extends JPanel {

    /** Title for this panel. */
    protected String title = "";

    private boolean saveRequired = false;

    public InstEditPanel() {
        super();
    }

    /**
     * Creates a new panel that will use the specified
          LayoutManager.
     * @param lm Manager to use.
     */
    public InstEditPanel(java.awt.LayoutManager lm) {
        super(lm);
    }

/**
 * Gets the title for this panel.
 * @return Title.
 */
public String getTitle() {
    return title;
}

/**
 * Sets the title for this panel.
 * @param title New title.
 */
public void setTitle(String title) {
    this.title = title;
}

/**
 * Saves the contents of this panel as appropriate
      for panel type.
 */
public abstract void save();

/**
 * A "save as" operation. By default, this is an
      alias for
 * <code>save()</code> but subclasses may override
      this
 * method to provide one which prompts
 * for a new location prior to saving.
 */
public void saveAs() {
    save();
}

/**
 * Sets a flag to indicate that this panel has
 * unsave changes made.
 */
public void requireSave() {
    saveRequired = true;
}

/**
 * Sets a flag to indicate that no changes need
```

```
     * saving in this panel.
     */
    public void unrequireSave() {
        saveRequired = false;
    }

    /**
     * Checks to see if all changes made in this panel
         are saved.
     * @return True if all changes are saved or false if
         there are
     * changes that need saving.
     */
    public boolean isSaved() {
        return !saveRequired;
    }

    /**
     * Checks to see if this panel is unsaved and if so
         prompts for a save.
     * (For use on close etc when current document is
         not saved.)
     */
    public void saveCheck() {
        if (saveRequired) {
            String title = "Save changes?";
            String msg = "\"" + getTitle()
                    + "\" is unsaved.";

            msg += "\nDo you wish to save the changes?";
            int returnVal =
                JOptionPane.showConfirmDialog(
                    this, msg, title,
                    JOptionPane.YES_NO_OPTION);

            if (returnVal == JOptionPane.YES_OPTION) {
                save();
            }
        }
    }

    /**
     * Used when the panel is about to be closed. This
         allows subclasses to
     * safely exit and free up resources etc if
         necessary. The default
     * implementation provided here does nothing
         (subclasses that do not need
     * to do anything need not override the method).
     */
    public void close() {// nothing is done by default
    }

}
```

C.1.3 InstalBuilder.java

```
package lch21.instal.gui;


import javax.swing.JFileChooser;
import java.io.*;

import lch21.instal.app.Project;
import lch21.instal.gui.components.FileFilterFactory;
import lch21.instal.translators.*;


/**
 * Responsible for controlling the build process. That
     is, calling
 * the various translators.
 * @author Luke Hopton
 */
public class InstalBuilder {
    private Project project;
    private MainForm mf;

    /**
     * Creates a new InstalBuilder.
     * @param mf Parent form for this builder.
     */
    public InstalBuilder(MainForm mf) {
        this.mf = mf;
    }

    /**
     * Generates ASP from the main project of the parent
         form of
```

```java
 * this builder.
 */
public void instalToAsp() {
    Project p = mf.getMainProject();

    if (p == null) {
        return;
    }

    OutputPanel out = mf.outputTab(true);
    PrintStream ps = out.getPrintStream();
    InstalTranslator it = new InstalTranslator(ps);

    // get filepath to ASP generated
    String[] paths = it.translate(p);

    if (paths == null) {
        return;
    }

    // display ASP generated
    TextAreaPanel tap = new TextAreaPanel();

    if (tap.open(paths[0])) {
        mf.addTab(tap, false);
    }

}

/**
 * Generares ASP from the main project of the parent
 *   form of this
 * builder and then computes the answer sets.
 * @param n Number of time instances to generate.
 */
public void instalToAnswerSets(int n) {
    Project p = mf.getMainProject();

    if (p == null) {
        return;
    }

    OutputPanel out = mf.outputTab(true);
    PrintStream ps = out.getPrintStream();
    InstalTranslator it = new InstalTranslator(ps);

    // get filepath to ASP generated
    String[] paths = it.translate(p);

    if (paths == null) {
        return;
    }

    AnswerSetSolver ass = new AnswerSetSolver(ps, n);

    ass.translate(paths[0], paths[1],
            p.isMultiInstitution());
}

/**
 * Computes answer sets of the specified program.
 * @param inst Path to ASP institutional model.
 * @param qry Path to ASP query.
 * @param multi If true, treats program as
 *   multi-institution model.
 * @param n Number of time instancees to generate.
 */
public void aspToAnswerSets(String inst, String qry,
    boolean multi, int n) {
    OutputPanel out = mf.outputTab(true);
    PrintStream ps = out.getPrintStream();

    AnswerSetSolver ass = new AnswerSetSolver(ps, n);

    ass.translate(inst, qry, multi);
}

/**
 * Prompts for a filename and generates a graph from
 *   it.
 */
public void graph() {
    String file = getAnswerSets();

    if (file == null) {
        return;
    }

    OutputPanel out = mf.outputTab(true);
    PrintStream ps = out.getPrintStream();
```

```
        Visualiser v = new Visualiser(ps);

        v.createGraph(file);
    }

    /**
     * Prompts for a filename and displays it in InstViz.
     */
    public void instviz() {
        String file = getAnswerSets();

        if (file == null) {
            return;
        }

        OutputPanel out = mf.outputTab(true);
        PrintStream ps = out.getPrintStream();

        Visualiser v = new Visualiser(ps);

        v.launchInstViz(file);
    }

    /**
     * Gets path to a file containing answer sets by
         prompting user.
     * @return Absolute filepath or null if selection
         cancelled.
     */
    private String getAnswerSets() {
        JFileChooser chooser = new JFileChooser();

        chooser.setDialogTitle("Choose answer sets to
            show");
        chooser.addChoosableFileFilter(
                FileFilterFactory.answerSets());
        int returnVal = chooser.showDialog(mf, "Show");

        if (returnVal == JFileChooser.APPROVE_OPTION) {
            return
                chooser.getSelectedFile().getAbsolutePath();
        } else {
            return null;
        }
    }
}
```

C.1.4 Translator.java

```
package lch21.instal.translators;


import java.io.*;


/**
 * Translator is the superclass for all components which
     perform some
 * translation. It provides normal and error output
     streams. Either of these
 * can be safely initialised to null, which prevents
     output being printed.
 * Translator also provides utilities for handling
     process I/O.
 * @author Luke Hopton
 */
public abstract class Translator {

    /** Output stream to use. */
    protected PrintStream out = System.out;

    /** Error stream to use. */
    protected PrintStream err = System.err;

    /**
     * Creates a Translator that will use the specified
         stream
     * for all output.
     * @param out Output stream.
     */
    public Translator(PrintStream out) {
        this.out = out;
        this.err = out;
    }

    /**
     * Creates a Translator that will use the specified
         streams
```

```
 * for output.
 * @param out Output stream for this Translator.
 * @param err Error stream for this Translator.
 */
public Translator(PrintStream out, PrintStream err) {
    this.out = out;
    this.err = err;
}

/**
 * Prints the specified string to the output stream,
     then terminates the
 * line. If the specified output stream is null,
     does nothing (i.e. printing
 * is null-safe).
 * @param s String to print.
 */
public void printOut(String s) {
    if (out != null) {
        out.println(s);
    }
}

/**
 * Prints the specified string to the error stream,
     then terminates the
 * line. If the specified output stream is null,
     does nothing (i.e. printing
 * is null-safe).
 * @param s String to print.
 */
public void printErr(String s) {
    if (err != null) {
        err.println(s);
    }
}

/**
 * Prints to the output stream for this translator
     all output
 * from the specified process,
 * @param p Process to handle output from.
 */
protected void processOutput(Process p) {
    try {
        // read output from the program
        InputStream in = p.getInputStream();
        BufferedReader read = new BufferedReader(
                new InputStreamReader(in));
        String line;
        StringBuilder builder = new
            StringBuilder(100);

        while ((line = read.readLine()) != null) {
            builder.append(line);
            builder.append("\n");
        }

        read.close();

        if (builder.length() > 0) {
            printOut(
                    "InstEdit:␣Info:␣External␣
                        process␣output...");
            printOut(builder.toString());
            printOut(
                    "InstEdit:␣Info:␣....␣end␣of␣
                        process␣output.\n");
            out.flush();
        }
    } catch (IOException ioe) {
        handleError(ioe);
    }
}

/**
 * Prints to the error stream for this translator
     all errors
 * from the specified process,
 * @param p Process to handle errors from.
 */
protected void processErrors(Process p) {
    try {

        // read errors from the programm
        InputStream in = p.getErrorStream();

        BufferedReader read = new BufferedReader(
                new InputStreamReader(in));
```

```
        StringBuilder builder = new
            StringBuilder(1000);
        String line = read.readLine();

        while (line != null) {
            builder.append(line);
            builder.append("\n");
            line = read.readLine();
        }

        read.close();

        if (builder.length() > 0) {
            printErr(
                    "InstEdit: Info: External 
                        process errors...");
            printErr(builder.toString());
            printErr(
                    "InstEdit: Info: .... end of 
                        process errors.\n");
            err.flush();
        }

    } catch (IOException ioe) {
        handleError(ioe);
    }
}

/**
 * Writes data from one stream to another.
 * @param in Source stream.
 * @param out Destination stream.
 */
public static void pipe(InputStream in, OutputStream
    out) throws IOException {
    BufferedReader r = new BufferedReader(
            new InputStreamReader(in));
    BufferedWriter w = new BufferedWriter(
            new OutputStreamWriter(out));

    String line = r.readLine();

    while (line != null) {
        w.write(line);
        w.newLine();
        line = r.readLine();
    }

    r.close();
    w.flush();
    w.close();
}

/**
 * Handles an exception by reporting it on the error
 *    stream.
 */
protected void handleError(Exception e) {
    printErr("InstEdit: Error: " + e.getMessage());
}

/**
 * Prints a message to output to inform of
 *    completion of task.
 */
protected void completeMessage() {
    printOut("InstEdit: Info: Done!");
}

/**
 * Flushes both output streams.
 */
public void flush() {
    out.flush();
    err.flush();
}

/**
 * Writes the output from a process to file.
 */
public void outputToFile(Process p, String file) {
    try {
        // read output from the program
        InputStream in = p.getInputStream();
        BufferedReader read = new BufferedReader(
                new InputStreamReader(in));
        String line;
        BufferedWriter writer = new BufferedWriter(
                new FileWriter(file));
```

```
            while ((line = read.readLine()) != null) {
                writer.write(line);
                writer.newLine();
            }

            read.close();
            writer.flush();
            writer.close();

        } catch (IOException ioe) {
            handleError(ioe);
        }
    }

    /**
     * Strips <code>ext</code> from <code>file</code>,
        if present.
     * @param file Filename to strip from.
     * @param ext Extension to strip off.
     * @return Stripped filename.
     */
    public static String stripExtension(String file,
        String ext) {
        if (!ext.startsWith(".")) {
            ext = "." + ext;
        }

        if (!file.endsWith(ext)) {
            return file;
        }

        int idx = file.length() - ext.length();

        return file.substring(0, idx);
    }
}
```

C.1.5 InstalTranslator.java

```
package lch21.instal.translators;


import lch21.instal.app.ApplicationSettings;
import lch21.instal.app.InstalToolsManager;
import lch21.instal.app.Project;

import java.io.*;
import java.util.Vector;


/**
 * InstalTranslator provides the functionality to
     translate InstAL code
 * into ASP. That is, in this version it calls the
     genasp utility (by Owen
 * Cliffe) as an external process.
 * @author Luke Hopton
 */
public class InstalTranslator extends Translator {

    // Genasp useage:
    // genasp [-m multifile] [-d domain] [-o output]
        <filename1>...

    /**
     * Creates an InstalTranslator that will use the
         specified stream
     * for all output.
     * @param out Output stream.
     */
    public InstalTranslator(PrintStream out) {
        super(out);
    }

    /**
     * Creates an InstalTranslator that will use the
         specified streams
     * for output.
     * @param out Output stream.
     */
    public InstalTranslator(PrintStream out, PrintStream
        err) {
        super(out, err);
    }

    /**
     * Takes a project and translates the InstAL files
         to ASP.
     * @param p Project to translate.
```

```
 * @return Path where output was written or
 * <code>null</code> if something went wrong at
     index 0,
 * query path (or <code>null</code>) at index 1.
 */
public String[] translate(Project p) {
    String[] result = new String[2];
    String cmd = buildCommand(p);

    if (cmd == null) {
        printErr(
                "InstEdit: Error: Error translating 
                    project \""
                        + p.getName() + "\"");
        flush();
        return null;
    }

    printOut("InstEdit: Info: Executing command:");
    printOut(cmd + "\n");

    // call the command
    try {
        Process proc =
            Runtime.getRuntime().exec(cmd);

        // print output, then errors, then report
            that we are done
        processOutput(proc);
        processErrors(proc);
        completeMessage();
        flush();

        result[1] = translateQuery(p);
        result[0] = buildOutputPath(p);
        return result;
    } catch (IOException ioe) {
        printErr("InstEdit: Error: " +
            ioe.getMessage());
        return null;
    }

}

/**
 * Builds a commond to generate ASP using
     <code>genasp</code>
 * for the specified project.
 * @param p Project to build output for.
 * @return Commannd to translate project into ASP, or
 * <code>null</code> if an error occurs.
 */
private String buildCommand(Project p) {
    InstalToolsManager man =
        InstalToolsManager.getManager();

    if (man == null) {
        printErr(
                "InstEdit: Error: I don't know where 
                    InstAL is!");
        printErr(
                "InstEdit: Error: Check application 
                    settings (Edit -> Settings).");
        return null;
    }

    // quotes mess up on linux, but may be required
        on windows....
    // String cmd = genasp + " -o \"" + outfile +
        "\" \"" + infile + "\"";
    String cmd = man.genasp();

    String m = p.getMultiPath();

    if (m != null && !m.equals("")) {
        cmd += " -m " + m;
    }

    String d = p.getDomainPath();

    if (d != null && !d.equals("")) {
        cmd += " -d " + d;
    }

    String o = buildOutputPath(p);

    cmd += " -o " + o;

    Vector<String> insts = p.getInstitutions();
```

```
        if (insts == null || insts.size() == 0) {
            printErr(
                    "InstEdit:␣Error:␣Project␣must␣have␣
                        at␣least␣"
                            + "one␣institution␣to␣
                                translate.");
            return null;
        }

        for (String s : insts) {
            cmd += "␣" + s;
        }

        return cmd;

    }

    /**
     * Creates a filepath for output for the given
         project.
     * @param p Project being translated.
     */
    private String buildOutputPath(Project p) {
        String o = p.getOutputDirectory();

        if (o != null && !o.equals("")) {
            String sep =
                System.getProperty("file.separator");

            if (!o.endsWith(sep)) {
                o += sep;
            }
            // kill whitespace in name and add extension
            o += p.getName().replaceAll("[␣\t]", "") +
                ".lp";
            return o;

        } else {
            // no output option so just use project name
            return p.getName().replaceAll("[␣\t]", "")
                    + ".lp";
        }
    }

    /**
     * Translates the query for a project.
     * @param p Project to translate.
     * @return Path to translated query.
     */
    private String translateQuery(Project p) {
        String qry = p.getQueryPath();

        // no query to translate!
        if (qry == null || qry.equals("")) {
            return null;
        }

        InstqlTranslator trans = new
            InstqlTranslator(out,
                err);

        return trans.translate(qry);
    }

}
```

C.1.6 InstqlTranslator.java

```
package lch21.instal.translators;


import java.io.*;

import lch21.instal.app.InstqlToolsManager;


/**
 * Provides the abiluity to translate InstQL queries
     into ASP
 * through the external InstQL toolset.
 * @author Luke Hopton
 */
public class InstqlTranslator extends Translator {
    public InstqlTranslator(PrintStream out) {
        super(out);
    }

    public InstqlTranslator(PrintStream out, PrintStream
        err) {
```

```java
    super(out, err);
}

/**
 * Translates an InstQL query into ASP.
 * @param file Filepath to InstQL query to translate.
 * @return Filepath to translated ASP query.
 */
public String translate(String file) {
    InstqlToolsManager man =
        InstqlToolsManager.getManager();

    if (man == null) {
        printErr(
                "InstEdit: Error: I don't know where 
                    InstQL is!");
        printErr(
                "InstEdit: Error: Check application 
                    settings (Edit -> Settings).");
        flush();
        return null;
    }

    String cmd = man.instql();

    cmd += " " + file;

    printOut("InstEdit: Info: Executing command:");
    printOut(cmd + "\n");

    String outFile = stripExtension(file, "iql");

    outFile += ".query.lp";

    try {
        File wd = new File(man.getPath());
        Process p = Runtime.getRuntime().exec(cmd,
            null,
                wd);

        processErrors(p);
        outputToFile(p, outFile);
        flush();
        return outFile;
    } catch (IOException ioe) {
            handleError(ioe);
            flush();
            return null;
        }
    }
}
```

C.1.7 AnswerSetSolver.java

```java
package lch21.instal.translators;


import java.io.*;

import lch21.instal.app.ApplicationSettings;
import lch21.instal.app.InstalToolsManager;
import lch21.instal.app.InstqlToolsManager;


/**
 * Takes ASP generated by InstAL (<code>genasp</code>)
     and
 * solves it to produce an output answer set. Solving is
     done by
 * an external call to LParse/Smodels. Assumes LParse
     and Smodels
 * are in path.
 * @author Luke Hopton
 */
public class AnswerSetSolver extends Translator {

    /** How to call LParse. */
    private static final String LPARSE = "lparse";

    /** How to call Smodels. */
    private static final String SMODELS = "smodels";

    // default number of answer sets to return:
    private static final int NUM_SETS = 10;

    private int instances = 0;
    private InstalToolsManager man;
    private InstqlToolsManager iqlman;
```

```java
/**
 * Creates an AnswerSetSolver that will use the
     specified stream
 * for all output.
 * @param out Output stream.
 * @param n Number of time instances to generate.
 */
public AnswerSetSolver(PrintStream out, int n) {
    super(out);
    instances = n;
}

/**
 * Creates an AnswerSetSolver that will use the
     specified streams
 * for output.
 * @param out Output stream.
 * @param err Error stream.
 * @param n Number of time instances to generate.
 */
public AnswerSetSolver(PrintStream out, PrintStream
    err, int n) {
    super(out, err);
    instances = n;
}

/**
 * Uses LParse/Smodels to compute the answer sets
     for the
 * file specified. The file should contain text
     suitable as
 * input for LParse (i.e. a <code>.lp</code> file).
 * @param inst Institutional model to solve.
 * @param qry Query to use. Pass <code>null</code>
     to not use a query.
 * @param multi If true, computes for a multi
 * rather than single institution.
 */
public void translate(String inst, String qry,
    boolean multi) {
    man = InstalToolsManager.getManager();
    iqlman = InstqlToolsManager.getManager();
    if (man == null) {
        printErr(
                "InstEdit:␣Error:␣I␣don't␣know␣where␣
                    InstAL␣is!");
        flush();
        return;
    }

    if (iqlman == null) {
        printErr(
                "InstEdit:␣Error:␣I␣don't␣know␣where␣
                    InstQL␣is!");
        flush();
        return;
    }

    // ground program and collect output
    InputStream in = ground(inst, qry, multi);

    // solve ground program
    solve(inst, in);

    completeMessage();
    flush();
}

/**
 * Creates a command that will start a new process
     to ground
 * the specified file.
 * @param inst Path to the filename to solve.
 * @param qry Query to use, or null.
 * @return Command to solve file, or null if
     something went wrong
 * (e.g. path was null/empty).
 */
private String buildGroundCommand(String inst,
    String qry, boolean multi) {
    if (inst == null || inst.equals("")) {
        printErr(
                "InstEdit:␣Error:␣Filename␣required␣
                    to␣ground.");
        return null;
    }

    String base = getBasePrograms(multi);

    if (base == null) {
```

```
        printErr(
                "InstEdit: Error: Problem finding 
                    base programs.");
        return null;
    }
    String time = buildTimeName(inst);

    if (!generateTime(time)) {
        return null;
    }

    String cmd = LPARSE + " " + inst + " " + base +
        " "
            + time;

    // add query and InstQL library files if query
        is present
    if (qry != null) {
        String qryTime = iqlman.timeProgram();

        cmd += " " + qry + " " + qryTime;
        String qryPredef = iqlman.predefsProgram();

        cmd += " " + qryPredef;
    }

    return cmd;
}

/**
 * Builds a filename for a file containing time
     instants.
 * @param filepath Path to program that time will be
     associated with.
 * @return A new filepath where the associated time
     program should be
 * stored.
 */
private String buildTimeName(String filepath) {
    filepath = stripExtension(filepath, "lp");
    return filepath + ".time";
}

/**
 * Builds a filename for answer sets output from
     smodels.
 * @param filepath Path to build from.
 * @return New filepath to store answer sets.
 */
private String buildAnswerName(String filepath) {
    filepath = stripExtension(filepath, "lp");
    return filepath + ".smout";
}

/**
 * Runs a process to ground the program specified.
 * @param inst Location of institution program to
     ground.
 * @param qry Query to use or null.
 * @param multi Is the program a multi-institution
     model?
 * @return Input stream containing output from
     grounding,
 * or <code>null</code> if something went wrong.
 */
private InputStream ground(String inst, String qry,
    boolean multi) {
    String cmd = buildGroundCommand(inst, qry,
        multi);

    // make sure we have a command to execute
    if (cmd == null) {
        return null;
    }

    printOut("InstEdit: Info: Executing command:");
    printOut(cmd + "\n");

    try {
        Process p = Runtime.getRuntime().exec(cmd);

        /* In some cases lparse was not indicating
            end of stream
         * when there were no errors which causes
             this to block
         * with lparse in the "pipe_wait" state */
        // processErrors(p); // display errors but
            not output

        return p.getInputStream();
```

```java
    } catch (IOException ioe) {
        handleError(ioe);
        return null;
    }
}

/**
 * Solves an ASP program.
 * @param in Input for solving - output from
 *     grounding.
 * @param filepath Original ASP file being solved.
 * @return Filepath to output from solver.
 */
private String solve(String filepath, InputStream
    in) {
    if (in == null) {
        printErr(
                "InstEdit: No output from grounding
                    - nothing to solve!");
        return null;
    }

    try {
        String cmd = SMODELS + " "
                + getNumberAnswerSets();

        printOut("InstEdit: Info: Executing
            command:");
        printOut(cmd);

        Process p = Runtime.getRuntime().exec(cmd);
        OutputStream out = p.getOutputStream();

        // copy data from in to out
        pipe(in, out);

        String outFile = buildAnswerName(filepath);

        outputToFile(p, outFile);
        processErrors(p);
        printOut(
                "InstEdit: Info: Answer sets written
                    on "
                        + outFile);
        return outFile;
    } catch (IOException ioe) {
        handleError(ioe);
        return null;
    }
}

/**
 * Gets the number of answer sets to compute.
 */
private int getNumberAnswerSets() {
    ApplicationSettings app =
        ApplicationSettings.getSettings();
    String n = app.getProperty(
            ApplicationSettings.NUM_ANSWER_SETS);

    // use default if no property is specified
    if (n == null || n.trim().equals("")) {
        return NUM_SETS;
    }

    try {
        int i = Integer.parseInt(n);

        return i;
    } catch (NumberFormatException nfe) {
        return NUM_SETS;
    }
}

/**
 * Returns the base programs required to compute all
 *     traces of an
 * institution. This is different for single and
 *     multi institutions.
 * @param multi If true, gets the multi-institution
 *     programs.
 * @return String giving locations to required
 *     programs.
 */
private String getBasePrograms(boolean multi) {
    String b = man.baseProgram();

    String t;

    if (multi) {
```

```java
            t = man.multiTraceProgram();
        } else {
            t = man.traceProgram();
        }

        return b + " " + t;
}

/**
 * Generates time instances using the gentime
     utility by Owen Cliffe.
 * The number of instances to generate is specified
     in the constructor
 * for AnswerSetSolver.
 * @param n Number of time instants to generate.
 * @param filepath Filepath to save time to.
 * @return True if and only if time generated
     successfully.
 * @see #AnswerSetSolver(PrintStream, int)
 * @see #AnswerSetSolver(PrintStream, PrintStream,
     int)
 */
private boolean generateTime(String filepath) {
    String gentime = man.gentime();

    String cmd = gentime + " " + instances;

    try {
        printOut(
                "InstEdit: Info: Generating "
                        + instances
                        + " time instance(s).");
        Process p = Runtime.getRuntime().exec(cmd);
        InputStream in = p.getInputStream();
        FileOutputStream out = new FileOutputStream(
                filepath);

        // write to file
        pipe(in, out);

        processErrors(p);
        return true;
    } catch (IOException ioe) {
        handleError(ioe);
        return false;
            }

        }
}
```

C.1.8 Visualiser.java

```java
package lch21.instal.translators;


import java.io.*;

import lch21.instal.app.ApplicationSettings;
import lch21.instal.app.InstalToolsManager;


/**
 * Allows visualisation of answer sets using either
 * InstViz (part of the InstAL toolset) or
 * <a href="http://www.graphviz.org">Graphviz</a>.
 * @author Luke Hopton
 */
public class Visualiser extends Translator {
    private static final String DOT = "dot -Tps";

    private InstalToolsManager man = null;

    /**
     * Creates a new Visualiser that will display
         information and
     * errors on the specified stream.
     */
    public Visualiser(PrintStream out) {
        super(out);
    }

    /**
     * Creates a new Visualiser that will display
         information and
     * errors on the specified streams.
     */
    public Visualiser(PrintStream out, PrintStream err) {
        super(out, err);
    }
```

```java
/**
 * Generates a Graphviz graph (in PostScript format)
     from the
 * specified file.
 * @param filepath Path to file containing answer
     sets.
 */
public void createGraph(String filepath) {
    man = InstalToolsManager.getManager();
    if (man == null) {
        printErr(
                "InstEdit: Error: I don't know where 
                    InstAL is!");
        flush();
        return;
    }

    String dotFile = runGengraph(filepath);

    if (dotFile == null) {
        return;
    }

    runDot(dotFile);
    completeMessage();
    flush();
}

/**
 * Runs gengraph to generate a Graphviz file from
     the specified
 * file containing answer sets.
 */
private String runGengraph(String filepath) {
    String cmd = man.gengraph();
    String oFile = stripExtension(filepath, "smout")
            + ".dot";
    String out = "-o " + oFile;

    cmd += " " + out + " " + filepath;

    try {
        printOut("InstEdit: Info: Calling command:");
        printOut(cmd);

        Process p = Runtime.getRuntime().exec(cmd);

        processOutput(p);
        processErrors(p);
        flush();
        return oFile;
    } catch (IOException ioe) {
        handleError(ioe);
        flush();
        return null;
    }
}

/**
 * Runs Graphviz (<code>dot</code>) to generate a
     PostScript graph
 * from the specified file.
 * @param filepath Path to a <code>.dot</code> file.
 */
private void runDot(String filepath) {
    String cmd = DOT;
    String oFile = stripExtension(filepath, "dot")
            + ".ps";

    cmd += " " + filepath + " -o " + oFile;

    try {
        printOut("InstEdit: Info: Calling command:");
        printOut(cmd);

        Process p = Runtime.getRuntime().exec(cmd);

        processOutput(p);
        processErrors(p);

        printOut("InstEdit: Info: Wrote " + oFile);
    } catch (IOException ioe) {
        handleError(ioe);
    }
}

/**
 * Launches InstViz to display answer sets.
 * @param filepath File containing answer sets.
```

```
    */
    public void launchInstViz(String filepath) {
        man = InstalToolsManager.getManager();
        if (man == null) {
            printErr(
                    "InstEdit: Error: I don't know where 
                        InstAL is!");
            flush();
            return;
        }

        String wd = man.getPath();

        try {
            String cmd = man.instviz() + " " + filepath;

            printOut("InstEdit: Info: Calling command:");
            printOut(cmd);

            // use InstAL directory as working directory
            // (InstViz requires config files in that
                directory)
            Process p = Runtime.getRuntime().exec(cmd,
                null,
                    new File(wd));

            processOutput(p);
            processErrors(p);
            completeMessage();
        } catch (IOException ioe) {
            handleError(ioe);
        } finally {
            flush();
        }
    }

}
```

C.2 InstQL Translator

C.2.1 Parser.pm

```
# InstQL::Parser.pm by Luke Hopton
# A recursive descent parser for InstQL.

package InstQL::Parser;

# TODO Error handling is poor. Well formed InstQL will
    produce
# correct ASP but badly formed InstQL can be
    "unpredictable".

use strict;
use Parse::RecDescent;
use InstQL::Util;

my $grammar = q{

# note that integers are only used in after(n) statements
# these require n > 0, this is enforced syntactically
INTEGER: /[1-9][0-9]*/

NAME: /[a-z][a-zA-Z0-9_]*/ { $return = $item[1]; }
VARNAME : /[A-Z][a-zA-Z0-9_]*/ { $return = $item[1]; }

varlist: VARNAME "," varlist { $return = "$item[1], 
    $item[3]"; }
        | VARNAME { $return = $item[1]; }

params: "(" varlist ")" { $return = "($item[2])"; }

identifier: NAME params
                {
                        $return = $item[1].$item[2];
                }
        | NAME { $return = $item[1]; }


var_identifier: identifier | VARNAME

NEG: "not"

predicate:
        "happens(" <commit> var_identifier ")"
                {
                        my $instant =
                            InstQL::Util::instant();
```

```
                        # a typo in InstAL spells
                            occurred as "occured" - this
                            is
                        # reflected here
                        $return = "occured($item[3],␣
                            $instant),␣event($item[3])";
                }
        | "holds(" <commit> var_identifier ")"
                {
                        my $instant =
                            InstQL::Util::instant();
                        $return = "holdsat($item[3],␣
                            $instant),␣
                            ifluent($item[3])";
                }
        | <error?>

literal:
        NEG predicate { $return = "$item[1]␣$item[2]"; }
        | predicate

condition_literal: NEG identifier { $return = "$item[1]␣
    $item[2]"; }
        | identifier

while_expr:
        literal "while" while_expr
                {
                        $return = "$item[1],␣$item[3]";
                }
        | literal
                {
                        my $i = InstQL::Util::instant();
                        $return = "$item[1],␣
                            instant($i)";
                        InstQL::Util::pushInstant();
                        InstQL::Util::nextInstant();
                }

AFTER: "after(" INTEGER ")" {
    InstQL::Util::pushAfter($item[2]); }
        | "after" { InstQL::Util::pushAfter(0); }

after_expr:
        while_expr AFTER after_expr
                {
                        $return = "$item[1],␣$item[3]";
                }
        | while_expr
                {
                        my $a = InstQL::Util::after();
                        if($a) { $a = ",␣$a"; }
                        $return = $item[1]. $a;
                }

# each time we try and match a term, empty instant stack
term: { InstQL::Util::emptyStacks(); }
        after_expr { $return = $item[2]; }
        | condition_literal

conjunction:
        term "and" conjunction
                {
                        $return = "$item[1],␣$item[3]";
                }
        | term
                {
                        $return = "$item[1]";
                }

disjunction:
        conjunction "or" disjunction
                {
                        my $ref = $item[3];
                        my @array = @$ref;
                        unshift(@array, $item[1]);
                        $return = \@array;
                }
        | conjunction { my @conj = ($item[1]); $return =
            \@conj; }

condition_decl: "condition" <commit> identifier ":"
    disjunction ";"
                {
                        my $name = $item[3];
                        my $ref = $item[5];
                        my @conjunctions = @$ref;
                        my @result;
                        my $i = 0;
                        foreach(@conjunctions) {
```

```
                                @result[$i] = $name . " :- " . $_ . ".";
                                $i++;
                        }
                        $return = \@result;
                }

constraint: "constraint" disjunction ";"
                {
                        my $name =
                            InstQL::Util::condition();
                        my $ref = $item[2];
                        my @conjunctions = @$ref;
                        my @result;
                        my $i = 0;
                        foreach(@conjunctions) {
                                @result[$i] = $name . " :- " . $_ . ".";
                                $i++;
                        }
                        @result[$i] = ":- not $name.";
                        $return = \@result;
                }

statement: condition_decl | constraint

start: statement(s)

};

sub getParser {
        my $parser = Parse::RecDescent->new($grammar);
        return $parser;
}

1;
```

C.2.2 Util.pm

```
# InstQL::Util.pm by Luke Hopton
# Provides helper functions for the InstQL parser.

package InstQL::Util;

my $instant = 0;
my $condition = 0;

my @instants = ();
my @afters = ();

# Increments the instant counter.
sub nextInstant {
        $instant++;
}

# Gets the current instant variable.
sub instant {
        return "I".$instant;
}

# Gets the next condition name.
# Condition counter is incremented.
sub condition {
        return "c".$condition++;
}

# Pushes current instant onto instant stack.
sub pushInstant {
        push (@instants, instant());
}

# Empties the instant and after stacks.
sub emptyStacks {
        @instants = ();
        @afters = ();
}

# Builds an expression of the form "after(i0, i1), after(i1, i2), ..."
# from the instant stack and then empties the stack.
sub after {
        # last while expression will have matched twice,
        # once in "after ::= while AFTER after" and then once in
        # "after ::= while" so discard penultimate instant from
        # first (incorrect) match
        my $last = pop(@instants);
        pop(@instants);
```

```
        push(@instants, $last);

        my $n = @instants;
        if($n <= 1) {
                return "";
        }
        my $i;
        my $s = "";

        for($i = 0; $i < $n - 1; $i++) {
                $s .= "after($instants[$i], 
                    $instants[$i+1]";

                my $a = $afters[$i];

                # specify how far apart instants are
                if($a) {
                        $s .= ", $a)";
                } else {
                        $s .= ")";
                }
                if($i < $n-2) {
                        $s .= ", ";
                }
        }

        # empty instant stack
        @instants = ();
        @afters = ();

        return $s;
}

# Pushes a number onto the after values stack.
sub pushAfter {
        my $n = shift;
        push(@afters, $n);
}

1;
```

Appendix D

AQL: A Query Language for Action Domains

In this appendix we present a paper on AQL – a reinterpretation of InstQL as a query language for general action domains modelled in ASP and not just for institutions. Note that the following paper is not the exclusive work of the author of this report.

A$\mathcal{QL}$: A Query Language for Action Domains Modelled using Answer Set Programming

Luke Hopton, Owen Cliffe, Marina De Vos, and Julian Padget

Department of Computer Science
University of Bath, BATH BA2 7AY, UK
lch21@bath.ac.uk,{occ,mdv,jap}@cs.bath.ac.uk

Abstract. We present a new general purpose query and constraint language for reasoning about action domains that allows the processing of simultaneous events or actions, definition of conditions and reasoning about fluents and actions. A$\mathcal{QL}$ provides a simple declarative syntax for the specification of constraints on the histories (the combination of action traces and state transitions) within the modelled domain. Its semantics is provided by the translation of A$\mathcal{QL}$ queries into $AnsProlog$ and thus A$\mathcal{QL}$ acquires the benefits of the reasoning power provided by ASP: the answer sets of programs obtained from combining the query and the domain description correspond to those histories of the domain changing over time that satisfy the query. The result is a simple, high-level query and constraint language that builds on ASP and through the synthesis of features offers a more flexible, versatile and intuitive approach compared to existing languages. Furthermore, due to the use of $AnsProlog$, A$\mathcal{QL}$ can also be used to reason about partial histories.

1 Introduction

Action domains are a useful mechanism for modelling a variety of domains such as planning, protocol definition, normative frameworks [1, 5, 6]. Given an action description we can use existing computational techniques, such as Answer Set Programming for verifying or examining properties of these models. it is desirable that such a system should allow designers to specify model properties with a high degree of flexibility while offering qualitative properties of succinctness and human readability.

Action languages[3, 12] are a way of describing the effects formally using a fragment of natural language. Central to action languages is notion of a transition system: with every action (or the combined effects of simultaneous actions) the environment changes. Traditionally action languages are split into two distinct parts: an action description language and a query language. As the names indicate, the former is used the describe the effects of actions resulting in the definition of the transition system. The latter serves to query or reason about the underlying transition system described by the action language.

In this paper, we present a new action query and constraint language A$\mathcal{QL}$ whose semantics is provided by ASP. A$\mathcal{QL}$ can be used in two ways: as a tool to select certain paths in the transition system or to model-check a particular path. A$\mathcal{QL}$ extends existing query language to allow for simultaneous actions, the definition of conditions which can

then be used to create more complex queries. Furthermore, AQL does not rely on the use of any action description language but can be used on top of a $AnsProlog$ description or used in conjunction with any action language description that maps to $AnsProlog$.

The remainder of the paper is set out as follows: in §2 and §3 we provide some context on answer set programming and on action domains, before giving a functional description of the AQL language in §4. Subsequently, we illustrate its usage for reasoning about action domains in general (§5) and in the setting of a small case study (§6). We conclude in §7 with a discussion of related work and future developments.

2 Answer Set Programming

In *answer set programming* ([2]) a logic program is used to describe the requirements that must be fulfilled by the solutions of a certain problem. For the purposes of this paper, we provide a brief and informal overview.

Answer set semantics is a model-based semantics for normal logic programs. Following the notation of [2], we refer to the language over which the answer set semantics is defined as $AnsProlog$.

An $AnsProlog$ program consists of a set of rules of the form $a : -B, \mathbf{not}\ C$. with a being and atom and B, C being (possibly empty) sets of atoms. a is called the head of the rule, while $B \cup \mathbf{not}\ C$ is the body. The rule can be read as: "if we know all atoms in B and we do not know any atom in C, then we must know a". Rules with an empty body are called facts, as the head is always considered known. An interpretation is a truth assignment to all atoms in the program. Often only those literals that are considered true are mentioned, as all the other are false by default (negation as failure).

The semantics of programs without negation (effectively horn clauses) are simple and uncontroversial, the T_p (immediate consequence) operator is iterated until a fixed point it reached. The *Gelfond-Lifschitz* reduct is used to deal with negation as failure. This takes a candidate set and reduces the program by removing any rule that depends on the negation of an atom in the set and removing all remaining negated atoms. *Answer Sets* are candidate sets that are also models of the corresponding reduced programs. The uncertain nature of negation-as-failure gives rise to several answer sets, which are all solutions to the problem that has been modelled.

Algorithms and implementations for obtaining answer sets of logic programs are referred to as *answer set solvers*. Some of the most popular and widely used solvers are DLV[9], SMODELS[15] and CLASP[11].

3 Action Domains

In action domains[12, 10] we are interested in the effect of actions or events on the environment. This can be modelled as a state transition where each state is a set of fluents that are considered true. Fluents not mentioned are considered false with respect to that state. The effect of an action or, when modelling simultaneous events and actions, the combined effects of all events taking place at a certain time, moves the domain to a new state. When moving between states, fluents may be initiated or terminated. For

every set of actions/events and a given state, there will be exactly one corresponding state.

A very simple example is a light-switch domain. In one state, the light is on. Flicking the switch moves the domain to a state in which the light is off.

When reasoning about action domains, we are not necessarily interested the effects of just one action. More often, we want to see the result of a sequence of actions. When allowing for simultaneous actions or events, this could become a sequence of sets of actions, one set for every time instance. Such a sequence is referred to as a trace. With each trace we have a corresponding sequence of states. By interleaving a sequence of states and the corresponding trace, we obtain a history of our action domain.

Planning[19] is probably the best known example of action domains and the reasoning used in such environments. Traces here correspond to the sequence of actions the planner needs take in order to achieve a goal.

Action domains can be studied using a variety of formalisms. In this paper we are interested in the use of answer set programming as a way of representing and reasoning about action domains, their traces and histories. When modelled using ASP, histories are obtained as answer sets.

Although action domains can be directly modelled using ASP, often action languages, languages especially designed for action domains, are used to provide a useful abstraction for the designer. This allows the designer to focus on the specifics of the domain rather than having to specify concepts like inertia that are common to every domain. The action language description can then be translated to an $AnsProlog$ program.

A variety of $AnsProlog$ based action languages exist ranging from a general languages like A [12], C [13] and DLV-K [8] to domain-specific ones like Inst*AL* [4] for normative multi-agent systems or [7] which was designed for biological networks.

The action language aside, a given action domain could give rise to a vast number of valid traces and associated histories. Often not all of them are equally useful for the task at hand and selection criteria have to be applied. While this can be done by adding rules and constraints to the answer set program, we would like to offer the designer the same sort of level of abstraction as is provided by action languages. Action query languages, like the language A*QL* which we will introduce later, provide a formal natural language-style of specifying properties of histories. Just like action description languages, the semantics of the query language can be provided by ASP.

Traditionally action query languages provide a model-checking functionality: given a history and a formula, it can be verified whether the formula is satisfied by the history. Our query language, A*QL* can be used this way but, additionally, can be used to produce only those histories that satisfy the given query. In a way, the query act as constraint on the set of all histories. This is the reason why we refer to A*QL* as a query *and* constraint language. However, what we are doing is the same as conventional query languages. We first state assumptions about the types of histories in which we are interested - the should have and should not have properties and then encode the conditions which we want to show or disprove over those models. But instead of starting with a given history, we are able to generate all the histories that would satisfy the query.

<table>
<tr><th>Expression</th><th></th><th>Definition</th></tr>
<tr><td><variable></td><td>::=</td><td>[A-Z][a-zA-Z0-9_]*</td></tr>
<tr><td><variable_list></td><td>::=</td><td><variable> , <variable_list> | <variable></td></tr>
<tr><td><name></td><td>::=</td><td>[a-z][a-zA-Z0-9_]*</td></tr>
<tr><td><param_list></td><td>::=</td><td>(<variable_list>)</td></tr>
<tr><td><identifier></td><td>::=</td><td><name> <param_list> | <name></td></tr>
<tr><td><predicate></td><td>::=</td><td>happens(<identifier>) | holds(<identifier>)</td></tr>
<tr><td><literal></td><td>::=</td><td>not <predicate> | <predicate></td></tr>
<tr><td><while_expr></td><td>::=</td><td><literal> while <while_expr> | <literal></td></tr>
<tr><td><after></td><td>::=</td><td>after(<integer>) | after</td></tr>
<tr><td><after_expr></td><td>::=</td><td><while_expr> <after> <after_expr> |
<while_expr></td></tr>
<tr><td><condition_literal></td><td>::=</td><td>not <identifier> | <identifier></td></tr>
<tr><td><term></td><td>::=</td><td><after_expr> | <condition_literal></td></tr>
<tr><td><conjunction></td><td>::=</td><td><term> and <conjunction> | <term></td></tr>
<tr><td><disjunction></td><td>::=</td><td><term> or <disjunction> | <term></td></tr>
<tr><td><condition_decl></td><td>::=</td><td>condition <identifier> : <disjunction>; |
condition <identifier> : <conjunction>;</td></tr>
<tr><td><constraint></td><td>::=</td><td>constraint <disjunction> ; |
condition <identifier> : <conjunction>;</td></tr>
</table>

Table 1. InstQL Syntax

4 AQL

In this section we introduce the query language, AQL , that can be used directly with an $AnsProlog$ program representing the action domain or with any action language.

Given an action domain $\mathcal{M}$, we use $\mathcal{E}_{\mathcal{M}}$[1] to denote the set of all actions and events in the action domain $\mathcal{M}$ while $\mathcal{F}_{\mathcal{M}}$ is the set of all available fluents. Following convention, we assume that the truth of a fluent $F \in \mathcal{F}$ at a given state I is represented as $\texttt{holdsat}(\texttt{F}, \texttt{I})$, while an event or an action $E \in \mathcal{E}$ is modelled as $\texttt{occurred}(\texttt{E}, \texttt{I})$. We further assume that the state instances are modelled as $\texttt{instant}(\texttt{I})$ for each instance I. The ordering of instances is established by $\texttt{next}(\texttt{I1}, \texttt{I2})$, with the final instance defined as $\texttt{final}(\texttt{I})$.

AQL has two basic concepts: (i) *constraint:* an assertion of a property that must be satisfied by a valid trace, and (ii) *condition:* a specification of properties that can be imposed on a trace. Conditions can be declared in relation to other conditions and constraints can involve declared conditions. Table 1 summarises the syntax of the language, while the remainder of this section discusses in detail the elements of the language and their semantics.

Variables and Identifiers: As terminal symbols, AQL provides a definition of various types of names which are built up as follows:

```
<variable>::= [A-Z][a-zA-Z0-9_]*
<variable_list>}::= <variable> , <variable_list> | <variable>
<name> ::= [a-z][a-zA-Z0-9_]*
<param_list>} ::= (<variable_list> )
<identifier>} ::= <name> <param_list> | <name>
```

[1] When it is clear from the context we will not mention the action domain

The definition of a variable conforms to that of Lparse/Smodels [16]. An *identifier* is an arbitrary name which may have variable parameters, that enables the parameterisation of events and fluents.

Predicates: A*QL* provides two *predicates* that form the basis of all A*QL* queries. The first is `happens(Event)`, meaning that the specified event should occur at some point during the lifetime of the institution. The second is `holds(Fluent)`, which means that the specified fluent is true at any point during the lifetime of the institution. That is:

```
<predicate> ::= happens( <identifier> ) | holds(<identifier>)
```

where the *identifier* corresponds to an event e (in the first case) or a fluent f (in the second case). $\mathcal{M}$.

Negation (as failure) is provided by the unary operator `not`:

```
<literal> ::= not <predicate> | <predicate>
```

To construct complex queries, it is often easier to break them up in sub-queries, or in A*QL* terminology, sub-conditions. For example, suppose we have defined a condition called `my_cond` which specifies some desired property. We can then join this with other criteria e.g. "`my_cond and happens(e)`". Sub-conditions may be referenced within rules as *condition literals*:

```
<condition_literal> ::= not <identifier> | <identifier>
```

Note that this allows for parameterised conditions to be defined by the definition of an *identifier*.

Conditions: The building block of query conditions is the *term*:

```
<term> ::=  <after_expr> | <condition_literal>
```

The after expression also allows for the more simpler constructs of `<literal>` and `<while_expr>`. *Terms* may be grouped and connected by the connectives `and` and `or` which provide logical conjunction and disjunction.

```
<conjunction> ::= <term> and <conjunction> | <term>
<disjunction> ::= <term> or <disjunction>  | <term>
```

On its own, this does not allow us create arbitrary combinations of *predicates* and named conditions and the logical operators `and`, `or`, `not`. To do so we need to be able declare conditions:

```
<condition_decl> ::=   condition <identifier> : <disjunction>
                     | condition <identifier> : <conjunction>;
```

This construction defines a `condition` with the specified name to have a value equal to the specified `disjunction` or `conjunction`. This allows the `condition name` to be used as a `condition_literal`.

Constraints: Constraints specify properties of the trace which must be true:

```
<constraint> ::= constraint <disjunction> | <conjunction> ;
```

For example, consider the following A*QL* query:

```
constraint happens(e);
```

This indicates that only traces in which event `e` occurs at some point should be considered. That is, we are only interested in those traces in which `e` occurs.

Example queries: To illustrate how this language is used to form queries, consider a simple light bulb action domain. The fluent `on` is true when the bulb is on. The event `switch` turns the light on or off. We can require that at some point the light is on:

```
constraint holds(on);
```

We can require that the light is never on:

```
condition light_on: holds(on);
constraint not light_on;
```

There is some subtlety here in that `light_on` is true if at any instant `on` is true. Therefore, if `light_on` is not true, there cannot be an instant at which `on` was true. And what if the bulb is broken – the switch is pressed but the light never comes on? This can be expressed as:

```
constraint not light_on and happens(switch);
```

Using condition names, we can create arbitrary logical expressions. The statement that event $e1$ and either event $e2$ or $e3$ should occur can be expressed as follow:

```
condition disj: happens(e2) or happens(e3);
condition conj: happens(e1) and disj;
```

Query Semantics: The semantics of an AQL query is defined by the translation function T which translates AQL into $AnsProlog$. This function takes a fragment of AQL and generates a set of (partial) $AnsProlog$ rules. Typically, this set is a singleton; only expressions involving disjunctions generate more than one rule. The semantics of predicates are defined as follows:

$$T(\texttt{happens(e)}) = \texttt{occurred(e, I)}$$
$$T(\texttt{holds(f)}) = \texttt{holdsat(f, I)}$$

For a literal of the form `not P` (where `P` is a predicate) the semantics are:

$$T(\texttt{not } P) = \texttt{not } T(\texttt{P})$$

while for a condition literal they are:

$$T(\texttt{conditionName}) = \texttt{conditionName}$$
$$T(\texttt{not conditionName}) = \texttt{not conditionName}$$

and a conjunction of terms is:

$$T(t_1 \texttt{ and } t_2 \texttt{ and} \cdots \texttt{ and } t_n) = T(t_1), T(t_2), \ldots, T(t_n)$$

A disjunction translates to more than one rule. However, this is defined slightly differently depending on whether it is within a condition declaration or a constraint.

$$\begin{aligned}
T(\texttt{condition conditionName :}\ & t_1 \texttt{ or } t_2 \texttt{ or } \cdots \texttt{or } t_n;) = \\
& \{\texttt{conditionName} \leftarrow T(t_i).\ |\ 1 \leq i \leq n\} \\
T(\texttt{constraint}\ & t_1 \texttt{ or } t_2 \texttt{ or } \cdots \texttt{or } t_n;) = \\
& \{\texttt{newName} \leftarrow T(t_i).\ |\ 1 \leq i \leq n\} \cup \\
& \{\bot \leftarrow \texttt{not newName.}\}
\end{aligned}$$

Note that the $AnsProlog$ term `newName` denotes any identifier that is unique within the $AnsProlog$ program that is the combination of the query and the action program. In addition, each time instant `I` generated in the translation of a predicate represents a name for a time instant that is unique within the A*QL* query. Recall that a condition name may be parameterised: since an A*QL* variable translates to a variable in Smodels, no additional machinery is required. For example, the condition "`condition ever(E): happens(E);`" (which just defines an alias for `happens`) is translated to "`ever(E) ← occurred(E, I), instant(I), event(E).`".

Concurrent Events and Fluents: We may wish to specify queries of the form "X and Y happen at the same time". That is, we may wish to talk about events occurring at the same time as one or more fluents are true, simultaneous occurrence of events or combinations of fluents being simultaneously true (and/or false). For this situation, A*QL* has the keyword `while` to indicate that literals are true *simultaneously*. Such `while` expressions are only defined over literals constructed from predicates (that is, `happens` and `holds`) and not condition literals involving condition names. This is because, for example, a condition may mean "event `e` never occurs". It does not make sense to define an expression that says something like "event `d` occurs at the same time as (`e` never occurs)". A while expression is defined as follows:

```
<while_expr> ::= <literal> while <while_expr> | <literal>
```

The `while`-operator has higher precedence than `and` and `or`.

Returning to the light bulb example, we can now specify that we want only traces where the light was turned off at some point:

```
constraint happens(switch) while holds(on);
```

Or that at some point the light was turned left on:

```
constraint holds(on) while not happens(switch);
```

The semantics for `while` is;

$$T(L_1 \texttt{ while } L_2 \texttt{ while } \cdots \texttt{ while } L_n) = T(L_1), T(L_2), \ldots, T(L_n), \texttt{instant}(\texttt{I})$$

Event and Fluent Ordering: The language allows for the expression of orderings over events. This is done with the `after` keyword. This allows statements of the form:

```
holds(f1) while not holds(f2) after happens(e1)
                              after happens(e2)
```

This should be read as: (i) at some time instant k the event `e2` occurs (ii) at some other time instant j the event `e1` occurs (iii) at some other time instant i the fluent `f1` is true but the fluent `f2` is not true (iv) these time instants are ordered such that $i > j > k$ (that is, k is the earliest time instant) However, in some cases we need to say not only that a given literal holds after some other literal, but that this is precisely one time instant later. Rather than just providing the facility to specify a literal occurs/holds in the next time instant, this is generalised to say that a literal holds n time instants after another. That is, for a fluent that does (not) hold at time instant t_i or an event that occurs between t_i and t_{i+1}, we can talk about literals that hold at t_{i+n} or occur between t_{i+n} and t_{i+n+1}. The syntax of an `after` expression is:

```
<after> ::= after | after( <integer> )
<after_expr> ::= <while_expr> <after> <after_expr> |
                 <while_expr>
```

An `after` expression may contain only the `after` operator or the `after(n)` operator, depending on how precisely the gap between the two operands is to be specified.

Once again returning to the light bulb example, we can now specify a query which requires the light to be switched twice (or more):

```
constraint happens(switch) after happens(switch);
```

Or that once that light has is on, it cannot be switched off again:

```
condition switch_off: happens(switch) after holds(on);
constraint not switch_off;
```

We give the semantics for the binary operator `after(n)`. This can easily be generalised for after expressions built of sequences of `after(n)` operators mixed with `after` operators.

$$T(W_i \; \mathtt{after(n)} \; W_j) = T(W_i),\; T(W_j),\; \mathtt{after(t_i, t_j, n)}$$

Where $\mathtt{t_i}$ and $\mathtt{t_j}$ are the time instants generated by W_i and W_j respectively. This is defined such that we require $\mathtt{n} > 0$.

We now provide a concrete example of the translation of an `after` expression to illustrate this process:

$$\begin{aligned} &T\;(\mathtt{happens(e)\;while\;holds(f)\;after\;happens(d)\;after(3)\;holds(g)}) = \\ &\quad \mathtt{occurred(e,\;t_i),\;event(e),\;holdsat(f,\;t_i),\;ifluent(f),} \\ &\quad \mathtt{instant(t_i),\;occurred(d,\;t_j),\;event(d),\;instant(t_j),} \\ &\quad \mathtt{holdsat(g,\;t_k),\;ifluent(g),\;instant(t_k),} \\ &\quad \mathtt{after(t_i,\;t_j),\;after(t_j,\;t_k,3).} \end{aligned}$$

5 Reasoning with A*QL*

Following the description of A*QL* in the preceding section, we now illustrate how it can be used to perform three common tasks[18] in computational reasoning: prediction, postdiction and planning.

Prediction is the problem of ascertaining the resulting state for a given (partial) sequence of actions and initial state. That is, suppose some transition system is in state S and a sequence $A = a_1, \ldots, a_n$ of actions occurs. Then the prediction problem (S, A) is to decide the set of states $\{S'\}$ which may result. Postdiction is the converse problem: if a system is in state S' and we know that $A = a_1, \ldots, a_n$ have occurred, then the problem (A, S') is to decide the set $\{S\}$ of states that could have held before A. The planning problem (S, S') is to decide which sequence(s) of actions, $\{A\}$, will bring about state S' from state S.

Identifying States: A state is described by the set of fluents that are true $S = \{f_1, \ldots, f_n\}$ where f_i are the fluents. States containing or not containing given fluents may be identified in A*QL* using the `while` operator:

```
holds(f_1) while ... while holds(f_n) while
not holds(g_1) while ... while not holds(g_k)
```

where $\texttt{f}_{1...\texttt{k}}$ are fluents which must hold in the matched state and $\texttt{g}_{1...\texttt{k}}$ are those fluents that do not.

Describing Event Ordering: A sequence of events $E = e_1, \ldots, e_n$ may be encoded as an `after` expression. If we have complete information, then we know that e_1 occurred, then e_2 at the next time instant and so on up to e_n with no other events occurring in between. In this case, we can express E as follows:

```
happens(e_n) after(1) ... after(1) happens(e_1)
```

This can be generalised to the case where e_{i+1} occurs after e_i with some known number $k \geq 0$ of events happening in between:

```
happens(e_i+1) after(1) ... after(k+1) happens(e_i)
```

Alternatively if we do not know k (that is, we know that e_{i+1} happens later than e_i but zero or more events occur in between) we can express this as:

```
happens(e_i+1) after happens(e_i)
```

We can combine these cases throughout the formulation of E to represent the amount of information available.

The Prediction Problem: Given an initial state S and a sequence of events E, the prediction problem (S, E) can be expressed in A*QL* as:

```
constraint E after(1) S;
```

This query limits traces to those in which at some point S holds after which the events of E occur in sequence. The answer sets that satisfy this query will then contain the states $\{S'\}$.

The Postdiction Problem: Given a sequence of events E and a resulting state S', the postdiction problem (E, S') can be expressed as:

```
constraint S after(1) E;
```

This requires S to hold in the next instant following the final event of E.

The Planning Problem: Given a pair of states S and S' the planning problem (S, S') can be expressed in A*QL* as:

```
constraint S' after S;
```

This allows any non-empty sequence of events to bring about the transition from S to S'. If we want to consider plans of length k (i.e. $E = e_1, \ldots, e_k$) then we express this:

```
constraint S' after(k) S;
```

6 A Case Study

As a case study we will look a fragment of the Dutch auction protocol with only one round of bidding. In this protocol a single agent is assigned to the role of auctioneer, and one or more agents play the role of bidders. The purpose of the protocol as a whole is either to determine a winning bidder and a valuation for a particular item on sale, or to establish that no bidders wish to purchase the item. Consequently, conflict — where two bids are received "simultaneously" — is treated as an in-round state which takes the process back to the beginning. The protocol is summarised as follows:

1. Round starts: auctioneer selects a price for the item and informs each of the bidders present of the starting price. The auctioneer then waits for a given period of time for bidders to respond.
2. Bidding: upon receipt of the starting price, each bidder has the choice whether to send a message indicating their desire to bid on the item at that price or not.
3. Bid processing: at the end of the prescribed period of time, if the auctioneer has received a single bid from a given agent, then the auctioneer is obliged to inform each of the participating agents that this agent has won the auction.
4. No bids: if no bids are received at the end of the prescribed period of time, the auctioneer must inform each of the participants that the item has not been sold.
5. Multiple bids: if more than one bid was received then the auctioneer must inform every agent that a conflict has occurred.
6. Termination: the protocol completes when an announcement is made indicating that an item is sold or that no bids have been received.
7. Conflict resolution: in the case where a conflict occurs then the auctioneer must re-open the bidding and re-start the round in order to resolve the conflict.

Based on the protocol description above, the following agent actions are defined: the auctioneer announces a price to a given bidder (`annprice`), the bidder bids on the current item (`annbid`), the auctioneer announces a conflict to a given bidder (`annconf`) and the auctioneer announces that the item is sold (`annsold`) or not sold (`annunsold`) respectively.

In addition to the agent actions we also include a number of time-outs indicating the three external events—that are independent of agents' actions—that affect the protocol. For each time-out we define a corresponding protocol event suffixed by `dl` indicating a deadline in the protocol:

`priceto`, `pricedl`: A time-out indicating the deadline by which the auctioneer must have announced the initial price of the item on sale to all bidders.

`bidto`, `biddl`: A time-out indicating the expiration of the waiting period for the auctioneer to receive bids for the item.

`decto`, `decdl`: A time-out indicating the deadline by which the auctioneer must have announced the decision about the auction to all bidders

Protocols provide a precise description what participants are allowed and not allowed to do. When agents perform prohibited actions, the protocol will generate a violation event `viol(E)` indicating that an action `E` has occurred which was not permitted by the protocol. The protocol could then enforce penalties on the violating participant. When the auctioneer violates the protocol, an event `badgov` occurs and the auction dissolves.

At the beginning of the protocol and each time the bidding has to start all over again due to a conflict, the protocol will initialise the environment or state such that bidding can proceed according to the rules.

A simple query for verifying this protocol is to look at those traces in which the auctioneer violates the protocol. This can be expressed as follows

```
condition bad: happens(badgov);
constraint bad;
```

Alternatively, we could look at all the traces in which the protocol is never violated by one of the bidders.

```
condition bad: happens(viol(E));
constraint not bad;
```

We would also like to be able to find those traces where more than one bidder entered a bid, resulting in a conflict, before the end of bidding is announced. This can be expressed as follows:

```
constraint happens(decdl) while holds(conflict);
```

The use of parameterised conditions is illustrated in the following statement that enumerates all the fluents that are true when the protocol has just started, which is indicated by the occurrence of the event `createdar`:

```
condition startstate(F): holds(F) after(1) happens(createdar);
```

The following query can be used to verify the protocol. The protocol states that if more than one bidder bids for the good, the protocol needs to restart completely. This implies that all the fluents from the beginning of the protocol need to be reinstated and all others have to be terminated. The query checks if this has been done. If we still obtain a trace with this query we know something has gone wrong with design of the protocol.

```
condition startstate(F): holds(F) after(1) happens(createdar);
condition restartstate(F): holds(F) after(1) happens(decdl)
                           while holds(conflict);
condition missing(F): startstate(F) and not restartstate(F);
condition added(F): restartstate(F) and not startstate(F);
constraint missing(F) or added(F);
```

7 Discussion

In [12], the authors present three query languages: $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$. Queries expressed in those languages can also be expressed using A*QL* . The action query language $\mathcal{P}$ has only two constructs : `now L` and `necessarily F after A1, ..., An`, where `L` refers to a fluent or its negation, `F` is a fluent and where `Ai` are actions. These queries can be encoded in A*QL* using the techniques discussed in Section 5. `now L` can be written as `constraint happens(An) after(1) ... after(1) happens(A1) after(1) holds(L)` while `necessarily F after A1, ..., An` is expressed as `holds(F) after(1) happens(An) after(1) ... after(1) happens(A1)`. Similar techniques can be used for the query languages $\mathcal{Q}$ and $\mathcal{R}$. Given the event ordering technique used, we can assign specific times to each of the fluents.

While A*QL* can perform the same reasoning as those query languages, it can do much more. Not only can simultaneous events and fluents be modelled, A*QL* allows us to construct more complex queries using disjunctions and conjunctions of conditions. Furthermore, A*QL* allows us to reason with incomplete information, thus fully exploiting the reasoning power of the underlying ASP.

LTL[17] is a logic commonly used for model-checking domains involving linear time. In [14], answer set programming was put forward as a mechanism for model checking asynchronous concurrent systems a generalisation. They demonstrated that answer set programming could be used as bounded LTL model-checking for 1-safe Petri nets. Originally, LTL only refers to states and as a general observation, the merging of actions and fluents inside LTL is non-trivial as one is merging state-relative and transition-relative concepts. This tends[3] to lead to restrictions on formulae that can be used. The same is true for A*QL* . It can be shown that for a subset of a formulae, LTL-queries can be mapped to A*QL* queries.

As it stands A*QL* is already a very intuitive and versatile query and constraint language for actions domains. The language is succinct and does not contain any overhead (i.e. no operator can be expressed as a function of other operators). However, from a software engineering point of view, we could make the language more accessible by providing commonly used constructs as part of the language. To this end, we plan to incorporate constructs such as `eventually(F), never(F), always(F), before(F), before(E)`, and an if-construct to express conditions on events or fluents. For the same reasons, we plan to add time specific `happens(E,I)` and `hold(F,I)` and the possibility to construct general logical expression without the need for condition statements.

At the moment A*QL* only supports linear time. For certain domains, other ways of representing time might be more appropriate. While linear time assumes implicit universal quantification over all paths in the transition function, branching time allows for explicit existential and universal quantification of all paths and alternating time offers selective quantification over those paths that are possible outcomes. While linear and branching time are natural ways of describing time in closed domains, alternative time is more suited to open domains.

References

[1] A. Artikis, M. Sergot, and J. Pitt. Specifying electronic societies with the Causal Calculator. In F. Giunchiglia, J. Odell, and G. Weiss, editors, *Proceedings of Workshop on Agent-Oriented Software Engineering III (AOSE)*, LNCS 2585. Springer, 2003.

[2] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.

[3] Diego Calvanese and Moshe Y. Vardi. Reasoning about actions and planning in LTL action theories. In *Proc. KR-02*, 2002.

[4] Owen Cliffe. *Specifying and Analysing Institutions in Multi-Agent Systems using Answer Set Programming*. PhD thesis, University of Bath, 2007.

[5] Owen Cliffe, Marina De Vos, and Julian A. Padget. Answer set programming for representing and reasoning about virtual institutions. In Katsumi Inoue, Ken Satoh, and Francesca Toni, editors, *CLIMA VII*, volume 4371 of *Lecture Notes in Computer Science*, pages 60–79. Springer, 2006.

[6] Owen Cliffe, Marina De Vos, and Julian A. Padget. Specifying and reasoning about multiple institutions. In Javier Vazquez-Salceda and Pablo Noriega, editors, *COIN 2006*, volume 4386 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2007.

[7] Steve Dworschak, Susanne Grell, Victoria J. Nikiforova, Torsten Schaub, and Joachim Selbig. Modeling biological networks by action languages via answer set programming. *Constraints*, 13(1-2):21–65, 2008.

[8] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. The DLV^{K} Planning System. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, editors, *European Conference, JELIA 2002*, volume 2424 of *LNAI*, pages 541–544, Cosenza, Italy, September 2002. Springer Verlag.

[9] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system `dlv`: Progress report, comparisons and benchmarks. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann, San Francisco, California, 1998.

[10] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence, Vol. 153, pp. 49-104*, 2004.

[11] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-Driven Answer Set Solving. In *Proceeding of IJCAI07*, pages 386–392, 2007.

[12] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electron. Trans. Artif. Intell.*, 2:193–210, 1998.

[13] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: preliminary report. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 623–630, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[14] Keijo Heljanko and Niemelä. Ilkka. Bounded ltl model checking with stable models. In *LPNMR2001*, LNAI, pages 200–212. Springer-Verlag Berlin Heidelberg, 2001.

[15] I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, Berlin, July 28–31 1997. Springer.

[16] Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In *LPNMR '97: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 421–430, London, UK, 1997. Springer-Verlag.

[17] A. Pnueli. The Temporal Logic of Programs. In *19th Annual Symp. on Foundations of Computer Science*, 1977.

[18] Marek Sergot. $(\mathcal{C}+)^{++}$: An action language for modelling norms and institutions. Technical Report 8, Department of Computing, Imperial College, London, June 2004.

[19] Lifschitz Vladimir. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year perspective*, pages 357–373. Springer, 1999.